

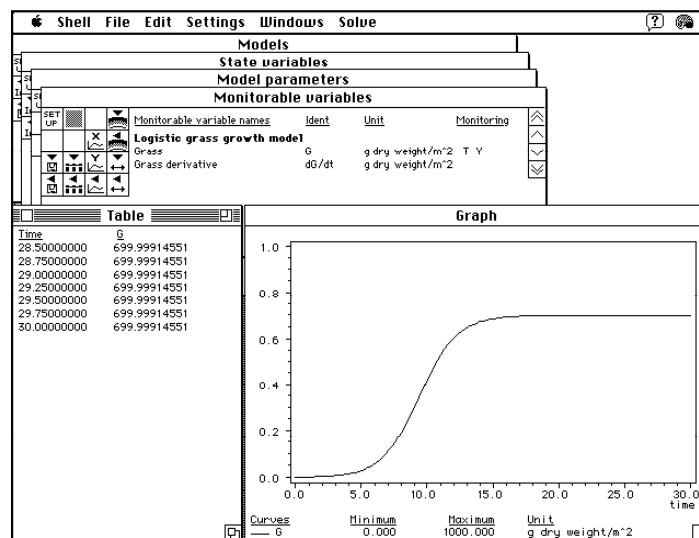
# ModelWorks 2.2

## An Interactive Simulation Environment for Personal Computers and Workstations

Andreas Fischlin

&

D. Gyalistras, O. Roth, M. Ulrich, J. Thöny,  
T. Nemecek, H. Bugmann and F. Thommen



Zürich, Mai / May 1994 (Revised Edition June 1996)

**Eidgenössische Technische Hochschule Zürich ETHZ**  
**Swiss Federal Institute of Technology Zurich**

Departement für Umweltnaturwissenschaften / Department of Environmental Sciences  
Institut für Terrestrische Ökologie / Institute of Terrestrial Ecology

The System Ecology Reports consist of preprints and technical reports. Preprints are articles, which have been submitted to scientific journals and are hereby made available to interested readers before actual publication. The technical reports allow for an exhaustive documentation of important research and development results.

Die Berichte der Systemökologie sind entweder Vorabdrucke oder technische Berichte. Die Vorabdrucke sind Artikel, welche bei einer wissenschaftlichen Zeitschrift zur Publikation eingereicht worden sind; zu einem möglichst frühen Zeitpunkt sollen damit diese Arbeiten interessierten LeserInnen besser zugänglich gemacht werden. Die technischen Berichte dokumentieren erschöpfend Forschungs- und Entwicklungsergebnisse von allgemeinem Interesse.

Adressen der Autoren / Addresses of the authors:

Dr. A. Fischlin, D. Gyalistras  
Systemökologie ETH Zürich  
Institut für Terrestrische Ökologie  
Grabenstrasse 3  
CH-8952 Schlieren/Zürich  
S W I T Z E R L A N D

F. Thommen  
Institut für Terrestrische Ökologie  
Institutsinformatik  
Grabenstrasse 3  
CH-8952 Schlieren/Zürich  
S W I T Z E R L A N D

Dr. O. Roth  
Widenstr. 3  
CH-8302 Kloten  
S W I T Z E R L A N D

EMAIL CONTACT:  
ramses@ito.umnw.ethz.ch

Dr. T. Nemecek  
Eidg. Forschungsanstalt für landw.  
Pflanzenbau Reckenholz  
Postfach  
CH-8046 Zürich  
S W I T Z E R L A N D

Dr. M. Ulrich  
Institut für Gewässerschutz und  
Wassertechnologie  
ETH Zürich  
EAWAG  
CH-8600 Dübendorf  
S W I T Z E R L A N D

J. Thöny  
AS-Informatik AG  
Mühlfangstrasse 16  
CH-8570 Weinfelden  
S W I T Z E R L A N D

Dr. H. Bugmann  
Potsdam-Institut für  
Klimafolgenforschung  
Postfach 601203  
D-11412 Potsdam  
G E R M A N Y

# **ModelWorks**

## **An Interactive Simulation Environment for Personal Computers and Workstations**

**Andreas Fischlin<sup>1</sup>**

**&**

**Dimitrios Gyalistras<sup>1</sup>, Olivier Roth<sup>2</sup>,  
Markus Ulrich<sup>3</sup>, Jürg Thöny<sup>1</sup>, Thomas Nemecek<sup>4</sup>,  
Harald Bugmann<sup>1</sup> and Frank Thommen<sup>1</sup>**

**ModelWorks Version 2.2  
Second Edition - Zürich, May 1994<sup>a</sup>**

### **Abstract**

ModelWorks is a modelling and simulation environment in Modula-2 specifically designed to be run interactively on modern personal computers and workstations. It supports modular modelling by featuring a coupling mechanism between submodels and an unrestricted number of state variables, model parameters etc. up to the limits of the computer resources. It allows for the formulation of continuous time, discrete time, discrete event models, as well as the free mixing of all these formalisms. Not only does ModelWorks offer the simulationist a handy user interface to experiment interactively with model systems, but also allows the modeller to use ModelWorks' functions via a client interface in any other programming context.

---

<sup>1</sup> Systems Ecology, Institute of Terrestrial Ecology, Department of Environmental Sciences, Swiss Federal Institute of Technology, Grabenstrasse 3, CH-8952 Schlieren/Zürich, Switzerland

<sup>2</sup> Widenstr. 3, CH-8302 Kloten, Switzerland

<sup>3</sup> EAWAG - Swiss Federal Institute of Water Resources, Water Pollution and Water Control, CH-8600 Dübendorf, Switzerland

<sup>4</sup> Station fédérale de recherches agronomiques, CH-1260 Nyon, Switzerland

<sup>a</sup> Revised Edition June 1996



# Contents

PREFACE.....	V
PREFACE TO THE SECOND EDITION .....	VIII
ACKNOWLEDGEMENTS .....	IX
READING HINTS .....	IX

## Part I - Tutorial

1 GENERAL DESCRIPTION.....	12
2 GETTING STARTED WITH THE SIMULATION ENVIRONMENT .....	19
2.1 The Sample Model .....	19
2.2 Simulating the Sample Model .....	20
2.2.1 Default simulation .....	22
2.2.2 Changing initial values .....	23
2.2.3 Changing parameters .....	23
2.2.4 Changing scaling .....	23
2.2.5 Changing monitoring .....	23
2.2.6 Changing parameters during simulation .....	26
2.2.7 Changing integration methods .....	26
2.2.8 Program termination .....	27
3 GETTING STARTED WITH MODELLING .....	28
3.1 The Model Definition Program of the Sample Model .....	28
3.2 Developing a New Model.....	31
3.2.1 The new model .....	31
3.2.2 Model definition program for the new model.....	32
3.2.3 Compilation of the new model .....	35
3.2.3 Simulation of the new model .....	36

## Part II - Theory

4 MODEL FORMALISMS .....	40
4.1 Elementary Models.....	40
4.2 Structured Models (Coupling of Submodels).....	43

5	FUNCTIONS .....	53
5.1	Simulation Environment .....	54
5.1.1	States of the Simulation Environment .....	54
5.1.2	Model Base .....	58
5.1.2.a	Model and model object installation and removal .....	58
5.1.2.b	Current values .....	58
5.1.2.c	Predefinitions, defaults, and resetting .....	59
5.1.2.d	Initialization of the simulation environment .....	63
5.1.3	Simulations and the Run-Time System .....	64
5.1.3.a	Elementary simulation run .....	64
5.1.3.b	Structured simulation (Experiment) .....	65
5.1.3.c	Integration respectively time step .....	68
5.1.3.d	Model objects and the run-time system .....	71
5.1.3.e	Client procedures and the simulation environment .....	71
5.1.3.f	Manipulating the model base at run-time .....	73
5.1.3.g	Monitoring .....	76
5.1.4	Standard User Interface .....	79
5.1.4.a	Multiple activations of the standard user interface .....	80
5.1.4.b	States of the standard user interface .....	82
5.1.4.c	IO-windows (Input-Output-windows) .....	82
5.1.5	User Interface Customization .....	84
5.2	Modelling .....	87
5.2.1	The Model Development Cycle .....	87
5.2.2	Structured Model Definition Programs (Modular Modeling) .....	88
5.2.3	Structured Simulations (Experiments) .....	88
5.2.4	Module Structure of ModelWorks .....	89

### Part III - Reference

6	STANDARD USER INTERFACE .....	94
6.1	Menus and Menu Commands .....	94
6.1.1	Overview over menus .....	95
6.1.2	Quit commands .....	95
6.1.3	Menu File .....	96
6.1.4	Menu Edit .....	99
6.1.5	Menu Settings .....	100
6.1.6	Menu Windows .....	107
6.1.7	Menu Solve .....	108
6.2	IO-Windows (Input-Output-Windows) .....	109
6.2.1	IO-window Models .....	110
6.2.2	IO-window State variables .....	113
6.2.3	IO-window Model Parameters .....	114
6.2.4	IO-window Monitorable variables .....	116
7	CLIENT INTERFACE .....	122
7.1	Declaring Models and Model Objects .....	124
7.1.1	Running a simulation session .....	124
7.1.2	Declaration of models .....	125
7.1.3	Declaration of state variables .....	128
7.1.4	Declaration of model parameters .....	130

7.1.5 Declaration of monitorable variables.....	131
7.1.7 Testing for the Presence of Objects .....	133
7.2 Accessing Defaults and Current Values .....	133
7.2.1 Global simulation parameters and project description .....	133
7.2.1.a Retrieval of read only current values .....	135
7.2.1.b Modification of defaults .....	135
7.2.1.c Modification of current values .....	136
7.2.1.c Resetting of current values to the defaults .....	137
7.2.2 Installed models and model objects .....	137
7.2.2.a Modification of defaults .....	138
7.2.2.b Modification of current values .....	138
7.2.2.c Resetting of current values to the defaults .....	139
7.2.2.d Model and model object attributes .....	139
7.2.2.e Access support for models and model objects .....	140
7.3 Removing Models and Model Objects .....	141
7.4 Simulation Control and Structured Simulation Runs .....	141
7.5 Display and Monitoring .....	144
7.5.1 Window operations .....	144
7.5.2 General monitoring .....	146
7.5.3 Stash filing .....	147
7.5.4 Graphical monitoring .....	148
7.5.5 Simulation environment modes .....	150
7.5.6 Setting of predefined defaults and global resetting .....	151
7.5.7 Customization of keyboard shortcuts for menu commands .....	151

## Appendix

A Sample Models .....	154
A.1 The Continuous Time Sample Models (DESS) of the Tutorial .....	155
A.1.1 The Sample Model “Logistic Grass Growth” - Logistic .....	155
A.1.2 The New Model - GrassAphids.....	156
A.2 A Discrete Time Model (SQM) - Insect .....	158
A.3 A Discrete Event Model (DEVS) - Diversity .....	162
A.4 Typical Applications .....	167
A.4.1 Batch Phase Portrait of Lotka-Volterra - LVPhasePlot .....	167
A.4.2 Interactive Phase Portrait of the Van-der-Pol Oscillator - VDPol ...	171
A.4.3 Animation of the Age Pyramid of the Swiss - SwissPop .....	175
A.4.4 Sensitivity Analysis - Sensitivity .....	182
A.4.5 Parameter Identification - GauseIdentif .....	187
A.4.6 Stochastic Simulations .....	195
A.4.6.1 Third Order Finite Markov Chain - Markov .....	195
A.4.6.2 Statistical Analysis of Simulation Results - StochLogGrow .....	205
A.4.7 Modular Modeling - GreenHouse .....	209
A.5 Mixed Type Structured Models .....	224
A.5.1 Mixing Continuous (DESS) and Discrete Time Models (SQM) .....	224
A.5.2 Mixing a Discrete Event System (DEVS) With a Continuous Time Model (DESS) - CarPollution .....	227
A.5.2.1 The Fiscrete Event System - Traffic(DEVS) .....	227
A.5.2.2 The Crossroad and the Traffic.....	231
A.5.2.3 Adding Traffic’s Air Pollution - Pollutants (DESS) .....	239
A.5.2.4 Putting All Together.....	241

A.6 Research Sample Models .....	243
A.6.1 Population Dynamics of Larch Bud Moth - LBM.....	243
A.6.2 Discrete Event Harvesting In a Continuously Growing Forest - ForestYield.....	254
B Literature .....	269
C ModelWorks Versions and Implementations .....	272
D Use and Definitions of ModelWorks and Library Modules.....	273
D.1 ModelWorks Mandatory Client Interface .....	273
D.2 ModelWorks Optional Client Interface.....	273
D.2.1 SimEvents.....	273
D.2.2 SimDeltaCalc.....	278
D.2.3 SimGraphUtils .....	281
D.2.4 SimIntegrate .....	288
D.2.5 SimObjects .....	290
D.3 Auxiliary Library .....	293
D.3.1 IdentifyPars.....	293
D.3.2 JulianDays .....	296
D.3.3 Queues .....	299
D.3.4 RandGen .....	301
D.3.5 RandGen0 .....	302
D.3.6 RandGen1 .....	304
D.3.7 RandNormal .....	306
D.3.8 ReadData .....	309
D.3.9 StochStat.....	313
D.3.10 StructModAux .....	317
D.3.11 TabFunc .....	320
D.3.11.a User Interface.....	320
D.3.11.b Declaration of table functions .....	323
D.3.11.c Modification of table functions .....	325
D.3.11.d Inter- and extrapolations with table functions .....	326
D.3.11.e Removing table functions .....	327
D.3.12 WriteDatTim.....	327
E Quick References.....	329
E.1 Auxiliary Library .....	329
E.2 Dialog Machine .....	335
E.3 ModelWorks Client Interface .....	342
INDEX .....	347



## Preface

ModelWorks is a simulation environment to solve dynamic systems as they are used in biology, physics, chemistry, environmental and engineering sciences to model various processes. It is also particularly well suited to be used by university students during a modelling course (FISCHLIN *et al.*, 1987; MANSOUR & SCHAUFELBERGER, 1989; FISCHLIN, 1992)<sup>1</sup>. ModelWorks can be used for simple didactic models as well as for very complex research models (NEMECEK, 1993). ModelWorks forms part of the even more powerful RAMSES<sup>2</sup> software (FISCHLIN, 1991), consisting of tools which are particularly tailored to aid researchers who wish to model and simulate complex, environmental or other so-called ill-defined systems (CELLIER & FISCHLIN, 1980).

ModelWorks allows to work with an arbitrary number of dynamic models described by differential, difference equation systems, or by the discrete event formalism. A global model can be separated into possibly hierarchically organized submodels which exist as independent units communicating via output-input coupling. Modular and hierarchical modelling is supported, which is particularly useful if for instance one wishes to keep experimental results clearly separated from a theoretical, mathematical model by formulating them as a parallel model, or to enhance model clarity, or to build model libraries. Discrete and continuous models can be combined in one global model system, with correct data exchange controlled by the simulation environment.

Simple mathematical models can be built with only minor programming knowledge, whereas programming experts have full access to a powerful programming language and may expand into any realm of sophisticated calculations still profiting from the simulation environment and numerical algorithms provided by ModelWorks. Hence in contrast to most existing simulation software ModelWorks fully supports the researcher during a model development process, which often starts with a first, crude model and ends with the most sophisticated, in every detail refined research model<sup>3</sup>.

ModelWorks is based on a high-level programming language which has been selected considering the following criteria: It has been formally defined; it is general and powerful enough to support not only numerical computations, but also a window based, graphical user interface; on the other hand it is also simple enough to be comprehended and mastered by the non-computer scientist having learned programming in a basic computer science course, such as for instance taught in Pascal programming courses; on the other hand it also offers support for the development of large and complex models for the expert; finally and not the least, the language is available in efficient implemen-

---

<sup>1</sup>For the cited literature see the chapter *Literature*

<sup>2</sup>RAMSES is an acronym for Research Aids for Modeling and Simulation of Environmental Systems. For more information on the concepts of RAMSES see FISCHLIN (1991).

<sup>3</sup>ModelWorks does not force the modeler to discard the simulation software together with all other investments in learning, implementation, and testing time, or any compatibility issues, when he reaches the limits of the simulation language; on the contrary, ModelWorks avoids the risk of having to restart with the model implementation all over again in a high-level programming language, since it does so from the very beginning. In contrast to a simulation language a well designed, general purpose, high-level programming language guarantees that anything which can be computed on a computer can be realized. It appears that one of the reasons why so many experienced researchers almost never use simulation software but use instead general-purpose high-level programming languages is that they avoid the risk to have to switch techniques in the middle of a project. However, to work in a high-level programming language only, requires to reinvent the functionality of a simulation software package, a task often surmounting the modeling research problem at hand by many orders of degree.

tations on many machines as e.g. Apple® Macintosh®<sup>1</sup>, IBM® personal computers<sup>2</sup>, or Sun® workstations<sup>3</sup>. Therefore we have chosen Modula-2 as the programming language to be used for ModelWorks, currently meeting all the listed requirements closest (WIRTH, 1988). Due to this approach ModelWorks could be designed as a fully open system, which can be expanded or customized by the user to any purpose she desires.

ModelWorks consists of a set of library modules written in Modula-2, which contain the program parts common to any simulation, such as numerical integration algorithms, and the tabular plus graphical display of the simulation results, or the interactive changing of model or other simulation parameters. The variable portion, the model of interest, is to be supplied by the user in the form of a standard Modula-2 program. It describes the model's properties and installs the model in the simulation environment by means of the so-called client interface of ModelWorks. Modelling and simulating with ModelWorks includes therefore several steps: a) In the role of the modeller the writing of the model definition by preparing a Modula-2 program, and b) in the role of the simulationist the execution of the model definition program within ModelWorks' simulation environment to produce and observe the model's behaviour.

Interactive modelling and interactive simulations are supported in ModelWorks in several ways. The standard user interface of ModelWorks provides an interactive access to the simulation environment. For instance it allows to change interactively all settings, including any simulation parameters such as the integration method or the step length, model parameters and/or initial values of the state variables, plus selection of the display of simulation results. Simulation results are made visible to the user by the so-called system behaviour monitoring concept: Values of any variable may be written onto a file for future reference, written into a table, or displayed as curves in line-charts. All data can be reset to a given default value. Further, the model's data structure are all stored dynamically. This allows the user to install an unlimited number of models of an arbitrary size, with an arbitrary number of variables each, up to the limits of the hardware. Finally, because of ModelWorks open system design, it allows to extend and customize the user interface, for instance by adding new functions or by using only some of its functions, so that the user needs can be met as closely as possible. ModelWorks software architecture has been especially designed to support such uses, needs which we consider to be generally of interest for researchers working with complex, non-linear model systems.

ModelWorks simulation environment is based on the "Dialog Machine"<sup>4</sup>, guaranteeing a consistent user interface and has originally been implemented using MacMETH, a fast and efficient Modula-2 language system for the Apple® Macintosh® computer (WIRTH *et al.*, 1992). ModelWorks simulation environment runs on any machine on which the "Dialog Machine" is available. If this is the case, an efficient and smooth port of ModelWorks in a few days work is possible. Currently ModelWorks is available for Macintosh computers with at least 512 KBytes of memory (RAM) plus at least two floppy drives and IBM® PCs which run under MS DOS and have 640 KBytes of memory (RAM) plus a hard disk. For more details on particular implementations and hard plus software requirements for specific versions, see the *Appendix*. This text serves as a manual for the ModelWorks software. Since all versions are very similar and differences are the exceptions, there exists only this one text. The remaining differences between the versions are only minor and therefore just briefly mentioned wherever the user is likely to encounter difficulties without knowing the details.

---

<sup>1</sup>Macintosh is a registered trademark of Apple® Computer, Inc.

<sup>2</sup>IBM is a registered trademark of International Business Machines Corporation.

<sup>3</sup>Sun is a registered trademark of Sun Microsystems, Inc.

<sup>4</sup>See the appendix for availability and the separate booklet «Installation Guide and Technical Reference of the RAMSES software» installation of the "Dialog Machine"

This text is subdivided into three parts: Part I is a *Tutorial* containing a little tour to be followed step by step. It suffices to learn all basic techniques, which are needed in order to model and simulate simple models with ModelWorks. Part II explains the *Theory* and concepts behind ModelWorks, in particular model formalisms and all functions of ModelWorks. Any advanced modelling, such as modular modelling, requires to study the theoretical part. Part III is a *Reference* manual containing a complete list and detailed description of all features of ModelWorks. Finally the *Appendix* contains sample models, the cited literature, a short explanation of the ModelWorks versions, descriptions of ModelWorks' client interfaces and library modules, convenient quick reference listings, and an index. For detailed instructions for the installation and other technical details refer to the separate booklet «Installation Guide and Technical Reference of the RAMSES software».

## Preface to the Second Edition

This second edition has been adapted to the changes and amendments made to the ModelWorks simulation environment during the last years. ModelWorks has been widely used in the context of several modelling and simulation research projects, the largest spanning three to four years. The hereby gained experience was used to redesign ModelWorks. Besides the many modifications the following are of major importance:

First the basic functionality of ModelWorks has been extended. It allows now to solve discrete event models (DEVS), which are formulated according to the so-called event scheduling paradigm (KREUTZER, 1986). In particular it is also possible to mix all three supported model formalisms in any combination, i.e. structured model systems can be built from continuous time, discrete time, as well as discrete event components. This includes the exchange of data among any model type.

Second, all ModelWorks functions can be used in a more flexible way than this was possible with the original design. In particular, all its functions can be used dynamically any time (e.g. model declaration or removal in the middle of a simulation run) and independently from each other (e.g. just the IO-window for the model parameters). The latter allows even to bypass the user interface completely and to use ModelWorks only as a batch simulator. Thus ModelWorks can be used also within RAMSES for interactive modelling, experiment definition, simulation, and post-simulation analysis (FISCHLIN, 1991) and can now support the concept of simulation servers (THOENY *et al.*, 1994) in a computer network.

Consequently, this text has been completely revised. In particular the part I *Tutorial* has been modified to explain the use of the «Mini RAMSES Shell». The whole of part II *Theory* as been completely rewritten: First, chapter *Model Formalisms* to define DEVS and the new coupling when building structured models from any of the three standard formalisms; second, chapter *Functions* to describe the new dynamic functionality of the simulation environment. The part III *Reference* has been adapted to the actual implementation of ModelWorks 2.2. The *Appendix* has been completely rewritten, in particular does it now contain many more sample models demonstrating a vast range of typical uses of ModelWorks.

Note, despite all these changes, no functionality available in former versions of ModelWorks had to be sacrificed. ModelWorks version 2.2 warrants full upward compatibility (including module keys) with all earlier versions.

Zürich, May 1994

Andreas Fischlin

## Acknowledgements

The authors wish to express many thanks to Prof. Dr. Walter Schaufelberger<sup>1</sup>, formerly Project Centre IDA from the Swiss Federal Institute of Technology Zürich (ETHZ), not only for his substantial support, but also for his unceasing encouragement, which made this research and development only possible.

The research behind this software has been supported by the Swiss Federal Institute of Technology Zürich (ETHZ) and the Swiss National Science Foundation grants Nr. 31-8766.86 and Nr. 31-31142.91.

## Reading Hints

Please be not irritated: Throughout this text references to persons are made by using the female form; yet, the text is valid not only for women but also for men.

Throughout this text *italics* are used to emphasize that this text is to be taken literally, in particular also case sensitive. This is the case for instance in the citation of an identifier, such as a module name like *SimMaster* or if the user has to open a file or directory with a given name such as *Logistic.OBM* or *\MW\SAMPLES*.

For easier orientation, the pages, figures and tables in Part I *Tutorial* and II *Theory* are prefixed with the letter T, in part III Reference with the letter R, and in the *Appendix* with the letter A. Within parts figures and tables are numbered separately, starting e.g. with Tab. T1 respectively Fig. A1, but pages are numbered consecutively throughout the whole text.

---

<sup>1</sup>Current address: Institute of Automatic Control, Swiss Federal Institute of Technology Zürich (ETHZ), ETH-Zentrum, CH-8092 Zürich, Switzerland



# Part I - Tutorial

This tutorial describes the elementary usage of ModelWorks, i.e. you learn how to develop and simulate models using ModelWorks.

The first chapter, *General Description*, describes the general, fundamental concepts of ModelWorks.

The second chapter, *Getting Started with the Simulation Environment*, contains a step by step explanation for running an existing model and getting familiar with the simulation environment of ModelWorks.

The third chapter, *Getting Started with Modelling*, teaches how to develop new models.

Having read this tutorial you will be able to develop and simulate your own, simple models. However, if you are interested in more complex models and more advanced techniques, this tutorial is not sufficient. In order to learn the more sophisticated features of ModelWorks you should read part II *ModelWorks Theory* and the second chapter, *Client interface*, of part III *Reference*. They contain a full and complete description of all possibilities ModelWorks offers.

This tutorial is best read while having access to a computer and the described steps are actually executed<sup>1</sup>. This requires that the reader is already familiar with her computer and the usage of its software, in particular the choosing of menu commands, clicking on objects (i.e. object selection), and the dragging of objects (e.g. moving the scroll box in a scroll bar). Moreover it is assumed that the user knows how to operate a simple programming editor (e.g. the desk accessory *MockWrite*), has a basic knowledge of the programming language Pascal or Modula-2 and is familiar with the mathematics involved with modelling and simulation of differential equation systems. No particular information is provided on these topics. Please refer to other texts if you should have any difficulties with any of these subjects<sup>2</sup>. The separate booklet «Installation Guide and Technical Reference for the RAMSES software» contains information on how to proceed in order to install the ModelWorks software.

**Reading Hint:** For easier orientation, the pages, figures and tables of Part I *Tutorial* are prefixed with the letter T.

---

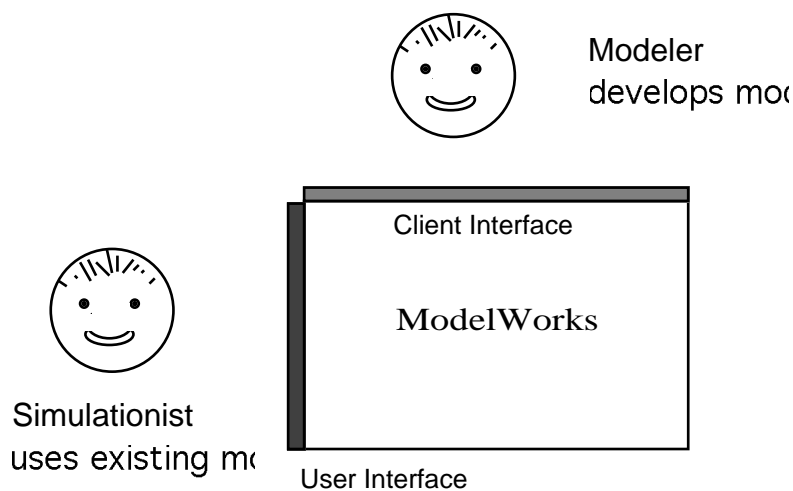
<sup>1</sup>Note that the following text assumes that you will work with the original ModelWorks version as available on the Macintosh<sup>®</sup> computer. If you have no access to a Macintosh<sup>®</sup> computer, the instructions are to be executed similarly, but may look a bit differently or behave slightly differently, since the IBM<sup>®</sup> PC version of the "Dialog Machine" is only a subset of the Macintosh<sup>®</sup> version. A few hints: On the IBM<sup>®</sup> PC folders become directories, object files ending with the extension "OBM" become linked GEM applications with the extension "APP", and in contrast to the Macintosh MS DOS file names are truncated to 8 characters (extension excluded); note that the latter may also affect module names. For more details see the appendix. Wherever necessary, IBM<sup>®</sup> PC specific information has been added in form of footnotes. Please interpret the text accordingly and accept our apology for not being able to offer an IBM<sup>®</sup> PC text version; note that we are a research institution, not a commercial software company, and hence not able to maintain more than that version we use ourselves in our daily research work; however, you should have no difficulties in following the tutorial text, since all essential features of ModelWorks are available on the IBM<sup>®</sup> PC version as well.

<sup>2</sup>We recommend: Operation of the computer: Your owner's guide, e.g. *Macintosh owner's guide*. Modula-2: WIRTH, N. 1988. *Programming with Modula-2*. Springer-Verlag, Heidelberg, New York, 4th corrected ed. Modelling: LUENBERGER, D.G., 1979. *Introduction to dynamic systems - Theory, models, and applications*. Wiley, New York, 446pp.

## 1 General Description

ModelWorks is an interactive modelling and simulation environment to study the behaviour of dynamic models, which are described by differential, difference equations, or discrete events. Any system described by a set of coupled, ordinary differential, ordinary difference equations, or instantaneous state transition functions formulated as discrete events can be modelled using ModelWorks. Since ModelWorks features modular modelling, it is also possible to mix models of different types or to integrate several differential equation systems simultaneously with different integration methods.

ModelWorks has two interfaces to communicate with the human user: the user interface of the simulation environment for the simulationist and the client interface for the modeller who builds models (Fig. T1).



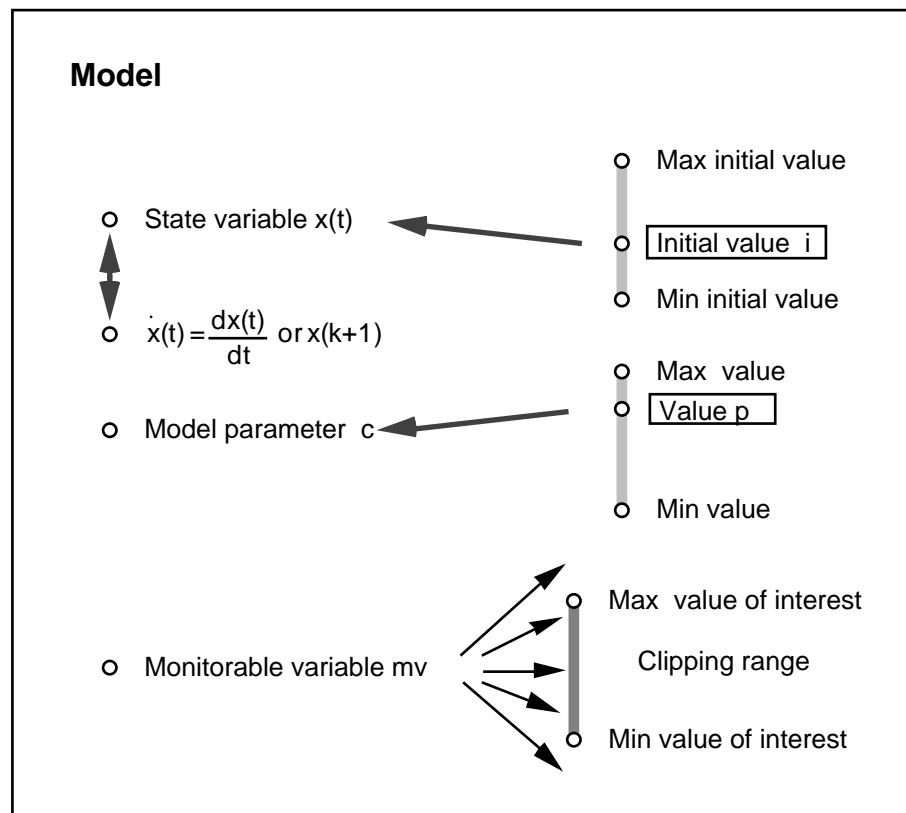
**Fig. T1:** The two interfaces of ModelWorks: The modeller uses the client interface for the model development, the simulationist uses the user interface of ModelWorks' simulation environment to perform simulation experiments with an already existing model. Typically the modeller and the simulationist are one and the same person changing just roles.

Typically the modeller and the simulationist are one and the same person. However their roles are distinct and should be clearly separated: The modeller defines all properties of a simulation model, i.e. she specifies a model definition. This includes the specification of the model's mathematical properties and its objects, such as equations, state variables, and parameters, plus the objects' default values and ranges. It is also the modeller who implements the model by writing a ModelWorks model definition program.

The simulationist runs interactive simulation experiments, hereby using one or several models, which have been constructed by the modeller. She is restricted to use these models within certain limitations which have been specified by the modeller, but within that range, she may interactively define and execute with the model any kind of experiment she wishes. For instance she may observe its temporal behaviour, sample points from particular trajectories, modify parameter values within a defined range, or run a sensitivity analysis. ModelWorks contains all elements and algorithms needed for computer simulations, such as numerical integration algorithms, the interactive changing of parameter values, and the display of simulation results. The only exception of course is the model itself, which has to be provided by the modeller.



Normally a ModelWorks model definition consists of several objects, which belong to various classes. First there must be present at least one model; but the model definition may consist of any number of models. Second, normally each model is associated with several objects like model equations, state variables, model parameters, auxiliary variables, and monitorable variables. Such objects are called model objects (Fig. T2).



**Fig. T2:** Model objects (○) of a ModelWorks model: A ModelWorks model definition must consist of at least one model and every model usually contains state variables, model parameters, and monitorable variables. Any initial value, parameter value, minimum, or maximum value becomes mandatory, if the associated variable or parameter is declared within the model definition. ModelWorks maintains the actual values of state variables, parameters, and monitorable variables and even remembers their initially specified values (default values): ← : ModelWorks automatically assigns the initial value  $i$  to the state variable  $x$  at the begin of every simulation run, and the value  $p$  is assigned to the model parameter  $c$  upon entering the ModelWorks simulation environment or after any interactive change. ⇕ : ModelWorks uses the derivative or new value in order to compute and repeatedly assign newly obtained values to the state variable during the course of a simulation run (numerical integration). → : During simulation experiments the unknown values, which the monitorable variable  $mv$  may obtain, shall be drawn in graphs only if they fit within a particular range of interest; otherwise ModelWorks will clip them from the display.

A model is always of a particular type, i.e. either continuous time or discrete time. This type is given by the kind of equations which belong to the model: In the case of continuous time the model equations are ordinary differential equations or discrete event instantaneous state transition functions, in the case of discrete time they are ordinary difference equations. Note however, that a ModelWorks model definition program may be structured, i.e. it consists of several models which may be of a differing type, i.e.

some models may be continuous time other discrete time. In the latter case results a so-called mixed continuous and discrete time model definition.

A model may consist of any number of model equations. However, they must be given as explicit, either first order differential equations, first order difference equations, or instantaneous state transition functions. E.g. the following differential equation describing the Van-der-Pol oscillator

$$\ddot{y} + \mu(y^2 - 1)\dot{y} + y = 0$$

is not in the proper form, since it is neither explicit nor is it first order. On the other hand, the same equation reformulated<sup>1</sup> as a system of explicit, coupled first order differential equations

$$\begin{aligned}\dot{x}_1 &= x_2 \\ \dot{x}_2 &= \mu(1 - x_1^2)x_2 - x_1\end{aligned}$$

is now suitable to be used directly as a set of ModelWorks model equations. The second form is called the state variable form. Most differential or difference equations can be formulated in this form.

Usually each model uses a number of state variables. Each state variable must be associated with a second variable used as its first order derivative in the case of continuous time, or its new value in the case of a discrete time model. The model equations are formulated as expressions capable of defining the values of the derivative or new value. The expression may be an arbitrary function of any of the other model objects, such as state variables, auxiliary variables, or model parameters. Every state variable must be associated with a particular initial value and a range within which it may be changed interactively (Fig. T2).

Every model may have any number of model parameters, each associated with a particular value and a range within which it may be changed interactively. Typically model parameters are not or only rarely changed in the middle of simulation experiments (Fig. T2).

Intermediate results from an expression may be stored in a variable which will be later used in another expression. Such auxiliary variables are often used to compute complex expressions defining the value of a derivative of a state variable. In a ModelWorks model definition program the modeller may use any number of auxiliary variables. However in the current version, ModelWorks does neither especially recognize or support such variables nor does it hinder the modeller to use them in whichever way she wishes.

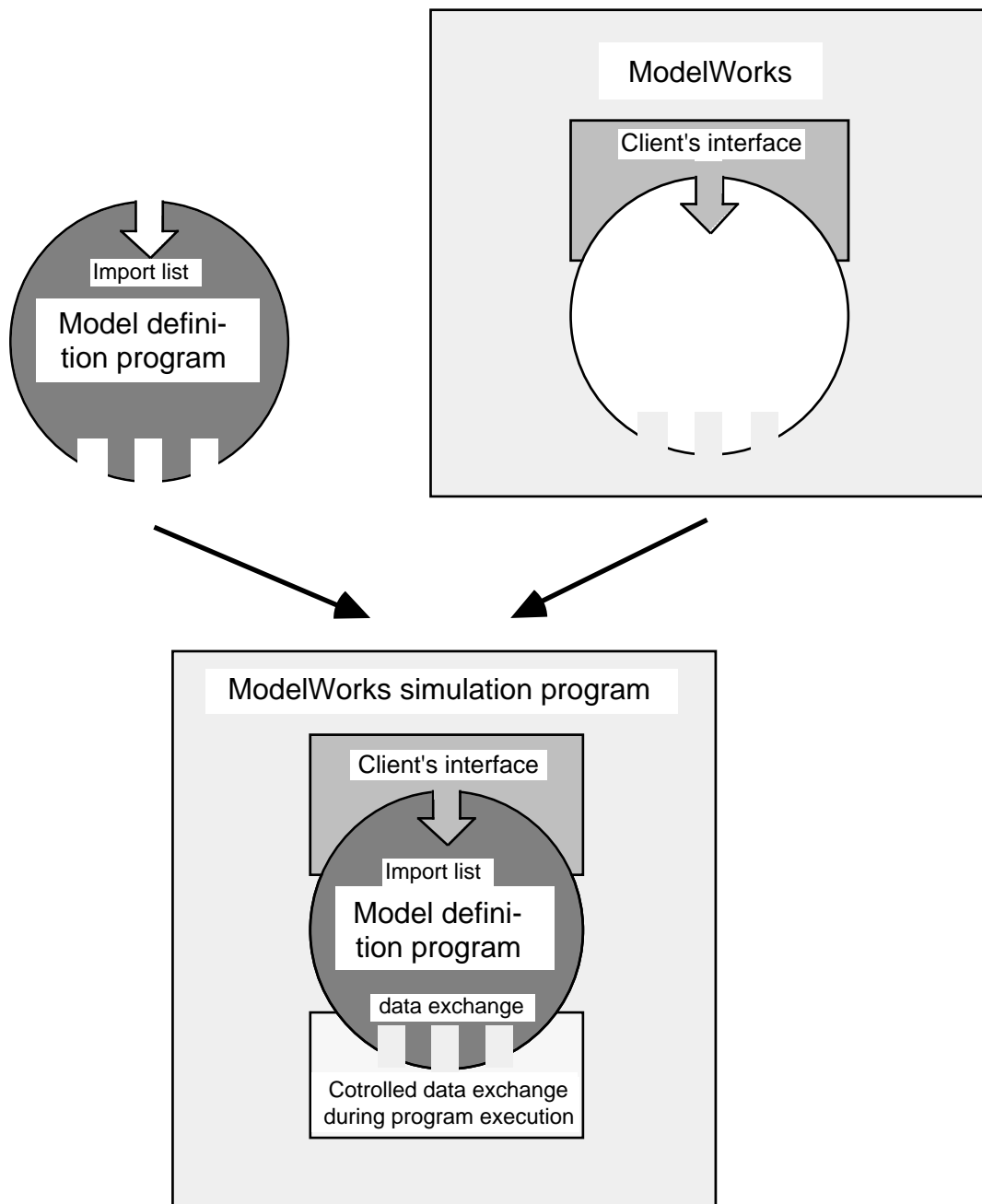
Finally models may have any number of monitorable variables. They are used to monitor the current values of any variable or otherwise accessible real numbers used in the ModelWorks model definition program. Each monitorable variable is associated with a clipping range used for the graphical display of the simulation results (Fig. T2).

All values specified by the modeller are remembered by ModelWorks as the so-called default values. The values currently in use by the simulation environment are called the current values. While starting the model definition program, ModelWorks assigns the default values to the current values. This is called a reset. Any time the simulationist wishes to do so, she may execute a further reset of a specific class of values, so that their current values are overwritten with their defaults. This mechanism is most useful

---

<sup>1</sup>From the definitions  $x_1 = y$  and  $x_2 = \dot{y}$  follows  $\dot{x}_2 = \ddot{y}$ , i.e. the variable substitutions  $\ddot{y} \rightarrow \dot{x}_2$ ,  $\dot{y} \rightarrow x_2$ ,  $y \rightarrow x_1$ ; rearrange resulting two equations to make the derivatives explicit.

if the simulationist wants to resume a well defined state before continuing with her work, especially after having made many and complex interactive changes.



**Fig. T3:** Organization of ModelWorks: ModelWorks is the constant part common to any simulation program forming the simulation environment. The variable part, the model definition program, describes the actual model to be simulated. Both units form together the final simulation program. They are linked by procedures provided via the client interface and which support mutual data exchange.

Ranges for initial values of state variables or model parameters are defined solely by the modeller. They become effective only in the simulation environment while the simulationist edits the current values of these model objects. ModelWorks guarantees that the simulationist assigns only values to an initial value of a state variable or a model parameter which lie within these ranges. Hence the modeller can use this mechanism to enforce limits within which the model equations are still valid in order to

reduce the danger that the simulationist runs a meaningless simulation experiment or encounters a fatal error condition. However, the clipping ranges for monitorable variables behave differently and should not be confounded with range limits: The simulationist can change clipping ranges interactively anytime.

ModelWorks has been designed to make modelling as easy as possible, yet as powerful and flexible as possible. Hence, for ModelWorks a model is a variable, not predefined portion of a simulation program, which has been left out so that the modeller may define it at a later time (Fig. T3). A user of ModelWorks wishes to define freely this open portion according to her current needs, for instance by specifying a new set of coupled differential equations. The modeller does it by writing simple Modula-2 statements, which are to be filled in and linked to the remaining, constant parts of ModelWorks. This is similar to a key which fits into a matching hole of a lock, only the two together rendering the lock into a fully functional unit.

With the model definition program the modeller provides the missing key. The key must conform to certain rules in order to fit into the hole. However, in all other aspects this analogy breaks down, since a key is not constructed before each use anew, or must not be extended, or has not its own particular functionality; the latter are all typical properties of ModelWorks model definition programs only.

The remaining parts of the simulation environment, i.e. the actual ModelWorks, can not be modified and constitute the pre-programmed ModelWorks software. They are general and hence common to any simulation program and resemble the lock with a hole for the key. When the simulationist starts a model definition program containing a model definition, the latter is inserted automatically into the hole of ModelWorks and what results is a fully functional simulation program (Fig. T3).

Technically a model definition program is a simple Modula-2 program module. Its main purpose is to define (declare) your model and its model objects, thus preparing the data exchange needed for simulations. ModelWorks does not care how the modeller organizes the structure of the model definition program and actually knows almost nothing about anything the modeller does in her program. The only objects ModelWorks cares about are: models, state variables to be integrated numerically, model parameters to be changed interactively from within the simulation environment, and monitorable variables for the monitoring of the simulation results. Hence, they are the only objects which have to be made known, i.e. declared, to ModelWorks.

The link of models and their model objects to ModelWorks is achieved via the client interface. In its essence it consists of two library modules: *SimBase* and *SimMaster*. These modules provide all Modula-2 objects (types and procedures) needed to describe a model in the model definition program.

Executing a ModelWorks model definition program means to start first the simulation environment. When it is entered, ModelWorks initializes the whole environment, in particular the global simulation parameters and typically executes all model and model object declarations as programmed by the modeller in the model definition program. It then performs a reset of all current values using all the defaults specified during the declarations. Subsequently ModelWorks is ready to execute commands entered by the simulationist, such as a simulation run, the execution of a simulation experiment, or the editing of the current values, e.g. of a model parameter or an initial value.

ModelWorks is not just another simulation language, since a model definition program is written as a plain Modula-2 program text. As a consequence ModelWorks can not automatically sort the statements which compute derivatives. Compared with other si-

mulation software, e.g. ACSL®<sup>1</sup>, this may be considered to be a draw-back. However, experience shows that automatic sorting of statements is error prone, if one models complex and ill-defined systems. Moreover, the greater flexibility offered by the host language Modula-2, a modern, powerful, and formally defined programming language, often outweighs the lack of automatic sorting, which is mostly not much more than a little inconvenience if the model definition has been carefully worked out before its implementation.

Most models maintain tight relationships among their objects such as state variables, parameters, and auxiliary variables etc. The modeller may keep logically connected objects close together, by defining related objects local to the model boundary. The latter normally coincides with the boundary of the scope of a Modula-2 module. Moreover, the modeller is free to use any Modula-2 feature she wishes: For instance model objects may be part of a complex data structure or the model definition may be spread over any number of modules, thus supporting modular modelling. This extendibility is one of the strongest features of ModelWorks.

Even if one is not familiar with the programming language Modula-2 but knows Pascal, it is feasible to use ModelWorks. On the other hand, ModelWorks is powerful and flexible enough to allow also the advanced modeller to develop sophisticated models.

Note that with ModelWorks the modeller has not only full access to all features of Modula-2, but also to those of the "Dialog Machine"<sup>2</sup>. The "Dialog Machine" is a generally applicable software layer between an application program such as ModelWorks and the system software respectively hardware. In this situation the user interacts via the latter (mouse, keyboard, screen) only indirectly with the application; the "Dialog Machine" intercepts all user interaction and filters it according to a simple user interface. The "Dialog Machine" substantially facilitates the writing of interactive programs. Not only does it simplify the programming of sophisticated dialogues, but also does it ensure automatically a consistent man-machine interface. Hence it allows the modeller to extend the standard, predefined ModelWorks simulation environment easily, efficiently, and without forcing her first to become a computer scientist; yet it supports an easy programming of windows, menus, bit-mapped graphics, plus mouse input. Moreover, the resulting program will be user-friendly: Thanks to the dialogue capabilities of the "Dialog Machine", the simulationist will be able to enjoy the use of a simulation program, which automatically conforms to a robust man-machine interface. This offers the advanced modeller to concentrate on the modelling process, instead of being distracted by the cumbersome and complex implementation details of user-interface problems. The easy access to the "Dialog Machine" is another strength of ModelWorks.

For instance the modeller may wish to extend the simulation environment by programming her own graphical monitoring in an additional, separate window or by adding further, customized functions to the simulation environment, i.e. by installing more menus offering additional menu commands. To give an example: ModelWorks and the "Dialog Machine" have been successfully used to program an interactive modelling environment<sup>3</sup>, which allows to enter differential equations and model objects at run time, without having to resort to any programming at all.

---

<sup>1</sup>ACSL® is a proprietary simulation software program that is leased with restricted rights according to license agreement and terms and conditions by Mitchell and Gauthier Associates, Inc. (USA), Concord, MA, respectively by Rapid Data Ltd. (Europe), Worthing, Sussex, UK.

<sup>2</sup>The "Dialog Machine" has been designed by Andreas Fischlin, implemented by Andreas Fischlin, Olivier Roth, Klara Vancso, and Alex Itten during the pilot project CELTIA under the auspices of Walter Schaufelberger. This work has been supported by the Swiss Federal Institute of Technology ETHZ, Zürich, Switzerland and by the Swiss National Science Foundation Grant Nr. 31-8766.86.

<sup>3</sup>This environment is the RAMSES session Modeling and Experiment Definition (FISCHLIN, 1991).

Despite the many features ModelWorks offers, typical model definition programs are written in a simple, standard format. Hence, as long as one develops models without any sophisticated extras, even the beginning programmer can quickly learn to use ModelWorks successfully. Finally, as a simulationist only, there is no need to know anything about the more advanced features of ModelWorks, since ModelWorks itself has been implemented by means of the "Dialog Machine". For instance, under-graduate students at the ETHZ have been able to work successfully with ModelWorks model definition programs within a learning time of only a few minutes.

## 2 Getting Started with the Simulation Environment

When you read this chapter and follow the instructions given, you learn step by step, how to run simulation experiments with ModelWorks. In particular you learn how to produce behaviour trajectories of a sample model and how to change a model's initial and parameter values using the ModelWorks simulation environment.

It is assumed that you know how to operate the computer you are using, its operating system, and typical application software, and that you have ModelWorks installed<sup>1</sup> and are ready in order to actually perform the described procedures on your computer while reading this chapter.

### 2.1 The Sample Model

The sample model is a simple growth model for grass. It models in a crude way the growth of real grass by assuming logistic growth. In the first phase, the plants grow exponentially under optimal conditions. Within a given, constant time interval (doubling time), the density doubles. With increasing density, limiting factors, such as nutrients, light energy, or competition by the neighbouring plants, become more important. This results in a decrease of the growth rate, expressed as a self-inhibition of the plants. Finally, the grass density reaches a maximum, the so-called carrying capacity determined by the plant's environment.

The following non-linear differential equation describes the model:

$$dG(t)/dt = c_1G(t) - c_2G(t)^2 \quad (1)$$

where

State variable:

grass (g dry weight per m <sup>2</sup> ):	G(t)
Initial amount of grass/initial value:	G(0) = 1.0 g/m <sup>2</sup>

Model parameters:

grass growth rate (day <sup>-1</sup> ):	c <sub>1</sub> = 0.7 day <sup>-1</sup>
Self-inhibition coefficient(m <sup>2</sup> g <sup>-1</sup> day <sup>-1</sup> ):	c <sub>2</sub> = 0.001 m <sup>2</sup> g <sup>-1</sup> day <sup>-1</sup>

Let us have a closer look at the model and its equation. The model has one state variable, the grass density G(t), which is a function of time. Further, it has two constant model parameters, c<sub>1</sub> and c<sub>2</sub>. The first term of the differential equation, c<sub>1</sub>G(t), describes the exponential growth phase of the plants; the second, - c<sub>2</sub>G(t)<sup>2</sup>, is responsible for the self-inhibition.

The unknown element in Eq. (1) is the function G(t). During a simulation, this function is approximated by calculating a sequence of values G(t<sub>0</sub>), G(t<sub>1</sub>), G(t<sub>2</sub>)... given the initial value G(t<sub>0</sub>). Since G(t) is defined by a differential equation these computations correspond to a particular solution of Eq. (1). In other words: By numerical integration ModelWorks produces the trajectory going through the point G(t<sub>0</sub>), i.e. solves an initial

---

<sup>1</sup>An exact description on how to install ModelWorks is given in the separate booklet «*Installation Guide and Technical Reference of the RAMSES software*». Please follow these instructions exactly, otherwise you may have difficulties while executing the described steps.

value problem. The sample model with the differential equation (1) has been pre-constructed and is ready for execution<sup>1</sup>.

## 2.2 Simulating the Sample Model

To simulate the sample model we recommend to use whenever possible the «RAMSES Shell». The «RAMSES Shell» represents a handy utility, which allows you to model and simulate more conveniently than this would be the case with ModelWorks alone. This is especially the case on the Macintosh® if you run the «RAMSES Shell» in the mode «Mini RAMSES Shell» under System 7. Note however, without the «RAMSES Shell»<sup>2</sup> all essential ModelWorks functions are always available, only the degree of convenience may vary.

A main purpose of the «RAMSES Shell» is to maintain for you a consistent working environment and to support you during your work. For instance does the «RAMSES Shell» execute automatically repetitive tasks, such as compiling or executing a model whenever it is needed or it remembers which model you worked with, i.e. the so-called work object, when you turned your machine off the last time etc.

The «RAMSES Shell» requires that there is a so-called work object present at all times. Technically speaking, the work object is an ordinary Modula-2 program running as a subprogram under the «RAMSES Shell». Typically it is just the model definition program with which you are currently working.

To run the sample model, you have first to start the «RAMSES Shell». Start it with a double click on its icon or adopt any other method you normally use to start application programs on your computer. In order to make the appearing message windows disappear and to resume the start-up process click into each of the automatically displayed windows or press any key. Unless the «RAMSES Shell» has been used before for other purposes<sup>3</sup>, you use it in the mode «Mini RAMSES Shell», then you enter immediately the simulation environment and the sample model *Logistic*, which should be the current work object, is made ready for simulations.

Once fully started, you see the initial screen of the ModelWorks simulation environment with its menu bar, and the four windows for models, state variables, model parameters, and monitorable variables (Fig. T4). ModelWorks is now ready to accept from you commands, which will cause it to execute a simulation, to change a parameter, or to

---

<sup>1</sup>On the Macintosh no preparations are necessary to follow this tutorial except that you should be using a working copy of the RAMSES software (Working through the tutorial will change the contents of your diskettes, so don't use your originals!).

On the IBM PC you are ready only if you have followed exactly the installation procedures described in the booklet «*Installation Guide and Technical Reference of the RAMSES software*», in particular those for the installation of the ModelWorks software. For instance when you are using GEM ModelWorks you should then have an executable GEM application made from the sample model *LOGISTIC.MOD* which is now called *LOGISTIC.APP*.

<sup>2</sup>Note that on the IBM PC there exists no RAMSES shell, hence skip in the following text any reference to the RAMSES shell. Instead follow the instructions described in your documentation, in particular the booklet «*Installation Guide and Technical Reference of the RAMSES software*».

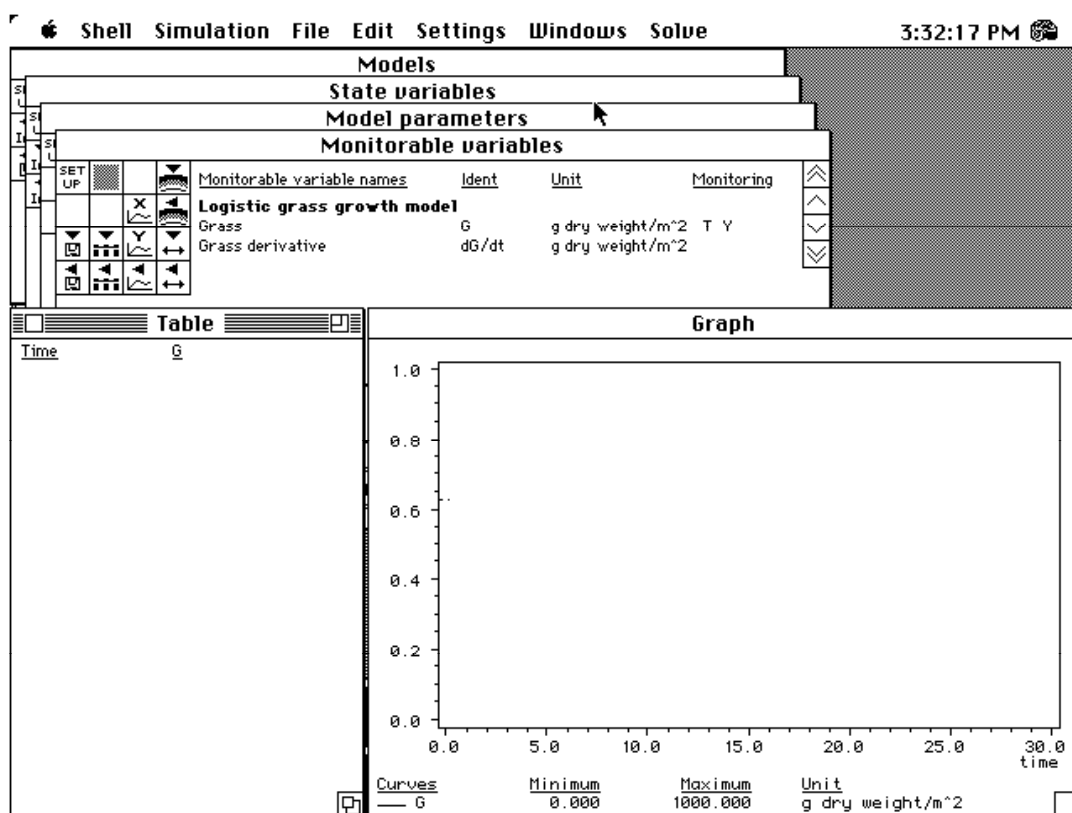
<sup>3</sup>Note the RAMSES shell remembers the settings (modes) it was in, when it was used the last time. Hence, in case the shell has been used before or you suspect wrong settings, confirm that you are really using it in the proper mode for following the instructions described in this tutorial. The needed settings (modes) are the following: Use the shell in the mode «Mini RAMSES Shell» with all other modes in the recommended default settings and the current work object should be *Logistic.MOD* (resides in the folder *Work*, or there is an additional copy also in the folder *Sample Models*). Consult further details in the help topic *Shell modes*, which you can access by choosing the menu command *Help...* or *Help RAMSES shell...* Follow the herein described instructions on how to change the settings (modes) of the RAMSES shell to the recommended defaults.



modify any other settings according to your needs. There are two basic techniques to issue commands to ModelWorks: Either you select a menu or you click with the mouse into a button from the button palette of the so-called IO-windows.

Throughout this manual read instructions as e.g. "choose menu command *Solve/Start run*" as "choose menu command *Start run* from menu *Solve*".

The menu-bar has five ModelWorks menus, each with several commands: *File* lets you print graphs, set preferences and quit the program; *Edit* allows you to access the clipboard to transfer graphs of simulation results to other programs or to desk accessories; *Settings* offers commands to set current values of the global simulation parameters or the so-called project description plus the resetting of current values to their defaults; *Windows* opens or activates the six windows of ModelWorks; and *Solve* is used to execute and control simulations. In the visible windows, the model objects of the activated model, the grass growth model, are displayed.



**Fig. T4:** Initial screen of the ModelWorks simulation environment obtained immediately after starting the model definition program, i.e. the module which contains the definition of the logistic grass growth sample model. All four IO-windows for the models, the state variables, the model parameters, the monitorable variables, plus the graph and the table window are open. The latter two windows have been slightly rearranged from their default size and position in order to give a better view onto the IO-windows.

The windows initially displayed serve two purposes: First they are used to display current values such as initial values or parameter values and secondly their button palettes are used to enter values or settings. Hence they are called IO-windows (input-output windows). Here are the common characteristics of the four IO-windows:

All IO-windows display a button palette in the upper left corner, a list of objects in the middle, and a scroll bar on the right side. Any model object can be selected by a simple

mouse click. All subsequent clicks on the buttons refer to the currently selected object. Selection of the bold model title is interpreted as selection of all elements belonging to this model. To select *all* objects of a list, click the button . All buttons with a down arrow are used to *set* a current value, whereas buttons with a left arrow are used to *reset* a value to its default as defined by the modeller. The button serves to specify which columns, i.e. current values of the model objects, are to be shown in the list.

The menu command *Settings/ All above* resets the program to its original state, i.e. exactly as it was when entering the simulation environment. If you should loose the orientation during a complex series of interactive changes, this command allows you to resume always to a well defined state. If you should have changed already any settings up to this point, reset it first with the menu command *Settings/Reset All above* before continuing with this guided tour.

### 2.2.1 DEFAULT SIMULATION

You can immediately start a simulation experiment (run), because ModelWorks ensures that any valid model definition program contains all necessary data for the so-called default simulation run. Choose the menu command *Solve/Start run* to actually start the simulation. The graph and the table windows are automatically opened, and a small time display window appears in the upper right corner (Fig. T5).

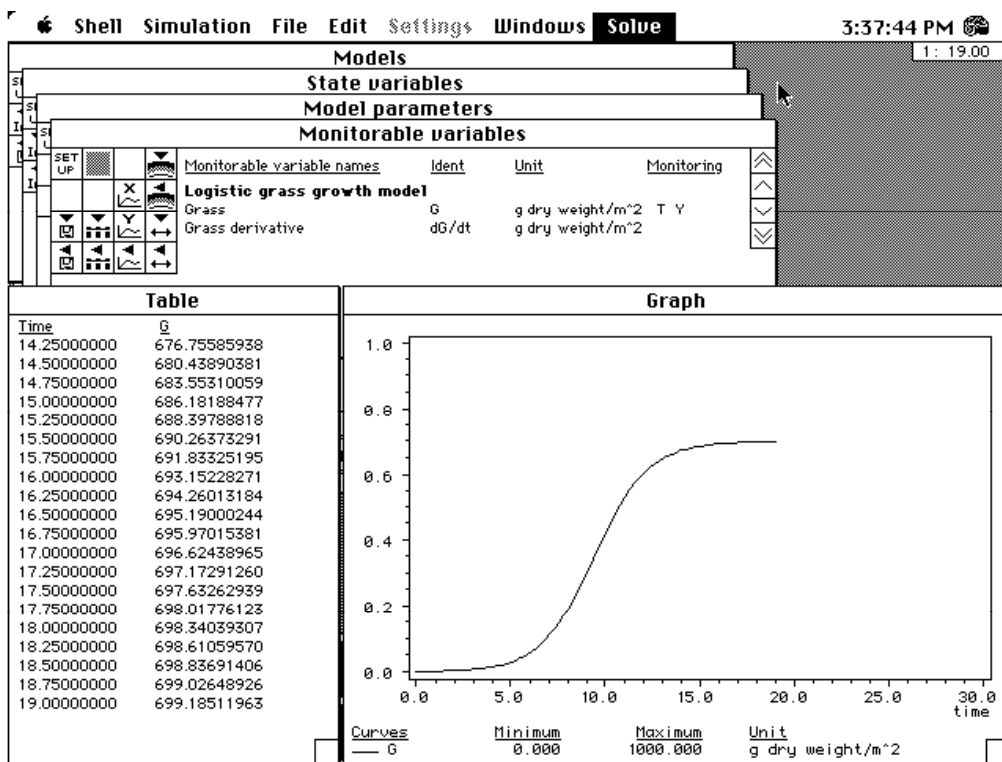




Fig. T5: ModelWorks simulation environment during a simulation run of the logistic grass growth sample model. In addition to the IO-windows the table plus graph windows are currently open. The current time is displayed in the upper right corner. The graph window shows the growth curve of the grass (g/m<sup>2</sup>).

Now, ModelWorks integrates the differential equation and displays simultaneously the results in the graph and table windows. In the graph window, you see how the grass grows at the beginning exponentially and how it reaches finally its equilibrium density.

### 2.2.2 CHANGING INITIAL VALUES


Initial values can be changed in the window for state variables: bring the state variable window to the front (click on it or choose the command *Windows/State variables*), and select the state variable *Grass*. Click on the button  and change in the appearing entry form the initial value to 4.0. This means, that the grass starts growing at a higher density. Verify this in another simulation run; the maximal density remains the same. Use other initial values to explore the model's behaviour (e.g. 200; 0.1). If you want to enter a initial value out of the allowed range [0,10'000], the program will refuse to accept it. For instance try to enter 10001 or -1 and see what happens.

After your explorations, reset the initial value with the menu command *Settings/Reset All model's initial values*, or with the button .

### 2.2.3 CHANGING PARAMETERS


Model parameter values can be changed in the window for model parameters in the same manner as described for initial values. Clear the graph with the menu command *Windows/Clear graph* and perform a simulation run for reference purposes. What will happen if you increase the growth rate  $c_1$  of the grass? Faster growth, or higher maximal density? Increase the growth rate from 0.7 to 1.2, and perform a simulation run. Now, the population grows faster, and reaches a higher equilibrium value. In the table output you can see the maximum value the grass density reached ( $\approx 1200 \text{ g/m}^2$ ).

### 2.2.4 CHANGING SCALING



As the grass curve exceeds the maximum value of 1000, ModelWorks clips these values. In order to avoid this clipping and to have also a look at the clipped portions of the curve, you should rescale the monitorable variable *Grass*. You may achieve this by increasing the upper limit of interest for the grass. This can be done in the window for monitorable variables. Bring it to the front, select *Grass*, and click on the button . In the appearing entry form, you can enter the new scaling value for the upper limit of interest, type 1200 and click into the OK button; ModelWorks writes the values automatically into the legend in the graph window. Perform another simulation run. This time, the curve should be fully visible and no longer be clipped.

### 2.2.5 CHANGING MONITORING

ModelWorks uses the expression *monitoring* for any kind of display of simulation results. Any variable which can be monitored is called a *monitorable variable*. Every monitoring definition is done in the window for monitorable variables. ModelWorks uses one window for numerical display (tabulated), and one window for the graphical display (line charts) of results, called the table window respectively the graph window. Storage of numerical results is also supported on the so-called stash file for the use of the data by other programs, e.g. a spread sheet program like Microsoft Excel™ or a program for statistical analysis or just to document a simulation run. At a time ModelWorks uses just one stash file only.




The model definition program of the sample model declares a second monitorable variable beside the state variable *Grass*. This is the derivative of grass listed in the IO-window *Monitorable variables* with the name *Grass derivative*. However, the defaults specified by the modeller for this variable are such that it is not displayed unless the simulationist activates it for actual monitoring. To see what the curve of the derivative looks like, bring the window for monitorable variables to the front, select *Grass derivative*, and click on the button  (Toggle function). In the column *Monitoring* appears a

"Y" in the row for the monitorable variable *Grass derivative* and the legend of the graph is accordingly updated<sup>1</sup>. The values of the variable *Grass derivative* will be drawn as another curve in the line chart of the window *Graph* during the next simulation run. Running another simulation displays the two curves *Grass* respectively *Grass derivative*.

You may generate also other graphs, e.g. *Grass derivative* versus *Grass*. Select the monitorable variable *Grass* in the window for the monitorable variables window and click onto the buttons  and then . In the column *Monitoring* disappears first the "Y" and then appears a "X" in the row for the monitorable variable *Grass*. This means that the values of the variable *Grass* will no longer be shown on the y-axis (ordinate) (toggle function) but will be used as x-values on the abscissa. The values of the variable *Grass derivative* should still be displayed as y-values (check the "Y" in the column *Monitoring* and the legends for the curves and the abscissa in the graph window). Run another simulation run and you should see a dome-shaped curve of *Grass derivative* vs. *Grass*.

Before you proceed, please select the command *Reset: All model's graphing* under menu *Settings*.

During the steps described in the previous two paragraphs you may have noticed that ModelWorks uses different colours<sup>2</sup> and line patterns if you have activated several monitoring variables at once. For instance the "Y" in the window *Monitorable variables* is drawn in the same colour as the corresponding curve, and curves are drawn using different patterns (important on monochrome screens and laser printers) in order to assist you in telling the curves apart. These characteristics of a monitoring variable are called curve attributes and they consist of first the line style (*LineStyle* - the pattern with which a line connecting two points is drawn), second the colour of a curve (*stain*), and thirdly the *symbol* with which points of a curve are marked. Unless explicitly specified, ModelWorks assigns curve attributes automatically, which is therefore called the automatic curve attribute definition strategy. For instance following this strategy ModelWorks assigns automatically the *stain coal* (black) to the first and *ruby* (red) to the second variable being activated for monitoring<sup>3</sup>. This helps the user to tell curves optimally apart under many circumstances.


However, the automatic curve attributes definition strategy has also its disadvantages, in particular the attributes may change all the time. For instance in one graph the *Grass* is black, in the other it is red but *Grass derivative* becomes black etc. The actual colour will depend only on the exact chronological sequence in which a monitorable variable has been activated for monitoring with the button . To try this out click on *Grass derivative* in the *Monitorable variables* window and toggle it with button  so that it becomes activated (Y). Run a simulation, e.g. this time by pressing the command key (clover-leaf key) simultaneously with key 'R'. Note that curve *Grass* is drawn in black (unbroken) and *Grass derivative* in red (broken). Then click on *Grass* in the *Monitorable variables* window and toggle it with the button  twice. Again both monitoring variables are activated (Y). Now rerun the simulation and note that this time colours are reversed, i.e. *Grass* is drawn in red (broken) and *Grass derivative* in black (unbroken). This is only because *Grass* has been activated for monitoring as the second curve after *Grass derivative* which has remained untouched during the toggling of *Grass*.



---





<sup>1</sup>This may depend on the currently set preferences, i.e. the immediate update of the graph takes place only if the option «Once changed, immediately redraw graph» available under menu command *File/Preferences* is currently checked; otherwise the redrawing of the graph will be deferred till the begin of the next simulation run.


<sup>2</sup>In GEM ModelWorks on IBM PCs are no colors available; sorry, but the memory limitations of MS DOS have forced us to sacrifice them.




<sup>3</sup> the third becomes *emerald* (green), and the fourth *sapphire* (blue). For more details see part *Reference*.

The convenient the automatic curve attributes definition strategy may be, the confusing it may become in complicated situations where the simulationist wishes to run many simulations and to compare the same monitoring variables. ModelWorks allows you to gain complete or partial control over the assignment of curve attributes, i.e. you can adopt your own curve attributes assignment strategy. You may achieve this by assigning explicitly to monitoring variables their particular curve attributes. For instance change the colour, of the curve *Grass*, to green and draw it with the symbol 'v' which may remind you of real grass tuft. Click on *Grass* in the window for monitorable variables, and click on the button  (rainbow toggle function). Choose the attributes *unbroken* as line style<sup>1</sup>, the *stain emerald* (green), and type 'v' in the symbol field; then click the "OK" push button. Finally select the command *Set: Global simulation parameters...* under the menu *Settings* and change the *monitoring interval* to 0.5; then click the "OK" push button. Now run another simulation run and you should see this time a green curve displaying the symbol 'v' at times 0, 0.5, 1, 1.5, 2 etc.

Note that from now on the curve *Grass* will always be drawn with exactly these curve attributes, i.e. in green regardless when and with how many other curves you currently display it. To see this behaviour click on *Grass* in the *Monitorable variables* window and toggle it with the button  twice. Run a simulation and note that *Grass* will be drawn in green (unbroken, 'v') and *Grass derivative* in black (unbroken)<sup>2</sup>. Then click on *Grass derivative* and toggle it with the button  once, so that it will no longer be activated for monitoring. Rerun the simulation and note that this time *Grass* is still drawn in green (unbroken, 'v').

Once again there is a disadvantage to this method if all your monitoring variables adopt it: you will run more often than you may first think into a situation where several curves currently in display happen to be all of the same colour or line style (may be important on a black-and-white laser printer or a publication). For instance click on *Grass* in the *Monitorable variables* window and click on the button , then select the line style broken, press the space bar to clear the symbol and hit return. Now click on *Grass* and activate it with button . Rerun the simulation and note that you can no longer separate the two curves on a printer or a monochrome screen. Of course it is also possible to switch back to the automatic assignment strategy: Select *Grass* and click the button ; in the appearing entry form click into the topmost radio button *automatic definition of curve attributes* and close it by pressing the enter key, or alternatively, reset the curve attributes of variable *Grass* with the button  or with the command *Reset: All model's curve attributes* under menu *Settings*.

Finally you can learn how to monitor the values of the variable *Grass derivative* in tabular form. Bring the window for monitorable variables to the front, select *Grass derivative*, and click on the button  (Toggle function). In the column *Monitoring* appears a "T" in the row for the monitorable variable *Grass derivative*. This means that the values of the variable *Grass derivative* will be written into a column of the table in the window *Table* during the next simulation run. Bring the table window to the front, enlarge it till you see all columns and rerun the simulation.

Before continuing reset this time the table and graph monitoring plus all curve attributes to their defaults with the following method: Bring first the window *Models* to the front, select the row containing the model title (*Logistic grass growth model*) and click on the buttons , , and . Note that the effect of this method is exactly the same as if you would have clicked on the buttons with the same pictures in the window *Monitorable variables* after having selected the model title (bold face *Logistic grass growth model*) in the latter window.

<sup>1</sup>Note that specifying a line style is crucial; if you should omit it, automatic definition of curve attributes would still remain active regardless of the settings of stains or plotting symbols.

<sup>2</sup>Note that on a monochrome screen or a non-color printer such as a laser printer both curves are drawn with the same line style, i.e. unbroken, and can only be separated by their different symbols 'v' resp. none.

### 2.2.6 CHANGING PARAMETERS DURING SIMULATION


Now, you will learn, how you can change model parameters even in the middle of a simulation run. We let the model simulate the grass growth as before; but, when the density has reached its maximal value, we increase the self-inhibition of the plants, the parameter  $c_2$  (this signifies, that the carrying capacity of the environment  $K = c_1/c_2$  decreases, for instance due to a sudden nutrient depletion or an unknown toxic substance). After this change, the grass density will tend to a lower equilibrium value.

To do this, start a simulation with the same settings as before. When the population has reached its maximal value (this happens approximately at time 15.0, watch the time window), interrupt the simulation with the menu command *Solve/Halt run (Pause)*. In the parameter window, you can now increase the value of the self-inhibition coefficient  $c_2$  from 0.001 to 0.002. Continue the simulation with *Solve/Resume run*, and observe the reaction of the system.

### 2.2.7 CHANGING INTEGRATION METHODS

To start with this section, reset the program to its initial state with *Settings/Reset All above*.

The numerical integration of the differential equation has to be done with special integration algorithms. ModelWorks offers several different methods for numerical integration. Each has its particular advantages and disadvantages. The default algorithm used in this example is *Euler*, which is shown in the model window. We shall compare two integration methods and record the results on the stash file.

First, we have to define the stash file output. Bring the window for monitorable variables to the front, select the variable *Grass*, and click on  (Toggle function). In the column *Monitoring* appears the letter "F" for stash filing (this is in addition to the "T" and "Y", which signify that this variable is written already into the table and drawn in the graph). Now, during a simulation run the values of the variable *Grass* will be written also onto the stash file. Note, by default every new simulation will overwrite the stash file's content.


Differences between integration methods become more obvious with large integration step sizes (this is the step which is internally used for numerical integrations). Therefore, we change this step to a higher value. Select the menu command *Settings/Global simulation parameters*, and change in the entry form the value for the integration step *and* the monitoring interval to 1.0. (The monitoring interval is the interval at which simulation results are displayed. If this is smaller than the integration step, the former is automatically reduced to generate the requested result display).

With these settings you can perform a simulation run. The integration method *Euler* is the simplest integration algorithm; therefore it is fast, but not very precise. After the integration, the stash file *ModelWorks.DAT*<sup>1</sup> contained in the same folder as the «RAMSES Shell» resides, is ready for inspection and you can open it with your favourite text editor, e.g. with the editor you normally use in conjunction with the «RAMSES Shell» or the desk accessory *MockWrite*. With the «Mini RAMSES Shell» simply select the menu command *Shell/Edit 'Logistic.MOD'*, close the work object and open the stash file with the editor's menu command *File/Open....* It should contain the same simulation results as the table window (look for *DATA-BEGIN of Run 1 ...*).

---

<sup>1</sup>On the IBM PC the name of this file is truncated to 8 characters and hence becomes MODELWOR.DAT. The file resides in the same directory as LOGISTIC.APP.

Once you have finished inspecting the results resume the simulation session, when using the «Mini RAMSES Shell» with the menu command *Macros/Clear, save & launch*<sup>1</sup>.

For the next simulation choose another algorithm: Bring the model window to the front, select the model, and click . Choose the more precise algorithm *Runge Kutta 4*: To prevent overwriting of the stash file, we change its name. Therefore, before starting the next simulation run, choose first the menu command *Settings/Select stash file*, and give the stash file a new name, e. g. *ModelWorks.DAT2*<sup>2</sup>. Only now start the simulation by choosing the menu command *Solve/Start run*.

The new run will give different results, as you can easily verify in the graph. For a more detailed, numerical analysis, you could use the values on the two stash files. ModelWorks would even allow to write the two time series onto the same stash file<sup>3</sup>.

### 2.2.8 PROGRAM TERMINATION

To return to the desktop of the Finder, use the menu command *Shell/Quit Mini RAMSES shell*.

In the «Mini RAMSES Shell» there exists also the possibility to terminate the simulation session with the menu command *Shell/Exit simulation*, i.e. to terminate only your running work object without quitting the «Mini RAMSES Shell». However, you may have little interest in selecting this menu command explicitly, since normally the «Mini RAMSES Shell» does this for you automatically, e.g. when you switch to the modelling session.

Note also, that in case your model encounters a run-time error, e.g. a numerical overflow because of a too large state variable value, you do also execute implicitly the same command as *Shell/Exit simulation*<sup>4</sup>.

---

<sup>1</sup>In case you should encounter problems while attempting to resume the simulation session, try to resume the «Mini RAMSES Shell» via the Finder and consult the help topic *Trouble shooting* (choose menu command *Shell/Help...*) or consult the booklet «*Installation Guide and Technical Reference of the RAMSES software*»

<sup>2</sup>On the IBM PC use e.g. *MODELWOR.DA2*

<sup>3</sup>This a more advanced technique, requiring multiple model declarations. For more details on this subject, please refer to the reference manual.

<sup>4</sup>This happens as soon as you click into the button *Abort* of the dialog box, which is displayed whenever a Modula-2 run-time error is detected.

### 3 Getting Started with Modelling

In this chapter you will get a closer look at the way ModelWorks models are defined. First it is explained, how the Modula-2 program defining the logistic grass growth sample model was written. Then you learn how to define a new model by modifying an existing model definition program by using the RAMSES modelling environment<sup>1</sup>. Finally, the new model's behaviour can be studied by resuming the simulation session and executing new simulation runs.

Again it is assumed that you know how to operate the computer and its software, and that you have the «RAMSES Shell» including ModelWorks installed and ready as described above in order to actually perform the described steps on your computer while reading this chapter.

#### 3.1 The Model Definition Program of the Sample Model

In the last chapter you worked with the grass growth model in the ModelWorks simulation environment as a simulationist. Now, we shall have a closer look at the used simulation program as a modeller. This program defines (declares) a logistic growth model and is called a model definition program. Choose *Shell/Edit 'Logistic.MOD'* to open the file *Logistic.MOD* for the subsequent inspection of its content<sup>2</sup>.

Step by step, we shall now go through this sample program and have a closer look at all its elements. Besides, the complete listings of the sample model definition program *Logistic*, and of the definition modules *SimMaster* and *SimBase*, which form the client interface used by *Logistic*, are also listed fully in the *Appendix* and are described in detail in the part III *Reference*. ModelWorks model definition programs have all the same basic structure as *Logistic.MOD*.

The import list contains all the items (types, constants, variables, and procedures) used within the program module. They are exported by the modules which form the client interface of ModelWorks:

```
FROM SimBase IMPORT
  Model, IntegrationMethod, DeclM, DeclSV, DeclP, RTCType,
  StashFiling, Tabulation, Graphing, DeclMV, SetSimTime,
  NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout,
  StateVar, Derivative, Parameter;

FROM SimMaster IMPORT RunSimEnvironment;
```

*RunSimEnvironment* is the procedure, which will start the simulation environment of ModelWorks. *DeclM*, *DeclSV*, *DeclMV* and *DeclP* are the procedures used to declare models and their objects. The types *Model*, *IntegrationMethod*, *StashFiling*, *Tabulation*, *Graphing*, *RTCType*, *StateVar*, *Derivative*, *Parameter* are needed in order to declare the model objects. All these objects, i.e. models, state variables, model parameters, monitorable variables, auxiliary variables and all associated variables, like derivatives, initial values and default values, are typically declared locally to the program module boundaries:

---

<sup>1</sup>On the IBM PC you will have to use the Modula-2 development environment as described in the booklet «*Installation Guide and Technical Reference of the RAMSES software*». For instance using the Windows-Version requires to work with the Logitech Modula-2 programming system.

<sup>2</sup>In case you work on a Macintosh with a system older than System 7.0, once you are in the model editor, e.g. *MEdit*, you will have to choose first the command *Macros/Open work file[s]* or simply type its keyboard equivalent `0`. This will open the current work object *Logistic.MOD*.



```

VAR
  m:      Model;
  grass:  StateVar;
  grassDot: Derivative;
  c1, c2: Parameter;

```

*m* is a variable of the opaque type *Model*. It instantiates an object of the class *Model* and allows also to reference that particular model, i.e. the logistic growth model, as a whole. Note however, that this declaration defines yet none of the model *m*'s properties nor does this associate with *m* any of its model objects.

As model objects the logistic growth model has one state variable, and two parameters. The type *StateVar* is used for state variables, the type *Parameter* for model parameters. For every state variable of a ModelWorks model, we also have to define an associated second variable of type *Derivative* (for continuous time models), or *NewState* (for discrete time models). It either corresponds to the derivative ( $\dot{x}(t) = dx/dt$  - continuous time) or the new value ( $x(k+1)$  - discrete time) of the state variable ( $x(t)$  respectively  $x(k)$ ). For the sample model, which is continuous time, this is the variable *grassDot* of the type *Derivative*.

In complex models often arises the need to introduce additional variables, which are not state variables. They are called auxiliary variables and ModelWorks offers the type *AuxVar* to denote variables of this category. Typically auxiliary variables hold the results of evaluations of terms from complex differential equations, i.e. they depend on state variables and on parameters. However, the logistic model is so simple, that there arises no need to introduce such a variable. On the other hand it is important to note, that any variable of the types *StateVar*, *Derivative* respectively *NewState*, *AuxVar*, and *Parameter* are fully compatible among themselves and with variables of the elementary Modula-2 data type REAL.

The procedure *Dynamic* is the heart of a ModelWorks model definition program. It contains the Modula-2 translation of the mathematical equations describing the model's dynamics, here Eq. (1); for a proper functioning of ModelWorks, it is very important, that this procedure computes the exact values of the derivatives or new values of all state variables as required by the given equation(s):

```

PROCEDURE Dynamic;
BEGIN
  grassDot:= c1*grass - c2*grass*grass;
END Dynamic;

```

The procedure *ModelObjects* contains the declarations of all model objects. For each of the four objects, models, state variables, parameters, and monitorable variables, there exists a special declaration procedure. Once such a procedure has been called, ModelWorks knows the variable, defaults, plus ranges corresponding to the model object, and can access it to maintain its values, or can show it in a window, or use it to display its current value in a graph. It is mandatory to declare a model if you wish to declare model objects (see below). The declaration of model objects, i.e. state variables, model parameters, or monitorable variables is optional and depends only on the current needs<sup>1</sup>.

The procedure *DeclSV* declares the state variable *grass*:

```

DeclSV(grass, grassDot, 1.0, 0.0, 10000.0,
      "Grass", "G", "g dry weight/m^2");

```

---

<sup>1</sup>For instance, you could use ModelWorks also for the plotting of a function, e.g. a time series measured during an experiment (parallel model to compare measured with simulated behavior). In this case you would need to declare only a monitorable but not a state variable.

The actual parameters are the two real variables for the state variable itself *grass*, and its derivative *grassDot*. Next, there are three real constants: the default initial value, and the upper and lower limit of the range of initial values. There is no such thing as negative grass, hence the lower limit has been set to 0.0, the upper to a value beyond which values are no longer plausible. The three strings are the name, an abbreviated name, and the unit of the state variable. These strings, and the initial value, will be displayed in the IO-windows for state variables. The limits for the initial value will be used during interactive changes: attempts by the simulationist to enter initial values out of the allowed range will be refused. With this mechanism the modeller can prevent the simulationist from entering values which would result in illegal simulation experiments for which the model is not defined or which could cause some other fatal run-time errors.

The procedure *DeclMV* declares the variables *grass* and *grassDot* as monitorable variables. This is necessary if we want to monitor the values of these variables on the stash file, in the table, or in a graph. Typically state variables, auxiliary variables, and output variables (used to couple submodels) are the model objects which are declared as monitorable variables. Our calls of *DeclMV*:

```
DeclMV(grass, 0.0, 1000.0,
      "Grass", "G", "g dry weight/m^2",
      notOnFile, writeInTable, isY);
DeclMV(grassDot, 0.0, 500.0,
      "Grass derivative", "dG/dt", "g dry weight/m^2/time",
      notOnFile, notInTable, notInGraph);
```

The first parameter denotes the real variable, which will be monitored. Next, there are two real constants: the default values for the scaling of the graphics output. The three strings are the same as for the state variables: the name, abbreviated name and the unit of the monitorable variables. The next three elements are default settings for file, table and graph output (e.g. *isY* means, that by default the variable *grass* will be plotted on the y-axis (ordinate) of the graph). These elements are imported with the enumeration types *StashFiling*, *Tabulation*, *Graphing*.

*DeclP* is the procedure for the declaration of model parameters. Since we have two model parameters,  $c_1$  and  $c_2$ , it is called twice:

```
DeclP(c1, 0.7, 0.0, 10.0, rtc,
      "c1 (growth rate of grass)",
      "c1", "/day");
DeclP(c2, 0.001, 0.0, 1.0, rtc,
      "c2 (self inhibition coefficient of grass)",
      "c2", "m^2/g dw/day");
```

The parameter list contains first the real variable of the parameter. Next, there are three reals: the default value of the parameter, and the upper and lower limit of its range within which the simulationist may enter a new parameter value. A parameter declared as *rtc* (*RTCType*) means that its value may be changed even in the middle of a simulation, not only before or after a run. The three strings are again: the name, the abbreviated name, and the unit of the parameter. Note that model parameters must not be implemented as constants; since they can be changed interactively during a simulation session, they must be Modula-2 variables.

The next procedure *ModelDefinitions* declares the model. It contains the following call to procedure *DeclM*:

```
DeclM(m, Euler, NoInitialize, NoInput, NoOutput, Dynamic,
      NoTerminate, ModelObjects, "Logistic grass growth model",
      "LogGrowth", NoAbout);
```

This declares the logistic grass growth model within ModelWorks. The first actual parameter is the model variable  $m$ . Then, *Euler* (type *IntegrationMethod*) defines the default integration method for this model. The next six parameters are all procedures; Modula-2 supports procedure types and therefore it is possible to use procedures as actual parameters when calling a procedure. This mechanism has to be used to install in ModelWorks all procedures, which describe the model dynamics and perform the model object declarations. It is then left to ModelWorks to actually call any of these procedures. The procedures *(No)Input*, *(No)Output*, *Dynamic* describe the model's dynamics, and the procedures *(No)Initialize*, *(No)Terminate* describe actions to be taken at the begin and end of every simulation run (more details on the purpose and usage of these procedures is given in the reference manual and in the definition of module *SimBase*). Some of these procedure identifiers have the prefix *No*, which means that these procedures have actually just empty bodies and are needed here only to call *DeclM* properly. The next procedure, *ModelObjects*, declares all model objects as explained above. The next two elements are strings for the name and an abbreviated name of the model. The last procedure, in our case *(No)About*, could be used to write information about the model in the help window of ModelWorks (this window is activated by clicking on the button  in the model window).

The procedure *SetSimTime* sets the default values for the simulation start and stop time.

Finally, we come to the short body of the program module:

```
BEGIN
  RunSimEnvironment(ModelDefinitions);
END Logistic.
```

The only action performed by this program is to call the procedure *RunSimEnvironment*. This starts the ModelWorks simulation environment, and passes the program control to ModelWorks. Its parameter, the procedure *ModelDefinitions*, contains the complete definition of the sample model. Note, how the procedures are nested: First ModelWorks will activate the simulation environment and call the procedure *ModelDefinitions*. Later on it will call the procedure *Objects*; which will result again in calls to the procedures *DeclSV*, *DeclMV*, and *DeclP*. This mechanism ensures that it is clear which objects belong to which model. Note also that while declaring an object, this object will also be immediately initialized with the given values. E.g. returning from procedure *DeclP(c,p,...)* will imply that the default value  $p$  for the model parameter is assigned to the variable  $c$ .

## 3.2 Developing a New Model

Instead of just reading an existing model definition program we will now develop a new model, hereby writing a new model definition program. However, before we start working on the new model definition program, we have to specify the mathematical properties of the new model.

### 3.2.1 THE NEW MODEL

The new model does not only include grass, but also herbivores as the second state variable *aphids*. Aphids feed on the grass and establish an ecological relationship, for the sake of simplicity, we assume somehow similar to other predator-prey relationships. The new model will consist of two coupled differential equations, each describing the dynamics of the two species, according to the Lotka-Volterra predator-prey model<sup>1</sup>: the grass is the prey, and the aphids are the predators.

---

<sup>1</sup>Early this century these models have first been formulated by LOTKA (1925) and VOLTERRA (1926). Their purpose is to describe the population dynamics of a prey and a predator species.

The model is described with the following non-linear second order differential equation system; note that the parameter and initial values are not the same as in the former model<sup>1</sup>:

$$\begin{aligned} dG(t)/dt &= c_1 G(t) - c_2 G^2(t) - c_3 G(t) A(t) \\ dA(t)/dt &= c_3 c_4 G(t) A(t) - c_5 A(t) \end{aligned} \quad (2)$$

where

State variables:

Grass (g dry weight [dw] per m <sup>2</sup> ):	G(t)
Initial amount of grass/initial value:	G(0) = 200 g/m <sup>2</sup>
Aphids (g dry weight [dw] per m <sup>2</sup> ):	A(t)
Initial number of aphids:	A(0) = 20 g/m <sup>2</sup>

Model parameters:

Grass growth rate (day <sup>-1</sup> ):	c <sub>1</sub> = 0.4 day <sup>-1</sup>
Self-inhibition coefficient(m <sup>2</sup> g <sup>-1</sup> day <sup>-1</sup> ):	c <sub>2</sub> = 8·10 <sup>-5</sup> m <sup>2</sup> g <sup>-1</sup> day <sup>-1</sup>
Grass consumption rate by aphids(m <sup>2</sup> g <sup>-1</sup> day <sup>-1</sup> ):	c <sub>3</sub> = 1.5·10 <sup>-3</sup> m <sup>2</sup> g <sup>-1</sup> day <sup>-1</sup>
Aphids birth rate per grass consumption (g g <sup>-1</sup> ):	c <sub>4</sub> = 0.1 g g <sup>-1</sup>
Death rate of aphids (day <sup>-1</sup> ):	c <sub>5</sub> = 0.2 day <sup>-1</sup>

Let us have a closer look at the new model and its equations: The first equation is the same as before, except that the term  $- c_3 G(t) A(t)$  has been added. This term is responsible for a decrease of the net grass growth, due to grass consumption by aphids. The second equation describes the dynamics of the aphids: They can grow by feeding on the grass, which is expressed with the term  $c_3 c_4 G(t) A(t)$ . The second term,  $- c_5 A(t)$ , accounts for the natural mortality of the aphids.

In the next section it will be explained how to alter step by step a copy of the logistic grass growth sample program to implement this new grass-aphids model. It is assumed that you know how to edit a program text<sup>2</sup>.

### 3.2.2 MODEL DEFINITION PROGRAM FOR THE NEW MODEL

The new model definition program will not be written completely anew, that would be too cumbersome. Instead we will simply modify a copy of the sample model definition program. The menu command *Shell/New...* of the «Mini RAMSES Shell» provides a simple mechanism to achieve exactly this goal. This is an easy, hence generally recommended way to develop ModelWorks model definition programs<sup>3</sup>.

When you choose the menu command *Shell/New...* a dialogue box appears, where you can specify the name of the new model definition program<sup>4</sup>. Type *GrassAphids.MOD* and make sure the file is stored in the folder *Work*.

<sup>1</sup>The new parameter and initial values are not necessarily realistic, since the sole purpose of the model is to help to learn ModelWorks.

<sup>2</sup>If you are using the full «RAMSES Shell» instead of the «Mini RAMSES Shell» or on the IBM PCs it is also assumed that you are familiar with the following terms and concepts: program text or source code, compilation, compiled object code, program linking, and the execution of programs.

<sup>3</sup>On the IBM PC make a copy of the source code of the sample program *Logistic.MOD* and rename this copy to *GrassAph.MOD*. Then open *GrassAph.MOD* and start editing.

<sup>4</sup>Note that the new model definition program is created by copying most of its content from a template. In case the template can not be accessed, for whichever reason, you are asked in an additional standard file open dialog to select also a new template file. You may select one of the templates, e.g. the file

Throughout the following explanations the affected program text is shown together with its context. The text portions actually having been altered or added are shown underlined. The begin of the model definition program looks as follows:

**Important hints:** First it is recommended to use the macro *Macros/Placeholder* ( $\sim G$ ) to go to the next place holder. A place holder starts with the character sequence '(\*.' and ends with '.\*)'. Once the macro *Macros/Placeholder* has selected such a place holder, e.g. (.\* Author .\*), simply overwrite it, e.g. by your name. Secondly, it is also recommended to save regularly your work during editing, e.g. by regularly choosing the menu command *File//Save*. Thirdly, in order to work conveniently with the «RAMSES Shell» make sure that the identifier (name) of the model definition program, here *GrassAphids* matches exactly the name of the work object, i.e. *GrassAphids.MOD*, at all times<sup>1</sup>.

```
MODULE GrassAphids;

( *****

MODEL: GrassAphids Lotka-Volterra grass and aphids model

Author2, date, ETHZ

***** )
```

There is no need to change the import list. All objects required are already imported from the modules *SimMaster* respectively *SimBase*.

Next declare the new state variable *aphids*, its derivative *aphidsDot*, and the three new parameters *c3*, *c4*, and *c5*:

```
VAR
  m: Model;
  grass, aphids: StateVar;
  grassDot, aphidsDot: Derivative;
  c1, c2, c3, c4, c5: Parameter;
```

Change the procedure *Dynamic* by adding the consumption term into the first statement plus inserting a second statement corresponding to the second differential equation:

```
PROCEDURE Dynamic;
BEGIN
  grassDot:= c1*grass - c2*grass*grass - c3*grass*aphids;
  aphidsDot:= c3*c4*grass*aphids - c5*aphids;
END Dynamic;
```

Now edit the procedure *ModelObjects*. Since several default values will be different from the ones of the old model, first change the parameters of the declaration procedures already present. The behaviour of the state variable *grass* is different. It needs another initial value:

```
DeclSV(grass, grassDot, 200.0, 0.0, 10000.0,
```

---

*ModDefProg.TEMPLATE*, contained in the folder *RAMSESLib*, or any other text file. In the latter case, please observe the syntax rules any template should follow. These rules are described in the help topic *Templates* (choose menu command *Shell/Help...*).

<sup>1</sup>This problem may occur only if you are not using the «Mini RAMSES Shell». The latter shell mode avoids any such problems automatically.

<sup>2</sup>Replace the place holder (.\* Author .\*) with your name. Similarly replace the place holder (.\* date .\*) with the actual date.

```
"Grass", "G", "g dry weight/m^2");
```

The monitorable variable *grass* needs a new upper limit for its clipping range:

```
DeclMV(grass, 0.0, 10000.0, "Grass", "G", "g dry weight/m^2",
notOnFile, writeInTable, isY);
```

The model parameters *c1* and *c2* need new default values:

```
DeclP(c1, 0.4, 0.0, 10.0, rtc,
"c1 (growth rate of grass)", "c1", "/day");
DeclP(c2, 8.0E-5, 0.0, 1.0, rtc,
"c2 (self inhibition coefficient of grass)", "c2", "m^2/g
dw/day");
```

Secondly, insert the procedures declaring the variable *aphids* as a state and as a monitorable variable, plus call the procedures declaring the new parameters:

```
DeclSV(aphids, aphidsDot, 20.0, 0.0, 1000.0,
"Aphids", "A", "g dry weight/m^2");

DeclMV(aphids, 0.0, 1500.0, "Aphids", "A", "g dry weight/m^2",
notOnFile, writeInTable, isY);

DeclP(c3, 1.5E-3, 0.0, 1.0, rtc,
"c3 (coupling parameter)", "c3", "m^2/g dw/day");
DeclP(c4, 0.1, 0.0, 10.0, rtc,
"c4 (ratio of grass net use by aphids)", "c4", "-");
DeclP(c5, 0.2, 0.0, 10.0, rtc,
"c5 (death rate of aphids)", "c5", "/day");
```

You could call these procedures in any order, mix declarations of state variables with those of monitorable variables or parameter declarations. However, consider that the sequence of declarations corresponds to the order in which they are listed in the I/O-windows of ModelWorks simulation environment.

The model declaration procedure *ModelDefinitions* remains the same except for minor changes in the actual parameters of the call to procedure *DeclM*. As the new model requires a better integration algorithm, we change the default method from *Euler* to *Heun*; further we change the model name strings:

```
DeclM(m, Heun, NoInitialize, NoInput, NoOutput, Dynamic,
NoTerminate, ModelObjects, "Aphid-grass model (Lotka-
Volterra)",
"GrassAphids", NoAbout);
```

Change the defaults for the simulation start and stop time as follows:

```
SetSimTime(0.0, 100.0).
```

The main program needs no changes.<sup>1</sup>

Once you have finished editing the new model, save your work plus resume the simulation session with the menu command *Macros/Clear, save & launch*<sup>2</sup>.

---

<sup>1</sup>A complete listing of the new model is contained in the Appendix.

<sup>2</sup>In case you should encounter problems while attempting to resume the simulation session, try to resume the «Mini RAMSES Shell» via the Finder and consult the help topic *Trouble shooting* (choose menu

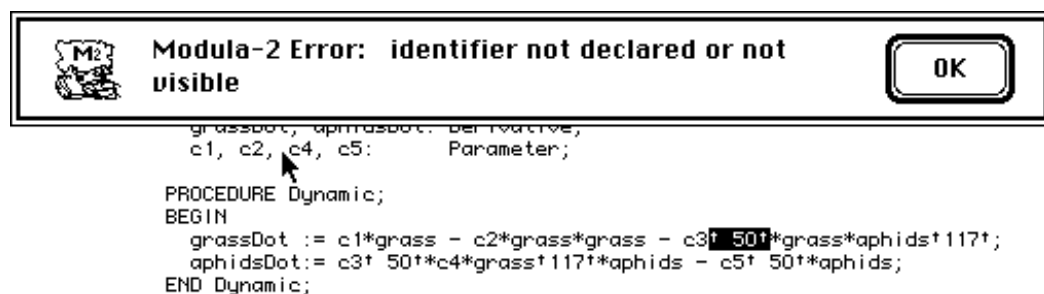
### 3.2.3 COMPILATION OF THE NEW MODEL

If your new model definition program contains no more errors, you will immediately resume the simulation environment with the new model GrassAphids loaded and ready for simulation (continue with section *Simulation of the new model*)<sup>1</sup>.

However, often this is not the case. Understand that the «Mini RAMSES Shell» compiles and executes automatically the current work object while resuming the simulation session. No action will be visible, except for the following two possibilities: Either the new model definition program contains no syntax errors, then it will be immediately executed, i.e. loaded into the simulation environment; or alternatively, the compiler detects errors and you will return into the editor.

In case compiler errors have been detected, you must first correct them before you can continue. Once you are back in the editor, the first error mark will be selected and the corresponding error message is displayed. Errors are marked in your program text with the characters '†'; they enclose also the error number. Acknowledge the error message, e.g. by pressing the carriage return or enter key, and overtype the error mark with the key backspace. Then correct the error by modifying your code appropriately.

E.g. if your model definition program is missing the declaration of the parameter  $c_3$ , then your file may look similar to this:



Follow above given instructions, then locate the next error by the command *Macros/Find next error* (or type  $\mathcal{E}$ ). The cursor jumps to the next error mark, selects it, and a message explaining the error kind will be shown again. Repeat correcting errors until you have no more error marks or you wish to ignore subsequent error marks, which might have been caused just as a consequence from the already corrected one. This is the case in the above example. In such a case you simply declare the missing parameter  $c_3$  and finish the editing by choosing the menu command *Macros/Clear, save & launch*. This macro command will clear your source code from all eventually still present error marks, save your code onto the disk, close the file and resume the simulation session. Hence, as a rule: Once you have finished editing the new model, save your work plus resume the simulation session by this technique only, i.e. choose the menu command *Macros/Clear, save & launch*.

---

command *Shell/Help...*) or consult the booklet «*Installation Guide and Technical Reference of the RAMSES software*»

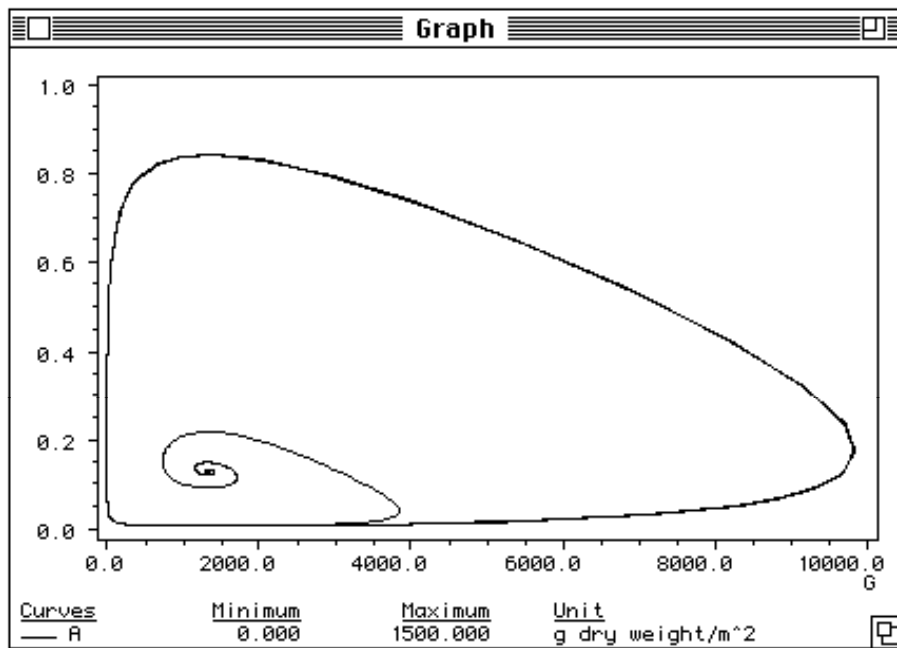
<sup>1</sup>This whole section applies only to the «Mini RAMSES Shell» as available on the Macintosh. On the IBM PC skip it completely and follow the specific instructions on how to make Modula-2 programs. For instance, GEM ModelWorks requires to use the JPI TopSpeed V1.17 Modula-2 development environment or the Windows-Version requires to work with the Logitech Modula-2 programming system. For more information on how to edit, compile, and correct programs with it, consult the booklet «*Installation Guide and Technical Reference of the RAMSES software*». Once you have compiled, and linked *GRASSAPH.MOD* you will have to perform one more step: rename the resulting applicaton *GRASSAPH.EXE* to *GRASSAPH.APP*.

If needed, repeat such edit and simulation cycles until there are no more errors and the model definition program satisfies your ideas about the new model.

### 3.2.3 SIMULATION OF THE NEW MODEL

Once your model definition program is error-free, you see the initial start-up screen of the ModelWorks simulation environment<sup>1</sup> with the new variables displayed in the I/O-windows. Execute the following steps to explore the behaviour of the new model:

- Run a simulation with the default settings (Choose *Solve/Start run*)
- Define a graph where the predator is plotted versus the prey (state space curve): Select the prey, and toggle its curve definition by clicking on the button . Click the button to define a plot which uses the x-axis (abscissa) to plot the prey values. Start a new simulation run. The resulting curve shows nicely how the grass and the aphids reach an equilibrium point.
- Set  $c_2 = 0.0$  (no self-inhibition of the prey population). This results in a different stability behaviour of the system: The oscillations of the population are no longer damped, but persist in a marginally stable limit cycle. In the state space you may observe closed trajectories, each corresponding to such a limit cycle. You should have obtained a graph similar to the one shown in Fig. T6.



**Fig. T6:** Graph of the simulation results produced with ModelWorks simulating the new, developed sample model . The graph shows a state space representation of a Lotka-Volterra like grass-aphids model system.

Marginally stable limit cycles can be easily perturbed; verify this by changing the integration method to *Euler*, or while using the method *Heun* by increasing

<sup>1</sup>On the IBM PC follow the same steps as when you executed the initial sample model *LOGISTIC.MOD* under chapter *Simulating the Sample Model*. For instance when you are using GEM ModelWorks you have to start first the GEM desktop and then to start the application *GRASSAPH.APP*.



the integration step and the monitoring interval up to 0.5 . How accurate is the numerical integration algorithm?

Congratulations! You have reached the end of the introductory tour through ModelWorks. You should have learned to develop and simulate simple models using ModelWorks.

In addition to the basic techniques you have just learned, ModelWorks features many more advanced modelling and simulation techniques. Among the more important features are modular, hierarchical modelling, including the coupling of several models and the mixing of discrete time with continuous time models. With ModelWorks it is easy to analyze results of complex simulation studies by means of a sensitivity analysis or a parameter identification. Moreover, thanks to its architecture open for extensions it allows for an unlimited number of possibilities. For a complete, full, and detailed description of all of ModelWorks features, please refer to the parts *Theory* and *Reference* of this text.

In case you would like to continue with the introductory example, here some suggestions how you could possibly explore it further on your own:

- introduce an auxiliary variable for the total biomass  $b(t)$ :

$$b(t) = G(t) + A(t) \quad (3)$$

Declare  $b(t)$  as a monitorable variable and compute its values within the procedure *Output*.



## Part II: Theory

Part II *Theory* contains a description and functional specification of every feature ModelWorks offers. However, it contains only little information on the elementary and typical usage of ModelWorks. In case you should not be familiar with the basic concepts of ModelWorks, please read first the ModelWorks tutorial. In particular you should read the first chapter of the tutorial: *General Description* since this part contains no technical information on the actual use.

Part II *Theory* explains the principles behind ModelWorks, not the details on the actual implementation and version of ModelWorks. Therefore it is typically studied once, at the begin of any serious work with ModelWorks. Implementation dependent details are listed and explained in Part III *Reference*. The latter has been written to support you during your daily work with ModelWorks.

Part II *Theory* contains two chapters:

The chapter *Model Formalisms* presents the mathematical formalisms in which ModelWorks models are to be formulated. The first section of this chapter treats elementary models, the second structured models, which are built from several elementary, coupled submodels.

The chapter *Functions* describes all basic functions of ModelWorks: First it describes the functionality of the simulation environment and secondly general aspects of the model development process.

Any serious modeling with ModelWorks requires to read at least this part of the manual and the section on the client interface of the manual Part III *Reference*.

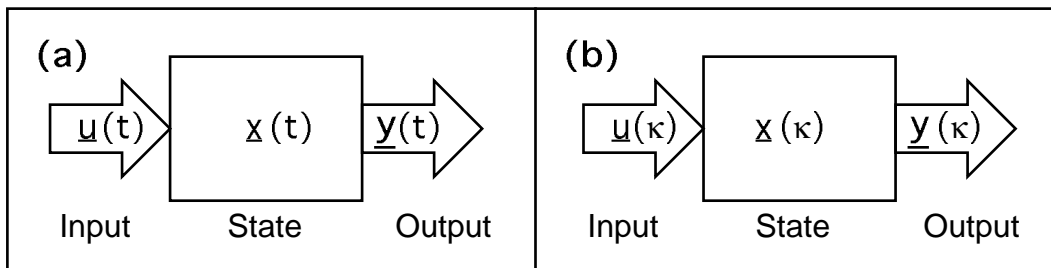
**Reading Hint:** For easier orientation, the pages, figures and tables of Part II *Theory* are prefixed with the letter T. Within this part the numbers of figures and tables follow those used in Part I *Tutorial*.

## 4 Model Formalisms

This chapter deals with theoretical aspects of modeling which are used in addition to the standard knowledge when developing models with ModelWorks. It explains only the mathematical formalisms in which the modeler should describe ModelWorks models. Please refer to a textbook for a general introduction to the modeling and simulation of dynamic systems<sup>1</sup>. ModelWorks distinguishes between two model types: Elementary models and structured models. Structured models are composed of several possibly coupled, elementary submodels.

### 4.1 Elementary Models

The elementary models which are used in ModelWorks are discrete or continuous time plus discrete event dynamic systems. They are formally described by a set of possibly coupled ordinary first order differential, difference equations, or instantaneous state transition functions. Normally the continuous or discrete independent variable is the so-called simulation time; however, it can represent any other independent variable like length or depth. Generally model parameters are considered to be time invariant, but ModelWorks supports also time variant parameters. However it is recommended to treat them either as auxiliary variables (becoming part of the differential or difference equations) or to treat them as an input. A graphical representation is given in Fig. T7, for more details see also Fig. T8.



**Fig. T7 :** Graphical representation of a dynamic system: **(a)** continuous time (DESS) or discrete event (DEVs) systems, **(b)** discrete time (SQM) systems. They constitute the basic types used in ModelWorks to describe models (s.a. Fig. T8).

A continuous time differential equation system specification (DESS) is given by the following system of differential equations (s.a. Fig. T8a):

$$\text{Dynamic equations:} \quad \dot{\underline{x}}(t) = \underline{f}(\underline{x}(t), \underline{u}(t), \underline{p}_f(t), t) \quad (4.1)$$

$$\text{Output equations:} \quad \underline{y}(t) = \underline{g}_y(\underline{x}(t), \underline{p}_y(t), t) \quad (4.2a)$$

$$\text{Event output:} \quad \partial\{<v, t_s, \tau, \alpha>\} = \underline{g}_\theta(\underline{x}(t), \underline{p}_\theta(t), t) \quad t_s=t \quad (4.2b)$$

$$\text{Initial conditions:} \quad \underline{x}(t_0) = \underline{x}_0 \quad (4.3)$$

$$\text{Input function:} \quad \underline{u}(t) \quad (4.4)$$

$$\text{Parameter set:} \quad \underline{p}(t) \quad (4.5)$$

<sup>1</sup> E.g. LUENBERGER, D.G., 1979. *Introduction to dynamic systems - Theory, models, and applications.* Wiley, New York, 446pp.

A discrete time difference equation system specification or **sequential machine (SQM)** is given by the following system of difference equations (s.a. Fig. T8b):

$$\text{Dynamic equations:} \quad \underline{x}(\kappa+c) = \underline{f}(\underline{x}(\kappa), \underline{u}(\kappa), \underline{p}_f(\kappa), \kappa) \quad (5.1)$$

$$\text{Output equations:} \quad \underline{y}(\kappa) = \underline{g}_y(\underline{x}(\kappa), \underline{p}_y(\kappa), \kappa) \quad (5.2a)$$

$$\text{Event output:} \quad \vartheta\{\langle v, t_s, \tau, \alpha \rangle\} = \underline{g}_\theta(\underline{x}(\kappa), \underline{p}_\theta(\kappa), \kappa) \quad t_s = \kappa \quad (5.2b)$$

$$\text{Initial conditions:} \quad \underline{x}(\kappa_0) = \underline{x}_0 \quad (5.3)$$

$$\text{Input sequence:} \quad \underline{u}(\kappa) = \underline{u}(\kappa_0), \underline{u}(\kappa_1), \dots, \underline{u}(\kappa_f) \quad (5.4)$$

$$\text{Parameter set:} \quad \underline{p}(\kappa) = \underline{p}(\kappa_0), \underline{p}(\kappa_1), \dots, \underline{p}(\kappa_f) \quad (5.5)$$

A continuous time **discrete event system specification (DEVS)** is given by the following system of equations based on instantaneous state transition functions (s.a. Fig. T8c):

$$\text{Dynamic equations:} \quad \underline{x}(t) = \begin{cases} \underline{\Phi}_v(\underline{x}(t^-), \underline{u}(t), \alpha, \underline{p}_f(t), t) & (\vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t\} \neq \emptyset) \\ \underline{x}(t^-) & (\vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t\} = \emptyset) \end{cases} \quad (6.1)$$

$$\text{Output equations:} \quad \underline{y}(t) = \begin{cases} \underline{g}_y(\underline{x}(t), \alpha, \underline{p}_y(t), t) & (\vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t\} \neq \emptyset) \\ \underline{y}(t^-) & (\vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t\} = \emptyset) \end{cases} \quad (6.2a)$$

$$\text{Event output:} \quad \vartheta\{\langle v, t, \tau, \alpha \rangle\} = \begin{cases} \underline{g}_\theta(\underline{x}(t), \alpha, \underline{p}_\theta(t), t) & (\vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t\} \neq \emptyset) \\ \emptyset & (\vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t\} = \emptyset) \end{cases} \quad (6.2b)$$

$$\text{Initial conditions, events:} \quad \underline{x}(t_0) = \underline{x}_0, \quad \vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s = t_0\} = \vartheta_0 \quad (6.3)$$

$$\text{Inputs, event input:} \quad \underline{u}(t), \quad \vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t\} \neq \emptyset \quad (6.4)$$

$$\text{Parameter set:} \quad \underline{p}(t) \quad (6.5)$$

where:

$\mathfrak{R}, \mathfrak{R}^+, \mathfrak{R}^n$ :	Set of real respectively positive real numbers in 1- respectively n-dimensional space		
$\emptyset$ :	The empty set		
$t$ :	Continuous time (independent variable) (DESS, DEVS)	$t \in [t_0, t_{\text{end}}]$	$t \in \mathfrak{R}$
$\kappa$ :	Discrete time (independent variable) (SQM only)	$\kappa = \kappa_0, \dots, \kappa_{f-1}$	$\kappa \in \mathfrak{R}, \kappa \subset t$
$c$ :	Discrete time step (SQM only)	$c > 0$	$c \in \mathfrak{R}^+$
$t^-$ :	Continuous left-hand side of time before and up to a discrete event (DEVS only)		
$\vartheta$ :	Set of discrete events (internal and external), events are quadrupels of the form $\langle v, t_s, \tau, \alpha \rangle$		
$v$ :	Event class (must be globally unique)		
$t_s$ :	Scheduling time of an event	$t_s \in [t_0, t_{\text{end}}]$	$t_s \in \mathfrak{R}$
$\tau$ :	Time advancement from $t_s$ till associated events $\vartheta$ are due	$\tau \geq 0$	$\tau \in \mathfrak{R}^+$
$\alpha$ :	Transaction data of an event		
$\underline{x}$ :	State vector $\underline{x}(t)$ at time $t$ respectively $\kappa$ . For DEVS exists also the state vector $\underline{x}(t^-)$ , it is the left-hand side of the discontinuity before and up to $t$		$\underline{x} \in \mathfrak{R}^n$
$\dot{\underline{x}}$ :	Derivative vector $d\underline{x}(t)/dt$ (DESS only)		$\dot{\underline{x}} \in \mathfrak{R}^n$
$\underline{f}, \underline{g}_y, \underline{g}_\theta \dots$ :	Linear or nonlinear function vectors		$\dim[\underline{f}] = n, \dim[\underline{g}_y] = m$
$\underline{\Phi}_v$ :	Linear or nonlinear instantaneous state transition function vector of event class $v$		$\dim[\underline{\Phi}_v] = n$
$\underline{u}$ :	Input vector		$\underline{u} \in \mathfrak{R}^l$
$\underline{y}$ :	Output vector		$\underline{y} \in \mathfrak{R}^m$
$\underline{p}$ :	Parameter vector, composed of the elements of $\underline{p}_f, \underline{p}_y$ , and $\underline{p}_\theta$		$\underline{p} \in \mathfrak{R}^r$

In the context of ModelWorks the term *output* is used in a different than the usual systems theoretical meaning. It is reserved to the output produced by a submodel to be connected with the input of another submodel (coupling of submodels). It should not be confounded with the display of simulation results for the simulationist. The latter is called monitoring. Note that the output  $\underline{y}$  defined by Eq. (4.2), resp. (5.2) does not depend on the input  $\underline{u}$ . This restriction guarantees the correct calculation of structured models. Structured models are explained below.

The discrete time  $\kappa$  of a SQM<sup>2</sup> is a real number, which may be interpreted as a subset of the continuous time  $t$ . The discrete time step  $c$  is interpreted as the continuous time interval elapsed between two adjacent discrete time points  $\kappa_j$  and  $\kappa_{j+1}$  (Fig. T8b). The classical case where  $\kappa$  is an integer and  $c$  may be interpreted to be of length 1 is just a special case.

For discrete event systems ModelWorks supports the event scheduling paradigm. Discrete events are time bound entities which have the potential to cause an instantaneous state transition  $\phi$  in a DEVS. Typical examples of events are the arrival of individuals such as animals at a feeding site, or the leaving of customers from a location which offers services. Hereby the state vector instantaneously changes from state  $\underline{x}(t^-)$  to state  $\underline{x}(t)$ , thus creating a state discontinuity. The vector  $\underline{x}(t^-)$  represents the left-hand side,  $\underline{x}(t)$  the right-hand side of these discontinuities. By definition states of DEVS do not change inbetween events. Usually discrete events are grouped into a finite set of classes, whereby each class  $v$  is characterized by a corresponding instantaneous state transition function vector  $\phi_v$  operating on the DEVS state vector.

Every event is described as a quadrupel  $\langle v, t_s, \tau, \alpha \rangle$ , where  $v$  describes the event's class,  $t_s$  the time at which it has been scheduled,  $\tau$  the time which may elapse till the event becomes due, and  $\alpha$  the transaction data. Through the event class  $v$  an event is uniquely associated with a particular instantaneous state transition function vector  $\phi_v$ . In simple cases<sup>3</sup>  $\tau$  can be interpreted as the time a DEVS is allowed to remain in its current state before the next change. Transactions allow to transmit data from the scheduling state to the due time  $t_s + \tau$ . Examples of transactions are attributes or properties describing individual animals or customers.

Event scheduling is needed for state changes and the time advancement of DEVS (Fig. T8c). At a given time there may exist any number of events. Thus events are grouped into sets, i.e. the set of events  $\vartheta\{\langle v, t_s, \tau, \alpha \rangle \mid t_s = t\}$  defines all events scheduled at time  $t_s$ , the set  $\vartheta\{\langle v, t_s, \tau, \alpha \rangle \mid t_s + \tau = t\}$  all events due at time  $t$ . Events which are scheduled and handled by the same DEVS are called internal events. The events, which are scheduled by a system which does not provide the instantaneous state transition function  $\phi_v$  associated with the event's class  $v$ , are called external events. The latter are only of importance in structured systems, which are explained below.

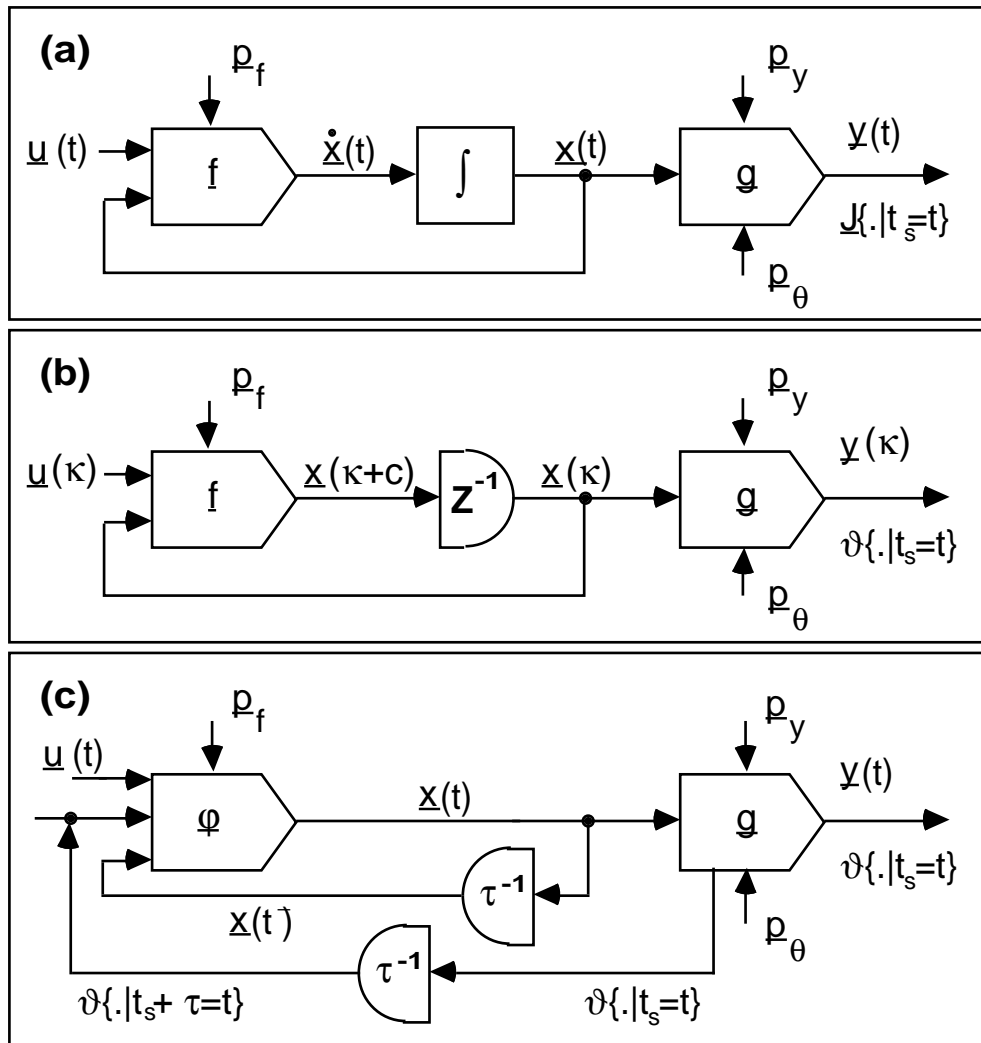
ModelWorks orders all events according to their due times, which allows to solve a DEVS solely by a sequence of instantaneous state changes given by the ordering of the events. Hereby the "independent variable" continuous time  $t$  is no longer a true "independent variable", but rather a byproduct; in other words: discrete events have the side effect of advancing the time  $t$  by a positive amount  $\tau$  ( $\tau \in \mathbb{R}^+$ ). Given any set of events  $\vartheta\{\langle v, t_s, \tau, \alpha \rangle \mid t_s \geq t\}$  at time  $t$ , the time is always only advanced to the the next event, i.e. the due time of the event with the smallest  $\tau$ . Note, an arbitrary number of events may be due at the same time, which may have been scheduled at various times  $t_s$  in the past; moreover,  $\tau$  may also be 0, i.e. an event may schedule immediate events, which are due at the same time they are scheduled<sup>4</sup>. Therefore

<sup>2</sup>Throughout this text a sequential machine is considered a synonym for the discrete time standard system formalism. However, the term sequential machine is sometimes also used to specify an automaton. Note the latter is usually defined to operate on discrete, explicit states and not on state variables as this is usually the case for discrete time difference equations.

<sup>3</sup>i.e. an autonomous DEVS, only one event class, cardinality of event output always 1

<sup>4</sup>Of course the modeler has to make sure that her equations specify no recursive event scheduling without a proper termination condition, otherwise a simulation run may not be able to progress resp. terminate at all.

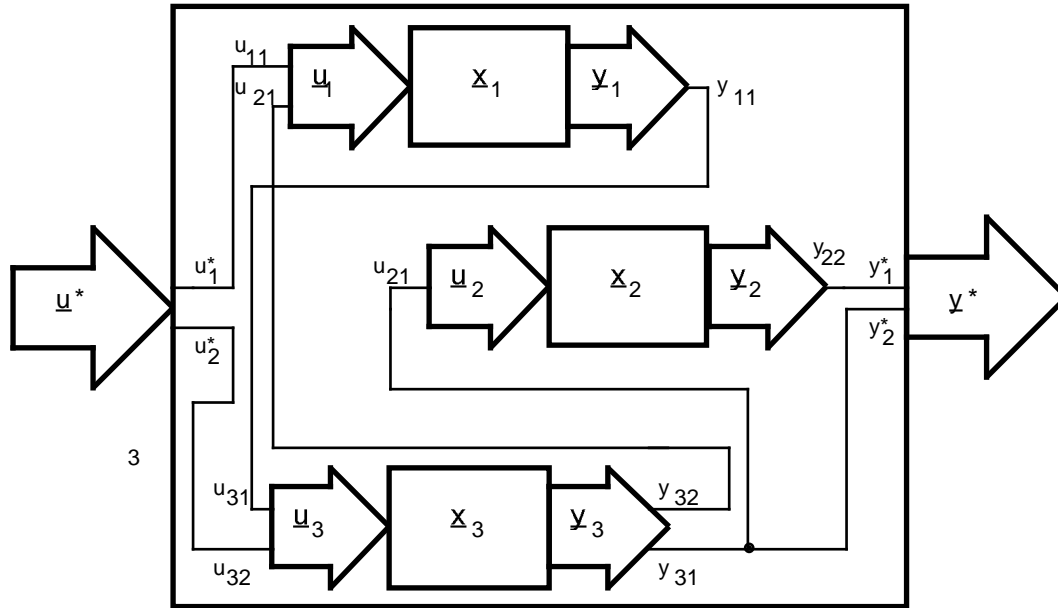
ModelWorks uses the scheduling sequence as an additional criteria to determine a unique order among events which are due at the same time.



**Fig. T8 :** Signal flow in (a) continuous time differential equation (DESS), (b) discrete time difference equation or sequential automaton (SQM), and (c) continuous time discrete event (DEVS) dynamic systems (s.a. Fig. T7). These systems constitute the basic types used in ModelWorks to describe elementary and structured models.

## 4.2 Structured Models (Coupling of Submodels)

Any number of elementary models, here called submodels, may be coupled to form complex, structured models. Any number of hierarchical levels may be introduced. Elementary models are defined exactly the same way as described in the previous chapter. The coupling is realized by connecting a submodel's output to another submodel's input. There are four cases to be distinguished: **(A)** all submodels are continuous time differential equation systems only (DESS), **(B)** all submodels are discrete time systems only (SQM), **(C)** all submodels are discrete event systems only (DEVS), **(D)** and there are some continuous (DESS and/or DEVS) as well as discrete time (SQM) submodels. Mixed structured models may be composed from any number and any type of models.



**Fig. T9 :** Example of a structured model system composed of three elementary, coupled submodel systems. The global input  $\underline{u}^*$  is defined by Eq. (7a, b, or c), the inputs of the subsystems  $u_{ij}$  by Eq. (8a, b, or c), the outputs of the subsystems  $y_{ij}$  by Eq. (10a, b, or c), and the global output  $\underline{y}^*$  by Eq. (11a, b, or c).

**(A)** A structured model composed of  $n$  elementary, coupled DESS submodels, each continuous time and defined according to Eq. (4.x) is defined as follows (for an example see also Fig. T9):

The input of the global system is given by

$$\underline{u}^* = \underline{u}^*(t) \quad \text{(global input)} \quad (7a)$$

The input to the submodel  $i$  depends on the global input  $\underline{u}^*$ , and on the output  $\underline{y}_i(t)$  of the  $n$  submodels (output-input coupling):

$$\underline{u}_i(t) = \underline{h}_i(\underline{u}^*(t), \underline{y}_1(t), \dots, \underline{y}_n(t)) \quad \text{(input of submodel } i) \quad (8a)$$

The dynamic equations for obtaining the derivative s of the state variables of the submodel  $i$  are given by:

$$\frac{d\underline{x}_i(t)}{dt} = \underline{f}_i(\underline{x}_i(t), \underline{u}_i(t), \underline{p}_{fi}(t), t) \quad \text{(dynamic equations of submodel } i) \quad (9a)$$

The output of the submodel  $i$  depends on the states and the parameter set of the submodel  $i$ . An output variable must not depend directly on an input variable (no direct output-input coupling). Since the input variables may depend directly on the output variables, a direct output-input coupling would lead to a circularity which could not be resolved generally.

$$\underline{y}_i(t) = \underline{g}_i(\underline{x}_i(t), \underline{p}_{gi}(t), t) \quad \text{(output of submodel } i) \quad (10a)$$

Some of these outputs are not connected to other submodels but are global outputs. These elements from  $\underline{y}_i(t)$  form for each submodel the global output vectors  $\underline{y}_i^*(t)$ .

The output of the global system is given by combining the global output vectors  $\underline{y}_i^*$  of the  $n$  submodels  $i$ :



$$\underline{y}^*(t) = \begin{bmatrix} y_1^*(t) \\ \vdots \\ y_n^*(t) \end{bmatrix} \quad \text{(global output)} \quad (11a)$$

**(B)** A structured model composed of  $n$  elementary, coupled submodels, each discrete time and defined according to Eq. (5.x) is defined as follows (for an example see also Fig. T9):

$$\underline{u}^* = \underline{u}^*(\kappa) \quad \text{(global input)} \quad (7b)$$

$$\underline{u}_i(\kappa) = \underline{h}_i(\underline{u}^*(\kappa), y_1(\kappa), \dots, y_n(\kappa)) \quad \text{(input of submodel i)} \quad (8b)$$

$$\underline{x}_i(\kappa+c) = \underline{f}_i(\underline{x}_i(\kappa), \underline{u}_i(\kappa), \underline{p}_{fi}(\kappa), \kappa) \quad \text{(dynamic equations of submodel i)} \quad (9b)$$

$$y_i(\kappa) = \underline{g}_i(\underline{x}_i(\kappa), \underline{p}_{gi}(\kappa), \kappa) \quad \text{(output of submodel i)} \quad (10b)$$

$$\underline{y}^*(\kappa) = \begin{bmatrix} y_1^*(\kappa) \\ \vdots \\ y_n^*(\kappa) \end{bmatrix} \quad \text{(global output)} \quad (11b)$$

**(C)** A structured model composed of  $n$  elementary, coupled DEVS submodels, each continuous time and defined according to Eq. (6.x) is defined as follows (for an example see also Fig. T9):

DEVS may also receive inputs and produce outputs.

The second part of event output occurs if a system schedules external events, i.e. events which are designated for a DEVS different from the scheduling system. This is the case if the scheduling system is a DEVS which provides for the class  $v$  of the scheduled event no instantaneous state transition function  $\underline{\phi}_v$ , or if the scheduling system is not a DEVS.

The input of the global system is given by

$$\underline{u}^* = \underline{u}^*(t) \quad \text{(global input)} \quad (7c)$$

All input to the submodel  $i$  of type DEVS depends on the global input  $\underline{u}^*$ , and on the output  $y_i(t)$  respectively  $\vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t\}$  of the  $n$  submodels (output-input coupling):

$$\underline{u}_i(t) = \underline{h}_i(\underline{u}^*(t), y_1(t), \dots, y_n(t)) \quad \text{(input of submodel i)} \quad (8c)$$

In contrast to DESS and SQM, DEVS can receive two types of input, i.e. ordinary input vector  $\underline{y}_i(t)$  and event input; the latter consists of external events which have been scheduled by other submodels.

$$\vartheta\{\langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t\} \quad \text{(event input of submodel i)} \quad (8c')$$

Note, state changes in a DEVS can only be caused by event input, but not by continuous input vectors  $\underline{u}(t)$ . The latter influence a DEVS only in the moment of an instantaneous state transition, i.e. a discrete event. The dynamic equations consist of a set of transition function vectors  $\underline{\phi}_v$ , one for each event class, which are capable of changing the state vector  $\underline{x}(t)$  of the submodel  $i$  instantaneously, given at least one event of the corresponding class  $v$  is due:

$$\underline{x}(t) = \begin{cases} \underline{p}_v(\underline{x}(t^-), \underline{u}(t), \alpha, \underline{p}_f(t), t) & (\vartheta\{ \langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t \} \neq \emptyset) \\ \underline{x}(t^-) & (\vartheta\{ \langle v, t_s, \tau, \alpha \rangle | t_s + \tau = t \} = \emptyset) \end{cases} \quad \text{(dynamic equations of submodel i)} \quad (9c)$$

The output of the submodel i depends on the states and the parameter set of the submodel i. An output variable must not depend directly on an input variable (no direct output-input coupling). Since the input variables may depend directly on the output variables, a direct output-input coupling would lead to a circularity which could not be resolved generally.

$$\underline{y}_i(t) = \underline{g}_i(\underline{x}_i(t), \underline{p}_{g_i}(t), t) \quad \text{(output of submodel i)} \quad (10b)$$

Some of these outputs are not connected to other submodels but are global outputs. These elements from  $\underline{y}_i(t)$  form for each submodel the global output vectors  $\underline{y}_i^*(t)$ .

The output of the global system is given by combining the global output vectors  $\underline{y}_i^*$  of the n submodels i:

$$\underline{y}^*(t) = \begin{bmatrix} \underline{y}_1^*(t) \\ \vdots \\ \underline{y}_n^*(t) \end{bmatrix} \quad \text{(global output)} \quad (11b)$$

Event classes must be globally unique.

Similarly DEVS can produce two types of outputs: As this is the case for states, by definition ordinary output vectors  $\underline{y}(t)$  remain also constant inbetween events. Note, that any event output can only be received by a DEVS, but the inverse is not the case, i.e. any model may produce event. In addition to such ordinary output vectors, ModelWorks supports also event output which consists of internal and external events. DEVS may produce so-called internal as well as external events. The events produced by a DEVS are called internal events; external events are produced by a model system different from the receiver. Moreover note, not only DEVS, but all other elementary model types may produce external event output. Event output consists of a set of events  $\vartheta\{ \langle v, t_s, \tau, \alpha \rangle \}$ . It can be done anytime by scheduling events at time  $t_s$ . The time  $\tau$  is the time span the DEVS is allowed to remain in the current state.

**(D)** A general structured model composed of n elementary, coupled submodels, each either continuous or discrete time and each defined according to Eq. (4.x) resp. Eq. (5.x) is defined as follows (for an example see also Fig. T9):

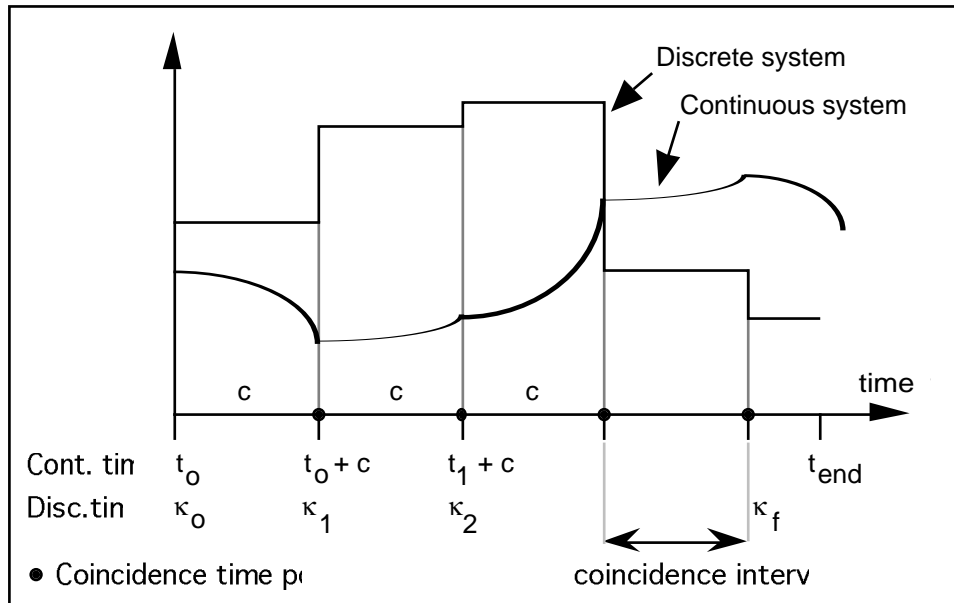
There are two different independent variables: the continuous time  $t \in \mathfrak{R}$  and the discrete time  $\kappa \in \mathfrak{R}$ , where  $\kappa$  is a subset of the continuous time t, i.e.  $\kappa \subset t$ . The discrete time step c of the discrete time submodel(s) is interpreted as a real time interval, the *coincidence interval* c, on the time axis t. The discrete time submodels are only defined at the endpoints of these intervals, the *coincidence time points*. The set of all coincidence points constitutes the discrete time  $\kappa$ . The continuous time submodel(s) describe continuous (or faster) processes which occur between the coincidence points. A communication between the two submodel types occurs only at every coincidence point (Fig. T10). The values of the two time variables match at every coincidence point exactly, i.e.  $t = \kappa$ . In particular, this implies that the following condition is satisfied by the continuous and the discrete start time:

$$t_0 = \kappa_0 \quad \text{where } t_0, \kappa_0 \in \mathfrak{R} \quad (12)$$

Given the coincidence interval c is constant the continuous time t may be mapped always to the discrete time  $\kappa = \kappa_j$  at the last coincidence point as follows:

$$\kappa_j = t_0 + j c = t_0 + \text{INT}\left(\frac{t - t_0}{c}\right) c \quad c = \text{const.}, \text{INT is integral part of argument} \quad (13)$$

However, ModelWorks does not require to keep the coincidence interval  $c$  constant, in which case Eq. (13) no longer holds. Finally note in the special case where  $\kappa$  shall be restricted to integer numbers, Eqs. (5.1) and (12) require that  $t_0$  as well as  $c$  must also be integers.



**Fig. T10 :** Coupling of discrete and continuous time submodels: The figure shows the results of a simulation of a structured model system composed of one discrete and one continuous time submodel. A communication between the two submodels occurs at every *coincidence time point*, when the output of the discrete submodel determines the rate of change of the continuous time submodel. No data exchange takes place during the *coincidence interval*, during which the rate of change of the discrete time submodel remains constant (sample and hold).

Any structured model mixed of continuous and discrete time submodels can be subdivided into two portions: the first is the continuous time portion  $\Xi$  consisting of the set of all continuous time submodels with their related continuous time inputs and outputs plus the continuous time global input and output; the second is the discrete time portion  $\Delta$  consisting of the set of all discrete time submodels with their related discrete time inputs and outputs plus the discrete time global input and output. At every coincidence point the system is fully defined and all submodels are fully coupled (System  $\sim \Xi + \Delta$ ). Between coincidence points, the dynamics of the system collapse or degenerate to the continuous time portion  $\Xi$ , the other portion of the system  $\Delta$  remains constant but is still accessible to  $\Xi$ . This corresponds to a sample and hold technique (sample at coincidence points, hold between) (Fig. T10).

The global input consists of two vectors, one for the global continuous time input  $\underline{u}^{*\xi}$  and the other for the global discrete time input  $\underline{u}^{*\delta}$ :

$$\begin{aligned} \underline{u}^{*\xi} &= \underline{u}^*(t) & (\in \Xi) \\ \underline{u}^{*\delta} &= \underline{u}^*(\kappa) & (\in \Delta) \end{aligned} \quad \begin{array}{l} \text{(global inputs)} \\ \end{array} \quad (7c)$$

At the coincidence points the inputs of all submodels  $i$  depend on the continuous as well as the discrete time global input  $\underline{u}^{*\xi}$  resp.  $\underline{u}^{*\delta}$ , and on the output of the continuous time submodels  $j^\xi \in \Xi$  as well as the discrete time submodels  $j^\delta \in \Delta$  ( $i, j^\xi, j^\delta \in \{1, 2, \dots, n\}$ ). Between coincidence

points the inputs to the continuous submodel  $i^\xi \in \Xi$  depend continuously on the continuous time global input  $\underline{u}^{*\xi}$  and on the output of the continuous time submodels  $i^\xi \in \Xi$ . Any dependence of the continuous time submodels  $j^\xi \in \Xi$  on the output of the discrete time submodels  $j^\delta \in \Delta$  is resolved by using the last defined values (sample from "sample and hold") of all variables of  $\Delta$  while mapping time  $t$  to  $\kappa$  using Eq. (13) (hold from "sample and hold"):

$$\begin{aligned} \underline{u}_i^\xi(t) &= \underline{h}_i^\xi(\underline{u}^{*\xi}(t), \underline{u}^{*\delta}(\kappa), \underline{y}_1^\xi(t), \underline{y}_2^\xi(t), \dots, \underline{y}_{n-1}^\delta(\kappa), \underline{y}_n^\delta(\kappa)) & (\in \Xi) \\ \underline{u}_i^\delta(\kappa) &= \underline{h}_i^\delta(\underline{u}^{*\xi}(t), \underline{u}^{*\delta}(\kappa), \underline{y}_1^\xi(t), \underline{y}_2^\delta(\kappa), \dots, \underline{y}_{n-1}^\delta(\kappa), \underline{y}_n^\xi(t)) & (\in \Delta) \end{aligned} \quad (8c)$$

The dynamic equations for the calculation of the derivatives for  $\Xi$  or the new values for  $\Delta$  of the state variables of the submodels  $i^\xi$  resp.  $i^\delta$ :

$$\begin{aligned} \underline{dx}_{i^\xi}(t) &= f_{i^\xi}(\underline{x}_{i^\xi}(t), \underline{u}_{i^\xi}(t), \underline{p}_{fi^\xi}(t), t) & (\in \Xi) \\ \underline{x}_{i^\delta}(\kappa+c) &= f_{i^\delta}(\underline{x}_{i^\delta}(\kappa), \underline{u}_{i^\delta}(\kappa), \underline{p}_{fi^\delta}(\kappa), \kappa) & (\in \Delta) \end{aligned} \quad \begin{array}{l} \text{(dynamics of submodels } i^\xi, i^\delta) \\ (9c) \end{array}$$

The output of the submodels  $i^\xi$  resp.  $i^\delta$  are calculated of the states and the parameter set of the particular submodel  $i^\xi$  resp.  $i^\delta$ . An output variable must not depend directly on an input variable (no direct output-input coupling, avoids unresolvable circularity).

$$\begin{aligned} \underline{y}_{i^\xi}(t) &= \underline{g}_{i^\xi}(\underline{x}_{i^\xi}(t), \underline{p}_{gi^\xi}(t), t) & (\in \Xi) \\ \underline{y}_{i^\delta}(\kappa) &= \underline{g}_{i^\delta}(\underline{x}_{i^\delta}(\kappa), \underline{p}_{gi^\delta}(\kappa), \kappa) & (\in \Delta) \end{aligned} \quad \begin{array}{l} \text{(outputs of submodels } i^\xi, i^\delta) \\ (10c) \end{array}$$

Some of these outputs are not connected to other submodels but are global outputs. These elements from  $\underline{y}_{i^\xi}(t)$  resp.  $\underline{y}_{i^\delta}(\kappa)$  form for each submodel the global output vectors  $\underline{y}_{i^\xi}^*(t)$  resp.  $\underline{y}_{i^\delta}^*(\kappa)$ .

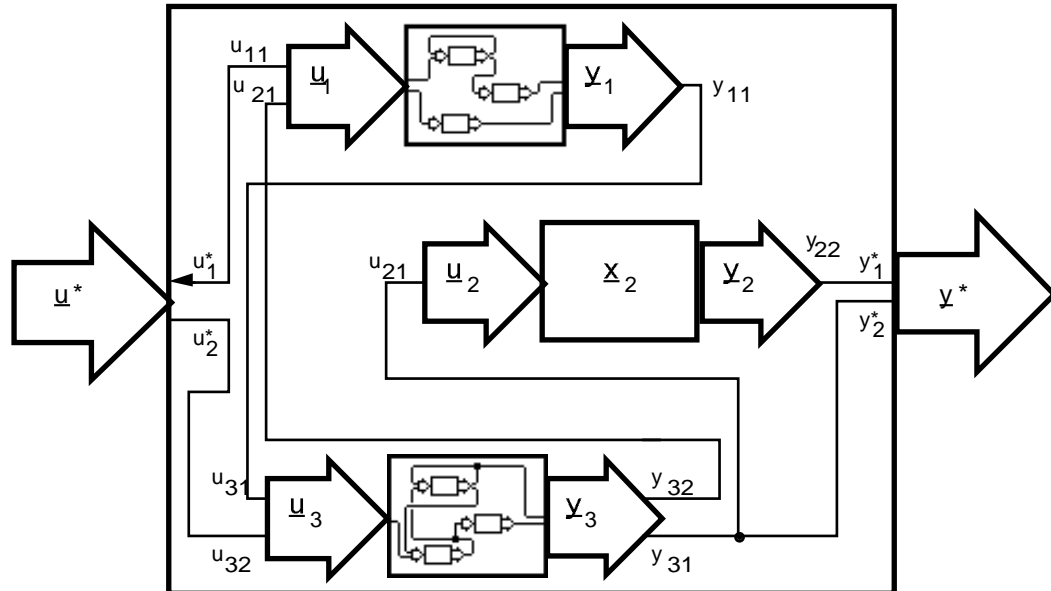
The global output  $\underline{y}^*$  of the structured model consists of two vectors, one for the global continuous time output  $\underline{y}^{*\xi}$  and the other for the global discrete time output  $\underline{y}^{*\delta}$ . Each is again composed from the global output vectors  $\underline{y}_{i^\xi}^*(t)$  of the continuous time submodels  $i^\xi \in \Xi$  resp. the global output vectors  $\underline{y}_{i^\delta}^*(\kappa)$  of the discrete time submodels  $j^\delta \in \Delta$ :

$$\begin{aligned} \underline{y}^{*\xi}(t) &= \begin{bmatrix} y_{1\xi}^*(t) \\ y_{2\xi}^*(t) \\ \vdots \\ \vdots \end{bmatrix} & (\in \Xi) \\ & & \text{(global outputs)} \\ \underline{y}^{*\delta}(\kappa) &= \begin{bmatrix} \vdots \\ \vdots \\ y_{n-1\delta}^*(\kappa) \\ y_{n\delta}^*(\kappa) \end{bmatrix} & (\in \Delta) \end{aligned} \quad (11c)$$

The general definition of the coupling has two special cases which are often of interest to the modeler:

- Structured model consisting of several, but uncoupled submodels: The inputs of the submodels do not depend on any output of another submodel:  $\underline{u}_i(t) = \underline{h}_i(\underline{u}^*(t))$  resp.  $\underline{u}_i(\kappa) = \underline{h}_i(\underline{u}^*(\kappa))$ . Such submodels coexist as completely independent units, yet implementing them within the same model definition program offers the advantage that they can be simulated in parallel at once. This might be useful when working with similar models, e.g. to test different model versions of the same real system, or to compare a measured time series (parallel model) with a simulated trajectory (model).

- The structured model is composed of hierarchically organized submodels (several levels): An example of such a hierarchical model system is given in Fig. T11 (two levels). Note however, that ModelWorks ignores the hierarchical organization, which is only of concern to the modeler. ModelWorks treats all models exactly the same way, regardless of the level on which they are defined.



**Fig. T11** : Example of a hierarchically organized structured model system composed of several submodels, which are themselves structured model systems consisting of several internally, coupled submodels.

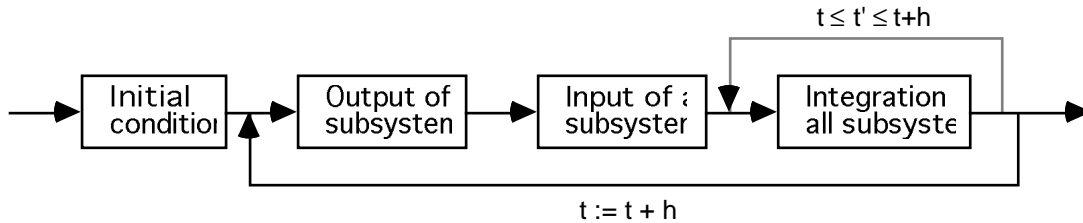
Structuring model systems as defined in Eq. (8a,b or c) requires a particular calculation sequence during simulation which may affect the results in a way which has to be considered by the modeler. In particular it must ensure that all input values are calculated first, i.e. at the begin of an integration step. Further, the results must be independent of the calculation order of the submodels. This can be guaranteed given that the following conditions are observed:

1. The calculation of a model is split into the following three parts:
  - a) Calculation of the input variables  $\underline{u}_i(t)$  for the submodel i: Eq. (8a, b or c)
  - b) Calculation of the derivatives resp. the new values of the state variables of the submodel i (integration): Eq. (9a, b, or c)
  - c) Calculation of the output variables  $\underline{y}_i(t)$  of the submodel i: Eq. (10a, b, or c)
2. The calculation order is that shown in Fig. T12.

ModelWorks guarantees that the prerequisite under point two is always met, but cannot ensure that none of the model equations are misplaced, e.g. that a derivative is calculated in a part reserved for the calculation of outputs.

Note also that the calculation order shown in Fig. T12 has a further consequence to be considered by the modeler: It may affect the precision of the numerical results depending on how the

equations are distributed among the continuous-time submodels. Differential equations coupled within a submodel are integrated differently from those coupled via submodel boundaries when using higher order integration methods. This fact should be considered when subdividing a model into several submodels unless the simulationist should restrict herself to single step integration methods only (s.a. the following example and Fig. T14).



**Fig. T12** : Calculation order applied by ModelWorks during integration. The larger loop corresponds to a single time step ( $h =$  current integration step); the inner loop is used only by integration methods with order  $> 1$  (e.g. Heun, Runge-Kutta 4<sup>th</sup> order) (s.a. Fig. T22).

Fig. T12 shows how coupling within a single submodel, i.e. formulated within the equation section dynamic, is defined at every point in time, whereas the coupling between submodels, i.e. formulated within the equation sections output respectively input, takes place only at the end points of an integration step. Note also that this phenomenon is different from the coupling between continuous and discrete time submodels, where the coupling is usually happening even more rare, i.e. only at the coincidence points. They are mostly much further apart than the current size of the integration step  $h$ . Both kinds of coupling, the one at the end points of the discretisation interval  $h$  as well as the one at the end points of the coincidence interval  $c$ , are of the same type, i.e. ModelWorks applies the so-called sample and hold technique (see also below under *Simulation environment* of the next chapter *Functions*).

Finally a simple example shall illustrate the whole concepts discussed in this chapter. The model is a system consisting of two ordinary, nonlinear first-order differential equations. First it shall be modeled simply and secondly it shall be modeled as a structured model built from submodels:

**Ex.:** The following model equations shall be modeled, first within a single model (Eq. 14):

$$\begin{aligned} \dot{x}_1 &= ax_1 - bx_1^2 - cx_1x_2 \\ \dot{x}_2 &= c'x_1x_2 - dx_2 \end{aligned} \tag{Model M} \tag{14}$$

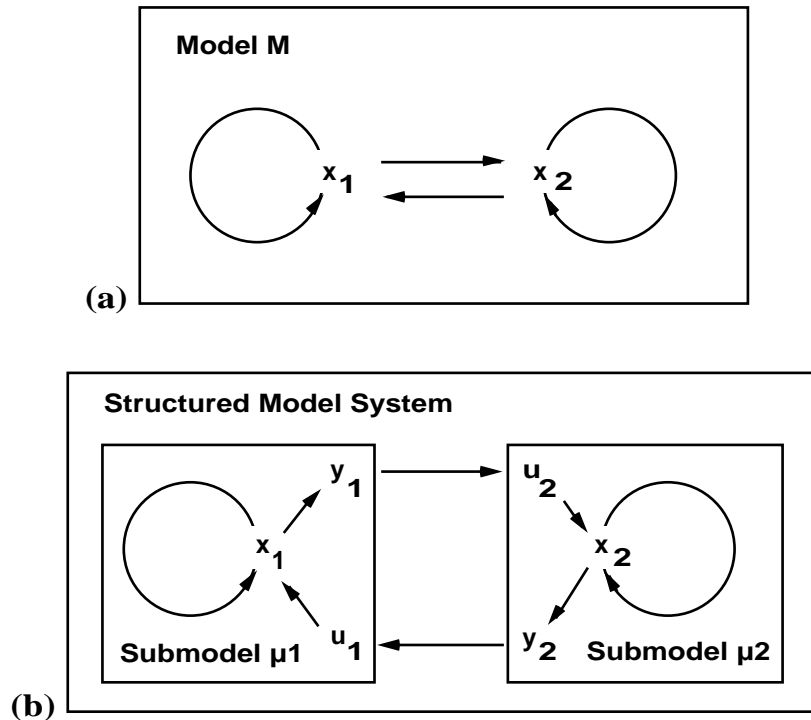
This system consists of two ordinary but coupled differential equations formulated according to Eq. (9a) with neither input nor output (autonomous system). See Fig. T13a for the relational diagram of this model system.

Secondly the two differential equations shall be distributed into two separated submodels (15) respectively (16), which are coupled with each other (Fig. T13b):

$$\begin{aligned} u_1 &= y_2 && \text{input according Eq. (8a)} \\ \dot{x}_1 &= ax_1 - bx_1^2 - cx_1u_1 && \text{dynamic according Eq. (9a)} \quad (\text{Submodel } \mu_1) \tag{15} \\ y_1 &= x_1 && \text{output according Eq. (10a)} \end{aligned}$$

respectively

$u_2 = y_1$	input according Eq. (8a)	
$\dot{x}_2 = c'x_1x_2 - dx_2$	dynamic according Eq. (9a)	(Submodel $\mu_2$ ) (16)
$y_2 = x_2$	output according Eq. (10a)	

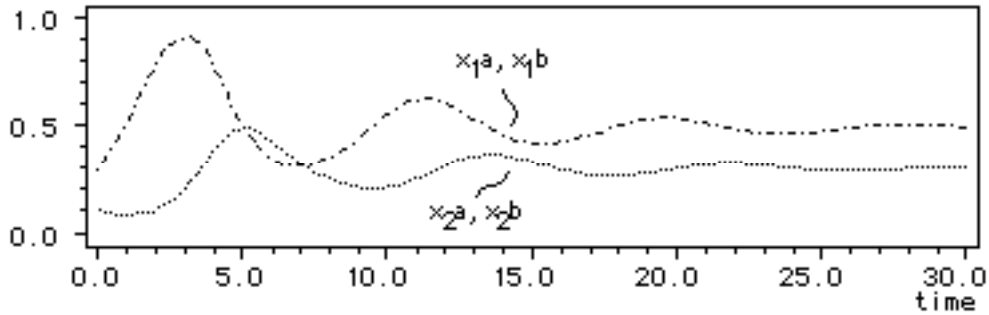


**Fig. T13** : Relational diagrams of a model once **(a)** formulated as a single elementary model M given by Eq. (14) and once **(b)** modeled as a structured model system consisting of two submodels  $\mu_1$  and  $\mu_2$  according to the Eq. (15) and (16).

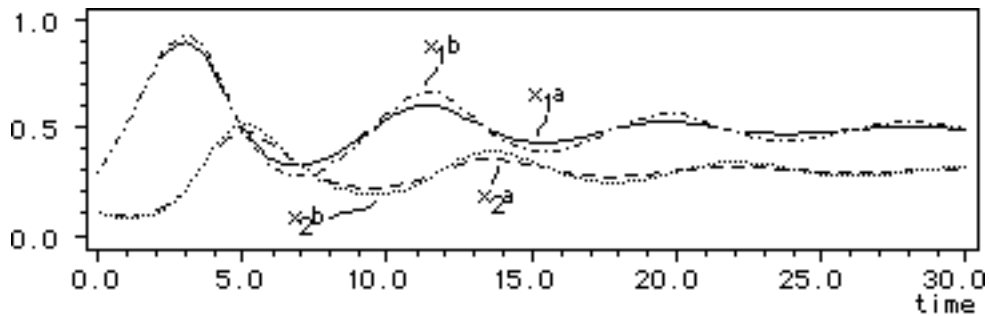
Both submodels are of the type continuous time and case **(A)** applies with the equations (7.a) till (11a), but no global inputs nor global outputs are present. Each of these submodels has one input and one output defined according to Eq. (8a) and (10a). These inputs and outputs have only been introduced in order to couple the two submodels. They form a structured model system, each submodel containing one of the differential equations from Eq. (14). Mathematically the structured model system formed with (15) and (16) is equivalent to the one given by Eq. (14). However, discretisation errors may result in the sample and hold effect described above (see also below under *Simulation environment* of the next chapter) (Fig. T14).

This is because no information exchange across submodel boundaries takes place during an integration step. Thus coupling among submodels occurs only at the endpoints of an integration step (s.a. Fig. T12). In case a higher order integration method is used, the coupling of differential equations within a submodel takes place even in the middle of an integration step. Hence simulation results of the continuous-time part of a structured model might slightly differ for non-single step integration routines depending on where the modeler has chosen the submodel boundaries between the differential equations. However the smaller the integration step, the smaller becomes this effect. E.g. in order to make the effect clearly visible, case **(ii)** of Fig. T14 has been computed with a rather large integration step of  $h=0.15$ .

(i)



(ii)



<u>Curves</u>		<u>Minimum</u>	<u>Maximum</u>	<u>Unit</u>
—	x1a prey (variant a)	0.000	27000.000	#
---	x2a predator (variant a)	0.000	1200.000	#
----	x1b prey (variant b)	0.000	27000.000	#
.....	x2b predator (variant b)	0.000	1200.000	#

**Fig. T14** : Simulation results of two mathematically equivalent model variants a and b as given by Eq. (14) respectively Eq. (15-16). Results obtained using **(i)** - the first order Euler, **(ii)** - the second order integration method Heun with steplength  $h = 0.15$ . Although the two model variants (s.a. Fig. T13) ought to behave identically, their two implementation variants a and b yield the same results only in case **(i)**, but differing ones in case **(ii)**. This is a consequence of the calculation order within a simulation step (Fig. T12) and the order of the integration method: In case **(ii)** the information exchange between submodels is not so often done for variant b than for variant a, because it takes only place at the begin of, not during an integration step.  $x_{1a}, x_{2a}$  - state variables of variant a;  $x_{1b}, x_{2b}$  - of variant b.



## 5 Functions

ModelWorks functions are provided by its simulation environment and are available in two ways: First by the simulationist via the user interface and second by the modeller via the client interface (Tutorial Fig. T1). The standard, interactive user interface of ModelWorks allows the simulationist to access the majority of the functions interactively. The client interface allows the modeller to access all functions, but in a static way, i.e. through the writing of a Modula-2 program, typically a model definition program (Tutorial Fig. T3). Both techniques have their unique advantages and disadvantages and can be freely mixed in any combination.

The simulation environment of ModelWorks provides at run-time the needed base and environment to produce model behaviour trajectories, e.g. by executing simulation runs, changing parameter values, monitoring settings, or defining simulation experiments etc. A simulation run of ModelWorks corresponds to the numerical solution of an initial value problem of a set of ordinary differential equations or difference equations alone or in any mixed form (s.a. chapter *Model Formalisms*). In particular note that this means that the current implementation of ModelWorks does neither provide a direct support for the solution of boundary value problems, nor partial differential equations, nor does it offer backward numerical integration. Hence, the simulation environment provides only one single independent variable, typically the time, and any model currently installed, regardless of the installation mechanism, will be solved in function of this variable only.<sup>5</sup>

ModelWorks' simulation environment comes into existence as soon as a module is executed which does import either from module *SimMaster* or module *SimBase*, or both. All activities from the starting of the simulation environment till its quitting are termed a simulation session<sup>6</sup>.

The ModelWorks simulation environment consists of the following components:

- Model base
- Global parameters and settings
- Simulation run-time system
- Standard user interface

The model base allows to install or deinstall any number of models together with their model objects and all associated data. The global parameters and settings allow to control the general properties and appearance of the simulation environment, e.g. to specify the time domain of a

---

<sup>5</sup>The module *SimIntegrate* (see *Appendix* section *Definition Modules*) provides an exception to this rule: Numerical integration of definite integrals can be computed without affecting the global independent variable.

<sup>6</sup>Any eventual interruption of a session, e.g. if the simulationist starts to work on something else, hereby switching the simulation environment into the background, is ignored and omitted from the simulation session. This is because such activities will in general not affect the state of the simulation environment. However, any activity, e.g. executing another model definition program, also operating on the same simulation environment, is considered to form part of the same simulation session. Note also: On the Macintosh under MultiFinder or System 7 the simulation environment will not cease to operate when switched into the background; in case a simulation run is still in execution while the background switch occurs, this run will continue till ModelWorks requires the next input by the simulationist. This feature can be very helpful, e.g. if a researcher has to execute lengthy simulations she does not need to observe interactively; in this case she can launch a simulation experiment and start writing a paper or analyzing data etc. The simulation can be fetched into the foreground anytime to check its progress and then be rereleased into the background again. Albeit, in background simulations run slower than in foreground, since foreground applications get a higher priority and more CPU-time than a background process. Of course any number of ModelWorks applications can coexist in such a fashion.

simulation experiment or to define the location of windows on a screen. The simulation run-time system provides all algorithms needed for numerical integration, model coupling, and graphics etc. The standard user interface allows the simulationist to access the typically needed functions of the simulation environment interactively (see subchapter *Simulation Environment*).

If the modeller wishes to give the simulationist access to the standard user interface (see subchapter *Standard User Interface*) she simply activates the latter by calling procedure *RunSimEnvironment* (exported by module *SimMaster*). The standard user interface may be quit anytime, even in the middle of a simulation, by the menu command *Quit*.

ModelWorks allows to extend the ModelWorks' standard user interface with additional, user specific functions, or to use ModelWorks even within a completely different user interface (see subchapter *User Interface Customization*). Since ModelWorks is based on the "Dialog Machine" the modeller may also access "Dialog Machine" routines herself and mix them with the functions provided by ModelWorks. For instance the modeller might want to have an additional kind of monitoring not offered by the standard ModelWorks functions. To accomplish this she may add a new menu with commands to open a window in which simulation results are to be displayed in a problem specific graph. E.g. by drawing a line chart of the progress of a parameter identification in the parameter space according to the current values of the performance index or to draw the age pyramid of an age structured population during the simulation of its population dynamics (s.a. chapter *Sample Models* in the *Appendix*).

The client interface must be used to define, i.e. to declare, the models, the model objects, the model equations, and the default values for all objects so that the run time system of ModelWorks may access and maintain them (Tutorial Fig. T2). It is important to note that ModelWorks does only know about those objects which have been made known to it, i.e. which have been explicitly declared by means of one of the following procedures from module *SimBase*: *DeclM*, *DeclSV*, *DeclP*, or *DeclMV* (see subchapter *Model Base*). Otherwise ModelWorks does not care what the modeller is doing with these objects, nor whether they are involved in a complex structure or operation. For instance a state variable might be part of a structured data type such as a Modula-2 record or an array or might be computed by first retrieving input values from a data base. On the other hand it is also important to understand that ModelWorks will operate on model objects, i.e. will repeatedly access their values. E.g. at the begin of a simulation run ModelWorks assigns automatically the initial values to all state variables or updates the values of state variables during the simulation by assigning to them the results of numerical integrations (see section *Model objects and the run-time system*). In order to use ModelWorks meaningfully it is therefore necessary that the modeller obeys a minimum number of rules, so that ModelWorks and the modeller use and access model objects in harmony.

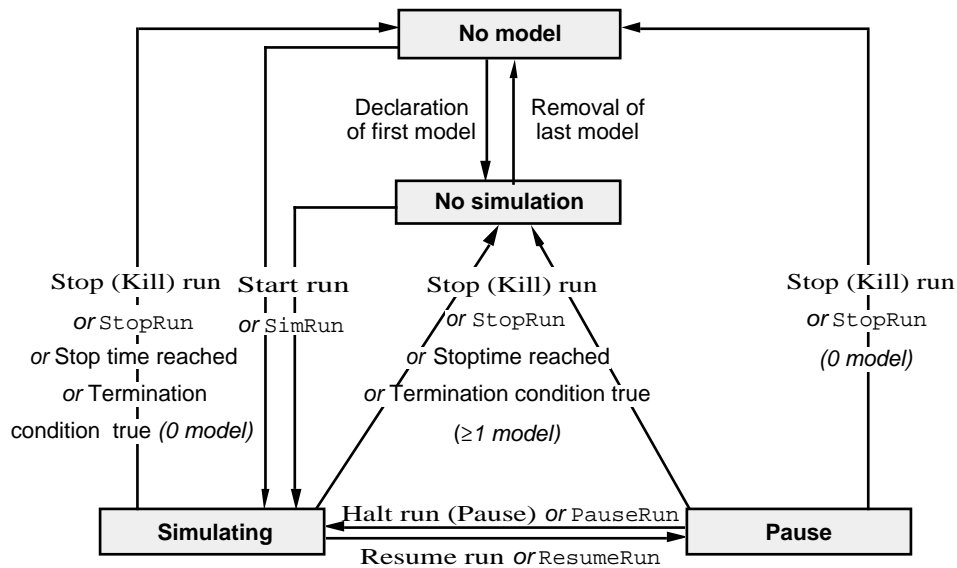
## 5.1 Simulation Environment

### 5.1.1 STATES OF THE SIMULATION ENVIRONMENT

During a simulation session the simulation environment is always only in one of four states: *No model*, *No simulation*, *Simulating*, or *Pause* (Fig. T15). States reflect what basic operations have been performed on the simulation environment and may also determine the availability of certain commands or functions.

In the state *No model* the simulation environment's model base is empty, i.e. no model has been installed. As a consequence any commands such as the menu command *Settings/Reset: All model's parameters* of the standard user interface are disabled; similarly a call to procedure *ResetAllParameters* (from module *SimBase*) is without any effect. Only if at least one model is installed, the simulation environment is set to the state *No simulation* where all functions requiring at least one model as a precondition such as the changing of model and model object attributes become possible. The state *No simulation* serves the starting of simulations and is typically used to change settings or values such as initial values, model parameters etc. In the state *Simulating*, i.e. once a simulation run has been started, the functions with the potential to

conflict with the running simulation, e.g. changing the simulation start time  $t_0$  resp.  $K_0$ , produce a slightly different result. In the state *Pause* the simulation is temporarily brought to a halt, for instance to allow for a closer inspection of the simulation results (Fig. T15).

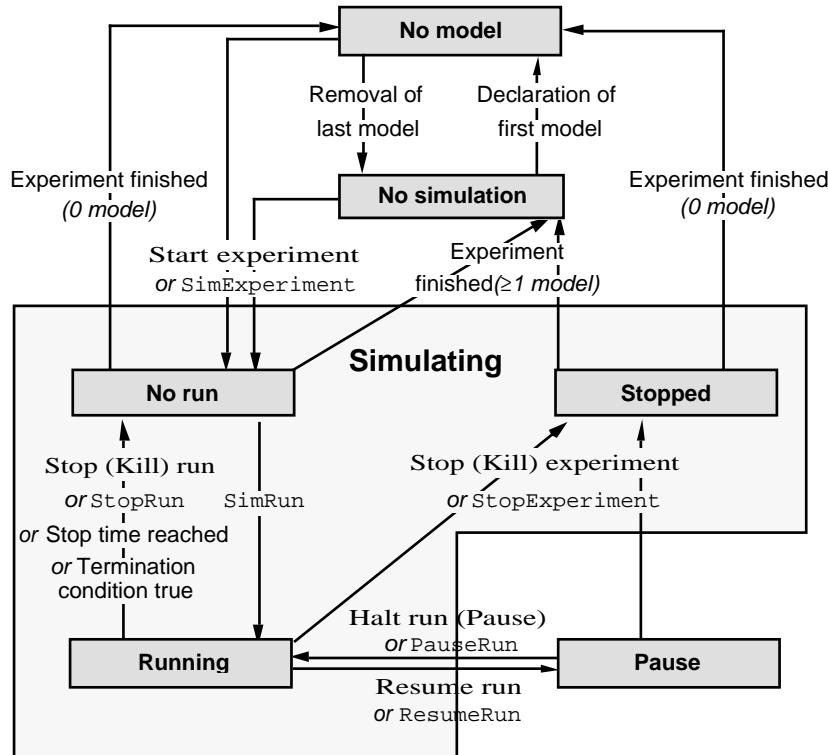


**Fig. T15** : State transition diagram of the simulation environment of ModelWorks when executing elementary simulation runs. The states are: *No model* - when there is no model present, *No simulation* - when there has been at least one model declared, but no simulation is running yet, *Simulating* - simulation is running, and *Pause* - running simulation has been temporarily halted. The standard user interface reflects these states by the enabling or disabling of commands accordingly (s.a. Fig. T24). The transitions controlled by the simulationist are labelled with the bold text of the corresponding menu commands of the standard user interface, e.g. **Start run**. The transitions under the control of the modeller are labelled with plain style procedure identifiers, e.g. *PauseRun*.

Since the effectiveness of certain functions may depend on the current state of the simulation environment, the standard user interface adjusts accordingly: Fig. T24 illustrates in which state which menu commands are enabled (black) or disabled (dimmed) and which so-called IO-windows are active (black title bar), i.e. respond to mouse clicks, or are inactive (grey title bar), i.e. do not respond to any mouse clicks. The fact that certain functions are only available in certain states could be perceived as an undesirable limitation. However, they have been merely introduced because certain commands are meaningful only in particular states and to ensure maximum consistency. For instance, the command to stop a simulation is meaningless if there is currently no simulation run in progress; thus the standard user interface offers this function only in the states *Simulating* or *Pause*. For more information on this topic see the section *States of the standard user interface*

Transitions from one state to another are accomplished in several ways: First most transitions are caused by the simulationist who chooses particular menu commands such as those of the menu *Solve* available in the standard user interface (Fig. T15). Secondly, as programmed by the modeller, procedures like *StopRun* from module *SimMaster* are called. Thirdly ModelWorks causes transitions by itself, for instance when leaving the state *Simulating* due to one of the following two reasons: The simulation time has reached the stop time, or the terminate condition provided by the modeller returns true. Note that this implies that state transitions are under the control of three masters: the simulationist, the modeller, and/or the ModelWorks software. For instance in the standard user interface the following state transitions are under the control of the simulationist (Fig. T15, T2New4): In the states *No model* and *No simulation* the menu command *Start run*; in state *Simulating* *Stop (kill) run*; in state *Pause* *Resume run* or

*Stop (kill) run.* All these transitions may also be controlled by the modeller via calls to the following procedures *SimRun*, *StopRun*, *PauseRun*, and *ResumeRun*. In case of multiple, conflicting control the simulationist has the highest priority, followed by the programming by the modeller, and lowest priority has the ModelWorks run-time system, but any request to stop the simulation will lead to a termination of the run as soon as the current integration step has been completed.



**Fig. T16 :** State transition diagram of the simulation environment of ModelWorks when executing structured simulation runs (experiments). The main states are the same as the ones shown in Fig 15. However, the state *Simulating* must still be split into three further sub-states: *No run*, *Running*, and *Stopped*. The substate *No run* allows the modeller in the middle of an experiment to modify values as freely as in the state *No simulation*. In the substate *Running* a few functions which might disturb the ongoing simulation are no longer possible. In the substate *Stopped* any attempt to execute more runs are ignored (see Fig. T15 for the meaning of the labels of the transitions).

When programming structured simulations (experiments) (see subchapter *Programming Structured Simulations (Experiments)*), the state *Simulating* is split into three further sub-states: *No run*, *Running*, and *Stopped* (Fig. T16). Whenever procedure *SimRun* from module *SimMaster* is executed, the simulation environment is in substate *Running*; in the remainder of the experiment procedure it is in the substate *No run* (unless the experiment has been stopped). The substate *No run* has been introduced to allow the modeller in the middle of an experiment to modify values like global simulation parameters, e.g. the simulation start time  $t_0$ , as freely as in the state *No simulation*. For instance to simulate in a row an agroecosystem model during several, by the winter separated growing seasons requires to set the next time domain  $[t_0, t_{end}]$  between two consecutive simulation runs. But the simulation start time  $t_0$  can't be set to a value bigger than the current time  $t$ , since this would require the simulation time to jump. The latter is only possible in the substate *No run*, which is in contrast to the substate *Running* less restrictive, a behaviour which may be essential for structured simulations.

The substate *Stopped* is reached if the experiment has been stopped and any subsequent runs are to be suppressed.

The following typical transitions take place in experiments which are executed from within the standard user interface or by a call to procedure *SimExperiment* from *SimMaster*. Once the simulationist has chosen the menu command *Solve/Start experiment*, this causes the simulation environment to enter the main state *Simulating* plus the substate *No run* (Fig. T16). As soon the procedure *SimRun* is called from within the procedure *DoExperiment*, the substate *No run* is left and ModelWorks enters the substate *Running*. If a simulation run is finished, either because the modeller has called *StopRun*, the simulation time has reached the stop time, or the termination condition has returned true, the sub state *Running* is left and the substate *No run* is re-entered (Fig. T16). Once all simulation runs have been completed and the end of the experiment procedure has been reached, the main state *No simulation* is resumed again unless there should all models have been removed; in the latter case the state *No model* is resumed instead.

In case the simulationist chooses from within the standard user interface the menu command *Solve/Stop (Kill) experiment* or the modeller calls *StopExperiment* from *SimMaster*, the substate *Stopped* is entered. In substate *Stopped* the modeller can program a final analysis of the experiment or perform any other house-keeping tasks similar to the possibilities in the substate *No run*. Note that in substate *Stopped* ModelWorks will still execute all remaining statements, including eventual calls to procedure *SimRun*, i.e. till the procedure *Experiment* is actually finished. However, no more integration and monitoring will take place. This is because ModelWorks empties the body of the procedure *SimRun*, so that the structured simulation will terminate without any further computations by the run-time system. The simple Boolean function procedures *ExperimentRunning* and *ExperimentAborted* from module *SimMaster* allow to determine whether the simulationist has started respectively aborted an experiment. This allows the modeller to program structured simulations accordingly, e.g. to exit from a loop calling *SimRun* as soon as *ExperimentAborted* returns true (s.a. subchapter *Programming Structured Simulations (Experiments)* and in the *Appendix* the sample models demonstrating *Stochastic Simulations* such as *Markov* or *StochLogGrow*).

The first group of procedures causing state transitions, i.e. *SimRun* and *SimExperiment* from module *SimMaster*, keep the program control during their whole execution. As long as one of these procedures is executing the simulation environment is either in the state *Simulating* or in the state *Pause*. Note, the second group of state transition causing procedures, i.e. *StopRun*, *StopExperiment*, *PauseRun*, and *ResumeRun* from module *SimMaster* set only a semaphore, i.e. they inform only ModelWorks run-time system about the wish, that a state transition ought to happen as soon as possible. This technique allows ModelWorks to complete first consistently an eventually already started integration step before actually making the transition. As a rule follows that the modeller's program calling one of the semaphore settings procedures should immediately relinquish control and return it to the "Dialog Machine". In particular no attempts should be made to call another procedure causing a state transition. This will allow the first transition to actually take place. For instance it is not possible to execute the statement sequence ... *StopRun*; *SimRun*; ... successfully because of the following reasons: First the call to *StopRun* signals to the run-time system to stop the run; but, this has only an effect on the semaphore and the program has not yet returned from procedure *SimRun*. The latter could only happen if the program control would be relinquished immediately after calling *StopRun*, which is of course not the case if *SimRun* is subsequently called. Moreover *SimRun* disallows any recursive calls to itself. Hence the subsequent call to *SimRun* will be without any effect and no new simulation run can be started by such a method.

The modeller can request to be informed about all state changes by installing a state change handler (see *InstallStateChangeSignaling* from module *SimMaster* and for an example the subchapter *User Interface Customization*). Whenever the simulation environment changes substates, the installed state change handler will also be called. The current state can then be inquired by the modeller via the client interface by calling the procedure *GetMWState* from module *SimMaster*. The substates can be inquired via procedure *GetMWSubState*. Substates are only defined while an experiment is executing, hence *GetMWSubState* returns otherwise always *no-SubState* (note the latter is also the case if the experiment has been temporarily paused).

## 5.1.2 MODEL BASE

### 5.1.2.a Model and model object installation and removal

ModelWorks allows to install or deinstall any number of models and model objects at any time. The actual limitations are not inherent in the software but are only given by the available computer resources, i.e. the currently available heap space and the computing power needed to numerically solve the models.

Every model definition program imports either from module *SimMaster* or *SimBase* and calls the model base of the simulation environment into existence. Initially there are no models installed in this base and the simulation environment resides in the state *No model*. Typically the actual definition of the models and model objects are contained in the body of declaring procedures. The procedure containing the call(s) to procedure *DeclM* from module *SimBase* is often either executed as one of the first statements in the body of the model definition program or indirectly executed by passing it as the actual argument to procedure *RunSimEnvironment* (Tutorial Fig. T3). Any successful call to procedure *DeclM* from module *SimBase* while no simulation is running will result in a state transition from state *No model* to state *No simulation* (given the simulation environment should not already be in the state *No simulation*). If a simulation is running a successful call to *DeclM* will cause no state transition until the simulation is ended. Then the simulation environment will enter the appropriate state, i.e. *No model* or *No simulation*, depending whether the model base contains at least one model or none (Fig. T15, T16).

Models and model objects can also be declared or removed in the state *Simulating* even in the middle of a simulation run (including substate *Running*; Fig. T15, T16). In the latter case apply a few rules, which are slightly different from the effect of a declaration outside state *Simulating* (see section *Manipulating the model base at run-time*). They ensure the consistency of the model base and all associated data during the whole simulation session.

Removing a model implies always the removal of all its model objects together with a loss of all data associated with this model and its objects.

ModelWorks' standard user interface provides no means to install and remove models; hence, once installed, neither the number of models nor that of the model objects can be changed without quitting first this user interface (Fig. T18). However, it is possible to extend the user interface by menu commands, which support the dynamic declaration and the removing of models and model objects. If the procedures *DeclM* resp. *RemoveM* are called from within procedures, which are attached to some additional menu commands, the simulationist gets the power to install and remove models dynamically (s.a. subchapter *User Interface Customization* and in the *Appendix* the research sample model *LBM*). For instance in its simulation session the RAMSES shell does provide a mechanism, which allows to load and unload models, in form of individual model definition programs, which may be even called on top of each other.

### 5.1.2.b Current values

To allow the simulationist for interactive experimentation with models, model objects, and the global settings of the simulation environment, ModelWorks provides for all these data scratch copies which may be manipulated freely. For instance the standard user interface allows her to manipulate not only the monitoring, but also the current global settings such as window positions, and numerical values or attributes of the models and the model objects. All these data are called current values. Any numerical integration works with the current values only.

Current values can either be interactively changed from within the standard user interface by means of the mouse (e.g. window positions), menu commands, entry forms, and IO-windows or via the client interface by calling procedures such as *SetSV*, *SetP* etc. from module *SimBase*. Changes can also be made in the middle of a simulation run (including substate *Running*; Fig. T15, T16). In the latter case apply a few rules, which are slightly different from the effect

of a change outside state *Simulating* (see section *Manipulating the model base at run-time*). They ensure the consistency of the current values during the whole simulation session.

In particular from within the standard user interface in the states *No model* or *No simulation* modifications are possible for the following settings and parameter values:

- Global parameters and settings (*No model* and *No simulation*):
  - start ( $t_o/\kappa_o$ ) and stop ( $t_{end}/\kappa_f$ ) time for simulation runs
  - integration step<sup>7</sup> respectively maximum integration step ( $h/h_{max}$ ) plus maximum relative local error ( $e_r$ )<sup>8</sup>
  - discrete time step or coincidence interval ( $c$ )<sup>9</sup>
  - monitoring interval ( $h_m$ )
  - project description consisting of a title, remark, and footer string plus parameters which control the display of strings in the graph respectively the recording of information on models, model objects and table functions together with their current values and settings on the stash file (recording flags)
  - stash-file name, type, and creator
  - Window positions and arrangements
- Model specific attributes (*No simulation*):
  - integration method
- Model objects specific attributes (*No simulation*):
  - initial values of state variables
  - values of model parameters
  - kind of monitoring, scaling, and curve attributes for monitorable variables

In addition ModelWorks offers a versatile reset mechanism which allows to reset any settings or parameters which may have been modified during the simulation session by the simulationist or via the client interface by the model definition program. The values or settings to which ModelWorks resets are the so-called default values, either originally specified by the modeller or later by a call to a procedure redefining defaults via the client interface. This allows the simulationist within the standard user interface to return any time to a well defined state of all parameters and settings, regardless of the degree to which they have been manipulated (see section *Predefinitions, defaults, and resetting*).

#### 5.1.2.c Predefinitions, defaults, and resetting

ModelWorks maintains for the global parameters and settings and for all data contained in its model base two copies: One is the default value, the other is the current value (s.a. section *Current values* Fig. T17). Two copies exist for the following kind of data:

- Global parameters and settings:
  - global simulation parameters  $t_o/\kappa_o$ ,  $t_{end}/\kappa_f$ ,  $h/h_{max}$ ,  $e_r$ ,  $c$ ,  $h_m$
  - project description and recording flags
  - stash-file name, type, and creator
  - Window positions and arrangements

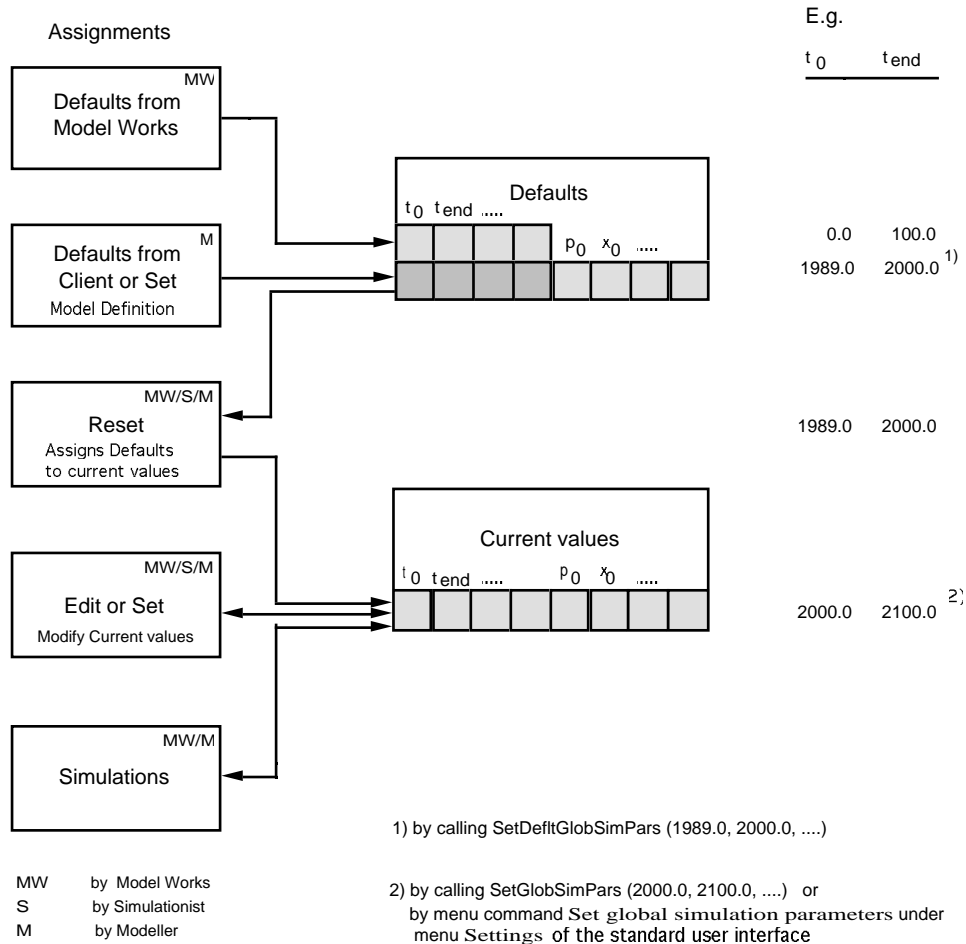
<sup>7</sup> Interactively via the standard user interface only if at least one continuous time (sub)model is present

<sup>8</sup> Interactively via the standard user interface only if at least one continuous time (sub)model with a variable step length integration method is present

<sup>9</sup> Interactively via the standard user interface only if at least one discrete time (sub)model is present

- Model and model object specific attributes:

- integration method
- initial values of state variables
- values of model parameters
- kind of monitoring, scaling, and curve attributes for monitorable variables



**Fig. T17 :** Relationship between default and current values and the reset mechanism of ModelWorks. ModelWorks maintains for most values two copies: One is the default, the second is the current value, which is actually used for simulations. Some defaults, for instance for the global simulation parameters, are predefined by ModelWorks, others such as those for model objects are only specified by the modeller. During a reset, also executed at program start up, the default values are assigned to the current values. Interactive modifications (editing) of values from within the standard user interface affect only the current values. Via the client interface it is possible to change the defaults as well as the current values.

The modeller is forced by the client interface to specify defaults for all models and model objects. They are the values passed to ModelWorks while declaring the particular objects. E.g. the value 0.1 is the default of the model parameter *c1*. This requires that the modeller declares the parameter *c1* with the following call: *DeclP(c1,0.1, ...* ModelWorks will keep a copy of the object's default value in order to be able to reassign it to the current value if a reset is requested by the simulationist. Choosing a menu command such as *Settings/Reset all model's parameters* from the standard user interface or calling procedure *ResetAllParameters* from module *SimBase* in a simulation session while running above example will then assign 0.1 to the variable *c1* (Fig. T17) regardless of what the value of *c1* currently might be.



Symbol	Meaning of parameter or variable	Predefined default
Global simulation parameters		
$t_o/\kappa_o$	Start time for simulation	0.0
$t_{end}/\kappa_f$	Stop time for simulation	100.0
$h/h_{max}$	Fixed integration step or maximum integration step for continuous time (sub)models	0.05
$e_r$	Maximum relative local integration error	0.001
$c$	Discrete time step for discrete time (sub)models or coincidence interval for mixed time structured models	1
$h_m$	Monitoring interval	0.25
	Descriptor, identifier, and unit for independent variable	"time" "t" ""
Project description		
	Project title string	""
	Use project title string in graph	TRUE
	Remarks string	""
	Use remarks string in graph	TRUE
	Footer string	"dd/mon/yyyy hh:mm Run 1" <sup>10</sup>
	Automatic update of date, time, and run # in footer	TRUE
	Recording of data about models in stash file	TRUE
	Recording of data about state variables in stash file	FALSE
	Recording of data about model parameters in stash file	FALSE
	Recording of data about monitorable variables in stash file	TRUE
	Recording of graph in stash file	FALSE
	Recording of table functions in stash file	FALSE
Stash filing		
	Stash file name	ModelWorks.DAT <sup>11</sup>
	Macintosh file type and creator (signature) actually determined by module DMFiles' default from the "Dialog Machine" file type creator	e.g. 'TEXT' e.g. 'MEDT'
Automatic definition of curve attributes		<b>Predefined value</b>
	colours and line-styles	$i \text{ MOD } 4 = 12$ 0: coal unbroken 1: ruby broken 2: emerald dashSpotted 3: turquoise spotted $i =$
	symbols	4: • 5: * 6: o 7: Δ else ""

Tab. T1: Predefined default s: Unless overwritten by the modeller, ModelWorks assigns the given default values to the listed parameters. For the defaults of models and model objects, the modeller is forced to specify them while declaring the models and the model objects in the model definition program. Predefined values can not be overwritten by the modeller.

<sup>10</sup>The abbreviations stand for: dd - current day, e.g. 01 for the first day of a month; mon - current month, e.g. Jan for January; yyyy - current year, e.g. 1989; hh - current hour, e.g. 22 for 10 pm; mm - current minute, e.g. 04 in 10:04 pm

<sup>11</sup>On the IBM PC *MODELWOR.DAT*. Will be created in the folder where the application resides, which has started the model definition program respectively on the IBM PC in the current working directory.

Interactive modifications of values from within the standard simulation environment by using entry forms or the IO-windows affect always only the current values, not the defaults. Calling a reset function from ModelWorks will then reassign the default values to the current values. All current values affected by the reset, e.g. all initial values of a particular model, will then be set to their defaults as have been defined latest via the client interface (Fig. T17).

Resets can be executed for particular classes of data. Resets may affect only a single model object, all objects of a single model, or all objects of all models (s.a. Fig. T26). Furthermore resets can be executed for a particular class of model objects only, e.g. only windows or only the curve attributes of monitorable variables. The declaration of a model or model object will always result in an implicit assignment of the default to the current value, i.e. an individual reset.

Generally defaults are defined by two mechanisms, but always via the client interface or a non-standard user interface only: The first mechanism is provided by the declaration of models and model objects; all these defaults are provided by the modeller only and belong to the individual model or model object only. The second mechanism is used for all global parameters of the simulation environment; these defaults are not necessarily provided by the modeller, hence ModelWorks provides them in form of predefined defaults or the so-called predefinitions (Fig. T17). Thus, whenever the simulation environment enters the state *No model*, ModelWorks assigns to every global simulation parameter, the project description, or the stash file name the appropriate predefinitions (Tab. T1). Then the model definition program may overwrite these predefinitions with defaults preferred by the modeller, i.e. she calls *SetDefl<sub>xyz</sub>* procedures.

E.g. does ModelWorks use a predefined default simulation start and simulation stop time of  $t_0 = 0.0$  respectively  $t_{\text{end}} = 100.0$ . If the modeller wishes to use a different default simulation time range, she calls the procedure *SetDeflGlobSimPars* from module *SimBase* to define it, e.g. with the statement:

```
SetDeflGlobSimPars (1989.0, 2000.0, ...)
```

typically in the procedure *initSimEnv*, which is passed as actual argument to procedure *RunSimEnvironment*.

When executing *RunSimEnvironment* for the first time (see also section *Initialization of the simulation environment*), ModelWorks assigns automatically all defaults, either provided by ModelWorks in form of predefinitions or overwritten by the modeller as her defaults, to the current values (Fig. T17, T18). Such an assignment is called a full reset, corresponding to a call to procedure *ResetAll* from module *SimBase*.

Via the client interface or a non-standard user interface defaults can be changed always, even during a simulation run (s.a. section *Manipulating the model base at run-time*). Note however, that such changes will not become effective until a corresponding reset is actually executed.

Unless curve attributes are assigned to the monitorable variables either interactively by changing the current curve attributes in the monitorable variable window or via the client interface by calling the procedures *SetCurveAttributesForMV* or *SetDeflCurveAttributesForMV*, ModelWorks adopts the so-called automatic definition of curve attributes. It has been designed so that curves can be optimally told apart on black and white as well as colour devices, such as monochrome or colour screens, on laser printers or on colour ribbon matrix printers, on slide recorders, plotters etc. However, this has the disadvantage that for a particular monitorable variable the curve attributes may change too often, i.e. as soon as the automatic curve attribute of another, previously activated monitorable variable is changed. To avoid the latter, the user has to override the automatic definition. Note that the curve attributes assigned by the automatic definition are predefined by ModelWorks only and can not be changed by the user. ModelWorks uses the values listed in Tab. T1. Attributes are distributed according to the position  $i$  in the sequence in which the monitorable variables have been activated for graphical monitoring.

---

<sup>12</sup> $i$  is the order of activation of the monitorable variables, the first variable's value  $i = 0$ .

### 5.1.2.d Initialization of the simulation environment

The simulation session consists of all activities done by means of the simulation environment during the existence of the importing module, e.g. a model definition program. Initially the simulation environment is in state *No model* (Fig. T15, T16) and all pre definitions are assigned. The simulation environment is now ready to accept model declarations (Fig. T18).

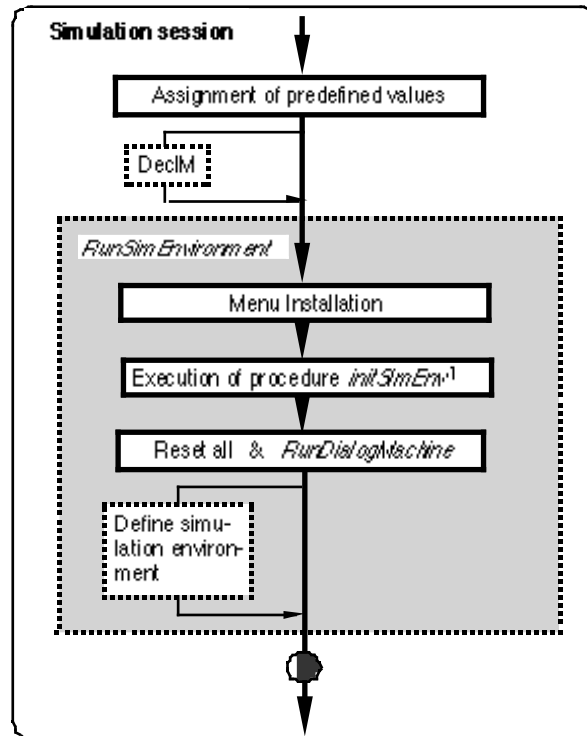


Fig. T18 : Flow chart of the initialization of the simulation environment by a typical model definition program using the ModelWorks standard user interface by a call to procedure *RunSimEnvironment*. Unless defaults are later changed (only possible via the client interface), the standard user interface allows to fully reset a simulation session's model base to the original start-up conditions (s.a. Fig. T17).

<sup>1</sup>argument passed in call to procedure *RunSimEnvironment*

The first successful model installation by means of procedure *DeclM* from module *SimBase* will bring the simulation environment into the state *No simulation* and ModelWorks is now ready to perform simulations, e.g. by a call to procedure *SimRun* from module *SimMaster*.

Typical model definition programs will use the standard interactive user interface by calling after the model declarations procedure *RunSimEnvironment* from module *SimMaster*. *RunSimEnvironment* will first install the standard user interface, e.g. its menu bar, and then execute the procedure *initSimEnv* which has been passed as its actual argument. Then it performs a full reset corresponding to a call of procedure *ResetAll* from module *SimBase* (Fig. T18). Hence, the best place to define defaults different from ModelWorks predefinitions, e.g. for the simulation time, or to extend the user interface, e.g. by installing an additional menu, is within procedure *initSimEnv*. The subsequently executed, automatic full reset ensures that all current values to be used during the subsequent simulation session have exactly the values as defined by all defaults (s.a. chapter *Predefinitions, defaults, and resetting*). Finally *RunSimEnvironment* calls procedure *RunDialogMachine* from module *DMMaster* (FISCHLIN, 1986a,b; KELLER, 1989). Note that the latter will then call implicitly any eventually installed simulation environment definition procedure *defineSimEnv* (see procedure *InstallDefSimEnv* from module *SimMaster*), before rendering control to the simulationist. *defineSimEnv* allows to customize

the interactive simulation environment, e.g. by reading data from a file (see *Appendix* e.g. sample model *SwissPop*) or opening an additional window, once the "Dia log Machine" has started to run the standard user interface.

Note that as long as the simulationist remains within the standard simulation environment of ModelWorks, a full reset resumes the initial program state which existed at start-up time. This is because, in contrast to the client interface, it is not possible to access and modify defaults via the standard user interface. However, if the modeller, by using the client interface, has programmed extensions (see subchapter *User Interface Customization*), which allow to change interactively also the defaults, the reset mechanisms provided by ModelWorks will no longer guarantee the simulationist to resume this initial start-up condition. Instead the state as defined by the last default specifications will be resumed. Note however, the client can install a simulation environment definition procedure *defineSimEnv* (see procedure *InstallDefSimEnv* from module *SimMaster*), which allows to implement this functionality: Since ModelWorks will not call *defineSimEnv* as part of the menu command *Settings/Reset all above* nor of the procedure *Reset-All*; the initial start-up conditions are resumed exactly if *defineSimEnv* first sets resp. reassigns all defaults, regardless of their eventual modification, and then calls procedure *ResetAll* from module *SimBase*. The simulationist can then resume exact initial start-up condition by simply choosing the menu command *Settings/Define simulation environment*.

### 5.1.3 SIMULATIONS AND THE RUN-TIME SYSTEM

After successful installation of a model by means of procedure *DeclM* from module *SimBase*, the simulation environment enters state *No simulation*. Eventually open IO-windows will then display the new information, current settings and values of all model(s) and all models' objects.

Once in state *No simulation*, from within the standard user interface the simulationist has the choice either to change interactively any settings or to start immediately a simulation with the predefined default values. The latter is possible without any further action, since the client interface has been designed such that the modeller is exhorted to provide all needed values required to define fully the initial value problem of a ModelWorks simulation run.

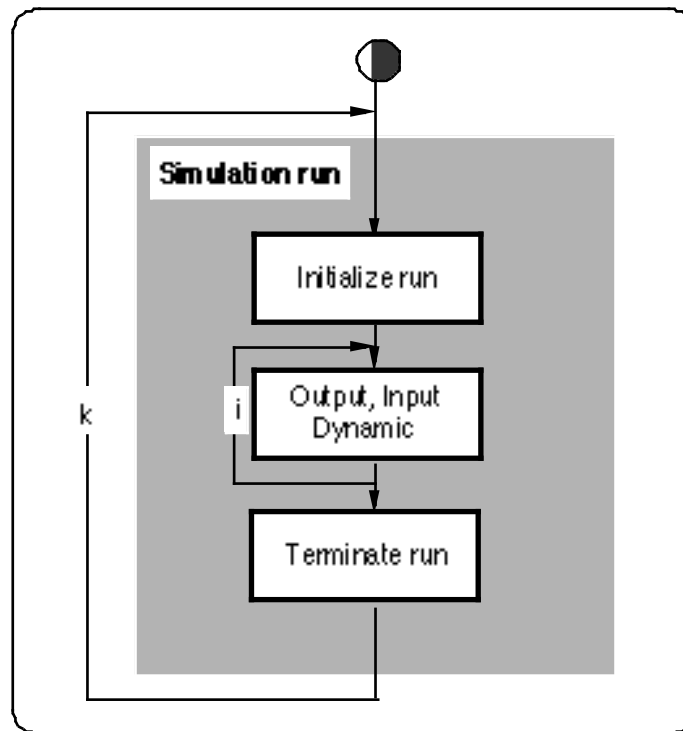
Simulations may be repeated with the same set of models as many times the simulationist wishes, but otherwise there are no relationships to other simulation tasks within the same or different simulation sessions. In particular ModelWorks does not support any communication of data from a simulation session to another one except for the simulation results contained in the stash file. Neither does the current version of ModelWorks support the direct reading of the stash file. However, it is possible to construct a particular model which reads a stash file and declares the models and model objects needed for a post-simulation analysis. The post-simulation analysis session of the RAMSES shell provides such a mechanism. ModelWorks writes data onto the stash file according to a syntax particularly designed for this purpose.

There hold certain relationships among the tasks which are performed by ModelWorks during a simulation session (Fig. T18, T19, and T21). Tasks such as elementary or structured simulation runs or resets can be executed in any order, some tasks such as simulation runs are nested (compare Figs. T19 and T21) and thus belong to a particular, hierarchical level: The simulation session represents the topmost level of all simulation tasks (Fig. T18), the next lower level is the experiment or structured simulation run (Fig. T21), on the next lower level resides the elementary simulation run (Fig. T19), and on the lowest level the integration step (Fig. T22).

#### 5.1.3.a Elementary simulation run

An elementary simulation run can be accessed by the simulationist directly without going through the experiment level. Otherwise this level is the next level below the level of the structured simulation (this level could also be understood as a structured simulation with  $k=1$ ; compare Figs. T19 and T21). ModelWorks supports this level by requiring the modeller to specify for each model an *initialize* and *terminate* procedure. The organization of an elementary simulation run is shown in Fig. T19 and in more details in Fig. T20. The mechanism to execute in -

teractively an elementary simulation run is provided by ModelWorks standard user interface by the menu command *Start run* under menu *Solve* and has not to be programmed by the modeller. Yet, choosing this menu command has exactly the same effect as the execution of procedure *SimRun* from module *SimMaster*.



**Fig. T19** : Flow chart of the elementary simulation run consisting of the run initialization (procedure *initialize*), the section dynamic (*output, input, dynamic*), and the run termination (*terminate*). The dynamic section is executed an arbitrary number of times  $i$ , which depends on the chosen time step and the simulation time.

Normally ModelWorks will execute for every model the *initialize* procedures once at the begin and the *Terminate* procedures once at the end of the simulation run. If a model is declared or removed during a running simulation, these procedures are called at declaration resp. removal time (for exact moments of execution see Tab. T4). Note that the execution of the *initialize* procedures happens at a moment when ModelWorks has already assigned the initial values to all state variables. This design makes it possible to overwrite the values assigned by ModelWorks with other values or to use these values for calculations. A typical use of *initialize* and *terminate* procedures is the opening and closing of a file at the begin respectively at the end of a simulation run in order to write simulation results onto a file different from the stash file.

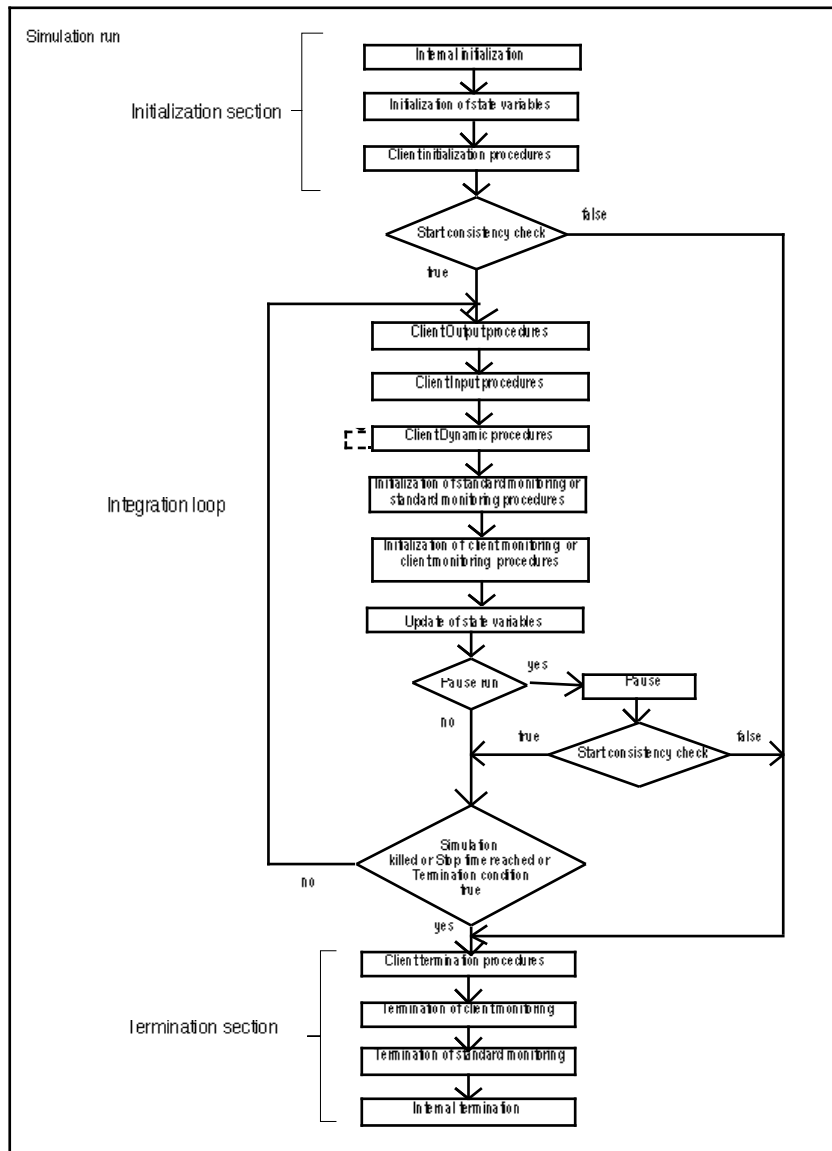
Note also that in contrast to the procedure *initSimEnv* passed to ModelWorks as actual argument in the call to procedure *RunSimEnvironment* (it is called only once), the procedures *initialize* and *terminate* may be called many, i.e.  $k$ , times during a simulation session (Fig. T19). The actual number depends on how many times the simulationist starts a simulation run directly (or via an experiment, see below) and is not known to the modeller.

Whenever ModelWorks calls client procedures such as procedures *initialize* or *terminate* from several models, the calling sequence is the same as the declaration order of the owning models.

#### 5.1.3.b Structured simulation (Experiment)

A structured simulation or experiment can be launched by the simulationist from within the standard user interface by choosing the menu command *Start experiment* under menu *Solve*.

Exactly the same result is obtained by a call to procedure *SimExperiment* from module *SimMaster*. The experiment level is the next level below that of the simulation session (Fig. T21). A structured simulation works similar to an elementary simulation run but differs slightly in the following aspects: Basically it is a procedure programmed by the modeller; typically it calls several elementary simulation runs by calling the procedure *SimRun* from module *SimMaster*. Since it is implemented as a client procedure, where the modeller has anyway already full control, ModelWorks offers no specific support for initialization and termination procedures for experiments (Fig. T21).



**Fig. T20** : Flow chart of an elementary simulation run as performed by ModelWorks. A run consists of the three basic steps: initialization, integration loop, and termination. Each step calls model specific procedures (s.a. Fig. T19).

Structured simulations are optional and have to be installed first by the modeller via the client interface before they can be executed by the simulationist from within the standard user interface. The corresponding menu command is only enabled if an experiment has actually been declared. If several subprogram levels are stacked on top of each other (s.a. section *Multiple activations of the standard user interface*), each level can install its specific experiment. The standard user interface supports the separate execution of each level's experiment by installing for each level a separate menu command.

The simulationist can execute experiments an arbitrary number of times  $n$  (Fig. 21). A structured simulation calls elementary simulation runs  $k$  times, i.e. structured simulations are only of some interest if  $k > 1$ . The total number of simulation runs then becomes  $k \cdot n$ .

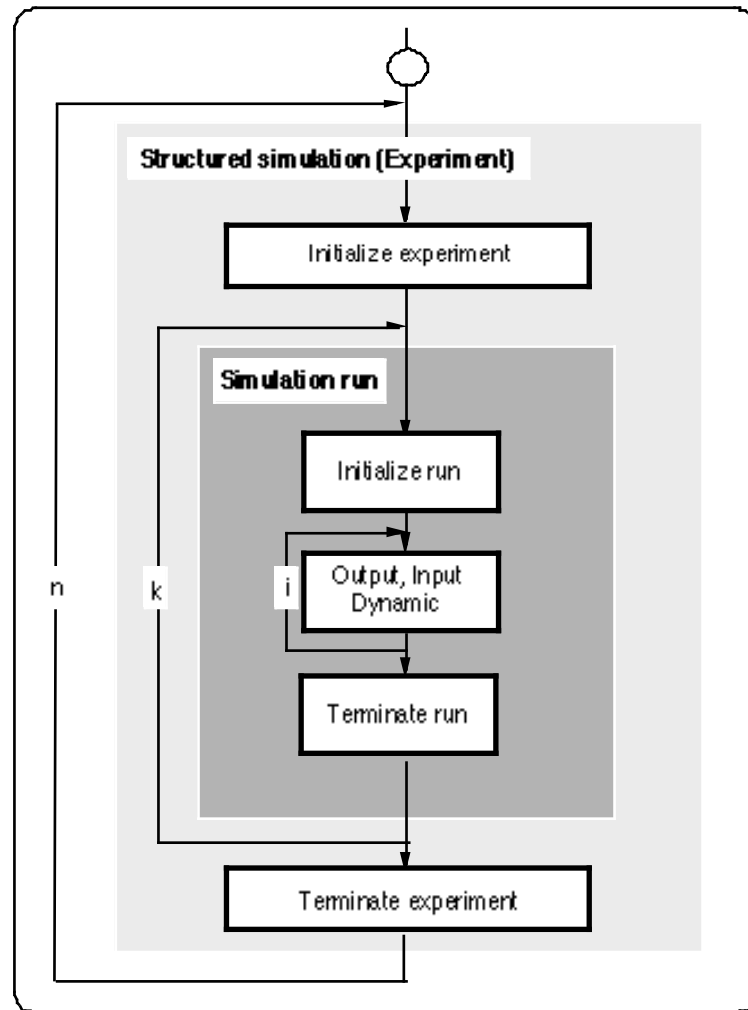


Fig. T21 : Flow chart of a ModelWorks structured simulation or experiment: From within the standard user interface the simulationist may execute an arbitrary number  $n$  of structured simulation runs. A structured simulation (experiment) consists itself again of a fixed or also a variable number  $k$  of elementary simulation runs, i.e. calls to procedure *SimRun* as programmed by the modeller (s.a. Fig. T19).

The main state of ModelWorks during a structured simulation is always *Simulating* and the substate is always different from *noSubState* (Fig. T16). Normally the environment switches only between the two substates *No run* and *Running*. The substate *No run* (outside execution of procedure *SimRun*) resembles the main state *No simulation* and allows to modify most data and settings freely. The substate *Running* represents the actual state "*Simulating*" (during execution of procedure *SimRun*) and offers a somewhat restricted modifying access to the simulation environment's data and model base (see section *Manipulating the model base at run-time*).

If an experiment is stopped by the simulationist e.g. by choosing from within the standard user interface the menu command *Solve/Stop (Kill) experiment* or by calling procedure *StopExperiment* ModelWorks reaches the substate *Stopped*. Not only the currently running elementary simulation is terminated but also all subsequently eventually following runs are "skipped". ModelWorks accomplishes this by emptying the body of the procedure *SimRun* from module *Sim-*

*Master*, hereby avoiding the use of hardware dependent interrupts. This means, ModelWorks does not actually interrupt the experiment procedure, but allows it to reach its end as programmed by the modeller. The latter should make sure that this procedure may terminate even if the body of procedure *SimRun* does no longer execute any statements. For further details see subchapter *Programming Structured Simulations (Experiments)*

### 5.1.3.c Integration respectively time step

The integration or time step resides at the lowest level of all simulation tasks. ModelWorks supports this level by requiring the modeller to specify for each model an *input*, *output*, and *dynamic* client procedure. ModelWorks will execute for every model the *output*, *input*, and *dynamic* procedures during every integration step at least once. Only *dynamic* may be called from once up to times the order of the model's integration method during a single integration step. Note also that in contrast to the procedures *initialize* and *terminate* (which are called only once per simulation run), the procedures *input*, *output*, and *dynamic* are called many, i.e.  $i$  times during an elementary simulation run (Fig. T19 and T20).

In the integration loop, user commands, such as pausing or stopping the simulation, are processed first. This enables an interactive control of the simulation. After that, the client procedures of the models are called. Their calling sequence guarantees a correct coupling of more than one model, independently of their installation order (see also chapter *Model formalisms*). The calculation order which meets all these requirements is shown in Fig. T22:

First, the *output* procedures of all models, then all *input* procedures are called. Thereafter, the numerical integration is performed. Depending on the integration algorithm, the procedure *dynamic* will be called once or several times for the evaluation of the derivatives or new states. Note, it is completely left to the modeller's responsibility to compute the derivatives or new states correctly. In particular ModelWorks offers no sorting of statements. This advantage of this method is that ModelWorks allows to compute derivatives or new states in any conceivable way. Every sub model is integrated as an independent unit. Therefore it is possible to integrate different submodels with different integration algorithms. This can be of interest if some models are numerically less stable than others or to solve stiff systems.

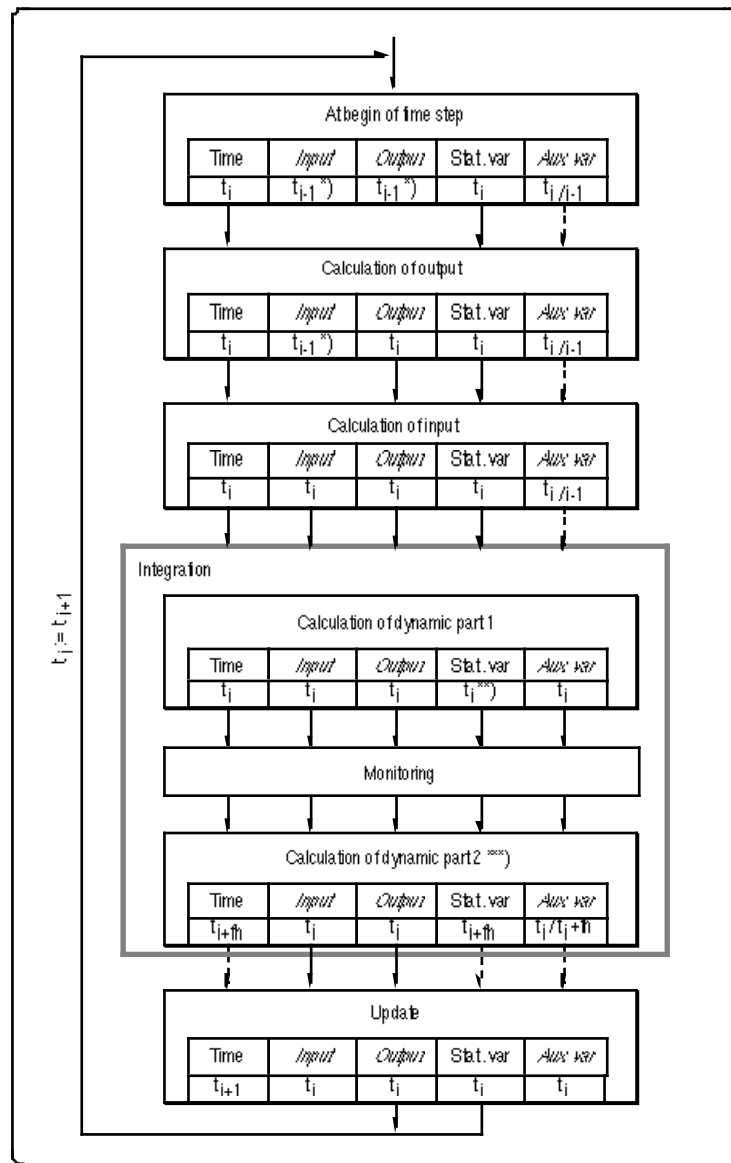
Once all dynamic client procedures, i.e. *output*, *input*, and *dynamic*, have at least been called once, all model variables, i.e. input, output, state, plus auxiliary variables, are defined and have a correct value valid at the point  $t_i$ . This is the moment ModelWorks does the monitoring (Fig. T22), i.e. the current value for any monitorable value is written onto the stash file, tabulated in the table, or drawn into the graph if the corresponding kind of monitoring is activated for the particular variable. The monitoring is followed by additional calls, now only of the client procedure *dynamic*, in case of higher order integration methods used to solve continuous time models. Discrete time or single step integration methods will skip this step.

Finally all state variables and the independent variable (time  $t$ ) are updated to their new values at time  $t := t_{i+1} = t+h$ . Afterwards the termination criteria is evaluated. The simulation will be terminated if either the simulation run was stopped (killed) by the simulationist, if procedure *StopRun* resp. *StopExperiment* from module *SimMaster* has been called, the termination condition from the model definition program has returned true, or the simulation stop time has been reached. Depending on the result, the simulation continues or stops, which will result in a state transition from the state *Simulating* into the state *No model* respectively *No simulation* or from the substate *Running* into the substate *No run* resp. *Stopped* (s.a. Fig. T15, T16, and T2New4).

Discrete time models are treated analogously to the continuous time models. For declaration, display, and monitoring, ModelWorks treats basically both the same, except that the discrete time submodels are «integrated» with a different integration method, i.e. *discreteTime* (see enumeration type *IntegrationMethod* from module *SimBase*): In the declaration of the state variable, the new value of a discrete-time difference equation replaces its continuous-time counterpart, i.e. the derivative (see procedure *DeclSV* from module *SimBase*). However, be aware,



that the body of the corresponding procedure *dynamic* needs also to be formulated accordingly, since expressions defining a system of continuous-time differential equations are fundamentally different from expressions defining a system of difference equations (s.a. chapter *Model Formalisms*); yet it is fully left to the modeller's responsibility to program them properly.



**Fig. T22** : Calculation of time, input, output, state, and auxiliary variables during an integration step. The calculation order guarantees that all calculations are based on valid values which have been calculated in a previous step. The arrows indicate which values have become valid. At the begin of the simulation run, only time and the state variables are available for  $t_i$ . These values are used to calculate the output variables for  $t_i$ . Next, the input variables can be calculated, since they depend on the previously calculated outputs. Next the numerical integration respectively the new state variables for time  $t_{i+1}$  are computed. Finally all state variables are assigned (updated) to the new values for  $t_i$  (s.a. Fig. T12).

- \*) Not defined the first time the integration loop is entered
- \*\*\*) Value for  $t_{i+1}$  is calculated, but not yet assigned to state variable field
- \*\*\*\*) Only calculated if an integration method of higher order used ( $f \in (0,1)$ , e.g.  $f=0.5$  for Runge-Kutta 4th order)

The situation is more complicated in case of continuous with discrete time mixed simulations; since the discrete time step might be several times larger than the integration step needed for the continuous time submodels, it is obvious that the two types of models must be treated separately. Typically the discrete time submodels will then not be called as often as the continuous time ones and the *output* of the discrete time submodels will have to be computed at the begin, the *input plus dynamic* only at the end of the coincidence interval.

Time steps usually vary, even if a fixed step integration method is used<sup>13</sup>. This is because the time steps depend not only on the integration step, but also on the coincidence interval and/or the monitoring interval. ModelWorks is computing values exactly at any of the time points given by the current values of these global simulation parameters. E.g. a fixed integration step of  $h = 0.75$  and a monitoring interval  $h_m = 1.0$  will result in the following actual sequence of integration step lengths: 0.75, 0.25, 0.5, 0.5, 0.25, 0.75 ...

Symbol	Meaning of variable	Action by ModelWorks
State variables		
x	State variable overwrite with initial value during declaration overwrite with initial value at begin of run	write write
integration	(continuous time only) update with new value obtained via integration	read and write write
$\dot{x}/x(k+1)$	Derivative (continuous time)/ new value (discrete time) integration	read
$x_0$	Initial value <sup>14</sup> overwrite with default during declaration overwrite with default while resetting initial values editing of value via IO-window	write write read and write
Model parameters		
p	Model parameter overwrite with default during declaration overwrite with default while resetting parameters editing of value via IO-window	write write read and write
Monitorable variables		
o	Monitorable variable monitoring Stash filing, Tabulation, Graphing, or curve attributes <sup>15</sup> overwrite with default during declaration overwrite with default while resetting parameters editing of value via IO-window	read write write read and write

Tab. T2: Actions of ModelWorks performed on model objects installed in the model base.

<sup>13</sup>However, in the current implementation of ModelWorks, at a particular time the same integration step length is used for all models to guarantee a co-ordinated data transfer between submodels.

<sup>14</sup>variable belongs to ModelWorks not to the client's model definition program

<sup>15</sup>see previous footnote

#### 5.1.3.d Model objects and the run-time system

Any model object is recognized by ModelWorks only if it has been declared, typically during execution of the declaration *declModelObjects* procedure passed as actual argument to *DeclM*.

Otherwise the models and model objects belong fully to the model definition program. Thanks to this method the modeller may define and access these variables in whichever way she likes, e.g. by using state variables as part of an array or a record data structure. Note however, since ModelWorks maintains the model objects also, e.g. during numerical integration (Tab. T2) it uses the following access mechanism: While executing declarations such as *DeclSV* ModelWorks stores the addresses of the declared variables. Later during simulations, ModelWorks will access the model objects and their associated variables (Tutorial Fig. T2) for reading or writing (Tab. T2) by assuming that these objects still exist. Therefore the modeller must be careful to ensure that any model object continues to exist within the model definition program as long as it remains declared. A model object such as a state variable remains declared within the simulation environment until it is removed, e.g. by a call to *RemoveSV*, or the program is terminated<sup>16</sup>. Thus, any attempt to start a simulation will cause the simulation environment to try to access all currently declared models and model objects. In case the model definition program should have discarded only locally any model or model object, an attempt to run a simulation, will produce unpredictable results or even crash the program. In particular does this imply that models or model objects such as state variables must not be declared as variables local to a Modula-2 procedure, unless the procedure calls at its end the corresponding remove procedure for any locally declared model or model object. It is recommended to declare models and model objects always globally (s.a. part I *Tutorial*, chapter *Getting Started with Modelling*).

Models are always calculated in parallel, regardless of the presence of any coupling among them, i.e. the calculation order of the client procedures is: first all *output* of all submodels, second all *input* of all submodels etc. (Fig. T22). The actual sequence of the computations of a particular kind of client procedures, e.g. the sequence with which ModelWorks calls the procedures *dynamic*, is given by the sequence of the declarations of their owning models.

In the current version of ModelWorks all types of auxiliary variables, i.e. input, output, and internal auxiliary variables, do not appear explicitly in the ModelWorks concept. Inputs and outputs were formally defined in chapter *Theory*, and the model definition program is responsible for a correct handling of them by the client procedures *input* respectively *output*. The remaining internal auxiliary variables may be used freely within the scope of the owning model definition program. Note, in contrast to state variables ModelWorks does neither initialize nor otherwise maintain auxiliary variables. Often auxiliary variables are computed in the procedure *dynamic*; hence, they will only hold a correct value if the procedure *dynamic* has at least been called once, i.e. only after a simulation run has already begun (s.a. Fig. T18, T20, and T22); in particular note, attempts to use them in the procedure *initialize* may lead to wrong results, if their values are only defined in the procedure *dynamic*.

#### 5.1.3.e Client procedures and the simulation environment

The modeller or client normally installs so-called client procedures into the simulation environment, which will then be called by ModelWorks at various occasions (Tab. T3). E.g. the run-time system calls repeatedly such client procedures, e.g. the procedure *initialize* to initialize a run or procedure *dynamic*, which contains the differential equations. Client procedures are installed into ModelWorks via installing procedures such as *DeclM* or *SetDefltM*.

Most of the client procedures are either called directly or indirectly from within the standard user interface or by the client interface while executing particular procedures, called callee (Tab. T3). Note first that some of the installing procedures may even function as callee and secondly that several menu commands of the standard user interface call often just a callee; e.g.

---

<sup>16</sup>Program termination will cause an implicit removal of all models and model objects owned by the program.

the menu command *Solve/Start run* calls procedure *SimRun*. It is then *SimRun* which will call client procedures such as *initialize* or *dynamic* (Tab. T3).

Called client proc.	Callee Standard user in- terface	<i>RunSim Environ- ment</i>	<i>SimRun</i>	<i>SimExp- eriment</i>	<i>DeclM</i>	<i>Re- moveM</i>	<i>Tile- or Stack- Windows</i>	Client procedure installed by
<i>initSimEnv</i>		x						<i>RunSimEnvironment</i>
<i>declModelObjeccts</i>					x			<i>DeclM</i>
<i>defineSimEnv</i>	x - D	x						<i>InstallDefSimEnv</i>
<i>startAllowed</i>	i - RE	i - SUI	x					<i>InstallStart- Consistency</i>
<i>initialize</i>	i - RE	i - SUI	x		x <sup>17</sup>			<i>DeclM, SetDefltM</i>
<i>output, input, dynamic</i>	i - RE	i - SUI	x					<i>DeclM, SetDefltM</i>
<i>terminate</i>	i - RE	i - SUI	x			x <sup>18</sup>		<i>DeclM, SetDefltM</i>
<i>about</i>	x <sup>19</sup>	i - SUI						<i>DeclM, SetDefltM</i>
<i>initClientMon, doClientMon, termClientMon</i>	i - RE	i - SUI	x					<i>InstallClient- Monitoring</i>
<i>isAtEnd</i>	i - RE	i - SUI	x					<i>InstallTerminate- Condition</i>
<i>doExperiment</i>	i - E	i - SUI		x				<i>InstallExperiment</i>
<i>doAtStateChange</i>	i - S	i - SUI	x	x	x <sup>20</sup>	x <sup>21</sup>		<i>InstallStateChange- Signaling</i>
<i>doAtTile or doAtStack</i>	x - W	i - SUI					x	<i>InstallTile- or InstallStackWin- dowsHandler</i>
<i>DialogMachine- Task</i> <sup>22</sup>	i - RE	i - RunDM	x					<sup>23</sup> "Dialog Machine"

Tab. T3: Relationships between calling callee and called client procedures: Each row lists a type of client procedure which can be installed into ModelWorks by the procedures listed in the rightmost column and their callees listed in the topmost row. Legend: x - client procedure directly called by callee; i - client procedure only indirectly called by callee, e.g. by choosing a menu command such as *Solve/Start run*; RunDM - *RunDialogMachine* from *DMMaster*; SUI - *Standard User Interface*. Menu commands: D - *Settings/Define simulation environment*; RE - *Solve/Start run* or *experiment*; RER - *Solve/Start run* or *experiment* and *Solve/Resume run* or *experiment*; E - *Solve/Start experiment*; S - all commands of menu *Solve*; W - *Windows/Stack windows* or *Tile windows*. For exact timing of calls see Tab. T4, Figs. T18 till T22.

<sup>17</sup> *DeclM* causes for the newly declared model also a call to its procedure *initialize* in case of state *Simulating* and substate *noSubState* resp. *Running*, or state *Pause* (Tab. 4).

<sup>18</sup> *RemoveM* causes for the model to be removed also a call to its procedure *terminate* in case of state *Simulating* and substate *noSubState* resp. *Running*, or state *Pause* (Tab. 4).

<sup>19</sup> Only callable via palette button in the IO-window *Models*. Note, the use of this window does not require the standard user interface.

<sup>20</sup> In case the simulation environment has been in state *No model* before calling *DeclM*.

<sup>21</sup> If *RemoveM* removes the last model such that the simulation environment enters state *No model*.

<sup>22</sup> *DialogMachineTask* from module *DMMaster* is not really a client procedure, however since it dispatches control to all procedures which have been installed via the "Dialog Machine", it is also a mean to call indirectly client procedures.

Normally it is ModelWorks which keeps the program control, but each time it calls a client procedure it relinquishes the control. Thus the program control is passed back and forth between the modeller and ModelWorks. If the modeller calls a non-callee procedure she will not gain control back until that procedure has been fully completed. However, if she calls a callee, she will regain control once or even several times. It is then important that she relinquishes control from a client procedure as soon as possible, so that the callee can actually continue and finish.

### 5.1.3.f Manipulating the model base at run-time

All functions offered by the client interface may also be used while a simulation is running. This is particularly important when programming structured simulation runs or experiments (s.a. sub chapter *Programming Structured Simulations (Experiments)*). However, since some procedures affect values currently in use or already used, such as the simulation start time  $t_0$ , a few additional rules are needed to handle firstly the changes of the structure of the model base, i.e. the declaration or removal of models and model objects, and secondly the changes of current values and defaults.

The following rules apply only while procedure *SimRun* is executing, i.e. in the states *Simulating* or *Pause* (Fig. T15), whereby the main state *Simulating* may or may not be subdivided into substates (elementary vs. structured simulation run). In case of structured simulation runs or experiments the restrictions apply only to the substate *Running* (Fig. T16):

If models or model objects are declared or removed in a client procedure such as procedure *input*, the overall system structure changes while procedure *SimRun* is still executing (s.a. section *Client procedures and the simulation environment*). In this case the basic principle is not to disturb the on-going integration for the current time step and to allow for its completion as if the system would not have been modified (Tab. T4). Thus any integration result available at the end of a single integration step is that which would have been produced by the system which has been present at the begin of the step. Only in the next integration step will the system modifications become effective (Tab. T4).

More precisely, if a model is declared during an elementary simulation run, e.g. in the procedure *input* of an already existing model, the new model and all its objects which are declared in its procedure *declModelObjects*, are instantiated immediately (Tab. T4). Hence, the modeller may subsequently operate freely on them upon returning from procedure *DeclM*. However note, according to above mentioned principle, neither the model's state variables nor its procedures *input*, *output*, or *dynamic* will be involved in the integration of the current time step. Moreover note also, that the procedure *initialize*, which has been passed as actual argument to *DeclM*, will be executed as a direct consequence of the call to *DeclM* while executing *SimRun*. Procedure *initialize* will be called individually for any newly declared model, despite the fact that the run has already been started. Similarly, during a simulation, the terminate procedure *terminate* of a model to be removed is called immediately before the model is actually removed, again as a direct consequence of the call to procedure *RemoveM* (Tab. T4). Since the procedures *initialize* or *terminate* may contain additional calls to the procedures *DeclM* or *RemoveM*, this will lead to further calls of the procedures *initialize* or *terminate* from the involved new or obsolete models. ModelWorks repeats this process until there remain no more procedures *initialize* or *terminate* to be executed. It is left fully to the modeller's responsibility to ensure that this condition is always met.

---

<sup>23</sup>Can not be installed by the client, but only imported from *DMMaster*, and is always in use by any "Dialog Machine" program, i.e. any Modula-2 program like ModelWorks which imports from the "Dialog Machine".

<b>Moment of execution of the callee</b>  (Callee)  Called procedure	<b>During initialization and In-between integration steps</b>  ( <i>initialize</i> , <i>DMTask</i> , <i>initialize</i> , or <i>terminate</i> )	<b>During integration step</b>  ( <i>output</i> , <i>input</i> , <i>dynamic</i> , <i>initClientMon</i> , or <i>doClientMon</i> )	<b>During termination</b>  ( <i>terminate</i> or <i>termClientMon</i> )	<b>Otherwise</b>  ( <i>initSimEnv</i> , <i>defineSimEnv</i> , <i>declModelObjects</i> , <i>doExperiment</i> , <i>doAtStateChange</i> , <i>DMTask</i> )
<b><i>DeclM</i></b>	immediate	immediate	immediate	immediate
<i>declModelObjects</i>	immediate	immediate	immediate	immediate
<i>initialize</i>	after <i>initialize</i> of all not yet initialized models	after completion of current integration step	deferred to initialization of next run	deferred to initialization of next run
<i>output</i> , <i>input</i> , <i>dynamic</i>	deferred to subsequent integration steps	deferred to subsequent integration steps <sup>24</sup>	deferred to integration of next run	deferred to integration of next run
<i>terminate</i>	deferred to termination of run	deferred to termination of run	deferred to termination of next run	deferred to termination of next run
<b><i>RemoveM</i></b> <sup>25</sup>	after <i>terminate</i> of all models to be removed	after completion of current integration step	after <i>terminate</i> of all models	immediate
<i>initialize</i>	–	–	–	–
<i>output</i> , <i>input</i> , <i>dynamic</i>	–	(if involved in ongoing integration step) <sup>26</sup>	–	–
<i>terminate</i>	after <i>initialize</i> of all models	after completion of current integration step	–	–

Tab. T4: Effects of calls to *DeclM* or *RemoveM* during a simulation run: The table lists the indirect calls of the client procedures *initialize*, *input*, *output*, *dynamic*, *terminate* and *declModelObjects* by ModelWorks plus some effects such as model instantiations or some procedure calls. Such calls partly take place while *DeclM* resp. *RemoveM* are still executing (immediate), partly they happen at a later moment (after), partly their execution is even deferred more to take place at the ordinary time of execution (deferred), e.g. the initialization of the next run. The tabulated sequences warrant that structured model systems can be solved consistently at all times, even if the modeller changes the system structure in the middle of a simulation.

Via the client interface or a non-standard user interface current values can be manipulated freely, however note the following particularities:

- The current substate of the simulation environment determines the exact effect of the called *Set<sub>xyz</sub>* procedures<sup>27</sup>. The following rules hold while *SimRun* is executing:
  - Depending on the current simulation time  $t$  resp.  $k$  an attempt to change the simulation start time  $t_0$  resp.  $\kappa_0$  or stop time  $t_{end}$  resp.  $\kappa_f$  may result in a different

<sup>24</sup>The current integration step is completed without considering the newly declared model; hence these procedures will only be called during the next integration step.

<sup>25</sup>Removal of a model implies the removing (including memory release) of all declared objects belonging to the model.

<sup>26</sup>In case of a mixed continuous and discrete time structured model the discrete time model's procedures *output*, *input*, *dynamic* are not necessarily called during every integration step.

<sup>27</sup>Since a reset is defined as *GetDeflt<sub>xyz</sub>(v)* followed by a *Set<sub>xyz</sub>(v)*, i.e. the default value  $v$  is made the current value, the mentioned particularities apply also to the corresponding *Reset<sub>xyz</sub>* procedures.

setting of the time domain than this would be the case in another state. The call *SetGlobSimPars*( $t_0, t_{end}, \dots$ ) is handled as if the following call *SetGlobSimPars*( $\text{MIN}(t_0, t), \text{MAX}(t_{end}, t), \dots$ ) would have been made (where *MIN* and *MAX* are function procedures, which return the minimum respective maximum of their actual parameters). This allows either to prolong, shorten or even to end the simulation if  $t_{end} \leq t$ , but not to jump to a new time domain when  $t_0 > t$  (procedures *SetGlobSimPars*, *SetSimTime*, *ResetGlobSimPars*, *ResetAll*).

- Changes to the current values will never affect the on-going integration step but will only have an effect in the subsequent integration step. In particular this means that the effect of the procedures *SetM*, *SetDefltM*, *SetSV*, *SetP*, and *SetMV* plus all corresponding reset procedures such as *ResetAllIntegrationMethods* will be delayed until the current integration step has been fully completed. This behaviour matches the rules which apply to changes in the structure of the system, i.e. declaration of removal of models or model objects (Tab. 4).
- Changing current values of monitorable variables, i.e. their minimum and maximum for the scaling and their settings (stash filing, tabulation, and graphing) (procedures *SetMV*, *ResetAllStashFiling*, *ResetAllTabulation*, *ResetAllGraphing*, *ResetAllScaling*) may have particular effects:
  - a) Changes to the attribute filing, i.e. adding or removing a monitorable variable from resp. to the current set of monitorable variables to be written onto the stash file (*SetMV*, *ResetAllStashFiling*), leads to a so-called subrun break, i.e. the begin of a new subrun. Since the stash file is written according to a formally defined syntax, which requires to write the simulation results always in form of a matrix (i.e. each row must contain the same number of values, s.a. section *Monitoring*), altering the number of columns in the middle of a run is disallowed. However, if a run is subdivided into subruns, where each adopts the syntax of a full run and contains the results only in form of a matrix with a fixed number of columns, ModelWorks can again support changes to the attribute filing.
  - b) Changes to the attributes tabulation or graphing (*SetMV*, *ResetAllTabulation*, *ResetAllGraphing*) may lead to a full redrawing of the table or the graph, hereby losing all previously monitored results. Note however, curve attributes can be changed without disturbing already existing monitoring results and it can be useful for the marking of different types of measurements in the middle of a simulation run by changing the colour of a curve on the fly (procedures *SetCurveAttrForMV*, *ResetAllCurveAttributes*).
- The following rule holds while *SimRun* or *SimExperiment* is executing:
  - Changes to the stash file affect the currently opened stash file, e.g. by renaming it. Note that this results in a slightly different behaviour opposed to calls in the states *No model* or *No simulation*. In the latter case *SetStashFileName* would produce a stash file with a different name and leave an eventually already existing stash file with the old name untouched. Use *SwitchStashFile* to force the closing of an already opened stash file and the opening of a new one. However, if there is currently no stash file open, *SwitchStashFile* will have the same effect as a call to *SetStashFileName* (procedures *SetStashFileName*, *SetStashFileType*, *ResetStashFile*, *SwitchStashFile*).
- The following hints or recommendations are valid in general:
  - Current values should not be changed directly, e.g. the modeller must not assign a new value to a state variable in the middle of an integration step, despite the fact that the state variable belongs fully to her model definition program. Instead the modeller is urged to call always the safe procedures *SetSV* or *SetP*. This is because only ModelWorks run-time system can assign

a new value to a state variable at the right moment, i.e. when the assignment won't disturb any other calculations, e.g. the integration of an other model which uses an output depending on the state variable to be changed (s.a. section *Integration respectively time step*).

- Some procedures have actually the desired effect, i.e. changing the current values, yet this will have no immediate effect, but only a delayed one. For instance the procedures *SetProjDescrs*, *SetTabFuncRecording*, or *SetIndepVarIdent* and the corresponding reset procedures all cause changes to some current values, but this has no visible effect until the next time these new values are actually used, often only in the next simulation run. For instance the project description is written to the stash file merely once at its begin, and it can't be corrected once written. Or if the simulation has already started, the start time  $t_0$  can not be altered anymore and the change of  $t_0$  does not become effective until the next simulation run is actually started.

All changes affecting default values (procedures *SetDeflt<sub>xyz</sub>*) will not become effective until a corresponding reset is actually executed.

### 5.1.3.g Monitoring

ModelWorks displays simulation results only via a monitoring concept. It is based on the so-called monitorable variables, which are declared in the model definition program. Once a variable has been declared as monitorable variable via the client interface, it can be selected interactively in the corresponding IO-window in order to activate a certain kind of monitoring. Any variable can be monitored as long as it is a real number. In this way the simulationist may observe the values of any variable, might it be an input, state, auxiliary or output variable. Monitorable variables might be understood as nothing else than probes attached to any information flow circulating within the model system. They measure anytime anywhere any quantity without disturbing the dynamics of the system, regardless whether this variable is an internal state, auxiliary, input or output variable. This is different from conventions in systems theory, where often additional outputs must be first introduced to monitor internal variables.

Since continuous time measurement would result in an exorbitant amount of data, monitoring is possible only at discrete points in time, the monitoring times. The time interval between monitoring times is global, i.e. the same for all models and all kinds of monitoring, the so-called monitoring interval  $h_m$ . Although  $h_m$  is normally kept constant, via the client interface the modeller may change this global simulation parameter freely. ModelWorks computes values exactly at the time points  $t_m$  for which monitoring is requested. In case of a constant  $h_m$  the monitoring occurs at  $t_m = t_0 + i \cdot h_m$  where ( $t_0$  - simulation start time;  $i = 0, 1, 2, \dots, i_{\max}$ ) plus an additional time for  $t_{\text{end}}$  in case that  $i_{\max} \cdot h_m \neq t_{\text{end}}$ . The monitoring takes place during an integration step only (Fig. T22). For instance if  $t_m = t_{\text{end}}$  even the very last monitoring occurs during the calculation of an additional last integration step, which is not fully completed, i.e. not updated, in order to retain the state of the system at  $t_{\text{end}}$ .

Standard monitoring of ModelWorks is available in one or any combination of the following three kinds: The simulation results, may be written and stored on a so-called stash file for later usage, tabulated as numbers in a table, or shown as curves in a graph.

During simulations, i.e. in state *Simulating*, unless disabled three windows, the graph, the table window, and the time window, are automatically opened and brought to the front to display the simulation results and the current simulation time.

At the begin respectively end of each elementary simulation run, the time window will always appear respectively disappear automatically in the upper right corner of the main screen. This is also the case in structured simulations or experiments. To facilitate the orientation of the simulationist during experiments, ModelWorks displays in the time window not only the current si-



mulation time  $t$ , but also the number  $k$  (Fig. T21) of the current simulation run (format:  $k: t$ ).  $k$  can also be obtained by calling function procedure *CurrentSimNr* from *SimMaster*.

The stash file can store an arbitrary amount of information about the current status of the model base, i.e. on models, model objects, and their associated current values and it is usually produced for further numerical, e.g. statistical analysis, of the simulation results or for future report generation to document simulation runs in any detail. The size to which the stash file may grow is limited only by the available disk space. The file is written in a formally defined syntax and contains several types of information, partly always included and partly included only selectively by means of the so-called recording flags. The content consists of:

- General information on the simulation session consisting of a) the ModelWorks version, type of computer, and the date and time of the session's begin, b) date and time of begin and end of simulation runs, c) project title, remarks and footer, d) date and time when the file was closed. This information is always written.
- Values of all global simulation parameters (Start and stop time of simulation ( $t_o/\kappa_o$ ,  $t_{end}/\kappa_f$ ), integration step ( $h$ ) respectively maximum integration step ( $h_{max}$ ) plus maximum local relative error ( $e_r$ ), discrete time step respectively coincidence interval ( $c$ ), and monitoring interval ( $h_m$ )). The parameters actually written on the stash file depend on the type of models currently present: continuous time only, discrete time only or both types mixed as well as the used integration methods (with or without variable step length methods). This type of information is written always and in particular also repeatedly for every simulation run.
- Lists of all models and their integration methods, of all state variables and their current values, of all model parameters and their current values, of monitorable variables and their settings, curve attributes and scaling are written selectively under control of the recording flags. Note that not all monitorable variables are recorded but only those for which either the stash filing is currently set (*F/writeOnFile*) or those which are present in the graph, given that the recording flag for graph dumping is currently set.
- Lists of the parameters of all table functions declared by means of module *TabFunc* under control of the recording flag *Table functions*.
- Numerical simulation results tabulated for those monitorable variables for which the stash filing is currently set (*F/writeOnFile*).
- Messages (procedure *Message*) or changes of the current values of models or model objects (procedures with identifiers commencing with *Setxyz*, e.g. *SetP*).
- Graphical simulation results (encoded, only machine readable), dumped under control of the recording flag *Graph*.
- A table of byte and line numbers at which an individual run starts and ends for speeding-up the reading from the stash file during a post-simulation analysis.

ModelWorks can handle only one stash file at a time. In the states *No model* and *No simulation* it is always closed to allow for the inspection of its content by the simulationist. ModelWorks automatically opens respectively closes the stash file at the begin respectively at the end of an elementary simulation run. However, this is not the case in a structured simulation experiment, where the stash file is only closed at the very end of the experiment. This allows to record the results of all elementary simulation runs involved in the experiment as a single sequence. Unless the stash file name is changed (its default name is *ModelWorks.DAT*), ModelWorks will use always the same file, i.e. if a file with the same name already exists, that file's old content will be lost and completely rewritten!

The stash file is written in a format which can be read by the user as well as scanned by a computer program (post-simulation analysis). Furthermore it is also possible to transfer the results into another program, e.g. a spreadsheet program, or into a document processing program

which understands the RTF<sup>28</sup> format. These formats are fully controlled by ModelWorks and can not be changed by the user.

At the heart of the information written to the stash file is the writing of the values of the monitorable variables for which the stash file monitoring has been set at every monitoring time  $t_m$ . The format is such that these results can be transferred directly, for instance via the clipboard, into another application: Only horizontal tab characters  $\tau$  (ASCII ht = octal 11C) separate the values and all values at a particular monitoring time are written on the same line terminated with an end of line symbol  $\rho$  ( $\rho$  = string `\par` followed by ASCII cr = octal 15C). E.g.:

```
(*"t      "  \tau      "Ident var 1"  \tau      "Ident var 2"  \tau *)  \tau \rho
0.000000  \tau      1.0000000  \tau      0.9025031  \tau      \tau \rho
0.200000  \tau      1.1764115  \tau      0.6883310  \tau      \tau \rho
0.400000  \tau      1.2954322  \tau      0.4211738  \tau      \tau \rho
0.600000  \tau      1.3516583  \tau      0.0882961  \tau      \tau \rho
0.800000  \tau      1.3498297  \tau      -0.3198467  \tau      \tau \rho
...
```

Normally the stash file is only opened and written if at least one monitorable variable has been requested for the recording. However, if the particular simulation environment mode (see preferences) is set appropriately, the stash file is opened during every simulation run and data are recorded according to the current settings of the recording flags.

Unless disabled the table window is used to tabulate the values of monitorable variables during a simulation. Values are written in a similar way as shown above under the stash file monitoring. Currently, only the values which fit into the window are displayed. Once the window is full, ModelWorks erases most of its content<sup>29</sup> and restarts tabulating from the top again (called a «page up»). In the current version of ModelWorks any erased values are lost and the simulation has to be repeated to display them again.

ModelWorks can display in the graph window one graph only. The graph has a linear abscissa (x-axis) with time or any monitorable variable as independent variable (allowing for state space curves), and a linear ordinate (y-axis) with a fixed scaling from [0,1]. According to the currently set minimum and maximum values for the range of interest, an arbitrary number of dependent variables (range shown in the legend), can be plotted simultaneously in the graph. An unlimited number of simulation runs can be recorded in one graph. The graph will be automatically cleared after changes of the graph definition, the global simulation parameters (e.g. if the start or stop time has been changed and time is the abscissa), or if the window is resized after a simulation. However, this behaviour may differ depending on the currently set mode of the simulation environment (preferences). An example graph is shown in Fig. T23.

The graph's size is automatically fit to the window's size. The actual graph is drawn as large as possible, which depends on the number of curves to be listed in the legend at the bottom of the window. However, if there are too many curves requested so that the legend would become too big and there would not be left a minimal space for the panel of the graph, ModelWorks will not be able to list all curves in the legend. Only the first ones will be visible, the remaining ones at the bottom of the list will be missing.

If another than the standard monitoring of ModelWorks is required, the modeller can program and install such a client monitoring by calling the procedure *InstallClientMonitoring*. Any type of monitoring will then be possible, e.g. the writing of simulation results onto a file or the

<sup>28</sup>RTF stands for Rich Text Format. It is based on ASCII characters only but contains coded formatting information and can be interpreted by many commercially marketed text processing applications.

<sup>29</sup> Actual number of rows erased depends on the currently set preferences or simulation environment modes: The number specified as *Common rows between page ups in table* defines what happens during a page up: First it specifies how many rows at the bottom are not erased but copied to the top of the next page. Second only the space below these now top rows will be used to add new rows. Hence this number specifies how many rows are common to two consecutive pages.

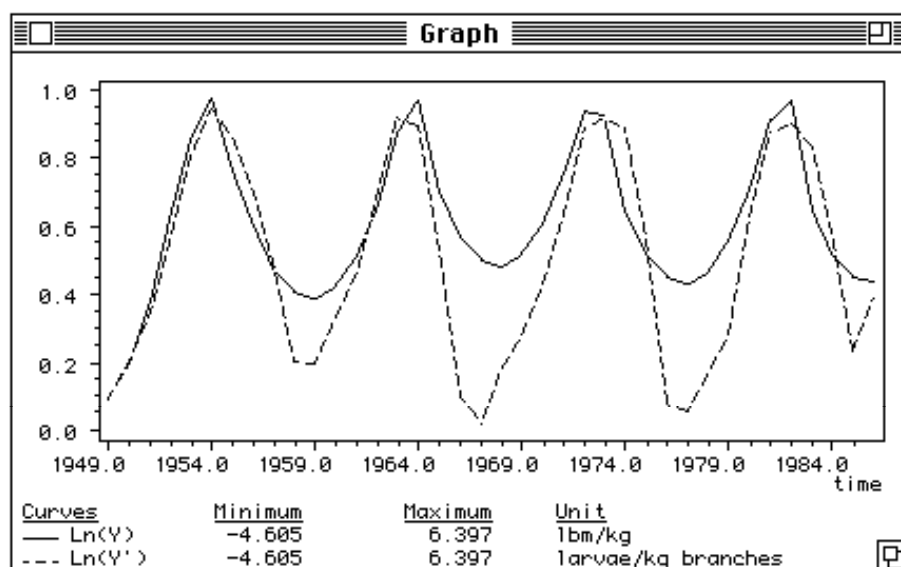


Fig. T23 : The graph window of ModelWorks (produced by the research sample model *LBM* from the *Appendix*).

drawing of animated graphical objects which move within a window according to computed positions etc. In order to accomplish such tasks, the modeller uses the "Dialog Machine", to which she has full access. Concerning values of state and other variables, the client monitoring will be done as often and at the same time as the standard monitoring. The exact sequence observed is that the client monitoring comes last, i.e. after ModelWorks has done its monitoring, so that it can also be used to customize or extend the standard monitoring by ModelWorks, e.g. by drawing tangents along a solution of a differential equation. Note however, at the end of a simulation run this sequence is reversed, i.e. the termination of the client monitoring occurs before ModelWorks terminates its monitoring; this offers the advantage that all objects such as the graph or table window, or the stash file can still be used for the client monitoring. Besides, the client monitoring is terminated only after all *terminate* procedures have been executed.

#### 5.1.4 STANDARD USER INTERFACE

The ModelWorks' standard user interface provides menus, menu commands, input-output-windows (IO-windows) and a series of entry forms which allow to edit various data.

The main purpose of the standard user interface is to allow the simulationist to issue a command to ModelWorks and to observe simulation results. She has the following options to enter commands: First an omnipresent menu bar offers a set of pull-down, pop-up, or tear-off menu commands; second some menu commands (their text is followed by "...") will open so-called entry forms offering from one to several editable fields to enter numbers or change settings via check boxes etc. Thirdly the so-called IO-windows allow to select particular models or model objects for modifications or to issue further commands. Efforts have been put into the design of menus, menu commands, button palettes, and entry forms to support the convenient use by researchers and to make the use as simple and as intuitively appealing as possible. The design follows the general purpose user interface of the "Dialog Machine", which is hardware and system software independent (FISCHLIN, 1986a,b; FISCHLIN *et al.*, 1987; FISCHLIN & SCHAUFELBERGER, 1987; MANSOUR & SCHAUFELBERGER, 1989).

The standard user interface is invoked by a call to procedure *RunSimEnvironment* from module *SimMaster*. ModelWorks installs then its menus and opens the IO-windows according to the modeller's specifications or internal predefinitions (s.a. Fig. T18). From now on and until the interactive environment is quit, all operations on the client interface will normally become visible on the user interface, e.g. as changes in the status of menu commands, updates in the I/O

windows and by display of simulation results into the table and graph windows. Once the interactive environment is quit, the client program may proceed with calls to any ModelWorks functions, the previously defined model base and the simulation environment's global settings remain unchanged and are still available until the calling program is actually terminated.

#### 5.1.4.a Multiple activations of the standard user interface

The interactive environment may be started (again) at any time from a client program, given that it is not already running on the subprogram level in which the client program resides<sup>30</sup>. This mechanism allows to stack model definition programs on top of each other, a behaviour which may be useful for research purposes. For instance it is possible to replace submodels at run time, while retaining its super model loaded; or it is possible to solve several models, which were developed independently from each other, simultaneously in order to compare their behaviour. The following program illustrates such a use of the simulation environment:

```
MODULE SimShell;
  FROM DMOpSys IMPORT GetFileDialog, CallM2SubProg, ProgStatus;      FROM DMStrings IMPORT Concatenate;
  FROM DMMenus IMPORT Menu, Command, AccessStatus, Marking, InstallMenu, InstallCommand,
    InstallAliasChar, InstallQuitCommand;
  FROM DMMaster IMPORT RunDialogMachine, CallSubProg, SubProgStatus;  IMPORT SimMaster; (*for preloading*)

  VAR theMenu: Menu; compCmd,editCmd,loadCmd: Command; pst: ProgStatus;

  PROCEDURE CompMDP; BEGIN CallM2SubProg('Compile',TRUE,pst) END CompMDP;
  PROCEDURE EditMDP; BEGIN CallM2SubProg('Edit2',TRUE,pst) END EditMDP;

  PROCEDURE LoadMDP;
    VAR path,mdp: ARRAY [0..127] OF CHAR; spst: SubProgStatus;
  BEGIN
    IF GetFileDialog('Select a model definition program','MOBJ|Mobj',path,mdp) THEN
      Concatenate(path,mdp,mdp); CallSubProg(mdp,spst)
    END;
  END LoadMDP;

  PROCEDURE Quitting(VAR rq: BOOLEAN); BEGIN rq:=TRUE END Quitting;

BEGIN
  InstallMenu(theMenu,'Shell',enabled);
  InstallCommand(theMenu,compCmd,'Compile',CompMDP,enabled,unchecked); InstallAliasChar(theMenu,compCmd,'C');
  InstallCommand(theMenu,editCmd,'Edit',EditMDP,enabled,unchecked); InstallAliasChar(theMenu,editCmd,'E');
  InstallCommand(theMenu,loadCmd,'Load...',LoadMDP,enabled,unchecked); InstallAliasChar(theMenu,loadCmd,'L');
  InstallQuitCommand('Quit shell',Quitting,0C); RunDialogMachine;
END SimShell.
```

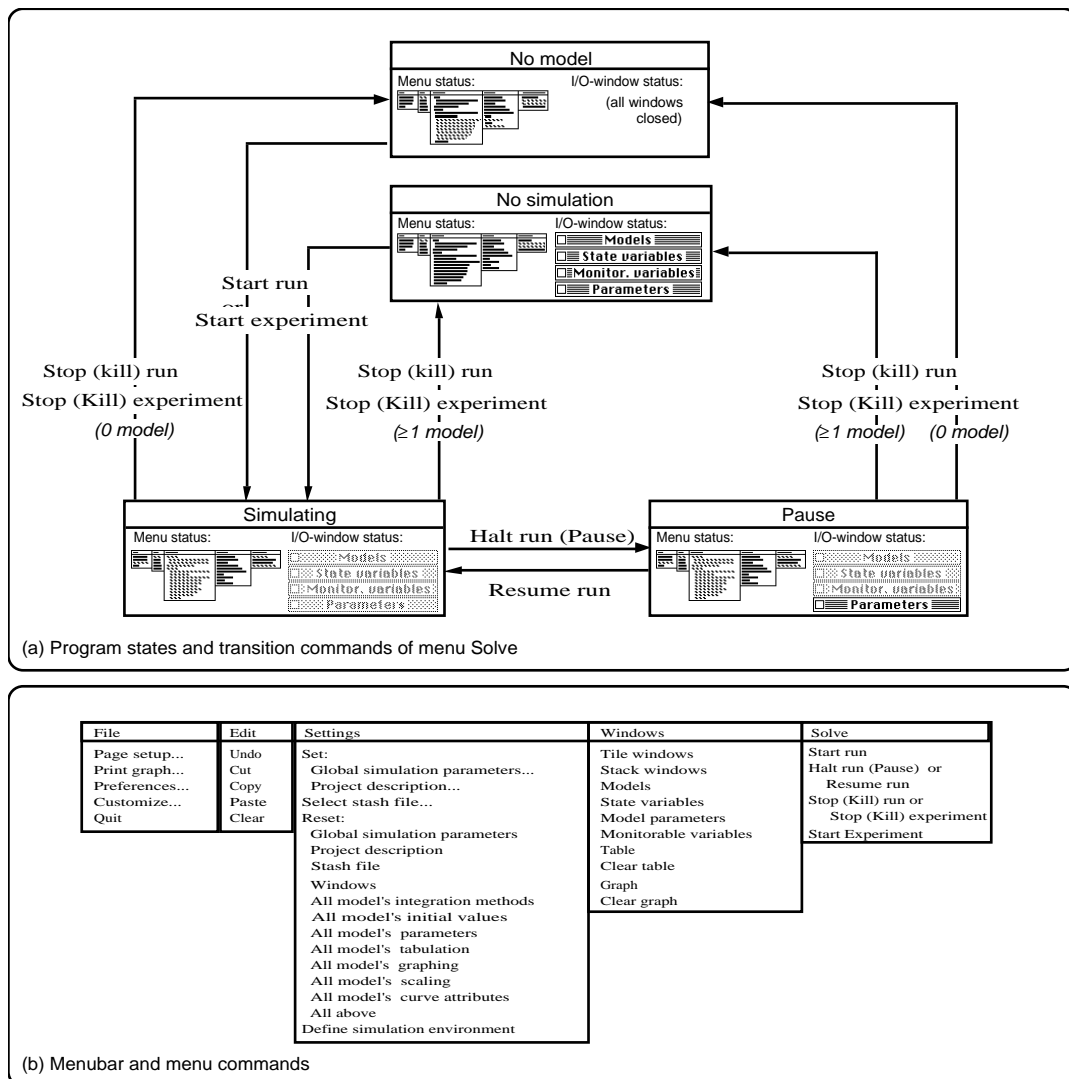
Program *SimShell* allows to load any number of model definition programs on top of each other, each on a different subprogram level. Once loaded, they can all be solved simultaneously, despite the fact that they belong to different subprogram levels. For example this technique allows to compare directly the trajectories of the same model equations, but produced with two different integration methods. A possible way to accomplish such a task is to follow these steps: First, prepare and compile two separate instantiations of the same model, where the second is just a copy except for the differently named program module. Then, load both modules by means of *SimShell* into the same simulation environment. Both models can now be solved simultaneously by choosing menu command *Solve/Start run*. Solving one with integration method *Euler*, the other with method *Runge-Kutta 4th order* allows to compare directly the performance of the two integration methods.

Note that the subprogram levels form a sort of program stack, where each level receives its own about entry in the Apple menu, its own menu commands for *Quit*, *Settings/Define simulation environment*, and *Solve/Start experiment*; all other standard menu commands are shared by all levels. As a consequence it is possible to unload individually the top-most sub program level by choosing its specific menu command *Quit*. The quitting of a program level will result in a selective removal of all models, model objects, and equations belonging to this level with out af-

---

<sup>30</sup>Separate (sub)program levels are only available if the ModelWorks version is based on a Dialog Machine which is implemented by means of a dynamic linking-loading Modula-2 language system. The MacMETH Modula-2 Language System for the Macintosh (WIRTH *et al.*, 1992) is such a Modula-2 system and fully supports the here described behaviour. Ignore any reference to subprogram levels if you are using an IBM PC version, which supports static linking only.

fecting any other parts of the simulation environment's model base. It is even possible to quit a subprogram level which is currently not on the top of the program stack; however this case actually results first in the quitting of all levels above, i.e. top-most levels are removed repeatedly till the chosen one becomes the top-most one and can now finally be removed also.



**Fig. T24** : (a) State transition diagram of the simulation environment of ModelWorks and its effect on the standard user interface. The simulation environment is always in one of the following four states: *No model* the state when no model is installed; *No simulation*, the state in which at least one model is present and no simulation is running. In this state the simulationist may change values or settings, e.g. simulation time, initial values of state variables, or values of model parameters. During a simulation run or a structured simulation experiment ModelWorks is in the state *Simulating*. In this state the simulationist may only temporarily pause or stop (kill) the running simulation. The state *Pause* allows to change model parameters with the attribute *RTC (Run Time Change)* set, or to resume respectively abort the simulation. For every state the status of the menu commands is symbolized as follows: A black line signifies an active, a grey an inactive or unavailable menu command (for the actual menu commands see (b)). The availability of the button commands of a particular IO-window is indicated by a black (object selection with mouse clicks possible, palette buttons can be pushed) or grey (disabled selection, inactive palette buttons) window title bar (s.a. Fig. T15, T16). (b) All menus and menu commands (separators omitted) of the standard user interface (except for the enlargement the same as in (a)).

Experiments can also be executed individually, yet note, that all involved simulation runs will be executed in the same simulation environment. In particular this implies that the ModeWorks run-time system will solve all currently declared models for a common domain of the independent variable, i.e. the currently set global simulation parameters ( $t_0, t_{end}, h_m$  etc.), regardless of their subprogram level. Any program level modifying the global simulation parameters or other global settings, such as a window position, will affect the simulation environment and immediately override any values, which might have been set by another program level, e.g. during earlier loading. However, to preserve current values already present in the simulation environment, every additional call to *RunSimEnvironment* on a new program level will only result in a conditional reset, i.e. in contrast to the very first initialization (Fig. T18) *RunSimEnvironment* calls *ResetAll* only if the simulation environment is in state *No model*. Whenever full cooperation among mathematically well co-ordinated submodels is to be implemented, proper measures can usually be programmed to overcome conflicts between multiple accesses to shared items of the simulation environment (s.a. *Appendix*, chapter *Sample Models*).

5.1.4.b States of the standard user interface

In the standard user interface the states of the simulation environment (Fig. T15 and T16) are characterized by the availability of certain commands (Fig. T24).

In the state *No model* it is not possible to choose any menu command, which requires at least one model to operate on (e.g. *Settings/Reset all model's parameters*). In the state *No simulation* model and model object attributes can be interactively changed. In the state *Simulating* user interactions are limited, e.g. IO-windows are in activated and will not respond to mouse clicks. In the state *Pause* the simulation is temporarily brought to a halt to allow for interactive changes of parameters only (s.a. Fig. T15, T16). Note that every state transition, regardless whether it is caused via the user or the client interface, will cause the simulation environment to reflect this fact properly in all its parts, in particular also in the standard menus.

5.1.4.c IO-windows (Input-Output-windows)

Unless customized the standard user interface provides IO-windows (Fig. T25). They serve two purposes:

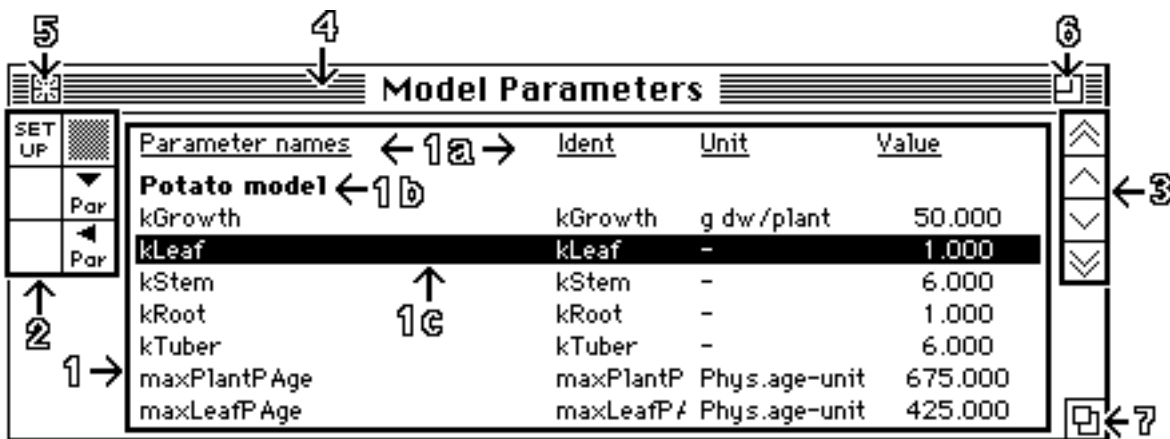


Fig. T25 : Basic structure of IO-windows subdivided into three fields: In the middle the list of model objects (1), on the upper left corner the palette of button commands (2), and on the upper right corner the scrollers to scroll the items in lists too large to show all items at once (3) (s.a. text).

First they display all models and all model objects plus their current values and settings (Output). Second they allow to modify interactively the current values and settings of these objects (Input). For instance, the value of an individual model parameter can be changed or reset, or

the kind of monitoring for a particular variable during simulations can be specified. There are four IO-windows: The first IO-window with the title *Models* lists all models ModelWorks currently holds in its model base, i.e. which have been declared by the modeller via the client interface. The second IO-window *State variables* lists all declared state variables, the third *Parameters* all parameters, and the fourth *Monitorable variables* all monitorable variables.

All IO-windows have a common structure: The content area of any IO-window is subdivided into three fields (Fig. T25). First the field in the middle of the window contains a list of ModelWorks objects (1). Its title line (1a) displays the headers of the columns currently in use, which describe, display, and designate ModelWorks objects and their values. Below, there is the actual list of the objects, e. g. the parameters, which have been installed in ModelWorks by the model definition program. The order follows the declaration order in the model definition program, and objects belonging to the same model appear together under the bold title of the corresponding model (1b). An object in the list can be selected as an operand by a mouse click on the corresponding line, which is confirmed by inverting the line (1c).

From the described behaviour follow scope rules for the selection of operands (Fig. T26).

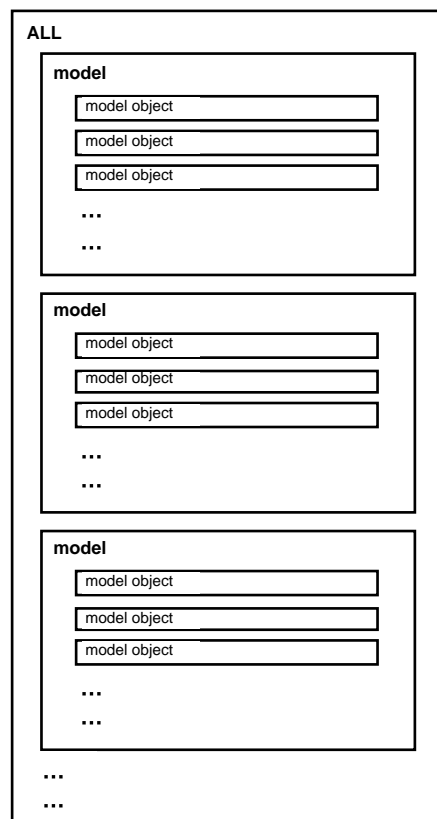


Fig. T26 : Scopes used for the selection of operands in the IO-windows. Selecting a model implies the selection of all model's objects.



Since model objects belong to models, the selection of a model in the models IO-window can be interpreted as the selection of all its objects. Hence the selection scopes in the models IO-window are:





- individual model respectively all objects of a model
- all models respectively all objects of all models

For the IO-windows of the state variables, model parameters, and monitorable variables exist the following selection scopes:

- all objects of a particular kind of all models
- all objects of a particular kind of a model
- individual object of a particular kind

Note that the operands actually affected by an operator are determined also by the operator itself. For instance: The selection of an individual model does not only allow to change an attribute such as the integration method of this model, but also to reset the values of all its objects, such as the resetting of all initial values of the model's state variables to their defaults.

Second the button palette (2) on the left side contains a palette of adjacent, square buttons. Each button has a separate function (operator), which can be activated by clicking on the little button picture with the mouse. There are two kinds of functions: basic window functions not requiring an operand, e.g. a window set up, and functions (operators) operating on the selected elements (operands). Two buttons are common to all windows: The button  activates an entry form where the columns to be displayed in the object field can be selected. The button  serves to select all objects of the particular kind listed in the IO-window, e.g. all parameters of all models (Scope *All* in Fig. T26). The simulationist will be informed while ModelWorks is executing a button function, e.g. by inverting the button picture or by any other appropriate mean.

Third on the right side, there are the scrollers to scroll lines individually (, ) or whole pages (, ) of the object list field up and down in case, that the window is too small to show all objects at once (3). During the actual scrolling the button picture will be shown inverted.

In an inactive IO-window no selection of operands is possible, nor can a button function be activated, nor is any scrolling possible. The simulationist can recognize this status if neither clicking within the list field nor on buttons or scrollers does invert the clicked object (Fig. T24).

The last group of elements are not specific to ModelWorks but are general and may be present in any window (Fig. T25)<sup>31</sup>: the title bar to move the window (4), the close box to close it (5), the zoom box to enlarge it to the size of the screen or back (6), and the grow box to change the size of the window to any shape (7).

### 5.1.5 USER INTERFACE CUSTOMIZATION

The standard user interface of ModelWorks can be customized, i.e. adapted to the users needs, in various ways:

First, it is possible to override the predefined settings, such as the default position of windows or the display of the lists in an IO-window. To this end the module *SimBase* exports various routines such as *SetDefltWindowPlace* or *SetDefltIOWColDisplay* (for an application of this technique see in the *Appendix* the sample model *Markov*).

Second, it is possible to disable certain functions within the standard user interface, e.g. the tabulation of simulation results. To this end the module *SimBase* exports the routines *DisableWindow* and *EnableWindow*. The corresponding menu commands will then still remain visible, but the simulationist can not choose them since they are disabled (dimmed).

Third, when the modeller uses also the optional table functions, the standard user interface is extended also by a graphical table function editor (s.a. *Appendix* chapter *Auxiliary Library* section *TabFunc*; for an application of this technique see the sample models *SwissPop* and *UseTabFunc*).

---

<sup>31</sup>The actual appearance of these elements may differ with the computer on which ModelWorks is running. However, thanks to the underlying 'Dialog Machine' the basic functionality of managing a window's position, size etc. remains the same.



Fourth, the modeller can customize the initialization of the standard user interface by installing an initialization procedure with a call to the procedure *InstallDefSimEnv* from *SimMaster* before calling procedure *RunSimEnvironment*. ModelWorks will then call the installed procedure as the very first step taken before awaiting commands from the simulationist (Fig. T18). The simulationist can request the execution of the installed procedure once more by choosing the corresponding menu command *Settings/Define simulation environment* (for an application of this technique see in the *Appendix* the sample models *SwissPop*, *Sensitivity*, and *Markov*).

Fifth, it is possible to insert before or to add after the menus of the standard user interface additional menus. In the first case the calls to the routines *InstallMenu* and *InstallCommand* from module *DMMenus* must precede the installation of the ModelWorks standard menus, i.e. the call of *RunSimEnvironment* (Fig. T18). In the latter case, i.e. adding menus to the right of the ModelWorks standard menus, the routines *InstallMenu* and *InstallCommand* must be called in the routine *initSimEnv* which has been passed as its actual argument to *RunSimEnvironment*. Such additional menu commands can then be associated with any kind of procedures, e.g. the opening of additional windows to display simulation results, the reading of data from files, the loading and unloading of model definition programs, the calling of a compiler etc. Since ModelWorks is only based on the "Dialog Machine", a co-operative coexistence of ModelWorks objects with other "Dialog Machine" items is already guaranteed by the "Dialog Machine"<sup>32</sup> (s.a. below subchapter *Module Structure of ModelWorks*) (s.a. *Appendix* chapter *Auxiliary Library* section *StructModAux*; for an application of this technique see in the *Appendix* the sample models *Markov*, *GreenHouse*, *CarPollution*, *LBM*, and *ForestYield*).

Sixth, it is possible to associate a handler with a ModelWorks window, e.g. the graph window. This handler will then be executed as soon as the simulationist clicks into the content of that window (s.a. *Appendix* chapter *ModelWorks Optional Client Interface* section *SimGraphUtils*; for an application of this technique see in the *Appendix* the sample model *VDPol*).

Seventh, it is not possible to remove a menu from the standard user interface; yet, with the "Dialog Machine" and ModelWorks it is possible to build a completely new user interface with relatively little effort, as the program module *MySimEnv* (following below) illustrates.

The user interface of *MySimEnv* provides all basic, i.e. the most frequently used functions, which are required for interactive simulations. Educational or demonstration programs can be built similarly, if they ought to offer only a few menu commands instead of the possibilities featured by the ModelWorks standard user interface, because the latter might only confuse the beginner.

---

<sup>32</sup>On how to work with the Dialog Machine see the *Appendix* section *Quick References*, the separate booklet «Installation Guide and Technical Reference of the RAMSES software», and FISCHLIN (1986a, b; *et al.*, 1989).

```

MODULE MySimEnv;

FROM DMMenu IMPORT Menu, Command, AccessStatus, Marking, Separator, InstallMenu, InstallCommand,
  InstallAliasChar, InstallSeparator, DisableCommand, EnableCommand;
FROM DMMaster IMPORT RunDialogMachine;
FROM DMEEntryForms IMPORT FormFrame, WriteLabel, DeflUse, RealField, UseEntryForm;

FROM SimBase IMPORT MWWindow, GetWindowPlace, SetWindowPlace, ResetAll, GetGlobSimPars, SetGlobSimPars;
FROM SimMaster IMPORT SimRun, StopRun, InstallStateChangeSignaling, MWState, GetMWState;

FROM MyMDP IMPORT ModelDefinitions;

VAR
  simMenu: Menu;
  setTCmd, resCmd, openWCmd, runCmd, stopCmd: Command;

PROCEDURE AskForGlobSimPars;
  CONST lem = 5; tab = 35;
  VAR ef: FormFrame; ok: BOOLEAN; cl: INTEGER; to,tend,h,er,c,hm: REAL;
BEGIN
  cl := 2; GetGlobSimPars(to,tend,h,er,c,hm);
  WriteLabel(cl,lem,"Global simulation parameters:"); INC(cl);
  WriteLabel(cl,lem,"to"); RealField(cl,tab,7,to,useAsDeflt,MIN(REAL),MAX(REAL)); INC(cl);
  WriteLabel(cl,lem,"tend"); RealField(cl,tab,7,tend,useAsDeflt,MIN(REAL),MAX(REAL)); INC(cl);
  WriteLabel(cl,lem,"h"); RealField(cl,tab,7,h,useAsDeflt,MIN(REAL),MAX(REAL)); INC(cl);
  WriteLabel(cl,lem,"hm"); RealField(cl,tab,7,hm,useAsDeflt,MIN(REAL),MAX(REAL)); INC(cl);
  ef.x:= 0; ef.y:= -1 (*display entry form in middle of screen*);
  ef.lines:= cl+1; ef.columns:= 55;
  UseEntryForm(ef,ok);
  IF ok THEN SetGlobSimPars(to,tend,h,er,c,hm) END;
END AskForGlobSimPars;

PROCEDURE OpenWindows;
  VAR x,y,w,h: INTEGER; enabl: BOOLEAN;
BEGIN
  GetWindowPlace(MIOW, x,y,w,h, enabl); SetWindowPlace(MIOW, x,y,w,h);
  GetWindowPlace(SVIOW, x,y,w,h, enabl); SetWindowPlace(SVIOW, x,y,w,h);
  GetWindowPlace(PIOW, x,y,w,h, enabl); SetWindowPlace(PIOW, x,y,w,h);
  GetWindowPlace(MVIOW, x,y,w,h, enabl); SetWindowPlace(MVIOW, x,y,w,h);
  GetWindowPlace(TableW, x,y,w,h, enabl); SetWindowPlace(TableW, x,y,w,h);
  GetWindowPlace(GraphW, x,y,w,h, enabl); SetWindowPlace(GraphW, x,y,w,h);
END OpenWindows;

PROCEDURE StateHasChanged;
  VAR s: MWState;
BEGIN
  GetMWState(s);
  CASE s OF
    | noSimulation: EnableCommand(simMenu,runCmd); DisableCommand(simMenu,stopCmd);
    | simulating: DisableCommand(simMenu,runCmd); EnableCommand(simMenu,stopCmd);
  ELSE
    END(*CASE*);
  END StateHasChanged;
END StateHasChanged;

BEGIN
  InstallMenu(simMenu,"Simulation",enabled);
  InstallCommand(simMenu,setTCmd,"Set time...",AskForGlobSimPars, enabled, unchecked);
  InstallSeparator(simMenu,line);
  InstallCommand(simMenu,resCmd,"Reset all",ResetAll, enabled, unchecked);
  InstallSeparator(simMenu,line);
  InstallCommand(simMenu,openWCmd,"Open windows",OpenWindows, enabled, unchecked);
  InstallSeparator(simMenu,line);
  InstallCommand(simMenu,runCmd,"Run",SimRun, enabled, unchecked);
  InstallAliasChar(simMenu,runCmd,"R");
  InstallCommand(simMenu,stopCmd,"Stop (kill)",StopRun, enabled, unchecked);
  InstallAliasChar(simMenu,stopCmd,"K");
  InstallStateChangeSignaling(StateHasChanged);
  ModelDefinitions;
  ResetAll;
  RunDialogMachine;
END MySimEnv.

```

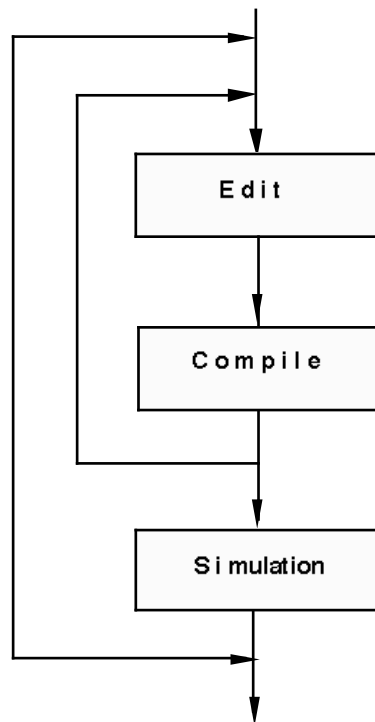
Finally it is also possible to use only some parts of the standard user interface, e.g. just the IO - window for the parameters, or even no user interface at all. The latter, a batch-type simulation program, might just declare a model and the corresponding model objects, in particular some monitorable variables with the stash filing activated (*writeOnFile*) and will then call *SimRun* from *SimMaster* to solve the model. The simulation results will be written to the stash file only. In combination with the "Dialog Machine", almost endless possibilities open up for the implementation of simulation tools tailored to specific applications.

## 5.2 Modelling

### 5.2.1 THE MODEL DEVELOPMENT CYCLE

The modelling process consists of the model development cycle with the steps editing, compilation, and execution of the model definition program (Fig. T27).

This process begins with a mathematical model given in form of the Equ. (4) or (5) respectively (6), (7) to (10). Then the modeller or client has to write the so-called model definition program, which represents an ordinary Modula-2 program, consisting of one or several modules, which import from ModelWorks client interface and are programmed accordingly to the rules described in this text. Typically at least one model with at least one state variable and at least one dynamic equation (Eq. 4.1, 5.1, or 6.1) already represents such a model definition. The resulting program is then capable to solve numerically the initial value problem of the currently declared system of differential equations (DESS), and/or difference equations (SQM), and/or discrete event system specification (DEVS). This corresponds to a translation process of the initial value problem of the mathematical model to a simulation model. The latter may also be termed a numerical problem with the initial values, model and global simulation parameters as inputs plus the monitorable variables as outputs. The algorithms are given by the run time system of ModelWorks.



**Fig. T27** : Flow chart of the development cycle of ModelWorks model definition programs. The modeller writes model definition programs by means of an editor, compiles and eventually links them, and executes them to obtain simulation results<sup>33</sup>.

The «Mini RAMSES Shell» and some sessions of the «RAMSES Shell» provide means to facilitate the development cycle. In particular does the «Mini RAMSES Shell» automatically

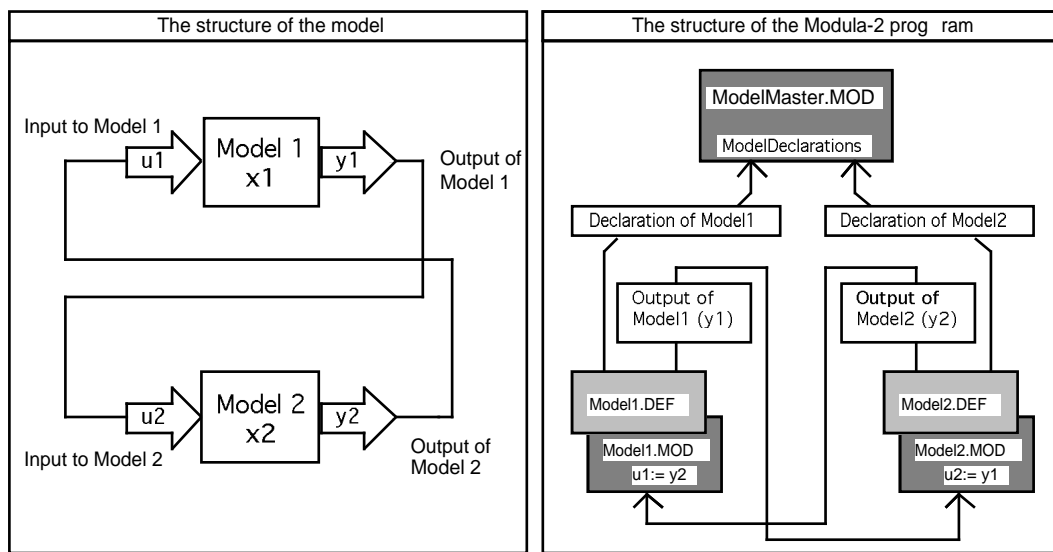
<sup>33</sup>Note, the «RAMSES Shell», in particular the «Mini RAMSES Shell», substantially simplify this cycle for the modeller and perform many tasks automatically. However, in order to obtain maximum efficiency during simulations, the principle remains the same.

switch between the steps shown in Fig. T27 and even hides the compilation step completely. The user just switches between the two roles (part I *Tutorial* Fig. T1) simulationist (simulation) and modeller (edit) only.

### 5.2.2 STRUCTURED MODEL DEFINITION PROGRAMS (MODULAR MODELING)

A model definition program may be built from as many modules as the modeller wishes. Typically structured models are built from several modules (external or library modules), each sub-model corresponding to a Modula-2 module (Fig. T28; see also *Appendix*, chapter *Sample Models* the sample model *GreenHouse* or *LBM* and FISCHLIN, 1991).

If the modeller makes use of modular modeling, the only thing to pay attention to, is to make sure that outputs from one submodel are computed in its procedure *output*, and the depending inputs of another submodel in its procedure *input* (Fig. R16).



**Fig. T28** : Mapping of a structured model composed of two subsystems (left) onto a Modula-2 model definition program (right). The outputs are exported by the definition modules (DEF) and imported by the implementation modules (MOD) of the other submodel. The program module *ModelMaster* links both submodels by importing and executing the submodel declarations. All modules together form the model definition program.

### 5.2.3 STRUCTURED SIMULATIONS (EXPERIMENTS)

The modeller may also program a structured simulation, a so-called ModelWorks experiment. Typically an experiment consists of many simulation runs and will call ModelWorks functions similar to the way the simulationist would use them. The latter is useful to relieve the simulationist from cumbersome, repetitive command sequences or if simulations are used as parts of complex algorithms. For instance in order to create a phase portrait the simulationist would have to assign a series of different initial values to the state variables as well as to start after each assignment a simulation run. The same can be accomplished by programming a structured simulation which the simulationist then can activate by a single command from within the simulation environment. During the development phase models are often thoroughly explored interactively; whereas, once fully developed, there arises the need for a sensitivity analysis or parameter identifications etc. Adding the needed program section in form of an experiment allows to accomplish such tasks without having to modify the existing model definition code.

An elementary simulation run can be started via the client interface by calling procedure *SimRun* from module *SimMaster*. A structured simulation or experiment consists typically of a sequence of calls to procedure *SimRun*, but all functions offered by the client interface may be used to program a structured simulation experiment (for the few exceptional effects of some functions see section *Manipulating the model base at run-time*). The following example illustrates a situation in which four initial state vectors ( $[x,y] = [1, 1], [2, 2], [-1, -1]$  and  $[-2, -2]$ ) for a second order system of differential equations are to be used to produce a phase portrait. Each combination will be used in a separate simulation run:

```
PROCEDURE MyExperiment;
BEGIN
  SetSV(m,x,1.0);   SetSV(m,y,1.0);   SimRun;
  SetSV(m,x,2.0);   SetSV(m,y,2.0);   SimRun;
  SetSV(m,x,-1.0);  SetSV(m,y,-1.0);  SimRun;
  SetSV(m,x,-2.0);  SetSV(m,y,-2.0);  SimRun;
END MyExperiment;
```

For more details see in the chapter *Sample Models* of the *Appendix* the sample model *LVPhasePlot*.

Structured simulations are useful for a sensitivity analysis (see also *Appendix*, chapter *Sample Models* the model *GauseIdentif*) or a parameter identification (s.a. the sample model *GauseIdentif*). To illustrate this point the example of a little sensitivity analysis is presented here: Given a set of  $n$  model parameters and for each parameter a triple of values, i.e. the lower boundary of a confidence interval, the mean, and the upper boundary of the confidence interval ( $\alpha = 5\%$ ) we can declare the following data structure:

```
CONST n = 3;
TYPE PVal = (cur, min, mean, max);
PType = RECORD
  v: ARRAY [cur..max] OF REAL;
  descr,ident,unit: ARRAY [0..64] OF CHAR;
END;
VAR p: ARRAY [1..n] OF PType;
```

The sensitivity analysis may then be implemented by the following procedure *MyExperiment*, which represents a generic recursive solution for any number of parameters:

```
PROCEDURE MyExperiment;
  PROCEDURE Sensitivity(i: CARDINAL);
    VAR j: [min..max];
  BEGIN
    FOR j:= min TO max DO SetP(m,p[i].v[cur], p[i].v[j]);
      IF i<n THEN Sensitivity(i+1) ELSE SimRun END;
    END(*FOR*);
  END Sensitivity;
BEGIN
  Sensitivity(1);
END MyExperiment;
```

For more details see in the chapter *Sample Models* of the *Appendix* the sample model *Sensitivity*.

#### 5.2.4 MODULE STRUCTURE OF MODEL WORKS

The ModelWorks client interface used by the modeller consists of a mandatory and an optional part: The mandatory part (kernel) consists of the two library modules *SimBase* and *SimMaster* and the optional part of the modules *SimDeltaCalc*, *SimGraphUtils*, *SimIntegrate*, and *SimObjects* (Fig. T29). Any model definition program has to import at least from the mandatory client interface and may import from the optional client interface and the auxiliary library *AuxLib*.

ModelWorks itself consists of the 5 modules providing the client interface and in the current implementation of 23 internal modules. All these modules import only from the "Dialog Machine", i.e. from the 11 kernel modules (*DMConversions*, *DMLanguage*, *DMMaster*, *DMMenu*, *DMMessages*, *DMStorage*, *DMStrings*, *DMSystem*, *DMWindowIO*, *DMWindows*, and *DM2DGraphs*), from 10 optional modules (*DMClipboard*, *DMClock*, *DMPEntryForms*, *DMFiles*, *DMMathLib* resp. *DMMathLib20*, *DMFloatEnv*, *DMPrinting*, *DMPTFiles*, *DMWPictIO*), and from the auxiliary library *AuxLib* of the RAMSES software, i.e. the modules *Matrices*

and *JumpTab*. Note that all modules from the auxiliary library do import only from the client interface of the "Dialog Machine", from other auxiliary library modules, or from the ModelWorks client interface. Thus, ModelWorks together with the auxiliary library can be ported without modification to any new computer system on which the "Dialog Machine" is available, regardless of the hardware and operating system.

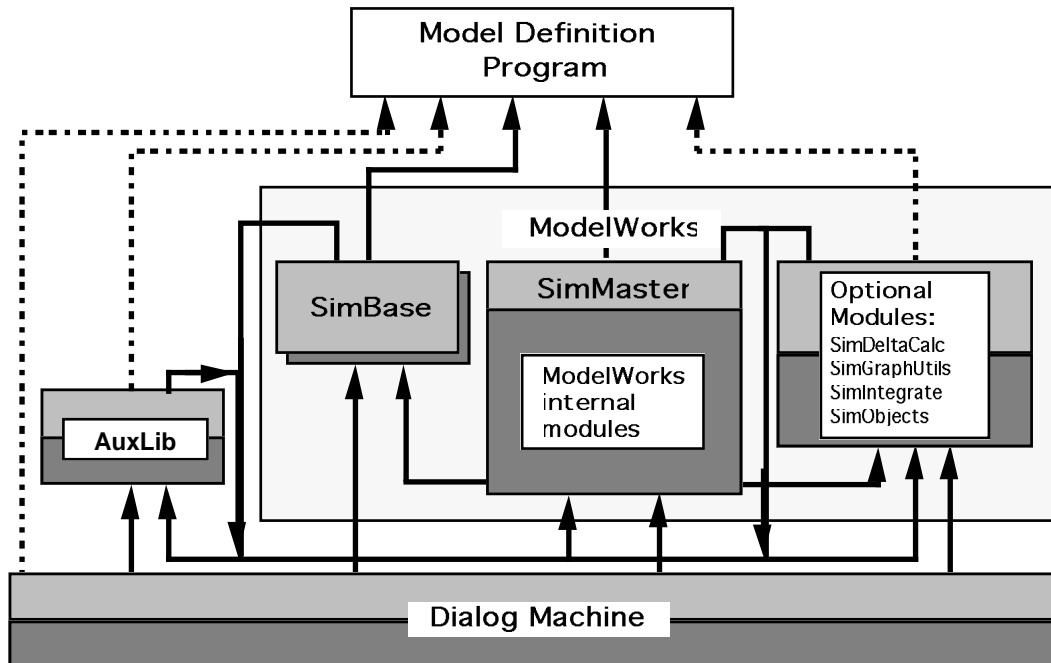


Fig. T29 : Module structure of ModelWorks programs: The model definition program imports from the client interface, which consists at least of the modules *SimBase* and *SimMaster* (mandatory part). The model definition program may actually consist of just one program module up to any number of modules. Some internal ModelWorks modules are prelinked in *SimMaster.OBM*. — mandatory imports; - - - optional imports.

Each module of the optional client interface serves a particular purpose: *SimDeltaCalc* allows to calculate deviations between simulated and observed data series, which is required for model validations or parameter identifications (see e.g. *Appendix* sample model *GauseIdentif*). *SimGraphUtils* can be used to draw into the standard graph window. This feature can be used to draw measurements with error bars into the graph or to customize the graph in any desired way by using also routines from the "Dialog Machine" module *DMWindowIO* (see *Appendix* sample models *VDPol*, *GauseIdentif*, and *Lorenz*). *SimIntegrate* can be used to integrate a model only numerically without actually running a simulation, in particular without any monitoring and without affecting the global independent variable of the simulation environment. *SimObjects* allows an efficient access to the models and model objects contained in the simulation environment's model base, for instance to associate additional data, such as an index, an identifier, or measurements, with a model or model object.

The auxiliary library consists of many modules, of which the following are of particular interest for simulations: *Identification*, *JulianDays*, *RandGen*, *RandNormal*, *ReadData*, *StructModAux*, *TabFunc*, and *WriteDatTim*. *Identification* provides optimization routines which allow to identify unknown model parameters. Given some measurements and specific model equations, the exported algorithms allow to minimize a performance index, e.g. the sum of squares of the differences between measured and simulated values (see *Appendix* sample model *GauseIdentif*). *JulianDays* provides functions useful for the mapping of the simulation time to calendar dates and vice versa. *RandGen* and *RandNormal* return uniformly (within (0,1)) respectively normally ( $N\sim(\mu, \sigma)$ ) distributed variates to support stochastic simulations (see *Appendix* sample

models such as *Diversity*, *Markov*, *StochLogGrow*, or *CarPollution*). *ReadData* facilitates the reading of data from text files (see *Appendix* sample model *SwissPop*), for instance when the user wishes to enter measured data into the simulation environment to compare them with simulation results (s.a. *Appendix* research sample model *LBM*). *StructModAux* supports the implementation of structured models where the submodels reside in separate modules (see *Appendix* sample models such as *GreenHouse* or *LBM*). *TabFunc* is useful if the modeller uses non-linear functions, which are defined by a table of supporting points, so-called table functions. During simulations the modeller can linearly interpolate or extrapolate needed values (see *Appendix* sample models *SwissPop* or *UseTabFunc*). *WriteDatTim*, together with the optional "Dialog Machine" module *DMClock*, allows to access the built-in computer clock in order to record real time events such as the begin and end of a long simulation experiment (see *Appendix* sample model *Markov*).

Since the auxiliary library is only based on defined client interfaces, i.e. either the "Dialog Machine" or ModelWorks, the modeller is free to add any modules she wishes. Note also that some of these auxiliary library modules may depend themselves again on some not yet used "Dialog Machine" parts, some ModelWorks client interface modules, or other auxiliary library modules. E.g. *TabFunc* requires optional "Dialog Machine" modules not used by ModelWorks, i.e. *DMEditFields*, and the auxiliary library modules *Matrices*. A parameter identification module *Identification* may import from the "Dialog Machine", from the ModelWorks modules *SimMaster*, *SimBase*, *SimObjects*, *SimDeltaCalc*, plus *SimGraphUtils*, and some auxiliary library modules such as *Matrices*, *Lists*, and *Optimizations*.

For detailed information on the aforementioned library modules see part III *Reference* subchapter *Client Interface*, the *Appendix* subchapters *Definition Modules* and *Quick References* and for a complete list of all technical aspects the separate booklet «Installation Guide and Technical Reference of the RAMSES software».





## Part III: Reference

This reference part contains a description of the usage of every feature ModelWorks offers. However, it contains only little information on the elementary and typical usage or the theoretical concepts of ModelWorks. In case you should not be familiar with the basic concepts of ModelWorks, please read first the ModelWorks tutorial (part I). In particular you should read the first chapter of the tutorial: *General Description*

The descriptions given in this reference are brief and relate only to specific properties of individual commands. In order to avoid redundancy they do not explain the general principles behind a class of commands and functions of ModelWorks which are described in the part II, *Theory*, in particular in the chapter *Functions*.

This part contains two chapters:

The chapter *User interface* lists all commands which are available to the simulationist via the user interface.

The chapter *Client interface* contains the specifications of the client interface used by the modeller. All functions and the use of all program objects exported by the ModelWorks modules *SimMaster* and *SimBase* are explained.

Any serious modelling with ModelWorks requires to read at least the Part II *ModelWorks Theory* and the second chapter on the client interface of the Part III *Reference*.

**Reading Hint:** For easier orientation, the pages, figures and tables of Part III *Reference* are prefixed with the letter R. Within this part figures and tables are numbered separately, starting with Fig. R1 respectively Tab. R1.

## 6 Standard User Interface


The standard user interfaces of the various ModelWorks versions differ slightly. This text has been made for the standard Macintosh version (see *Appendix*). As long as just the appearance is affected by the differing implementations (holds in particular for the IBM PC versions, which have a slightly different appearance), the following information should be easy to interpret. In all other cases particular explanations have been added.

**Reading Hint:** If there is a functional difference to the standard version, this fact will be stated in a phrase within brackets with the same font as this example: [Not available in Reflex and PC versions].

This chapter describes all features and implementational details of ModelWorks standard user interface and all its parts. This standard interactive environment is activated by calling the procedure *RunSimEnvironment* from module *SimMaster* in a model definition program [MDP]; since the modeller may customize or extend it easily, it may differ in some cases; in particular it may offer more functionality to the simulationist than provided by the standard interface, functions which of course can not be described herein (see also part II Theory chapter *User Interface Customization*).

Note: In the Macintosh versions it is possible to load several model definition programs [MDP's] which all call *RunSimEnvironment* on top of each other<sup>1</sup> [feature not available in static linking versions such as the PC versions]. This allows for example to dynamically replace a model by simply quitting the topmost program and by loading an alternative model definition program instead, hereby leaving models declared on lower program levels untouched. For each additional program loaded dynamically, now called subprogram, the standard user interface is extended by four additional menu commands named *About MDP n...*, *Quit MDP n*, *Define simulation environment n* and *Execute Experiment of MDP n*, which are again removed when the program is quit (*n* counts the number of subprograms loaded which have called *RunSimEnvironment*). In particular, the last two commands allow the simulationist to execute a separate simulation environment definition procedure or an experiment for each program level currently loaded. For more information on running the interactive simulation environment and for loading of several subprograms on top of each other see also part II *Theory* section *Multiple activations of the standard user interface*

### 6.1 Menus and Menu Commands

This section explains all menu commands in detail. Many and often used menu commands can also be invoked by using the keyboard instead of the mouse. For easier remembering the keys to be used for the keyboard shortcuts are shown together with the texts of the menu commands (Fig. R1). Keyboard shortcuts or so-called keyboard equivalents are entered by pressing the command key (clover-leaf key ) simultaneously with another key "X"<sup>2</sup>.

**Reading Hint:** Throughout this reference manual such keyboard equivalents are abbreviated as "/ X".

The following keyboard commands are globally available in the simulation environment: In all entry forms the simulationist may press the key *Return* or *Enter* instead of clicking into the push button *OK*. Pressing the keys ". ." or *Escape* is equivalent to the clicking into the push button

---

<sup>1</sup> A new Modula-2 program module is loaded by calling the procedure *DMMaster.CallSubProg(moduleName...)*, e.g. via an extra installed menu.

<sup>2</sup> In the PC GEM-Version press the Ctrl-key simultaneously with the key "X", in the PC Windows-Version press the Alt-key simultaneously with the key "X".

*Cancel*<sup>3</sup>. The latter two keyboard equivalents may also be used to stop a simulation run (*Stop (Kill) run*). Pressing the key *tab* in an entry form allows to move to the next edit field plus to fully select its content. In some edit fields the key combination *Shift tab* is available to move backwards from field to field (e.g. in the data table provided by the module *TabFunc*). While a selection is currently made, the simulationist may use within an edit field the key equivalents " C" for copying the selection into the clipboard, " X" to cut (copy plus delete) the selection into the clipboard, and " B" to blank (delete without copy) the selection. The current content of the clipboard (if text) may be pasted into an edit field at the current location of the insertion bar or as a replacement for the current selection by pressing " V". This technique allows also to transfer textual data between different entry forms and between different applications (given they support the clipboard for text). If no entry form or other dialogue box is currently in use, the clipboard accessing keyboard equivalents have the usual meaning (see below *Menu Edit*). By pressing " W" the currently active window will be closed, if ModelWorks is run from within the RAMSES Shell<sup>4</sup>.

### 6.1.1 OVERVIEW OVER MENUS

Fig. R1 shows an overview of all menus and menu commands of ModelWorks' standard user interface [In the PC GEM-Version any of the menu commands starting with the phrase *All model's...* are missing, but note that these functions are also available in the IO-windows]. If the modeller imports from module *TabFunc*, an additional menu will appear (see Fig. A1 in the *Appendix* section *Auxiliary Library module TabFunc*). A detailed explanation of all menus is given below.

File	Edit	Settings	Windows	Solve
Page setup...	Undo ⌘Z	Set:	Tile windows	Start run ⌘R
Print graph...	Cut ⌘H	Global simulation parameters... ⌘I	Stack windows	Halt run (Pause) ⌘H or Resume run (⌘R)
Preferences...	Copy ⌘C	Project description... ⌘D	Models	Stop (Kill) run ⌘K or Stop (Kill) experiment (⌘K)
Customize...	Paste ⌘V	Select stash file... ⌘F	State variables ⌘S	Start experiment ⌘E
	Clear ⌘B	Reset:	Model parameters ⌘P	
		Global simulation parameters	Monitorable variables ⌘M	
		Project description	Table ⌘T	
		Stash file	Clear table	
		Windows	Graph ⌘G	
		All model's integration methods	Clear graph ⌘B	
		All model's initial values		
		All model's parameters		
		All model's stash filing		
		All model's tabulation		
		All model's graphing		
		All model's scaling		
		All model's curve attributes		
		All above		
		Define simulation environment		

**Fig. R1 :** All ModelWorks standard menus. The two grey menus at the very left indicate menus which are only present if ModelWorks is run from within the «RAMSES Shell».

### 6.1.2 QUIT COMMANDS

The *Quit* command appears at the bottom of the second leftmost menu, i.e. the menu to the right of the " "- or "\*" -menu<sup>5</sup>.

<sup>3</sup> In the PC GEM-Version keyboard shortcuts function only if a letter is involved, hence the cancel function with "Ctrl^." is not available. Use *Escape* instead.

<sup>4</sup>In the PC Windows-Version use the keyboard shortcut "Ctrl^F4" to close a window.

<sup>5</sup>This menu will correspond to the ModelWorks menu *File.*, unless ModelWorks is used from within another environment which has already installed menus to the left of menu *File*, e.g. as does the simulation session of

Indeed, this menu will offer as many *Quit* commands as program levels currently exist [In the PC versions only one program level is supported]. Selecting a quit command which is not the current top-most program level, results in quitting at once all sub-programs from the selected till the top-most program level. The keyboard equivalent **Q** is always assigned to the last command of the menu and allows quitting of the current topmost program level.

*Quit / Q*: Quits the interactive simulation environment of ModelWorks. According to how ModelWorks is currently used, the application will change either to the next lower program level within the interactive simulation environment, or to the calling program of the model definition program<sup>6</sup> [PC versions: The MDP is an application which always returns to the calling program, for instance the operating system].

### 6.1.3 MENU *FILE*

Allows to control the printing of the content of window *Graph* and the current settings and modes of the simulation environment (preferences) (Fig. R2).

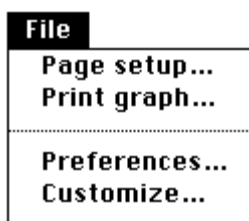


Fig. R2 : Menu *File*.

*Page setup...*: Usual page set up dialogue box used for the printing of the graph on the currently chosen printer. [Not available in Reflex and PC GEM-Version].

*Print graph...*: Prints the graph on the currently chosen printer. [Not available in Reflex and PC GEM-Version].

*Preferences...*: Allows to set the modes of the simulation environment (Fig. R3).

#### Filing

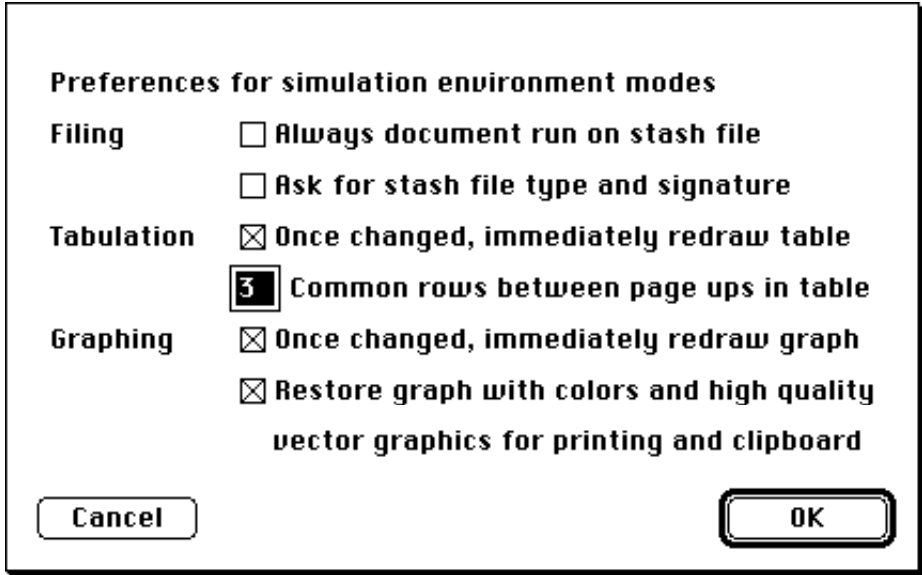
If the simulation environment mode *Always document run on stash file* is active, the stash file is opened at the begin of every simulation regardless of the current setting for stash filing of the monitoring variables. If this mode is inactive, the stash file will only be opened in case that at least one monitoring variable has the stash filing currently set (F). In case you rather use the stash file for run documentation purposes than for post run analysis purposes, it is recommended to have this simulation environment mode active. It will then force the opening of the stash file always and document every simulation experiment, e.g. by documenting the current parameter settings together with some key results. The recording flags and the stash file settings (F) of the monitorable variables will then no longer affect the file opening, but only determine which data are to be written to the stash file (s.a. below menu command *Set Project description...* recording flags in section *Menu Settings*).

---

the «RAMSES Shell». However, note the «Mini RAMSES Shell» suppresses this quit command; instead it offers the menu command *Shell/Exit simulation*.

<sup>6</sup> Typically the RAMSES- or the MacMETH-Shell

If the mode *Ask for stash file type* is activated, every time the simulationist selects a new stash file, a dialogue is displayed allowing to specify the file's type and signature (s.a. below menu command *Select stash file...*).



**Fig. R3 :** Entry Form of the menu command *Preferences...* shown with settings recommended for the beginner.

**Tabulation**

If the simulation environment mode *Once changed, immediately redraw table* is active, the table is redrawn immediately after each change in the tabulation settings. Otherwise the last table will be kept untouched until the next simulation run is started. Only at that time the old table is cleared and a new one will be drawn.

The number *Common rows between page ups in table* defines what happens during a page up. A page up occurs when the table window is full but more rows should be written; then ModelWorks attempts to erase most of the table and restarts tabulating from the top again. This number specifies first how many rows at the bottom are not erased but copied to the top of the next page. All remaining space below is then used to add the rows of the new page. Thus this number specifies how many rows are common to two consecutive pages.

**Graphing**

If the simulation environment mode *Once changed, immediately redraw graph* is active, the graph is redrawn immediately after each change in the graph settings. Otherwise the last graph will be kept untouched until the next simulation run is started. Only at that time the old graph is cleared and a new one will be drawn.

Activating the simulation environment mode *Restore graph with colours and high quality vector graphics for printing and clipboard* is active, the graph is restored with colours when a previously covered graph portion becomes visible again or will be printed or transferred to the clipboard in colours and with high quality. If this mode is turned off, a bitmap is used for restoring, printing, or transfer into the clipboard (pixel based raster graphics). Graph restoration becomes necessary whenever the simulationist moves, rearranges, or closes windows and the graph window is involved. Note

that with this option active, graphs may be restored slower and more memory may be needed. Otherwise graphs are restored in black and white only<sup>7</sup>. Vector graphics contain data about particular objects and the coordinates defining them; e.g. a line is stored as a line object together with the coordinates of its begin and end point. Hence vector graphics are usually of a higher quality than raster graphics. However, the printing of a vector graph may require too much time if draft quality of a graph would be sufficient. Note that with this option active complicated graphs, particularly if drawn during large experiments, may use up tremendous amounts of memory. On black and white monitors activate this mode if you wish to use colour printers or transfer the graph via the clipboard to other colour devices. [Not available in Reflex and PC GEM-Version].

*Customize...* : Allows to customize alias characters (i.e. keyboard equivalents or shortcuts) for the ModelWorks menu commands. First the simulationist is asked which type of customization she wishes to perform (Fig. R4).



Fig. R4 : Initial question asked when customizing keyboard shortcuts.

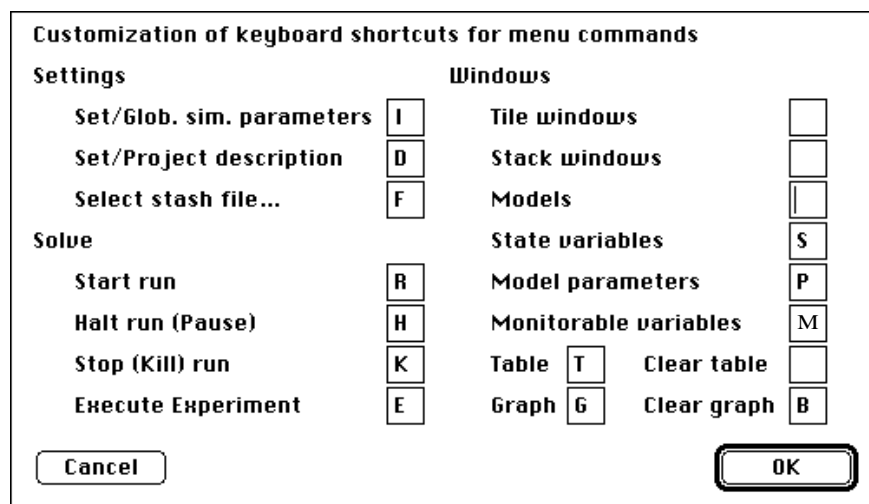


Fig. R5 : Customization of keyboard shortcuts.

The choice “Edit” allows to modify the keyboard shortcuts for the frequently used menu commands (the “core menu commands”) listed in the entry form shown in

<sup>7</sup>Note that these simulation environment modes have no default values. The current values are written in the resource fork of the MacMETH-Shell. They are read from there at the start-up of your model definition program.

Fig. R5. For the actual meanings of these commands see the explanations of the menus *Settings*, *Solve* and *Windows*.

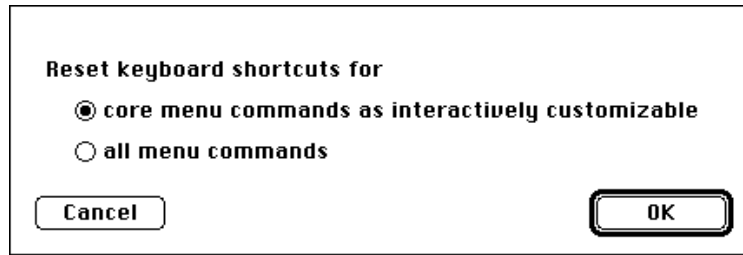


Fig. R6 : Resetting of keyboard shortcuts.

“Reset” offers the possibility to reset the keyboard shortcuts for the core menu commands, or for both, the core and all other ModelWorks menu commands to predefined values (Fig. R6). The predefined values for the core menu commands are shown in Fig. R5. For the other menu commands, which may be modified through the ModelWorks client interface, the default consists in no keyboard shortcuts being installed.

#### 6.1.4 MENU *EDIT*

Menu *Edit* allows to transfer texts such as parameter values or the content of the window *Graph* within ModelWorks' simulation environment respectively to import or export such objects among ModelWorks and other applications (Fig. R7).

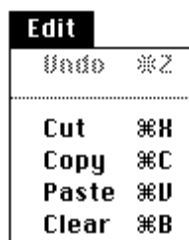


Fig. R7 : Menu *Edit*

[The whole menu is not available in the Reflex and PC GEM-Version].

*Undo / Z*: not available (present only for compatibility with user interface guide-lines).

*Cut / X*: Clears the graph and copies it into the clipboard if no other window than a ModelWorks window is the front most window. Otherwise, e.g. if a desk accessory is the front most window, this command will perform the standard *Cut* command as described in the computer owner's handbooks<sup>8</sup>. The quality of the transferred graph depends on the current simulation environment mode as described under menu command *File/Preferences...*

*Copy / C*: Copies the graph into the clipboard if no other window than a ModelWorks window is the front most window. Otherwise, e.g. if a desk accessory is the front most window, this command will perform the standard *Copy* command as described

<sup>8</sup>For instance *Macintosh owner's guide*

in the computer owner's handbooks. The quality of the transferred graph depends on the current simulation environment mode as described under menu command *File/Preferences....*

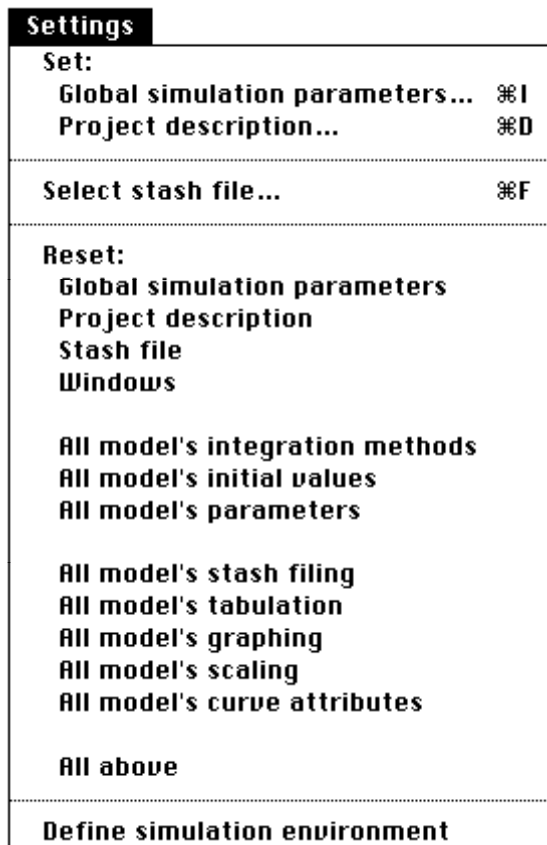
*Paste/ V*: If a ModelWorks window is the front most window, the current content of the clipboard is pasted into the graph window.

*Clear/ B*: Clears the graph if no other window than a ModelWorks window is the front most window. Otherwise, e.g. if a desk accessory is the front most window, this command will perform the standard *Clear* command as described in the computer owner's handbooks.

Keyboard equivalents of these commands are available often even when the simulation environment is in a mode which prohibits choosing menu commands, e.g. when an entry form is momentarily in display. The meaning of these commands is then such that textual objects such as parameter values and not the graph are exchanged with the clipboard. For a more detailed description of these commands see the first section of this chapter.

### 6.1.5 MENU *SETTINGS*

This menu consists of four parts: *Set*, *Select stash file...*, *Reset* and *Define simulation environment* (Fig. R8).



**Fig. R8** : Menu *Settings* used to set or reset current values and to define current settings of the simulation environment.



The first part lets you set the global simulation parameters such as the simulation start and stop time, plus the project description. The second determines which file is going to be used as the stash file. The third is used to reset the current values of global parameters, settings, and of model objects; resetting means to copy the defaults to the corresponding current values (Fig. T17 part II *Theory*). [The PC GEM-Version will not offer any of the menu commands starting with the phrase *All model's...*(but note, these functions are also available via the IO-windows)]. The menu's last part allows you to execute a procedure which may have been installed by the modeller via the ModelWorks client interface and which is typically used to (re)define simulation environment settings according to your individual needs.

### Set:

*Set Global simulation parameters./* I: Displays an entry form (Fig. R9a-b) to set the global simulation parameters such as the integration step. Note that all of these parameters are valid globally, i.e. they determine time and integration parameters for all present (sub)models together. Hereby, the start and stop time for the simulation and the monitoring interval can always be edited, whereas all remaining parameters may be ignored in the entry form, depending on the kind of models which are present (see explanations below and Fig. R9a-b).

*Start time for simulation* [ $t_o/k_o$ ]: The next simulation run will start with this time.

*Stop time for simulation* [ $t_{end}/k_f$ ]: The next simulation run will stop with this time.

*Integration step* [ $h$ ]: If at least one continuous time (sub)model is present,  $h$  is the fixed time step<sup>9</sup> for the numerical integration of the differential equations. Moreover, if a variable step length integration method is in use by at least one of the continuous time (sub)models, this simulation parameter becomes the

*Maximum integration step* [ $h_{max}$ ]: The actual integration step will be determined by the variable step numerical integration algorithm and globally used as the integration step for all other submodels, even if they should be solved with a fixed step length method.

*Maximum relative local error* [ $e_r$ ]: If at least one continuous time (sub)model is using a variable step length integration method, this simulation parameter determines the maximum relative local integration error  $\underline{\epsilon}$  estimated by comparing a higher order result with a lower order result. If a norm of this error vector  $|\underline{\epsilon}|$  divided by a norm of the state vector  $|\underline{x}|$  exceeds  $e_r$ , the integration step length  $h$  is halved till  $|\underline{\epsilon}| / |\underline{x}| \leq e_r$ . Otherwise  $h$  is doubled unless one of the following two conditions would become true  $|\underline{\epsilon}| / |\underline{x}| > e_r$  or  $h > h_{max}$ .

*Discrete time step* [ $c$ ]: If only discrete time (sub)models are present (case B)  $c$  may be edited in place of the integration step  $h$ . In this case however, the actual value of  $c$  is irrelevant, since the length of an interval between two discrete time points has no true meaning. If not only discrete time, but also continuous time (sub)models are present,  $c$  may be edited in addition to  $h$  and becomes the *Coincidence interval* [ $c$ ] (see also part II *Theory* chapter *Model Formalisms*).

*Monitoring interval* [ $h_m$ ]: Interval at which the values of all monitorable variables are either written onto the stash file, tabulated in the table, or drawn into the graph, depending on their current monitoring settings. Note, that if a discrete event model is present with at least one monitorable variable activated for stash filing, tabulation, or

<sup>9</sup>The smaller the step  $h$  is, the more accurate is the calculation (unless  $h$  gets so small that truncation errors become dominant); the larger  $h$  is, the faster runs the simulation. Therefore the simulationist has to select a good compromise, which depends on the integration method used and on the nature of the model.

graphing, in addition to the regular monitoring given by  $h_m$ , monitoring will also take place at every occurrence of a discrete event.

Start time for simulation:	0.0
Stop time for simulation:	30.0
Integration step h:	0.05
Maximum relative local error:	(ignored)
Coincidence interval c:	(ignored)
Monitoring interval:	0.25

**Fig. R9a** : Entry form *Global simulation parameters.*/ I for a case where only continuous time (sub)models with a fixed step length integration method are present.

Start time for simulation:	0.0
Stop time for simulation:	30.0
Maximum integration step h:	0.05
Maximum relative local error:	0.001
Coincidence interval c:	1.0
Monitoring interval:	0.25

**Fig. R9b** : Entry form *Global simulation parameters.*/ I for a case where some continuous as well discrete time (sub)models are present. In addition a variable step length integration method is used.

*Set Project description...*/ D: Displays the entry form to edit a global project description and control the recording of data on the stash file (Fig. R10).

*Project title*: String which can be freely used to describe the on-going project, i.e. for instance a title for the current simulation session. If the menu command *Print graph...* is chosen, this *Project title* string will always be printed in bold above the graph. However, the graph displayed and transferred into the clipboard will contain this string only if the flag *Use in Graph* has been checked. In the graph window this string will be displayed in the middle and at the top of the data panel.

*Remarks:* String which can be freely used to add some remarks on the on-going project. For instance it may be used as a sub-title similar to the project title string or it may contain some information on specific model parameter settings used in the simulations. If the menu command *Print graph...* is chosen, this *Remarks* string will always be printed just below the title in a smaller font and with style plain. However, the graph displayed and transferred into the clipboard will contain this string only if the flag *Use in Graph* has been checked. In the graph window this string will be displayed to the right of the legend at the bottom of the window.

**Project title**  Use in Graph

**Remarks**  Use in Graph

**Footer**  Automatic date & time update in footer

10/Mar/1993 16:02 Run 1

**Record data on the stash file during simulations for:**

Models  State variables  Table functions

Model parameters  Monitorable variables  Graph

Cancel OK

Fig. R10 : Entry form *Project description...*/ D

*Footer.* By default the footer contains the date, the time, and the simulation run number, but it may also be used to store any other information. If the flag *Automatic data & time update in footer* is turned on, ModelWorks will update this information at the begin of each simulation run. If the menu command *Print graph...* is chosen, this footer string will always be printed in a small font size below the graph. However, the graph transferred into the clipboard will never contain this string.

*Record data on the stash file during simulations for.* With these recording flags the simulationist may control which information and values are written onto the stash file. Check the appropriate boxes for models, model parameters, state variables, monitorable variables, table functions and the graph if you wish to have them written onto the stash file at the begin (for all except the graph) and at the end (graph only) of simulation runs.

Note that the flags *Models*, *Model parameters*, *State variable*, and *Monitorable variables* mean that information about these objects is written to the stash file. They are: the descriptor, the identifier, the unit (unless a model), and the object specific current values.

Except for the monitorable variables, information about all objects will be recorded. In case of the monitorable variables only the information about those monitorable variables is recorded, which are involved in the stash filing ( F ) or in the graph ( X or Y ). The latter requires also that the corresponding recording flag (*Graph*, see below) has been set.

The recording flag *Graph* controls whether graphical simulation results are written to the stash file. [Graph recording not available in Reflex and PC GEM-Version].

Note that ModelWorks will record information on the stash file at the begin and end of each simulation run, in particular also during experiments.

The optional recording flag *Table functions* is only shown if at least one table function is present (s.a. see *Appendix* section *Auxiliary Library* module *TabFunc*). It then allows to control whether the current values and settings of all table functions are written to the stash file.

The stash file is written in the so-called RTF-Format<sup>10</sup> which can be opened by the Microsoft® Word, WriteNow™, or MacWrite II document processing software<sup>11</sup>. Opening the file with other text editors which cannot interpret RTF is also possible; however, neither the graph nor the RTF control strings can be interpreted and remain dispersed throughout the text and distort its appearance. However data from simulation results are written in a format which allows to paste or import them directly into many other applications, such as the spreadsheet program Excel from Microsoft® or the presentation graphics program Cricket Graph<sup>12</sup>. The format of the stash file has also been designed to allow for an efficient and simple post simulation analysis. In particular check the recording flags for models and monitoring variables if you wish to produce a stash-file which can be used successfully by a post analysis<sup>13</sup>.

Note that in case the simulation environment mode *Always document run on stash file* is currently not active, the recording flag settings are irrelevant if no monitoring variable has currently the stash file setting active (F). Only as soon there is at least one variable, the stash file will be actually opened and the recording flags then control which information is written onto the stash file in addition to the simulation results. If you plan to run a post analysis from the stash file, you should at least have the recording flags *Models*, and *Monitorable variables* active. If you rather use the stash file for run documentation purposes it is recommended to have the simulation environment mode *Always document run on stash file* active. The recording flags together with the stash file setting (F) of the monitorable variables will then solely control the kind of information and data written to the stash file (s.a. menu command *Preferences...*).

*Select stash file...* / F Allows to select a stash file (Fig. R11, left) with the usual open file dialogue box. Note that this command will not really open the file until the simulation starts. This behaviour offers the advantage, that the simulationist may open it for inspection whenever the simulation environment is in state *No simulation*.

In case the mode *Ask for stash file type* is activated, the stash file selection is followed by the dialogue shown in Fig. R11 (right). The file's type and signature

---

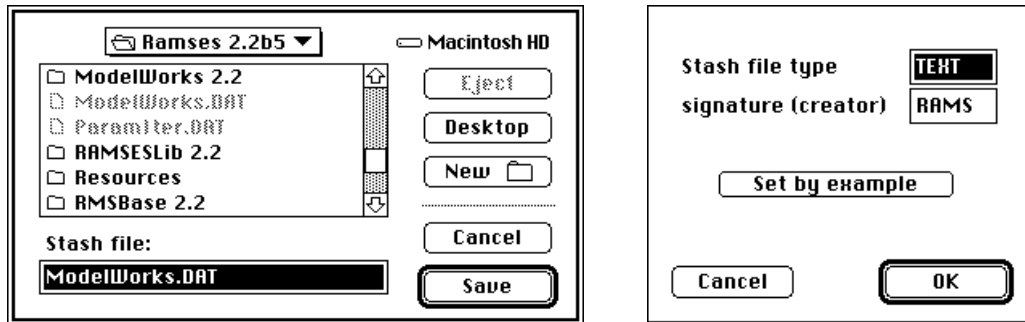
<sup>10</sup>RTF stands for Rich Text Format. It is based on ASCII characters only but contains coded formatting information that can be interpreted by many commercially marketed text processing applications on various computer platforms.

<sup>11</sup>Microsoft® Word is available from Microsoft® Corporation. WriteNow™ has been written by Anderson, D.J., Tschumy, B. & Stinson, C. and is available from NeXT Inc. MacWrite II is available from Claris Corp.

<sup>12</sup>Cricket Graph is a program to edit and produce presentation graphics for science and business by Rafferty, J. & Norling, R. and is available from Cricket Software Inc.

<sup>13</sup>The current version of ModelWorks does not feature a post analysis session. However, the RAMSES post-simulation analysis tool allows to explore, interactively or under program control, simulation results and other data contained in any ModelWorks stash-file.

may then be typed in the respective fields or set by example, the latter by selecting any application or document by means of the standard open file dialogue box. [file types and signatures are not available in Reflex and PC versions]



**Fig. R11** : Dialogue box *Select stash file...* / F (left) and dialogue for specification of stash file type and signature (right) [file types and signatures are not available in Reflex and PC versions].

Once a simulation starts, i.e. ModelWorks enters the program state *Simulating*, the stash file will be automatically opened and remains open till the state *Simulating* will be left. Since during a whole structured simulation ModelWorks remains in the state *Simulating* this means that all results from all simulation runs are normally written on the same stash file. The default name used in the file selection dialogue box is the current stash file name. Note that if there is not at least one monitorable variable present for which the current stash filing setting is activated (*F/writeOnFile*), ModelWorks will never open a stash file regardless of the current settings of the recording flags (this behaviour may be overridden by means of the simulation environment mode *Always document run on stash file*, see above).

### Reset:

The reset menu commands (Fig. R8) assign to the selected element(s) the default value(s) (s.a. section on Resetting, Fig. T17 part II *Theory*). The latter have been defined by the modeller in the model definition program or have been predefined by ModelWorks. All commands in this menu operate on the scope of all models respectively all objects of all models (Fig. T26 part II *Theory*); other scopes are available in the IO-windows only.

*Reset Global simulation parameters* Resets all global simulation parameters.

*Reset Project description:* Resets all strings and flags used to describe the current project to their defaults.

*Reset Stash file* : Resets the stash file name, type and signature.

*Reset Windows.* All windows are reset to their default status. Typically, this will reconstruct the state entered after start up of the interactive simulation environment. Hereby, individual windows may be (re)shown, hidden, or repositioned, and for all IO-windows the original set up for the display of columns is assumed.

*Reset All model's integration methods.* Resets the integration methods of all models. [Not available as a menu command in Reflex and PC GEM-Version].

*Reset All model's initial values* : Resets the initial values of all state variables of all models. [Not available as a menu command in Reflex and PC GEM-Version].

*Reset All model's parameters:* Resets all parameters of all models. [Not available as a menu command in Reflex and PC GEM-Version].

*Reset All model's stash filing :* Resets the stash file setting (*writeOnFile/ notOnFile*) of all monitorable variables of all models. The stash file name and directory as defined with the menu command *Select stash file...* is not affected. [Not available as a menu command in Reflex and PC GEM-Version].

*Reset All model's tabulation:* Resets the tabulation settings (*writeInTable/ notInTable*) of all monitorable variables of all models. [Not available as a menu command in Reflex and PC GEM-Version].

*Reset All model's graphing:* Resets the graph settings (*isX/isY/notInGraph*) of all monitorable variables of all models. [Not available as a menu command in Reflex and PC GEM-Version].

*Reset All model's scaling:* Resets the minimum and maximum values used for the scaling of all monitorable variables of all models on the ordinate. These scaling extremes define the range of interest (Fig. T2 part *Tutorial*) and are used during the drawing of values of the monitorable variables in the graph. [Not available as a menu command in Reflex and PC GEM-Version].

*Reset All model's curve attributes:* Resets the *curve attributes* of all monitorable variables of all models to their default values. [Not available as a menu command in Reflex and PC GEM-Version].

*Reset All above :* Encompasses a reset of all reset commands listed above, in particular: resetting of the global simulation parameters, of the project description, of the stash file, of the windows, of all integration methods for all models, of all initial values, of all parameters, and of all monitoring settings (stash filing, tabulation, graphing, scaling and curve attributes). See Resetting in the part *Theory*. If the modeller has not changed any defaults by calling a *SetDeflt*-procedure (see this part chapter *Client Interface* section *Modification of defaults*) since the simulation environment has been started up, the simulation environment's status and the current values of all objects will be exactly the same as they were right after the start up of the model definition program. However, since table functions are optionally added objects, note that this command does not affect, i.e. not reset, the current values of a table function; use the separate reset function provided by module *TabFunc* (see *Appendix* section *Auxiliary Library* module *TabFunc*).

*Define simulation environment.* Calls the procedure which was installed by the modeller by means of *InstallDefSimEnv* (see this part chapter *Client Interface* section *Running a simulation session*, in particular procedure *InstallDefSimEnv* from module *SimMaster*). If no such installation has been done, the menu command will appear dimmed (inactive) and can not be chosen. For every additional program level at which the simulation environment has been started, an additional *Define simulation environment.* command is appended to the *Settings* menu [In the static linking versions such as the PC versions only one program level is supported]. Note that a procedure associated with the menu command has already been called at least once during the start-up of the interactive simulation environment on the respective level. Be warned that it should not erroneously contain calls to procedures, which must not be called repeatedly. For instance, the installation of an additional menu should normally only be done once when starting up the interactive simulation environment (see also chapter *Simulation environment* section *Simulations* in the previous part II *Theory* and the next chapter *Client interface* of this part).

6.1.6 MENU *WINDOWS*

This menu contains all commands which operate on windows (Fig. R12). The commands can be used to rearrange all windows, to open a window, or to bring it to the front, and to clear the graph or table window. If the simulationist closes a window, ModelWorks remembers its size and position and will reopen it at the same place it was positioned before its closing.

<b>Windows</b>	
<b>Tile windows</b>	
<b>Stack windows</b>	
-----	
<b>Models</b>	
<b>State variables</b>	⌘S
<b>Model parameters</b>	⌘P
<b>Monitorable variables</b>	⌘M
-----	
<b>Table</b>	⌘T
<b>Clear table</b>	
-----	
<b>Graph</b>	⌘G
<b>Clear graph</b>	⌘B

Fig. R12 : Menu *Windows*

*Tile windows*: All IO-windows, plus the table and graph window are closed and re-opened so that they do no longer overlap. On small screens the IO-windows for the state variables, model parameters, and monitorable variables are shown beside each other on top of the screen, on larger screens all four IO-windows are displayed in two rows on top of the screen. The remaining windows are fit into the bottom portion of the screen, making the graph window as large as possible. The column display in the IO-windows is also affected. Only the short identifiers (*ident*) and the current value columns are shown: current integration method for models, current initial values for state variables, current values for model parameters, and current monitoring settings for the monitorable variables.

*Stack windows*: All IO-windows, plus the table and graph window are closed and re-opened in a stacked way. The locations and sizes of all windows, plus the columns displayed in the IO-windows are the same as at the begin of a simulation session. However in contrast to that situation, the table and the graph window are also opened.

*Models*: The IO-window *Models* is opened respectively brought to the front.

*State variables* S: The IO-window *State variables* is opened respectively brought to the front.

*Model parameters* P: The IO-window *Model parameters* is opened respectively brought to the front.

*Monitorable variables* M: The IO-window *Monitorable variables* is opened respectively brought to the front.

*Table* T: The table window is opened respectively brought to the front. If there are no monitorable variables which have an active tabulation setting (T), some ModelWorks versions may not allow to open this window in the program state *Simulating*.

*Clear table* This command clears the table, i.e. erases the content of the table window if it is currently open.

*Graph/ G*: The graph window is opened respectively brought to the front. If there are no monitorable variables which have an active graphing setting (*Y/isY*), some ModelWorks versions may not allow to open this window in the program state *Simulating*.

*Clear graph/ B*: Clears (**l**anks) all curves in the panel of the graph if the graph window is currently open. If the latter condition is true it is the same command as *Edit/Clear*(see above *Menu Edit*).

### 6.1.7 MENU SOLVE

If a simulation is started by any of the menu commands available under this menu (Fig. R13), ModelWorks will enter the state *Simulating* (Figs. T15, T16, and T24 part II *Theory*)

Solve	
Start run	⌘R
Halt run (Pause) or Resume run	⌘H (⌘R)
Stop (Kill) run or Stop (Kill) experiment	⌘K (⌘K)
Start experiment	⌘E

Fig. R13 : Menu *Solve*

*Start run/ R*: Starts an elementary simulation run with the current settings of all values. Previously drawn curves are not erased unless demanded by a change of the graph settings since the last simulation (a curve added or removed, scaling changed). In the upper right corner, the current run number (*k*) and the current simulation time (*t*) are displayed in a small window ('*k: t*'). This command lasts as long as the simulation runs. It may be terminated by the simulationist (menu command *Stop (Kill) run*) or by the modeller, i.e. if the simulation time reaches the stop time or if the installed termination condition returns true. Note that the menu command *Halt run (Pause)* does not really terminate this command.

*Halt run (Pause)/ H* or *Resume run / R*: Temporarily halts or pauses a simulation run if the current program state is *Simulating*. The new program state entered is *Pause*. If the current program state is *Pause*, this menu command will resume the interrupted simulation run where it has been left, i.e. reenter the state *Simulating* and continue with the integration, monitoring etc.. A pause can be used to study a curve or a tabulated result in more detail, or can be used to change model parameters in the middle of a simulation. Note however, that current values of model parameters can only be modified if the flag *rtc* (run time change) has been set for that particular parameter. [Keyboard equivalent for *Resume run* is not available in the PC GEM-Version].

*Stop (Kill) run / K* or *Stop (Kill) experiment / K*: This command terminates the simulation before the simulation time reaches the stop time  $t_{end}/k_f$ . It is the only menu command within the standard simulation environment which allows to terminate a single simulation run or a structured simulation (experiment) interactively.



*Start experiment/* E: Executes the currently installed structured simulation, a so-called experiment. It enters the program state *Simulating* and calls the procedure which has been declared as the *doExperiment* procedure by the model definition program (see (see this part chapter *Client Interface* section *Simulation Control and Structured Simulation Runs*, in particular procedure *InstallExperiment* from module *SimMaster*). If no such procedure has been installed, this command will appear dimmed (inactive) and can not be chosen. For every additional program level at which the simulation environment has been started, an additional *Start experiment/* command is appended to the *Solve* menu [The PC versions support only one program level].

If the monitoring settings for the stash filing of at least one monitorable variable is set (F/*writeOnFile*), a stash file with the current stash file name is automatically opened and data are written onto it according to the current recording flags (see menu command *Project description...*). However, this behaviour depends also on the current simulation environment mode (see above menu command *File/Preferences...*). In case that the mode *Always document run on stash file* is currently active, the stash file is opened even if the stash filing is set for no monitorable variable.

**I m p o r t a n t n o t i c e :** In case there exists already a file with the same name as the current stash file name, this file will be overwritten without any warning! <sup>14</sup> Due to the nature of interactive simulation, the overwriting is quite normal and frequent and causes usually no harm. Hence, the display of an alert would be too cumbersome, but the quiet overwriting can become dangerous if the modeller programs the stash file name erroneously (see this part chapter *Client Interface* section *Display and Monitoring*, in particular procedures *SetStashFileName* and *SwitchStashFile* from module *SimBase*).

If the monitoring settings for tabulation or graphing are activated, at begin of a single simulation run or of a structured simulation the corresponding windows are automatically opened or brought to the front. If already any of the windows *Table* or *Graph* is the front most window, ModelWorks will not automatically open the other window or bring it to the front. This allows the simulationist to suppress the automatic opening or bringing to the front of either the table or graph window by bringing first the other one to the front before she starts or resumes a simulation. If there are no monitorable variables which have an active tabulation setting (T/*writeInTable*), the table window may not remain open and will be automatically closed in case it should be already open at the begin of the simulation. If there are no monitorable variables which have an active graphing setting (Y/*isY*), the graph window may not remain open and will be automatically closed in case it should be already open at the begin of the simulation. In all other cases ModelWorks leaves the windows where they are.













## 6.2 IO-Windows (Input-Output-Windows)

IO-windows serve the entering of new current values (Input) and the display of the current values (Output) of all models and model objects momentarily present in ModelWorks model base. For a description of the general operation of IO-windows see also the section on IO-windows and for the states in which IO-windows accept input see Fig. T24 in part II *Theory*.

If an IO-windows is currently active, i.e. it is the front most window and enabled, the keyboard's navigation keys (page up/down, home etc.) can be used to scroll and the cursor keys (cursor up/down) plus key A to select items respectively models or model objects (Tab. R1).

---

<sup>14</sup>When using the menu command *Select stash file...* the simulationist will always be first asked if she really wants to overwrite in case there should already exist a file with the same name as the stash file.

Key (keyboard shortcut)	Button (push button)	Requires selection	Effect
<i>Cursor up</i>	–  (  or  )	no	Selects the adjacent line respectively item (model or model object) above the momentarily selected one. In case none is momentarily selected or no selection is presently visible, the first at the top of the current page is selected. In case the item to be selected is on the adjacent page, the content of the window is implicitly scrolled by one line (equivalent to button  or  ). In case all are momentarily selected (scope <i>All</i> , Fig. T26 part II <i>Theory</i> ), none is selected.
<i>Cursor down</i>	–  (  or  )	no	Like <i>Cursor up</i> but selects the adjacent item below the momentarily selected one.
<i>Page up</i>		no	Scrolls to the adjacent page above the one shown presently, i.e. scrolls the list of items down by as many items as given by the current size of the IO-window. Note, eventual selections are not lost when scrolled out of sight, but remain active.
<i>Page down</i>		no	Like <i>Page up</i> but scrolls to the adjacent page below respectively scrolls the items up.
<i>Home</i>	– {  }	no	Jump to the very first page of the list of items (equivalent to many clicks into button  )
<i>Tail</i>	– {  }	no	Like <i>Home</i> but jumps to the very last page
A		no	Select all items including those currently not shown (scope <i>All</i> , Fig. T26 part II <i>Theory</i> ) as operand for a subsequent action such as editing current values.

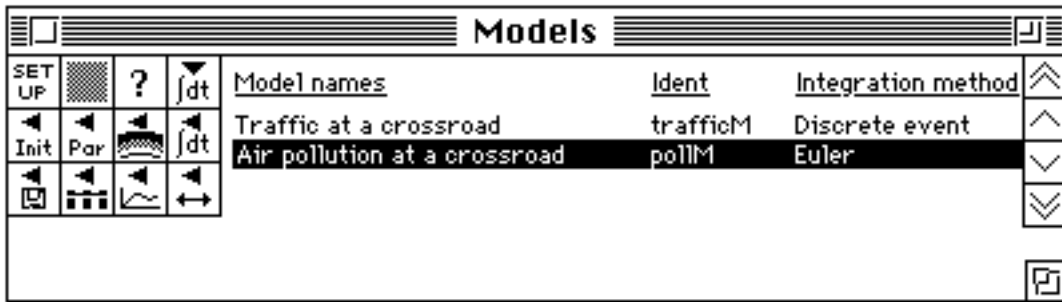
Tab. R1: In an IO-window generally available keyboard shortcuts, the equivalent palette buttons, and the corresponding effects. Note, these keyboard shortcuts are only effective if a currently enabled IO-window is actually the front most window.

By pressing one of the keys *Return* or *Enter*, an IO-window specific default action is launched, given the particular IO-window is active. Depending on the IO-window the default action allows to edit a model's integration method, a state variable's initial value, or a parameter's current value, respectively to toggle the drawing of a monitorable variable in the graph. In case of the IO-window *Monitorable variables* additional actions may be launched by pressing one of the keys *C*, *F*, *S*, *T*, *X*, or *Y* (for details see below).

Note that the keyboard shortcuts of the IO-windows differ from keyboard equivalents for menu commands, since they do not require to press the key simultaneously with the short cut.

### 6.2.1 IO-WINDOW MODELS

The IO-window *Models* (Fig. R14) displays information (name, identifier, current integration method) about the installed models. Furthermore it offers a mechanism to select models and to execute functions, which operate on the selected models as well as their model objects.

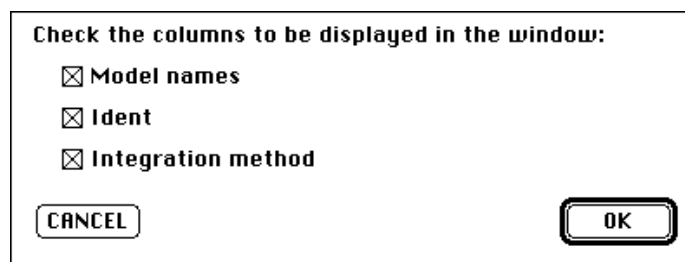



**Fig. R14** : IO-window *Models* showing the list of all (sub)models and offering a palette of button functions operating on the momentarily selected model(s) and their model objects. In the example shown the (sub)model *pollM* and implicitly all its model objects such as state variables, parameters etc. are momentarily selected.



**Columns set-up:** Activates an entry form in which the display of the columns in the IO-window *Models* can be controlled (Fig. R15). The columns of which the display can be turned on or off are:

- *Model names*: Full names of the models.
- *Ident*: Short identifiers of the models.
- *Integration method*: Current integration method.



**Fig. R15** : Entry form opened by the  button in the IO-window *Models*.



**Selects all models.** All subsequent button functions will operate on the scope *All* (Fig. T26 part II *Theory*), i.e. on all models respectively all objects of all models currently installed in the model base.

This function can also be activated by pressing key A, given IO-window *Models* is front most.



**Help respectively model information:** Opens or brings a window with the title *Model Help/Info* to the front and executes the help or about procedure (formal parameter *about*) for the momentarily selected model (works only on a single model and not on multiple selections). ModelWorks opens only the window, but writes nothing into it. It is up to the procedure *about* to write information into this window by output procedures from module *DMWindowIO*. The procedure *about* has been installed by the modeller while declaring the model (see this part chapter *Client Interface* section *Declaring Models and Model Objects*, in particular procedure *DeclM* from module *SimBase*). Typically the modeller uses this help function to inform the simulationist about some model characteristics. For instance the written information may consist of the model equations, the name of the author(s), or some help information on the model. This *Model Help/Info* window remains open until the simulationist closes it. The simulationist may close, move, or resize it freely.



*Set integration method* Opens an entry form in which the integration method for the momentarily selected model(s) can be set (Fig. R16). This is possible only for continuous time models (DESS); of course the «integration method» for discrete time models (SQM) or discrete event models (DEVS) can not be altered.

The following integration methods are available:

**Fig. R16** : Entry form to select the numerical *integration method* with which a continuous time (sub)model is to be integrated during simulations.

- *Euler*: Simple first order, one-step integration method with fixed integration step (also Euler-Cauchy method or Runge-Kutta 1st order). This method is fast, but should be used with caution as it is more likely to produce numerical errors.
- *Heun*: Second order one-step integration method with fixed integration step (also Runge-Kutta 2nd order). This method is similar to the trapezoidal rule, but differs from it inasmuch as it is not iterative.
- *Runge-Kutta-4th order*: Fourth order one-step integration method with fixed integration step.
- *Runge-Kutta-4/5th order, variable step length* 4th/5th order, variable step length Runge-Kutta-Fehlberg method (ATKINSON & HARLEY, 1983; ENGELN-MÜLLGES & REUTTER, 1988). The local error is estimated comparing the 5th order with the 4th order result. Depending on the error estimate the step length is increased or reduced to obtain optimal results in terms of efficiency and accuracy. This method is most useful for solving models with derivatives, which strongly vary during the course of a simulation. [Not available in Reflex and PC GEM-Version]

The number of times the procedure *Dynamic* is called is equal to the order of the integration method. Be aware that despite their higher computing load, high order methods are often more efficient than those of lower order. This is because higher order methods allow for larger integration steps while retaining the same accuracy. The latter may result in a reduction of the total number of times the procedure *Dynamic* has to be computed. However, since the actual performance depends also on the characteristics of the model, the best method for a particular model has often to be identified by theoretical considerations or numerical experiments.

This editing function can also be activated by pressing either the key *Return* or *Enter*, given the IO-window *Models* is front most.



*Reset integration method*: Resets the integration method of the momentarily selected model(s) to their default.



**Init** *Reset initial values:* Resets all initial values of the state variables of the momentarily selected model(s) to their default values.



**Par** *Reset parameters* Resets all parameters of the momentarily selected model(s) to their default values.



**Filing** *Reset stash filing:* Resets the monitoring settings for the stash filing (F/writeOnFile/notOnFile) of all monitorable variables of the momentarily selected model(s) to their default settings.



**Tabulation** *Reset tabulation:* Resets the monitoring settings for the tabulation (T/writeInTable/notInTable) of all monitorable variables of the selected model(s) to their default settings.



**Graphing** *Reset graphing:* Resets the monitoring settings for the graphing (X/Y/isX/isY/notInGraph) of all monitorable variables of the selected model(s) to their default settings.



**Scaling** *Reset scaling:* Resets the scaling (minimum and maximum on ordinate) of all monitorable variables of the selected model(s) to their default values.



**Curve attributes** *Reset curve attributes:* Resets all curve attributes (colour or stain, line style, and symbol) of all monitorable variables of the selected models to their default values.

### 6.2.2 IO-WINDOW STATE VARIABLES

The IO-window *State variables* displays information (name, identifier, unit, current initial value) about the installed state variables of all models (Fig. R17). Furthermore it offers a mechanism to select state variables and to execute functions, which operate on the current initial values of the selected state variables.

State variables				
SET UP	State variable names	Ident	Unit	Initial value
▼	<b>Larch Bud Moth model b1 V3.0 (Larch</b>			
Init	Raw fiber content (% fresh weigh rf		%	15.00
Init	Larch bud moth eggs (individuals) eggs	eggs	numbers	4765975

**Fig. R17** : IO-window *State variables* showing the list of all state variables and offering a palette of button functions operating on the current initial values of the momentarily selected state variables. In the example shown the model *Larch Bud Moth...* has two state variables, i.e. *rf* and *eggs*. Momentarily all state variables are selected, which has been accomplished by clicking button



**Columns set-up** Activates an entry form in which the display of the columns in the IO-window *State variables* can be controlled (Fig. R18). The columns of which the display can be turned on or off are:

- *State variable names* Full names of the state variables.
- *Ident*: Short identifiers of the state variables.
- *Unit*: Unit in which to measure the values of the state variables.
- *Initial value*: Current initial value of the state variable used to initialize the state variables at the begin of each simulation run.

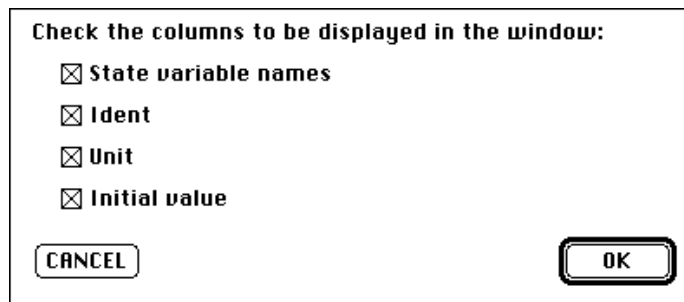


Fig. R18: Entry form opened by the  button in the IO-window *State variables*



**Selects all state variables** All subsequent button functions will operate on the scope *All* (Fig. T26 part II *Theory*), i.e. on all state variables of all models currently installed in the model base.

This function can also be activated by pressing the key *A*, given the IO-window *State variables* is front most.



**Set initial value:** Opens an entry form in which the current initial value for the selected state variable(s) can be edited. ModelWorks rejects any attempt to enter a value out of the range as defined by the modeller in the model definition program. If a multiple selection of state variables has been made, not only one but a series of entry forms will be offered, one form for each state variable. This sequence can be terminated by pressing the push-button *Cancel*. Note however, that in the latter case all changes which have been made to state variables before, can no longer be reversed (since their editing has already been made final by pressing the push button *Ok*). Only eventual changes made to the state variable momentarily in display will be discarded.

This editing function can also be activated by pressing either the key *Return* or *Enter*, given the IO-window *State variables* is front most.




**Reset initial value:** Resets the initial values of the momentarily selected state variable(s) to their default values.

### 6.2.3 IO-WINDOW *MODEL PARAMETERS*

The IO-window *Model parameters* displays information (name, identifier, unit, value, change at run time enabled/disabled) about the installed model parameters of all models (Fig. R19). Furthermore it offers a mechanism to select model parameters and to execute functions, which operate on the current values of the selected model parameters.

SET UP	Parameter names	Ident	Unit	Value
	<b>Potato model</b>			
Par	kGrowth	kGrowth	g dw/plant	50.000
Par	kLeaf	kLeaf	-	1.000
	kStem	kStem	-	6.000
	kRoot	kRoot	-	1.000
	kTuber	kTuber	-	6.000
	<b>Weather input module</b>			
	Weather input source	Inp	Year	1985.000

**Fig. R19** : IO-window *Model parameters* showing the list of all model parameters and offering a palette of button functions operating on the current values of the momentarily selected model parameters. In the example shown the first (sub)model *Potato...* has five model parameters (*kGrowth*, *kLeaf* etc.), the second *Weather...* has more than could be displayed here, thus only the first, i.e. *Inp*, is visible in this page of the IO-window. To see more parameters the list would have to be scrolled by clicking into the button  or by enlarging the size of the window. Momentarily no model parameter is selected.



**Columns set-up**: Activates an entry form in which the display of the columns in the IO-window *Model parameters* can be controlled (Fig. R20). The columns of which the display can be turned on or off are:

- *Parameter names* Full names of the model parameters.
- *Ident*: Short identifiers of the model parameters.
- *Unit*: Unit in which to measure the values of the model parameters.
- *Value*: Current value of the model parameters.
- *RTC*: (*rtc/noRtc* - runtime change/no runtime change). The value in this column shows whether a parameter may be changed during a simulation run or not. In order to warrant consistency, the modeller can prevent the changing of a parameter value in the middle of a simulation by disabling this flag (*noRtc*) when declaring the model parameter. By default, this column is not shown.

Check the columns to be displayed in the window:

Parameter names

Ident

Unit

Value

RTC

CANCEL OK

**Fig. R20** : Entry form opened by the  button in the IO-window *Model parameters*.



**Selects all model parameters** All subsequent button functions will operate on the scope *All* (Fig. T26 part II *Theory*), i.e. on all model parameters of all models currently installed in the model base.

This function can also be activated by pressing the key A, given the IO-window *Model parameters* front most.



*Set model parameter value* Opens an entry form in which the current parameter value for the selected model parameter(s) can be edited. ModelWorks rejects any attempt to enter a value out of the range as defined by the modeller in the model definition program. If a multiple selection of model parameters has been made, not only one but a series of entry forms will be offered, one form for each model parameter. This sequence can be terminated by pressing the push-button *Cancel*. Note however, that in the latter case all changes which have been made to model parameters before, can no longer be reversed (since their editing has already been made final by pressing the push button *Ok*). Only eventual changes made to the model parameter momentarily in display will be discarded.

This editing function can also be activated by pressing either the key *Return* or *Enter*, given the IO-window *Model parameters* front most.




*Reset model parameter*. Resets the parameter values of the momentarily selected model parameter(s) to their default values.

#### 6.2.4 IO-WINDOW *MONITORABLE VARIABLES*

The IO-window *Monitorable variables* displays information (name, identifier, unit, minimal and maximal value for scaling, current output selection) about the installed monitorable variables of all models (Fig. R21). Furthermore it offers a mechanism to select monitorable variables and to execute functions, which operate on the current monitoring settings, scaling, and curve attributes of the selected monitorable variables.

Monitorable variable names				Ident	Unit	Monitoring
<b>Forest submodel</b>						
		X		Biomass (dry weight) of forest	Qj	t/ha T Y
				Biomass derivative	dQj/dt	t/ha/a T
<b>Wood sector submodel</b>						
				Endurable forest products	Pj	t/ha T Y
<b>Observer submodel</b>						
				Total carbon ever fixed (pooled on	accCPool	t/ha FT
				Total carbon fixed	totCFixed	t/ha T Y
				Average total carbon fixed	avgTotCFix	t/ha T Y
<b>Harvesting submodel</b>						
				Harvested wood	Hj	t/ha T


**Fig. R21** : IO-window *Monitorable variables* showing the list of all monitorable variables and offering a palette of button functions operating on the current monitoring settings, scalings, and curve attributes of the momentarily selected monitorable variables. In the example shown the third (sub)model *Observer...* is momentarily selected, which implies that actually all its three monitorable variables (*accCPool*, *totCFixed*, *avgTotCFix*) are selected. Any subsequent action affects only these selected objects. For instance the button  would toggle the graphing according to the current setting of the first monitorable variable (*accCPool*) and copy that result to all remaining selected objects (*totCFixed*, *avgTotCFix*); here the result would then be that during the next simulation all monitorable variables of submodel *Observer* will be plotted in the window *Graph*.





**Columns set-up:** Activates an entry form in which the display of the columns in the IO-window *Monitorable variables* can be controlled (Fig. R22). The columns of which the display can be turned on or off are:

- *Monitorable variable names* Full names of the monitorable variables.
- *Ident:* Short identifiers of the monitorable variables.
- *Unit:* Unit in which to measure the values of the monitorable variables.
- *Minimum scaling:* Lower value used to scale the values of the monitorable variable on the ordinate of the graph. By default this column is not shown.
- *Maximum scaling:* Upper value used to scale the values of the monitorable variable on the ordinate of the graph. By default this column is not shown.
- *Monitoring:* This column shows the current output settings, where:
  - F: Monitorable variable is written onto the stash file
  - T: Monitorable variable is tabulated in the table
  - X: Monitorable variable is used as independent or abscissa variable (x-values). If there is no monitorable variable selected as the abscissa variable, ModelWorks provides the so-called default independent variable, the simulation time.
  - Y: Monitorable variable is used to draw a curve. Its values are drawn as ordinate values (y-values) versus the current independent variable (x-values).

Fig. R22 : Entry form opened by the  button in the IO-window *Monitorable variables*.



**Selects all monitorable variables** All subsequent button functions will operate on the scope *All* (Fig. T26 part II *Theory*), i.e. on all monitorable variables of all models currently installed in the model base.

This function can also be activated by pressing the key A, given the IO-window *Monitorable variables* is front most.



**Set/delete stash filing (F/writeOnFile/notOnFile):** Adds the selected monitorable variable(s) to the list of variables which are to be written onto the stash file. The function toggles actually the setting, i.e. if the current setting is on (F) it is disabled, otherwise enabled. In the monitoring-column of the IO-window the monitorable variables to be written onto the stash file are marked with an F (Fig. R21). If a multiple selection is active, the function adds or removes *all* momentarily selected variables to or from the list, reversing the current setting of the first variable in the selection.

This toggling function can also be activated by pressing the key F, given the IO-window *Monitorable variables* is front most.



**Reset stash filing:** Resets the stash filing of the momentarily selected monitorable variable(s) to their defaults.

**Tabulation:** Generally note, that in case that the simulationist makes any fundamental changes to the table such as removing or inserting columns, ModelWorks redraws the table not later than the begin of the next simulation run. The actual redrawing time (immediate or deferred) is determined by the current settings of the mode *Once changed, immediately redraw table (RedrawTableAlwaysMode)* of the simulation environment (see also menu command *File/Preferences*).



**Set/delete tabulation (T/writeInTable/notInTable):** Adds the selected monitorable variable(s) to the list of variables which are to be tabulated in the table window. The function toggles actually the setting, i.e. if the current setting is on (T) it is disabled, otherwise enabled. In the monitoring-column of the IO-window the monitorable variables to be tabulated are marked with an T (Fig. R21). If a multiple selection is active, the function adds or removes *all* momentarily selected variables to or from the list, reversing the current setting of the first variable in the selection.

This toggling function can also be activated by pressing the key T, given the IO-window *Monitorable variables* is front most.



**Reset tabulation:** Resets the tabulation of the momentarily selected monitorable variable(s) to their defaults.

**Graphing:** Generally note, that in case that the simulationist makes any fundamental changes to the graph such as a redefinition of scales (may also be caused by changing the start ( $t_o/\kappa_o$ ) or stop ( $t_{end}/\kappa_f$ ) time, see menu command *Settings/Set Global simulation parameters.*), removal or new activation of curves, ModelWorks redraws the graph never later than the begin of the next simulation run. The actual redrawing time (immediate or deferred) is determined by the current settings of the mode *Once changed, immediately redraw graph (RedrawGraphAlwaysMode)* of the simulation environment (see also menu command *File/Preferences*).



**Set/cancel variable as x-axis in the graph (X/isX):** Sets the selected variable as independent or abscissa variable (x-values) for the graph. The function toggles actually the setting, i.e. if the current setting is on (X) it is disabled, otherwise enabled. In the monitoring-column of the IO-window the monitorable variable to be used as abscissa variable is marked with an X. If another variable was already selected as the abscissa variable, that is automatically deselected, since ModelWorks allows only one independent variable at a time. In case that no monitorable variable is selected as abscissa variable, ModelWorks uses the default independent variable *time*. This function will completely erase and redraw the content of the graph window. This is because it has to redraw the x-axis. This function does not work on a multiple selection.

This toggling function can also be activated by pressing the key X, given the IO-window *Monitorable variables* is front most.



*Set/delete curve (Y/isY)*: Adds the selected monitorable variable(s) to the list of variables which are to be drawn as curves in the graph window. The function toggles actually the setting, i.e. if the current setting is on (Y) it is disabled, otherwise enabled (Fig. R21). In the column *Monitoring* of the IO-window the symbols F, T, or Y are shown in the same colour as that which is used to draw values of the corresponding variable in the graph. If a multiple selection is active, the function adds or removes all momentarily selected variables to or from the list, reversing the current setting of the first variable in the selection (for an example see Fig. R21). This function will completely erase and redraw the content of the graph window, if the mode *Once changed, immediately redraw graph (RedrawGraphAlwaysMode)* of the simulation environment is currently active (see also *File/ Preferences*).

This toggling function can also be activated by pressing either the key *Return*, *Enter*, or *Y*, given the IO-window *Monitorable variables* is front most.



*Reset graphing*: Resets the graphing (*XN/isX/isY/notInGraph*) of the selected monitorable variable(s) to their defaults.




*Set scaling*: Opens an entry form in which minimum and maximum values for the scaling of the selected monitorable variable in the graph can be edited (Fig. R23).

**Scaling of the following monitorable variable:**

**Grass**

Min:  Max:

**Fig. R23** : Entry form opened by the  button in the IO-window *Monitorable variables*. It allows to set the scaling of a particular monitorable variable in the window *Graph*. Only values within these limits will be displayed, i.e. all parts of the curve which are outside the range [ Min, Max ] will be clipped.

If a multiple selection of monitorable variables has been made, not only one but a series of entry forms will be offered, one form for each monitorable variable. This sequence can be terminated by pressing the push-button *Cancel*. Note however, that in the latter case all changes which have been made to variables before, will not be reversed (since their editing has already been made final by pressing the push button *Ok*). Only eventual changes made to the monitorable variable momentarily in display will be discarded. This function causes sooner or later a complete erase and redrawing of the content of the graph window, because it always affects the legend. The actual redrawing takes place according to the current settings of the mode *Once changed, immediately redraw graph (RedrawGraphAlwaysMode)* of the simulation environment (see also *File/ Preferences*).

This editing function can also be activated by pressing the key *S*, given the IO-window *Monitorable variables* is front most.



*Reset scaling*: Resets the scaling (minimum and maximum) of the selected monitorable variable(s) to their default values.



*Set curve attributes*: Opens an entry form in which the attributes for the drawing of a monitorable variable's curve in the graph window can be edited (Fig. R24). This editing serves to set or override the automatic definition of curve attributes by ModelWorks. [Since no colours are available in the PC GEM-Version any settings of the curve attribute stain (colour) is without effect].

**Curve attributes of the following monitorable variable:**

**Grass derivative**

**automatic definition of curve attributes**

or use following curve attributes:

<input type="radio"/> <b>unbroken</b>	<input checked="" type="radio"/> <b>coal</b>	<input type="radio"/> <b>snow</b>
<input type="radio"/> <b>broken</b>	<input type="radio"/> <b>ruby</b>	<input type="radio"/> <b>sapphire</b>
<input type="radio"/> <b>dashSpotted</b>	<input type="radio"/> <b>emerald</b>	<input type="radio"/> <b>pink</b>
<input type="radio"/> <b>spotted</b>	<input type="radio"/> <b>turquoise</b>	<input type="radio"/> <b>gold</b>
<input type="radio"/> <b>invisible</b>		
<input type="radio"/> <b>purge</b>	<input checked="" type="checkbox"/> <b>symbol</b>	

Fig. R24 : Entry form opened by the button in the IO-window *Monitorable variables* used to edit curve attributes.

The following curve attributes are available: A colour with which curves are drawn on colour screens, printed on colour printers, or exposed on slide recorders. Note, the colours are effective regardless of the current screen; for instance, despite a black-and-white screen, once set, a colour curve attribute, e.g. *ruby*, will result in the printing of a red curve when the graph is printed on a colour printer. Line styles affect the style with which connecting lines between points are drawn. Points can be emphasized by drawing plotting symbols.

The colours are *coal* (black), *snow* (white), *ruby* (red), *emerald* (green), *sapphire* (blue), *turquoise* (cyan), *pink* (magenta), and *gold* (yellow). The graph monitoring procedure produces line charts, i.e. points are connected with lines, which can be drawn with one of the following styles:

<i>unbroken</i>	_____
<i>broken</i>	-----
<i>dashSpotted</i>	-.-.-.-.-
<i>spotted</i>	.....
<i>invisible</i>	no drawing of lines at all, may be used to draw scatter plots or to stop the drawing of a particular section of a curve
<i>purge</i>	used to erase already drawn curves
<i>autoDefStyle</i>	line style will be determined by ModelWorks according to the automatic definition mechanism of curve attributes

Any character can be used as a plotting symbol, for instance using the plotting symbol "\*" results in curves like ---\*---\*---. Note that using a blank (or 0C) will result in the drawing of no plotting symbol at all.

If automatic definition of curve attributes is active for one or several monitorable variables, ModelWorks follows a strategy to distribute four colours (stains), four line styles and four symbols to draw curves (see Tab. T1 in part II *Theory*) for these monitorable variables. This strategy helps the simulationist to tell curves better apart, regardless whether a particular graph is displayed on a colour screen or is printed on a black and white printer only. Colours, line styles, and symbols are distributed among the monitorable variables which currently have the automatic definition of curve attributes setting active. Which attribute, e.g. colour, is used for which curve is influenced by the position of a monitorable variable within the chronological sequence in which it has been activated (Y-toggling) for the graphing. For instance if four monitorable variables have been activated for graphing, the first activated will automatically be drawn in black, the second in red, the third in blue, and the fourth in green. However, if the second is removed from the monitoring, the previously third will be drawn in red and the previously fourth in blue. If the previously second is now reactivated, it will be drawn in green (no longer red)!

In case this automatic definition does not please the simulationist, she may override it anytime and use a particular colour, line pattern, and marking symbol for a curve of a given monitorable variable. This allows e.g. to use always the same colour for the same variable, such as green for state variable *Grass* or blue for a water level. Mark symbols are characters drawn exactly at the points as defined by the monitoring, line patterns are used to connect these points with lines. With this technique it is also possible to draw only scatter-grams, instead of line charts (line style = *invisible*). The colour currently in use for a particular monitorable variable is not only shown in the legend and used to draw curves in the graph, but also used to display the current monitoring settings (F, T, or Y) in the column *Monitoring* of the IO-window *Monitorable variables* (of course only visible on a colour screen).

In order to toggle between automatic definition and its overriding, the simulationist must click into one of the line style radio buttons or into the radio button *Automatic definition*. In particular note, that in the current version of ModelWorks it is not sufficient to select just another colour (stain) to turn automatic definition off without selecting simultaneously also a new line style. This is because automatic definition can not be set individually for the various curve attributes but hold for all curve attributes of a monitorable variable at once; either the curve attributes of a particular monitorable variable are all defined automatically or all are user defined.

If a multiple selection of monitorable variables has been made, not only one but a series of entry forms will be offered, one form for each monitorable variable. This sequence can be terminated by pressing the push-button *Cancel*. Note however, that in the latter case all changes which have been made to variables before, can no longer be reversed (since their editing has already been made final by pressing the push button *Ok*). Only the eventual changes made to the monitorable variable momentarily in display will be discarded.

This function causes sooner or later a complete erase and redrawing of the content of the graph window, because it always affects the legend. The actual redrawing takes place according to the current settings of the mode *Once changed, immediately redraw graph* (*RedrawGraphAlwaysMode*) of the simulation environment (see also menu command *File/Preferences*).

This editing function can also be activated by pressing the key C, given the IO-window *Monitorable variables* is front most.



*Reset curve attributes:* Resets the curve attributes of the momentarily selected monitorable variable(s) to their default values.

## 7 Client Interface

The client interface consists of a mandatory part and an optional part (Fig. T29 part II *Theory*). The mandatory part consists of the two modules *SimBase* and *SimMaster*, the optional part of the modules *SimDeltaCalc*, *SimEvents*, *SimGraphUtils*, *SimIntegrate*, and *SimObjects*. Although every model definition program must import from both modules of the mandatory part, only a small subset of the exported Modula-2 objects are actually needed always. These few Modula-2 Objects, five procedures and six data types, form the core of the ModelWorks client interface (Fig. T29 Part II *Theory*).

All other types and procedures also exported from this interface are optional. Their purpose is either to serve the convenience of the simulationist or to support the modeller in the programming of advanced structured simulations. For instance, if the simulationist wishes to run a model in a time range different from the one predefined by ModelWorks, the modeller can overwrite the ModelWorks predefined defaults (Tab T1 part II *Theory*) with the values the simulationist prefers. This is much more convenient as if the simulationist would always have to assign the desired values interactively at the begin of each simulation session. If a simulation study has advanced to a later stage, it is often desirable to be able to run multiple simulation runs in a systematic, well defined way. The interactive control of the simulations becomes then rather an obstacle than a help. On the other hand, if much effort has been invested in the development of a complex model it is desirable to be able to run structured simulations under program control using the same model implementation. To support the modeller in such tasks is exactly the purpose of most of the additional objects exported by the client interface.

The principles behind the usage of the client interface have been described elsewhere (Manual Part II *Theory*, in the chapter *Modelling*). Please consult also the listings of the client interface definition modules<sup>1</sup> plus the sample model definition programs in the appendix while reading the following explanations of the Modula-2 objects exported by the client interface.

The two **mandatory client interface** modules which are explained later in more detail serve the following purposes:

*SimBase*: Provides procedures for the declaration or modification of models and model objects; provides control over global simulation parameters, project description, recording and monitoring options, windows, various preferences, menu key board equivalents and simulation environment modes; allows to specify defaults or current values and to reset the current values to the defaults.

*SimMaster*: Exports procedures for starting of the interactive simulation environment and for the control of single simulation runs or of structured simulations (experiments).

The following five modules constitute the **optional client interface** of ModelWorks and will only be briefly described as follows<sup>2</sup>:

---

<sup>1</sup>The appendix contains the definitions of all optional ModelWorks modules and of all auxiliary library modules listed below. The definitions of *SimBase* and *SimMaster* are not in the appendix since all objects exported by these modules are discussed in the following chapters.

<sup>2</sup>For more details on the functions provided by these modules refer to the listings of the definition modules in the appendix.

*SimDeltaCalc*: Provides utilities to calculate deviations between simulated and observed time series. It is typically used in model validations or in model parameter identifications.

*SimEvents*: Supports discrete event simulations.

*SimGraphUtils*: Provides utilities to make output to the graph window and the graph such as drawing of additional curves and displaying of validation data at discrete time points with or without error bars.

*SimIntegrate*: Provides means to integrate autonomous models without any monitoring and without affecting the global simulation time of the simulation environment.

*SimObjects*: Allows for a lower-level, efficient access to models and model objects and also for attaching of additional attributes to these objects. Using *SimObjects* may improve performance of object management substantially, because the mandatory module *SimBase* stresses safety over efficiency and traverses the internal data base before each access: when calling e.g. *SetP* or *SetSV*, the owner model's existence is always tested and the parameter or state variable is searched in the model's object list. Hence, using these procedures from within a loop may be very tedious and cumbersome. To better support object access, for instance in order to change the default and current values of model objects in a loop, you may better use the access mechanisms provided by *SimObjects*. CAUTION: wrong usage of these mechanisms can have serious consequences! Particularly do not remove or add any objects, nor change the addresses of any objects in the lists available. Usage of this module is recommended for advanced programmers only.

The following modules belong to the **auxiliary library**. They may also be used independently of ModelWorks, but are briefly described below since they are frequently needed in research involving modelling and simulation studies<sup>3</sup>:

*Identification*: to identify model parameters of a ModelWorks model definition program. It minimizes a performance index (see also *SimDeltaCalc*) between a given particular model behavior and the current model behavior by various minimization methods (for an example see *Appendix* sample model *GauseIdentif*).

*JulianDays*: Provides calendar procedures to convert calendar dates into Julian days and vice versa. Julian days are needed for calculations on dates, e.g. to compute an elapsed time between two dates.

*RandGen*: Contains a pseudo-random number generator for variates uniformly distributed within interval (0,1] (for examples see *Appendix* sample models *Diversity*, *Markov*, *StochLogGrow*, or *CarPollution*).

*RandNormal*: provides a pseudo-random number generator for normally distributed variates  $\sim N(\mu, \sigma)$  (for an example see *Appendix* sample model *StochLogGrow*).

*ReadData*: Allows to read data from an input file and to test various conditions (such as a minimum-maximum range) in an easy way. It is typically used to enter measurements stored on text data files into ModelWorks, for instance to compare simulated with observed data (for an example see *Appendix* sample model *SwissPop*).

---

<sup>3</sup>For more details on the functions provided by these modules refer to the listings of the definition modules in the appendix.

*StructModAux*: Provides support for the implementation of structured models where the submodels reside in separate modules. The module allows to install a custom menu to activate/deactivate models or submodels, and supports the maintenance of the global simulation parameters by a master model definition program. (for examples see *Appendix* sample models *GreenHouse* or *LBM*)

*TabFunc*: Allows for usage and graphical editing of non-linear functions which are defined by piece wise linear inter- resp. extrapolation according to a table of x-y values (for examples see *Appendix* sample models *Swiss Pop* or *UseTabFunc*).

*WriteDatTim* Can be used to write dates and times as accessed by means of *DMClock*. This may be useful to record begin and end of a long simulation experiments (for an example see *Appendix* sample model *Markov*).

Since ModelWorks has been designed as an open architecture and is based on Modula-2, the modeller is free to extend the auxiliary library by any module she wishes. In particular, there is also the possibility to use any object from the "Dialog Machine". The majority of the latter is contained in the kernel of ModelWorks and resides together with any model definition program already in the memory. Hence this part of the "Dialog Machine" can be used by the modeller without any penalty. Those modules of the "Dialog Machine" which are not in use by ModelWorks can then be considered as a particular extension of the auxiliary library (Fig. T29 in part II *Theory*).

## 7.1 Declaring Models and Model Objects

This section describes the core of the ModelWorks client interface, i.e. those procedures and types which are used by every model definition program.

Fundamental functions, for instance the activation by the standard, interactive simulation environment or states of the environment, are exported by module *SimMaster*. The instantiation respectively declaration of models and model objects, the accessing, i.e. retrieval and new setting, of these objects, as well as the removal of them are functions exported by module *SimBase*.

### 7.1.1 RUNNING A SIMULATION SESSION

The standard, interactive simulation environment is started whenever a program calls the procedure from the client interface module *SimMaster*.

```
PROCEDURE RunSimEnvironment( initSimEnv: PROC );
```

This is the only statement which the model definition program must execute in order to use the standard ModelWorks interactive simulation environment. The argument *initSimEnv* refers to a procedure which will typically declare the models including all their model objects by calling the procedure *DeclM* from module *SimBase* (s.a. below section *Declaration of model*). The procedure often contains also calls to procedures which set defaults for the global simulation parameters such as *SetDefltGlobSimPars*. Upon returning from this procedure, all elements of the standard user interface (menus and windows) are removed, but the calling program may proceed with the full ModelWorks functionality still present at the client interface.

It is also possible to use *RunSimEnvironment* only for the installation of extra menus and menu commands in the "Dialog Machine" to expand the standard simulation environment. For instance the procedure may contain no calls to any model object declarations at all, but the menu commands it installs allow for the activation of models, since they are bound to procedures which call the model and model object declaration procedures *DeclM*, *DeclSV*, *DeclP*, and *DeclMV* (s.a. below section *Declaration of model*). There may also be menu command



procedures installed which remove models, so that a full dynamic model loading and unloading becomes possible during a simulation session (for an example see the research sample model *Population dynamics of larch bud moth* in the *Appendix*). Any menu installation called within procedure *initSimEnv* will be placed on the right side of the menu bar as it is installed by ModelWorks' simulation environment. After the execution of the *initSimEnv* procedure *RunSimEnvironment* starts the "Dialog Machine" by calling procedure *RunDialogMachine* from module *DMMaster*.

The modeller can declare a "simulation environment definition (or customization) procedure" *defineSimEnv* to ModelWorks by calling *DeclDefSimEnv* from *SimMaster*. ModelWorks will then call this procedure after having called *initSimEnv* and after having performed a global reset. Execution *defineSimEnv* is the last action performed at starting up of the interactive environment. Also, it represents the first event that will be handled by the "Dialog Machine".

```
PROCEDURE InstallDefSimEnv( defineSimEnv: PROC );
```

The installed *defineSimEnv* procedure will be also executed whenever the simulationist chooses the command *Settings/Define simulation environment.*, e.g. in order to (re)show a window, to read data (anew) from a file, or to customize monitoring settings by means of calls to *SetMV*.

The currently installed *defineSimEnv* procedure may be executed from the client interface by calling

```
PROCEDURE ExecuteDefSimEnv;
```

Since the standard simulation environment may not be started more than once on the same program level, the procedure

```
PROCEDURE SimEnvRunning( progLevel: CARDINAL ): BOOLEAN;
```

allows to find out whether the simulation environment is currently running on a particular program level or not.

### 7.1.2 DECLARATION OF MODELS

Models are represented in ModelWorks by means of variables of the type

```
TYPE Model;
```

It is recommended to initialize all model-variables in the body of a model definition program using the variable

```
VAR notDeclaredModel: Model; (* read only variable *)
```

as follows:

```
MODULE MyModelDefProg;
...
VAR
  myModel: Model;
...
BEGIN
  myModel := notDeclaredModel;
...
END MyModelDefProg.
```

A model or a submodel is declared to ModelWorks or installed in the interactive simulation environment with the procedure *DeclM*:

```
PROCEDURE DeclM (VAR m: Model;
  defaultMethod: IntegrationMethod;
  initialize, input, output, dynamic, terminate: PROC;
  declModelObjects: PROC;
  descriptor, identifier: ARRAY OF CHAR;
  about: PROC);
```

This procedure can be called any number of times, but should be called for each model only once<sup>4</sup>, unless it has been removed in the meantime. Normally, *DeclM* will be called in the states *No Model* or *No Simulation* of the simulation environment. However, it may also be called in any other state of the simulation environment. Hereby *DeclM* may be called either from within a client procedure (e.g. *initialize* or during integration (*dynamic*) of an other model), or by the simulationist, based on an extension of the standard user interface (e.g. by means of a separately installed menu) (s.a. part II *Theory* Fig. T15, T16 and Tab. T4).

*m* is a variable of the opaque type *Model* exported by *SimBase*. It may be used for further references to the model, e.g. when accessing a model in order to change its integration method with procedure *SetM*. It must be declared in the model definition program. It does not matter where, but *m* must be a global variable which exists as long as the model definition program.

```
IntegrationMethod = (Euler, Heun, RungeKutta4,
  RungeKutta45Var, stiff,
  discreteTime, discreteEvent);
```

*defaultMethod* is the default integration method with which the model will be solved during simulations<sup>5</sup>. Moreover the modeller defines with this parameter also the type of model, i.e. whether it is a continuous time, a discrete time, or a discrete event model. If the method *discreteTime* or *discreteEvent* is specified, the model is declared as a discrete time or discrete event model, respectively. Note that models of type *discreteEvent* must not be declared by means of *DeclM* but should be declared with the corresponding procedure *DeclDEVM* from module *SimEvents*<sup>6</sup>. All remaining integration methods are used for the class of the continuous time models. The default integration method is (re)assigned to the current integration method by ModelWorks when the model is declared, at starting up of the interactive simulation environment<sup>7</sup>, or after every reset of the integration methods. During a simulation session the current integration method may be changed by the modeller via the procedure *SetM*, or by the simulationist via the models IO-window. Note that this mechanism makes it possible, that every continuous time model uses a different integration method. Though the simulationist may change only the integration method of continuous time models, the modeller may do so for all types of models; e.g. she may change a discrete time model to a continuous time model and vice versa at any time using *SetM* or preferably *SetDefltM* (the latter is preferable since it implies a change of the model equations, thus the procedure *dynamic* needs also to be changed).

The following five formal procedure parameters are procedures which will be called by ModelWorks during simulations

---

<sup>4</sup> Should *DeclM* be called for the same model variable *m* more than once, ModelWorks will display an error message and the simulation program will be halted.

<sup>5</sup> Integration method *stiff* is not available in the current implementation of ModelWorks, such that any attempts to assign this method to a model will lead to an error message and a halt of the simulation program.

<sup>6</sup> See the description of the definition module in the *Appendix*, section *ModelWorks Optional Client Interface*.

<sup>7</sup> Reset is however not performed if the interactive simulation environment is already running and is only restarted on a new program level.

*initialize* is called only once at the begin of each simulation run (Fig. T18 part II *Theory*). It may be used freely to execute any task at the begin of a run, such as opening a file for writing data during the simulation run or assigning new initial values to state variables by calling *SetSV*.

*input* calculates the input variables of model *m* (Eq. 4.4 resp. 5.4). It is called only once during a time step, but many times during a simulation run (Fig. T19).

*output* calculates the output variables of model *m* (Eq. 4.2 resp. 5.2). It is called only once during a time step, but many times during a simulation run (Fig. T19). Note the implementation restriction that output variables must not depend directly on input variables (see Manual Part II *Theory*, chapter *Model formalisms*).

*dynamic* contains the model equations of model *m* (Eq. 4.1 resp. 5.1 or 8a, 8b, and 8c). In the case of a continuous time model it calculates the new derivatives from the current values of the state variables (Eq. 4.1 or 8a and 8c). Depending on the order of the integration method, this procedure is called at least once up to several times during a time step. In the case of a discrete time model (Eq. 5.1 or 8b and 8c) it calculates the new state vector directly and is only called once during a time step (Fig. T19).

*terminate* is called once at the end of each simulation run (Fig. T18). It may be used freely to execute any task at the end of a run, such as closing a file which has been written during the simulation run etc.

*declModelObjects* declares all model objects, i.e. state variables, model parameters, and monitorable variables of model *m*. Typically this procedure contains calls to the procedures *DeclSV*, *DeclP*, and *DeclMV*. It is also possible to leave the body of this procedure empty, e.g. by using *NoModelObjects*, and to defer all model object declarations to a later time (note that this requires proper programming of such a feature, since it is not available in the standard simulation environment).

See to it that calculations, which should be performed only once per time step, are included in the procedure *input* or *output* only, not in the procedure *dynamic*, which may be called more than once per time step. For further information on the correct use of these procedures see part II *Theory*, chapter *Simulations and the Run-Time System* in particular section *Integration respectively time step* (s.a. Figs. T19-T22).

Note that if ModelWorks resides in either the state *No Model* or *No Simulation*, *DeclM* calls the *declModelObjects*-procedure only. However, if *DeclM* is called in one of the states *Simulating* or *Pause*, it behaves differently and calls more client procedures than just *declModelObjects*. In order to synchronize the newly declared model with the already existing ones, one or several of the client procedures *initialize*, *output*, *input*, or *dynamic* will also be called. The calling sequence will depend on which part of the model integration loop is currently executed (see part II *Theory*, chapter *Simulations and the Run-Time System* in particular Figs. T19-T22, section *Manipulating the model base at run-time* and Tab. T4)<sup>8</sup>.

The last three formal procedure parameters are used to identify and describe a model, so that the simulationist may recognize it during simulation sessions:

*descriptor* String containing a long description of the model *m*

---

<sup>8</sup>For example, if a model is declared from within an other model's *terminate* procedure, the procedures *initialize*, *output*, *input* and *dynamic* of the new model will be called by *DeclM* (*dynamic*: may be hereby called more than once, depending on the model's integration method). Then the run-time system will proceed with the termination of the run by calling the *terminate* procedures of any remaining, already declared models, followed by the *terminate* procedure of the new model, which is always inserted at the end of the models list.

*identifier* Short string identifying the model *m*. Although there is no limit to the actual size of this string, it is advisable to keep it as short as possible.

*about* Procedure allowing to write information about the model, e.g. by using routines such as *WriteString*, *WriteLn* etc. from the "Dialog Machine" module *DMWindowIO*, into the help window. This procedure is called for the currently selected model whenever the simulationist clicks into the *Help/model information* button of the *Models* IO-window (s.a. chapter 6.2.1, Part III, *Reference/User Interface*).

If the modeller wishes to share the same *about* procedure for several models, the actually requested model may be found out by means of

```
PROCEDURE CurAboutM(): Model;
```

which returns the reference to this model. If called outside an *about*-procedure, *CurAboutM* will return *notDeclaredModel*

For more convenience the following procedures from module *SimBase* with an empty body can be used as actual arguments when calling *DeclM*.

```
PROCEDURE NoInitialize;
PROCEDURE NoInput;
PROCEDURE NoOutput;
PROCEDURE NoDynamic;
PROCEDURE NoTerminate;
PROCEDURE NoModelObjects;
PROCEDURE NoAbout;
PROCEDURE DoNothing;
```

In particular, the currently set procedures *initialize*, *input*, *output*, *dynamic*, *terminate* and *about* may be dynamically changed by calling *SetDefltM* at a later stage (see below).

Once a model has been declared, it is ready for the declaration and the attaching of model objects to it. Typically model object declaration procedures are called immediately following the call to *DeclM*. However, the following procedure can be used to change this behaviour, so that model objects can be attached to models in any sequence:

```
PROCEDURE SelectM (m: Model; VAR done: BOOLEAN);
```

The latter is particularly important if models and their model objects are declared dynamically during a simulation session. This feature is not supported by the standard simulation environment, but such an extension can be easily programmed via the client interface by the modeller. The modeller will then use procedure *SelectM* to attach model objects to the proper model.

### 7.1.3 DECLARATION OF STATE VARIABLES

The following types should be used to declare state variables and their derivatives or new states:

```
TYPE StateVar = REAL; Derivative = REAL; NewState = REAL;
```

These types are equal to and fully compatible with the type *REAL*. Their usage is recommended since the identifiers are self-documenting and enhance the readability of the model definition program.

State variables are declared as variables of the type *StateVar* in the model definition program, derivatives are of type *Derivative*. In case of discrete time models it is recommended to use the types *StateVar* for  $x(k)$  and *NewState* for  $x(k+1)$ . They may be declared anywhere in the

program and may be part of a structured data type. E.g. the following variables  $x$  and  $z$  may be used as state variables respectively state vector:

```
CONST MaxStateVars = 16;
VAR
  x   : StateVar;
  xDot: Derivative;
  z   : ARRAY [1..MaxStateVars] OF StateVar;
  zDot: ARRAY [1..MaxStateVars] OF Derivative;
```

In order to declare a state variable  $s$  to ModelWorks and install it in the simulation environment, the procedure *DeclSV* must be called. This procedure may not be called another time with the same variable  $s$  (see below), unless the state variable has been removed in the meantime. Typically, *DeclSV* is called in the states *No Model* or *No Simulation* of the simulation environment (s.a. part II *Theory* Fig. T15). This is done either from within the parent model's *declModelObjects* procedure, or at some later time point, once the parent model has been declared.

```
PROCEDURE DeclSV(VAR s: StateVar; VAR ds: Derivative (*or NewState*);
  defaultInitial, minCurInit, maxCurInit: REAL; descriptor,
  identifier, unit: ARRAY OF CHAR);
```

The state variable will belong to the last declared model (procedure *DeclM*), unless the procedure *SelectM* has been called to select another model. The meanings of the formal procedure parameters of *DeclSV* are:

$s$  Variable to be declared as a state variable. *DeclSV* assigns to  $s$  the value *defaultInitial*. The real variable  $s$  can be declared anywhere in the model definition program and may be even part of any data structure. However make sure that it is declared as a global variable and that it does exist as long as the model definition program.

$ds$  Variable to be declared as the derivative  $ds/dt$  (for continuous time models using time  $t$  as independent variable) or the new value  $s(k+1)$  (for discrete time models using time  $k$  as independent variable) of  $s$ . For every state variable the derivative or the new value must be assigned to this variable by the procedure *dynamic*, which is called during numerical integration. Normally  $ds$  appears only on the left side of the dynamic equations in procedure *dynamic*. *DeclSV* assigns to  $ds$  the value 0.0.

*defaultInitial* Default initial value for state variable  $s$ . ModelWorks uses the current initial value at the beginning of each simulation run to initialize  $s$ . The default initial value is assigned to the current initial value at declaration of the state variable, at starting up of the interactive simulation environment<sup>9</sup>, or after every reset of the model's state variables. The default initial value may be changed only by the modeller, using procedure *SetDefltSV*, whereas the current initial value may be changed by both, the modeller and the simulationist, via the procedure *SetSV*, or via the state variables IO-window, respectively. Note that in case the modeller directly overwrites variables within procedure *initialize* (see procedure *DeclM*), the current initial value displayed in the IO-window will be inconsistent with the initial value actually used in the simulation. This is because ModelWorks assigns the current initial value to  $s$  just before the *initialize* procedure is called. Avoid direct overwriting of  $s$  and use the procedure *SetSV* instead.

*minCurInit, maxCurInit* Lower and upper bounds for the current initial value. Attempts by the simulationist to assign values out of this range are not accepted.

---

<sup>9</sup>Reset is however not performed if the interactive simulation environment is already running and is only restarted on a new program level.

*descriptor* String containing a long description of the state variable  $s$ . This string may have any length, but might not be visible till its end when it is too long to fit into the IO-window column where it is displayed during a simulation session (see also *identifier*). Example: "Density of alga *Scenedesmus obliquus*".

*identifier* Short string identifying the state variable  $s$ . This string should be kept as small as possible in order to ensure full visibility for the display in small IO-windows during a simulation session. In particular on small screens, IO-windows become small in the tiled window position (see menu command *Tile windows*) and they will display only this *identifier* to denote the state variable  $s$ . Example: "sa".

*unit* String containing the unit used to measure values of the state variable  $s$ . This string is displayed in IO-windows during a simulation session. Example: "cells/ml".

#### 7.1.4 DECLARATION OF MODEL PARAMETERS

A time invariant model parameter  $p$ , which the simulationist should be able to change interactively during simulation sessions, should be declared as being of the type

```
TYPE Parameter = REAL;
```

Declaration is done by means of the procedure

```
PROCEDURE DeclP(VAR p: Parameter; defaultVal, minCurVal, maxCurVal: REAL;
  runTimeChange: RTCType; descriptor, identifier,
  unit: ARRAY OF CHAR);
```

*DeclP* may not be called with the same variable  $p$ , unless the model parameter has been removed in the meantime. It will typically be used in a similar manner as already described for *DeclSV* above. The value of the parameter  $p$  can be changed within the range [ $minCurVal$ ,  $maxCurVal$ ], and be reset to its default value *defaultVal*. A parameter change in the middle of a simulation run can lead in some application to data inconsistencies. It can therefore selectively be allowed or prevented with the parameter *runTimeChange* of the type

```
TYPE RTCType = (rtc, noRtc);
```

The meanings of the parameters of procedure *DeclP* are:

$p$  Variable of type *Parameter* to be declared as model parameter. *DeclP* assigns to  $p$  its default value *defaultVal*. The real  $p$  can be declared anywhere in the model definition program and may be even part of any data structure. However make sure that it is declared as a global real variable and does exist as long as the model definition program.

*defaultVal* Default value for the model parameter  $p$ . The default value is (re)assigned to the current parameter value  $p$  by ModelWorks at the parameter's declaration, at starting up of the interactive simulation environment<sup>10</sup>, or after every reset of the model parameters. During a simulation session the current parameter value  $p$  may be changed by the simulationist (using an IO-window) or by the modeller via overwriting the value of  $p$  with another value, e.g. within procedure *initialize* (see procedure *DeclM*) by calling procedure *SetP*.

---

<sup>10</sup>Reset is however not performed if the interactive simulation environment is already running and is only restarted on a new program level.

*minCurVal*, *maxCurVal* Lower and upper value bounds for *p*. Attempts by the simulationist to assign values out of this range are not accepted.

*runTimeChange* *rtc* (=run time change) allows for interactive changing of values of model parameter *p* during a simulation run in the program state *Pause*. *noRtc* (=no run time change) disallows completely any changing of values of the model parameter *p* during a simulation run, even in the program state *Pause*.

*descriptor* String containing a long description of the model parameter *p*. This string may have any length, but might not be visible till its end when it is too long to fit into the IO-window column where it is displayed during a simulation session (see also *identifier*). Example: "Half saturation constant for algal growth".

*identifier* Short string identifying the model parameter *p*. This string should be kept as small as possible in order to ensure full visibility for the display in small IO-windows during a simulation session. In particular on small screens, IO-windows become small in the tiled window position (see menu command *Tile windows*) and they will display only this *identifier* to denote the model parameter *p*. Example: "Ks".

*unit* Unit in which to measure values of the model parameter *p*. This string is displayed in IO-windows during a simulation session. Example: "µg/l".

Besides ordinary, i.e. time independent, model parameters and state variables, complex systems contain other classes of variables. They are time variant parameters, inputs, outputs, and the so-called auxiliary variables. The latter are internal, time dependent variables which are neither state variables, nor inputs nor outputs. In order to support these types of variables, module *SimBase* exports three additional real types:

```
TYPE AuxVar = REAL;      InVar = REAL;      OutVar = REAL;
```

Time variant parameters which do not depend on the state of any model are best treated as input variables or auxiliary variables. Variables of type *AuxVar* are typically used to store intermediate results during complex calculations of derivatives or new states. If a time variant parameter depends on the state of a model, it is best treated as an output variable of the model on which it depends, and as an input variable in all other models. If the simulationist wishes to edit the values of time variant parameters interactively during a simulation session, it is recommended to use the series of values as a table function, where the independent variable is time<sup>11</sup>. Note, since auxiliary, input, and output variables are fully compatible with the standard type REAL, all can also be declared as monitorable variables (see *DeclMV* below).

### 7.1.5 DECLARATION OF MONITORABLE VARIABLES

Every real variable may be declared as a monitorable variable. This allows the simulationist to monitor or observe its values from within the interactive simulation environment, as well as the modeller to document simulation results on file, independent from whether the standard interactive simulation environment is running or not. There apply no restrictions nor does the monitoring exert any influence on the variables monitored. Simply call procedure

```
PROCEDURE DeclMV(VAR mv: REAL; defaultScaleMin, defaultScaleMax: REAL;
  descriptor, identifier, unit: ARRAY OF CHAR;
  defaultSF: StashFiling; defaultT: Tabulation;
  defaultG: Graphing);
```

---

<sup>11</sup> See the *Appendix*, section *Auxiliary Library* for a detailed description on how to work with table functions.

and the real *mv* passed as actual argument is associated with the ModelWorks monitoring mechanism, i.e. its values may be written onto the stash file, tabulated or plotted in the graph from within the simulation environment. You can also pass variables of types compatible with the type REAL (e.g. *StateVar*, *Derivative* or *NewState*, and *AuxVar*, *InVar*, or *OutVar*) as arguments. *DeclMV* may not be called another time for the same real variable *mv*, unless it should have been removed in the meantime.

Note that calling *DeclMV* in one of the simulation environment states *Simulating* or *Pause* (s.a. part II *Theory* Fig. T26) – either directly, or implicitly from within a model's *declModelObjects* procedure – may affect the monitoring settings of the ongoing simulation. If tabulation or graphing are requested for the new variable, the respective window will be cleared at the next monitoring event and any previously displayed results will be lost. If stash filing is requested, a new sub-section of simulation results containing the new monitorable variable will be started in the stash file.

The following types are used to control the actual monitoring settings for each kind of monitoring:

```
TYPE
  StashFiling = (writeOnFile, notOnFile);
  Tabulation  = (writeInTable, notInTable);
  Graphing    = (isX, isY, isZ, notInGraph);
```

The monitoring settings can be independently activated or deactivated and for every monitorable variable the simulationist can control them interactively during simulation sessions. The meaning of the formal procedure parameters of *DeclMV* are:

*mv* The variable to be declared as monitorable variable. The real *mv* can be declared anywhere in the model definition program and may be even part of any data structure. However make sure that it is declared as a global real variable and does exist as long as the model definition program.

*defaultScaleMin/defaultScaleMax* Default minimum and maximum values used for the scaling of the curve to the ordinate while drawing values of the monitorable variable *mv* in the graph. The default minimum and maximum of the ordinate scale are (re)assigned to the current scale minimum and scale maximum by ModelWorks at the monitorable variable's declaration, at starting up of the interactive simulation environment<sup>12</sup> or after every reset of the scaling. During a simulation session the current scale minimum and scale maximum may be changed by the simulationist (using an IO-window) or by the modeller via procedure *SetMV*. There apply no restrictions to the values of these variables. During interactive changes ModelWorks will use the range boundaries MIN(REAL) and MAX(REAL).

*descriptor* String containing a long description of the monitorable variable *mv*. This string may have any length, but might not be visible till its end when it is too long to fit into the IO-window column where it is displayed during a simulation session (see also *identifier*). Example: "Density of alga *Scenedesmus obliquus*".

*identifier* Short string identifying the monitorable variable *mv*. This string should be kept as small as possible in order to ensure full visibility for the display in small IO-windows during a simulation session. In particular on small screens, IO-windows become small in the tiled window position (see menu command *Tile windows*) and they will display only this *identifier* to denote the monitorable variable *mv*. Example: "xa".

---

<sup>12</sup>Reset is however not performed if the interactive simulation environment is already running and is only restarted on a new program level.



*unit* String containing the unit used to measure values of the monitorable variable *mv*. This string is displayed in IO-windows during a simulation session. Example: "cells/ml".

*defaultSF*, *defaultT*, *defaultG* Default settings for the kind of monitoring for the monitorable variable *mv*. If *defaultSF*, *defaultT*, *defaultG* are selected to be written on a file, tabulated or to be plotted, the values of the variable *mv* is written in the default stash file, resp. table, or drawn in the graph as a curve versus the current independent variable, usually simulation time. The defaults for the kind of monitoring are (re)assigned to the current kind by ModelWorks at the monitorable variable's declaration, at starting up of the interactive simulation environment or after every reset of the stash filing, tabulation respectively graphing. During a simulation session the current kind of monitoring may be changed by the simulationist (using the IO-window for monitorable variables) or by the modeller via procedure *SetMV*.

### 7.1.7 TESTING FOR THE PRESENCE OF OBJECTS

The following procedures can be used to check, whether models or model objects have already been declared:

```
PROCEDURE MDeclared (m: Model) : BOOLEAN;
PROCEDURE SVDeclared(m: Model; VAR sv: StateVar) : BOOLEAN;
PROCEDURE PDeclared (m: Model; VAR p : Parameter): BOOLEAN;
PROCEDURE MVDeclared(m: Model; VAR mv: REAL) : BOOLEAN;
```

Note that the last three procedures will also return FALSE in case the corresponding model is not known to ModelWorks.

## 7.2 Accessing Defaults and Current Values

During simulations ModelWorks uses many internal parameters, settings and other variables, the so-called defaults and current values (Fig. T22). They can be accessed by the modeller in order to control simulations in a similar way the simulationist may access them. One class of procedures lets the modeller retrieve values, but not change them (read only values), e.g. the simulation time or the default independent variable. Another class of procedures lets the modeller get and set values, e.g. *GetGlobSimPars* or *GetP* respectively *SetGlobSimPars* or *SetP*. The accessible values are grouped into several categories: the global simulation parameters (including the flags controlling the types of data written to the stash file), the variables associated with the models and the model objects, the settings of the ModelWorks windows, and the name and signature of the stash file. Each of these categories exists in two copies, the defaults and the current values. The following subchapters explain the procedures available for all above categories, except for the window-settings and attributes of the stash file, which are explained later.

### 7.2.1 GLOBAL SIMULATION PARAMETERS AND PROJECT DESCRIPTION

The global read-only variables are given in Tab. R1 and the global simulation parameters used to control simulations in Tab. R2. The variables listed in Tab. R3 are used for the project description.

The first column in each table contains the identifiers used to designate the variables in the client interface. The second column contains the symbols used to denote the variables in this manual, in particular part II, *Theory*.

Identifier	Symbol	Meaning
CurrentTime	t	Current simulation time or independent variable for continuous time models
CurrentStep	k	Current simulation time or independent variable for discrete time models
LastCoincidenceTime	–	Last simulation time point at which the state of all discrete time models was updated, or would have been updated if such models were present
CurrentSimNr	–	Number of the current simulation run

Tab. R1 : Read-only global simulation variables internally used by ModelWorks.

Identifier	Symbol	Meaning
t0	$t_0/k_0$	Simulation start time
tend	$t_{end}/k_f$	Simulation stop time
h	$h/h_{max}$	Integration step (if fixed step length method) maximum integration step (if at least one variable step length method in use) (h is only used if at least one continuous time model is present)
er	$e_r$	Maximum relative local error ( $e_r$ is only used if at least one variable step length method is in use)
c	c	Discrete time step (if only discrete time models present). Coincidence interval (if continuous as well as discrete time models present)
hm	$h_m$	Monitoring interval

Tab. R2 : Global simulation parameters of ModelWorks.

Identifier	Symbol	Meaning
title	–	Project title string
remark	–	Remark string
footer	–	Footer string
wtitle	–	With title in graph
wremark	–	With remarks in graph
autofooter	–	Automatic update of date, time, and run number in footer
recM	–	Recording of data on models in stash file
recSV	–	Recording of data on state variables in stash file
recP	–	Recording of data on model parameters in stash file
recMV	–	Recording of data on monitorable variables in stash file
recG	–	Recording of graph in stash file at end of run
recTF	–	Recording of data on table functions in stash file

Tab. R3 : Global project description of ModelWorks.

### 7.2.1.a Retrieval of read only current values

A user can access internal variables (Tab. R1) of ModelWorks by means of special procedures. This guarantees undisturbed data consistency. For instance, the procedure

```
PROCEDURE CurrentStep(): INTEGER;
```

exported by module *SimMaster*, returns the current simulation step, i.e. the current value of discrete time (must not be confounded with the integration step used by numerical integration for continuous time models). Note that this simulation step can only be read but not changed.

```
PROCEDURE CurrentTime(): REAL;
```

Returns the current simulation time, i.e. the current value of continuous time. Note that the simulation time can only be read but not changed.

```
PROCEDURE LastCoincidenceTime(): REAL;
```

Returns the last time point at which the state of all discrete time models was (would have been) updated. Note that this time point may be retrieved even if no discrete time models are currently declared within ModelWorks. This time point can only be read but not changed.

```
PROCEDURE CurrentSimNr(): INTEGER;
```

Returns the current simulation run number  $k$  during structured simulations ( $k = 1, 2, 3\dots$ ) (Fig. T16). Note that even aborted runs are numbered. This procedure is typically called in the client procedure *initialize*, e.g. to assign parameter values depending on the current run. Note  $k$  can only be read but not changed.

### 7.2.1.b Modification of defaults

The predefined values ModelWorks uses as defaults are listed in Tab. T1 (manual part II *Theory*). If the modeller wishes to change, i.e. overwrite, them she may access any of the variables listed in the tables Tab. R2 or R3 with a *SetDeflxyz* procedure, i.e. a procedure with an identifier starting with *SetDeflt*.

```
PROCEDURE SetDefltGlobSimPars(t0, tend, h, er, c, hm: REAL);
PROCEDURE GetDefltGlobSimPars(VAR t0, tend, h, er, c, hm: REAL);
```

```
PROCEDURE SetDefltProjDescrs(title, remark, footer: ARRAY OF CHAR;
                             wtitle, wremark, autofooter,
                             recM, recSV, recP, recMV, recG: BOOLEAN);
```

```
PROCEDURE GetDefltProjDescrs(VAR title, remark, footer: ARRAY OF CHAR;
                             VAR wtitle, wremark, autofooter,
                             recM, recSV, recP, recMV, recG: BOOLEAN);
```

```
PROCEDURE SetDefltTabFuncRecording(recTF: BOOLEAN);
PROCEDURE GetDefltTabFuncRecording(VAR recTF: BOOLEAN);
```

Above procedures set or get the defaults for the global simulation parameters, the project description, or the recording flags. The meanings of the formal procedure parameters are listed in the tables Tab. R2 and R3.

Calling one of the procedures *SetDefltGlobSimPars*, *SetDefltProjDescrs* or *SetDefltTabFuncRecording* will have no effect until the global simulation parameters respectively the project description or table function recording are reset.

ModelWorks solves equations, e.g. differential equations, by using an independent variable, by default named "time". The independent variable is used as the default abscissa variable in the graph, if no monitorable variable has been selected for this purpose.

```
PROCEDURE SetDefltIndepVarIdent( descr,ident,unit: ARRAY OF CHAR);
PROCEDURE GetDefltIndepVarIdent(VAR descr,ident,unit: ARRAY OF CHAR);
```

*SetDefltIndepVarIdent* overwrites the defaults of the descriptor *descr*, identifier *ident*, and the unit *unit* of the independent variable. The predefined default values ModelWorks uses are the *descr* "time", *ident* "t", and no *unit* (empty string). The call of this procedure will have no effect until the global simulation parameters are reset.

The following procedures are actually only kept for convenience and upward compatibility with previous versions of the ModelWorks client interface. In ModelWorks versions later than V1.1 their functions are also available by using the procedures *SetDefltGlobSimPars* respectively *SetGlobSimPars*.

```
PROCEDURE SetMonInterval(hm: REAL);
```

Sets the default of the monitoring interval only, not the current value. The call of this procedure will have no effect until the global simulation parameters are reset.

```
PROCEDURE SetIntegrationStep(h: REAL);
```

Sets the default integration step only, not the current value. The call of this procedure will have no effect until the global simulation parameters are reset.

```
PROCEDURE SetSimTime(t0,tend: REAL);
```

Sets the defaults for the simulation start and stop time as well as the current simulation start and stop time. It differs in this respect from all other parameter setting routines, which affect either only the defaults or only the current values.

### 7.2.1.c Modification of current values

Modification of current values of the parameters and variables listed in Tab. R2 and R3 can be accomplished by the following procedures whose identifiers start with *Set*:

```
PROCEDURE SetGlobSimPars( t0, tend, h, er, c, hm: REAL);
PROCEDURE GetGlobSimPars(VAR t0, tend, h, er, c, hm: REAL);

PROCEDURE SetProjDescrs( title,remark,footer: ARRAY OF CHAR;
                        wtitle,wremark,autofooter,
                        recM, recSV, recP, recMV, recG: BOOLEAN);
PROCEDURE GetProjDescrs(VAR title,remark,footer: ARRAY OF CHAR;
                        VAR wtitle,wremark,autofooter,
                        recM, recSV, recP, recMV, recG: BOOLEAN);

PROCEDURE SetTabFuncRecording( recTF: BOOLEAN);
PROCEDURE GetTabFuncRecording(VAR recTF: BOOLEAN);
```

The above procedures set or get the current values for the global simulation parameters, the project description, or the recording flags. The meanings of the formal procedure parameters are listed in the tables Tab. R2 and R3. Note that calls to the procedures *SetGlobSimPars* and *SetSimTime* only have effect if  $t_0 < t_{\text{end}}$  holds. If these procedures are called in one of the sub-states *Running* or *Pause* (s.a. part II *Theory* Fig. T15-T16) the additional restrictions  $t_0 \leq t$  and  $t_{\text{end}} \geq t$  apply.

The effects of calling one of the procedures *SetProjDescrs* or *SetTabFuncRecording* in the sub-states *Running* or *Pause* will not become visible to the simulationist until the next simulation run, except if the call occurs just before the very first monitoring of the current run has taken place<sup>13</sup>. Changes in the flags *wtitle* and *wremark* and/or the associated *title* and *remark* will become visible only if the graph window is resized during the simulation. Changes in the flag *recG* which controls whether the graph will be dumped to the stash file will apply to an ongoing simulation run, since the graph is dumped to the stash file at termination of the run.

```
PROCEDURE SetIndepVarIdent( descr,ident,unit: ARRAY OF CHAR);
PROCEDURE GetIndepVarIdent(VAR descr,ident,unit: ARRAY OF CHAR);
```

*SetIndepVarIdent* and *GetIndepVarIdent* allow to set and get the current descriptor *descr*, identifier *ident*, and the unit *unit* of the independent variable.

Note that calling of one of the procedures *SetSimTime*, *SetIndepVarIdent*, or *SetGlobSimPars* during a simulation may result in redrawing of the graph, such that any previously displayed simulation results will be lost.

### 7.2.1.c Resetting of current values to the defaults

The following two procedures have exactly the same effect as the execution of the menu commands *Reset Global simulation parameters* and *Reset Project description.*, respectively, from the menu *Settings* (s.a. part III *Reference/User Interface*).

```
PROCEDURE ResetGlobSimPars;
```

resets the current values for *t0*, *tend*, *h*, *er*, *c*, *hm* as well as the *descr*, *ident* and *unit* of the default independent variable to their defaults. This procedure will typically be used within the simulation environment definition procedure which may be installed in ModelWorks by means of *DeclDefSimEnv* (see above).

```
PROCEDURE ResetProjDescrs;
```

resets the project *title*, *remark*, *footer* as well as the flags *wtitle*, *wremark*, *autofooter*, *recM*, *recSV*, *recP*, *recMV*, *recG* and *recTF* to their respective default values. This procedure may be called at any time. It will have the same effects as described for *SetProjDescrs* and *SetTabFuncRecording* above.

## 7.2.2 INSTALLED MODELS AND MODEL OBJECTS

Once declared, model and model objects may be modified in any way, except for their binding to a particular variable in the model definition program. In order to break even this binding, you have to remove the model or model object completely by calling a remove procedure (see below section *Removing models and model objects*). Modifications affect attributes and values associated with a model or model object. To support model and model object editing there exists for each object class a procedure pair: a get and a set procedure. The get procedure retrieves the objects attributes, the set procedure modifies (overwrites) them. Moreover the procedures are grouped into two sets: The first set is to modify the defaults, the other to modify the current values. The meanings of the formal procedure parameters are the same as described under the declaration procedures *DeclM*, *DeclSV*, *DeclP* and *DeclMV*. Also, the parameter lists were kept similar to the ones used by the declaration procedures.

---

<sup>13</sup>This will be the case if one of these procedures is e.g. called from within a model's *initialize*-procedure.

### 7.2.2.a Modification of defaults

Setting of defaults (signified by formal parameter names starting with “default...”) in the procedures listed below will not imply a setting of the current values, i.e. no implicit reset. That is, changes of the defaults will not become effective or visible until the next corresponding reset. Setting of all other values however will have immediate effects. In particular, this concerns a model’s *initialize*, *input*, *output*, *dynamic*, *terminate* and *about* procedures, the descriptors, identifiers, and unit strings of models or model objects, as well as changes of range boundaries (used during the interactive changing of initial values or model parameter values via IO-windows) or a parameter’s *runTimeChange* option.

```

PROCEDURE GetDeflM(VAR m: Model;
                  VAR defaultMethod: IntegrationMethod;
                  VAR initialize, input, output, dynamic, terminate: PROC;
                  VAR descriptor, identifier: ARRAY OF CHAR;
                  VAR about: PROC);
PROCEDURE SetDeflM(VAR m: Model;
                  defaultMethod: IntegrationMethod;
                  initialize, input, output, dynamic, terminate: PROC;
                  descriptor, identifier: ARRAY OF CHAR;
                  about: PROC);

PROCEDURE GetDeflSV(m: Model; VAR s: StateVar;
                   VAR defaultInit, minCurInit, maxCurInit: REAL;
                   VAR descriptor, identifier, unit: ARRAY OF CHAR);
PROCEDURE SetDeflSV(m: Model; VAR s: StateVar;
                   defaultInit, minCurInit, maxCurInit: REAL;
                   descriptor, identifier, unit: ARRAY OF CHAR);

PROCEDURE GetDeflP (m: Model; VAR p: Parameter;
                   VAR defaultVal, minVal, maxVal: REAL;
                   VAR runTimeChange: RTCType;
                   VAR descriptor, identifier, unit: ARRAY OF CHAR);
PROCEDURE SetDeflP (m: Model; VAR p: Parameter;
                   defaultVal, minVal, maxVal: REAL;
                   runTimeChange: RTCType;
                   descriptor, identifier, unit: ARRAY OF CHAR);

PROCEDURE GetDeflMV(m: Model; VAR mv: REAL;
                   VAR defaultScaleMin, defaultScaleMax: REAL;
                   VAR descriptor, identifier, unit: ARRAY OF CHAR;
                   VAR defaultSF: StashFiling; V
                   VAR defaultT: Tabulation;
                   VAR defaultG: Graphing);
PROCEDURE SetDeflMV(m: Model; VAR mv: REAL;
                   defaultScaleMin, defaultScaleMax: REAL;
                   descriptor, identifier, unit: ARRAY OF CHAR;
                   defaultSF: StashFiling;
                   defaultT: Tabulation;
                   defaultG: Graphing);

```

### 7.2.2.b Modification of current values

Most of the so-called *Set...* procedures affect the corresponding current values immediately. They may be called freely at any simulation environment state. Note however that for reasons of consistency if they are called in the environment state *Simulating*, their effect will take place only at the end of the current integration loop (see part II *Theory*, chapter *Simulations and the Run-Time System* in particular section *Manipulating the model base at run-time*). Any new or changed values will be displayed in the corresponding IO-windows. Note that the updating of some changes may require some time before they become actually visible on the screen,

because the "Dialog Machine" may need several integration steps till all updates have been completed.

```

PROCEDURE GetM (VAR m: Model; VAR curMethod: IntegrationMethod);
PROCEDURE SetM (VAR m: Model; curMethod: IntegrationMethod);

PROCEDURE GetSV (m: Model; VAR s: StateVar; VAR curInit: REAL);
PROCEDURE SetSV (m: Model; VAR s: StateVar; curInit: REAL);

PROCEDURE GetP (m: Model; VAR p: Parameter; VAR curVal: REAL);
PROCEDURE SetP (m: Model; VAR p: Parameter; curVal: REAL);

PROCEDURE GetMV (m: Model; VAR mv: REAL;
                 VAR curScaleMin, curScaleMax: REAL; VAR curSF: StashFiling;
                 VAR curT: Tabulation; VAR curG: Graphing);
PROCEDURE SetMV (m: Model; VAR mv: REAL;
                 curScaleMin, curScaleMax: REAL; curSF: StashFiling;
                 curT: Tabulation; curG: Graphing);

```

It is recommended to avoid the direct modification of state variables, parameters etc. by directly assigning a new value to the respective variable. There are two reasons why: First, there may result a confusing discrepancy in the value actually used for simulations and the one visible in the IO-window. Secondly, ModelWorks is likely to overwrite the value, so that the assignment is fictitious and the modeller may have difficulties to understand subsequent simulation results. To avoid any such problems, use always the set procedures and they will preserve consistency between the model definition program and ModelWorks.

#### 7.2.2.c Resetting of current values to the defaults

The following procedures correspond to the menu commands *Reset All model's integration methods, initial values, parameters, stash filing, tabulation, graphing, scaling and curve attributes*, respectively, from the menu *Settings* (s.a. part III *Reference/User Interface*).

```

PROCEDURE ResetAllIntegrationMethods;
PROCEDURE ResetAllInitialValues;
PROCEDURE ResetAllParameters;

PROCEDURE ResetAllStashFiling;
PROCEDURE ResetAllTabulation;
PROCEDURE ResetAllGraphing;
PROCEDURE ResetAllScaling;

```

The first three procedures reset the integration methods of all currently declared models, the initial values of all state variables and the values of all parameters, respectively, to their defaults. The remaining procedures operate on the respective attributes of all currently declared monitorable variables. If you wish to reset a single model or model object only, call *GetDeflt...* to retrieve a default followed by *Set...* to assign it to the object's current value.

Same as for *Set...*, calls to these *Reset...* procedures will have a slightly delayed effect if occurring in the environment state *Simulating*. Note also that (re)setting of monitorable variables may affect an ongoing simulation's monitoring in a similar way as described for *DeclMV* above.

#### 7.2.2.d Model and model object attributes

ModelWorks provides a mechanism to attach one integer-type attributes to each model or model object. These attributes are typically used as array indices to access some data associated with the object, which are stored in an array.

```

TYPE
  Attribute = INTEGER;
CONST
  noAttr = MIN(Attribute);

PROCEDURE SetModelAttr(m: Model; val: Attribute);
PROCEDURE GetModelAttr(m: Model): Attribute;

PROCEDURE SetObjAttr(m: Model; VAR o: REAL; val: Attribute);
PROCEDURE GetObjAttr(m: Model; VAR o: REAL): Attribute;

```

In case there is currently no attribute attached to a model or model object, the procedures *GetModelAttr* and *GetAttr* will return the value *noAttr*.

See also the optional module *SimObjects* listed in the *Appendix* which allows for the installation and efficient retrieval of an additional attribute of the generic type *ADDRESS* for each model or model object.

### 7.2.2.e Access support for models and model objects

The following procedures allow to access all currently declared models and model objects. They are especially useful for manipulating objects from program modules which are not possessors of the variables. For exactly this reason however, they must be applied carefully, i.e. the modeler should ensure that no inconsistencies occur. In order to allow for an efficient usage of attributes, the attribute values currently attached to an object are also passed in the procedures of type *ModelProc* respectively *ModelObjectProc* which are repeatedly called by the respective *DoForAll...* procedures.

```

TYPE
  ModelProc      = PROCEDURE( VAR Model, VAR Attribute );
  ModelObjectProc = PROCEDURE( Model, VAR REAL, VAR Attribute );

PROCEDURE DoForAllModels( p: ModelProc );
PROCEDURE DoForAllSVs   ( m: Model; p: ModelObjectProc );
PROCEDURE DoForAllPs    ( m: Model; p: ModelObjectProc );
PROCEDURE DoForAllMVs   ( m: Model; p: ModelObjectProc );

```

The following program fragment exemplifies the usage of the *DoForAll*-mechanism. The procedure named *SetReducedMonitoring* is used to reduce all monitoring to the current stash filing settings of all monitorable variables, e.g. in order to increase simulation performance for a sensitivity experiment in batch mode.

```

...
FROM SimBase IMPORT
  Model, Attribute, StashFiling, Tabulation, Graphing,
  GetMV, SetMV, DoForAllMVs, DoForAllModels;
...

PROCEDURE ReduceMonitoringForMV ( m: Model; VAR mv: REAL; VAR dummy: Attribute );
  VAR curScaleMin, curScaleMax: REAL; curSF: StashFiling; curT: Tabulation; curG: Graphing;
BEGIN
  GetMV( m, mv, curScaleMin, curScaleMax, curSF, curT, curG );
  SetMV( m, mv, curScaleMin, curScaleMax, curSF, notInTable, notInGraph );
END ReduceMonitoringForMV ;

PROCEDURE ReduceMonitoringForModel( VAR m: Model; VAR dummy: Attribute );
BEGIN
  DoForAllMVs( m, ReduceMonitoringForMV );

```



```

END ReduceMonitoringForModel;

PROCEDURE SetReducedMonitoring;
BEGIN
  DoForAllModels( ReduceMonitoringForModel);
END SetReducedMonitoring;

...

```

### 7.3 Removing Models and Model Objects

Models and model objects can be removed by calling any of the procedures listed below. Note that removing means only that the linkage of, e.g. a state variable *s* to the simulation environment is removed, not the real variable *s* itself, which remains a part of the model definition program. Once removed, a model or model object is completely unknown to ModelWorks and has become inaccessible by ModelWorks' routines. E.g. removed model objects are no longer listed in IO-windows and can no longer be integrated.

```

PROCEDURE RemoveM      (VAR m: Model);
PROCEDURE RemoveAllModels;

PROCEDURE RemoveSV     (m: Model; VAR s : StateVar);
PROCEDURE RemoveMV     (m: Model; VAR mv: REAL);
PROCEDURE RemoveP      (m: Model; VAR p : Parameter);

```

Remove procedures may not be called another time, unless the model or the model object has been redeclared in the meantime. Remove procedures may also be called in the sub-states *Running* or *Pause*, but with a slightly delayed effect (s.a. part II *Theory*, section *Manipulating the model base at run-time* and Tab. T4). Note that *RemoveMV* may affect a running simulation's monitoring in the way already described for *DeclMV* above.

Calling procedure *RemoveM* results in an implicit removal of all model objects belonging to this model. In case there is a simulation currently running, in order to guarantee consistent simulation results at least the model's *terminate* procedure will be called prior to removing the model objects. For example, if a model *m* is removed from within an other model's *output* procedure, first the current integration step for *m* will be accomplished by calling its *output* (if not already called), *input* and *dynamic* procedures, than its simulation is terminated by calling its *terminate* procedure, and than its model objects are removed (s.a. part II *Theory*, section *Manipulating the model base at run-time* and Tab. T4).

### 7.4 Simulation Control and Structured Simulation Runs

The following Modula-2 objects serve the control of simulations.

```

PROCEDURE SimRun;

```

This procedure performs an elementary simulation run with the current parameter and other variable settings. Typically this routine is used to execute a series of simulation runs, e.g. in a loop within procedure *InstallExperiment* (see below this section). Simulation runs can then be executed under the control of the modeller, for instance to construct a whole phase portrait by means of a single menu command or to identify a model parameter. Note that *SimRun* may be called without the standard interactive simulation environment currently running.

```

PROCEDURE CurrentSimNr(): INTEGER;

```

Returns the current simulation run number  $k$  during structured simulations ( $k = 1, 2, 3\dots$ ) (see part II *Theory*, Fig. T21). A typical usage of this procedure looks similar to the following statement:

```
REPEAT SimRun UNTIL CurrentSimNr(=maxSimNr
```

Note however that even aborted runs are numbered. To handle properly abortion of structured simulation runs see below procedure *ExperimentAborted*.

```
TYPE
  StartConsistencyProcedure = PROCEDURE(): BOOLEAN;
  TerminateConditionProcedure = PROCEDURE(): BOOLEAN;

PROCEDURE InstallStartConsistency( startAllowed: StartConsistencyProcedure );
```

Procedure *startAllowed* is called at the begin of a simulation run, right after the execution of the procedure *initialize* (see procedure *DeclM*), and after resuming a run from the state *Pause*. If it returns FALSE, the simulation will be aborted and the simulation environment immediately returns into the program state *No simulation*. Otherwise the simulation is normally continued. Typically this procedure is used to check consistency in the initial conditions, e.g. to test relations among parameters and initial values. Since the simulationist may interactively change values of parameters independently from each other (entry forms test only syntax and ranges), this consistency test is important in case the model equations would become undefined if the conditions were not met. Moreover, the modeler may use this procedure to compute values of auxiliary variables, which depend on the current values of parameters.

```
PROCEDURE InstallTerminateCondition(isAtEnd: TerminateConditionProcedure);
```

Procedure *isAtEnd* is called at the end of each time (integration) step in the program state *Simulating* and continuously in the state *Pause*. If it returns TRUE, the simulation will be terminated. This behaviour can be used to program state events which lead to the simulation termination. Note however, that this does not fully conform to a proper handling of state events, since ModelWorks performs no iterations to find the exact location of the event. You have to program *tc* such that the value returned is correct even if the current time is not exactly that of the event, i.e. the procedure *tc* must be able to detect the state event even if it occurs anywhere in the time interval of the current integration step  $h$ .

```
PROCEDURE CurCalcM(): Model;
```

Procedure *CurCalcM* returns the model of which the *initialize*, *output*, *input*, *dynamic* or *terminate* procedure is currently being called by ModelWorks. It is typically used in a situation, where several models use common procedures e.g. for *input* or *dynamic*. Knowing the model, the model definition program can set e.g. array indices or exhibit a different behaviour.

```
PROCEDURE PauseRun;
```

Makes a state transition from the program state *Simulating* into the program state *Pause* (part II *Theory*, Fig. T15, T16 and T24) and will only return after the simulationist has chosen the menu command *Resume run* under menu *Solve*. This feature allows to temporarily interrupt a simulation run exactly at a particular point, such as a state event (e.g. a state variable becomes negative), and allows the simulationist to take some action, e.g. changing a parameter value, before resuming the simulation.

```
PROCEDURE ResumeRun;
```

Makes the opposite state transition than *PauseRun*, i.e. from the program state *Pause* into the program state *Simulating*. This procedure allows to resume simulation after e.g. execution of a menu command which does not belong to the standard interactive simulation environment.

```
PROCEDURE StopRun;
```

Makes a state transition from the program state *Simulating* into the program state *No Simulation* (part II *Theory*, Fig. T15), i.e. stops the current simulation run.

```
PROCEDURE InstallExperiment(doExperiment: PROC);
```

Installs an experiment *doExperiment* which may be executed by the simulationist by selecting the menu command *Execute Experiment* under menu *Solve*. The procedure *doExperiment* is provided by the modeller and contains typically calls to the procedure *SimMaster.SimRun*. If the procedure *InstallExperiment* has at least been called once in the course of a simulation session, the menu command *Execute Experiment* under menu *Solve* will no longer appear dimmed but will be active and can be chosen by the simulationist in the state *No Simulation*.

```
TYPE
```

```
  MWState = (noSimulation, simulating, pause, noModel);
```

```
PROCEDURE GetMWState(VAR s: MWState);
```

The current state of the simulation environment can be determined by calling procedure *GetMWState* from *SimMaster*. The meaning of the returned value *s*, either *noSimulation*, *simulating*, *pause*, or *noModel* corresponds exactly to the program states shown in Fig. T15 (part II *Theory*).

```
TYPE
```

```
  MWSubState = (noRun, running, noSubState, stopped);
```

```
PROCEDURE GetMWSubState(VAR ss: MWSubState);
```

The current substate of the simulation environment while a structured simulation (experiment) is currently in execution, can be determined by calling procedure *GetMWSubState* from *SimMaster*. The meaning of the returned value *ss*, either *noRun*, *running*, *noSubState*, or *stopped* corresponds exactly to the program substates shown in Fig T16 (part II *Theory*). If the value *noSubState* is returned, no experiment is currently running, i.e. the simulationist has reached state *simulating* by choosing the menu command *Solve/Start run* (s.a. below procedure *ExperimentRunning*).

```
PROCEDURE InstallStateChangeSignaling(doAtStateChange: PROC)
```

Installs the client's procedure *doAtStateChange* in ModelWorks which will be called each time a change in *MWState* or *MWSubState* occurs.

```
PROCEDURE ExperimentRunning(): BOOLEAN;
```

*ExperimentRunning* from *SimMaster* returns TRUE if a structured simulation (experiment) is currently in execution, i.e. if the simulationist has reached the state *simulating* by choosing the menu command *Solve/Execute Experiment* (s.a. above procedure *GetMWSubState*).

```
PROCEDURE ExperimentAborted(): BOOLEAN;
```

*ExperimentAborted* from *SimMaster* returns TRUE if the simulationist has stopped (killed) a running structured simulation (experiment). A typical use of this procedure is to skip superfluous calls to procedure *SimRun*. E.g.:

```
REPEAT
```

```
  SimRun;
```

```
UNTIL (CurrentSimNr)=maxRunNr) OR ExperimentAborted()
```

## 7.5 Display and Monitoring

### 7.5.1 WINDOW OPERATIONS

The following Modula-2 objects serve to control the display on the screen, e.g. the arrangement of windows or the monitoring.

```
PROCEDURE TileWindows;
PROCEDURE StackWindows;
```

The two procedures stack or tile windows on the screen. Stacking is with overlapping windows similar to the ModelWorks predefined start-up display. Tiled windows don't overlap and fill the screen as much as possible. The actual arrangement may depend on the screen in display.

```
PROCEDURE InstallTileWindowsHandler(doAtTile:PROC);
PROCEDURE InstallStackWindowsHandler(doAtStack:PROC);
```

The above two procedures allow to install a procedure which is executed when windows are tiled or stacked. *doAtTile* or *doAtStack* will be called immediately after windows are tiled or stacked, e.g. in order to rearrange additional windows managed by the model ler.

The following type enumerates all windows of the ModelWorks simulation environment

```
TYPE
  MWWindow = (MIOW, SVIOW, PIOW, MVIOW, TableW, GraphW, AboutMW, TimeW);
```

*MIOW*, *SVIOW*, *PIOW* and *MVIOW* designate the IO-windows for the models, state variables, model parameters, and the monitorable variables. *TableW*, *GraphW* and *AboutMW* are the table, graph, and the about model window with the title "Model Help/Info". The latter window is displayed if the simulationist clicks into the question mark button of the models IO-window. *TimeW* is the window in the top right corner of the main screen used to display the current simulation number and time while in substates *Running* or *Pause*. Since the time-window only exists in these substates, no default attributes are maintained for it. *TimeW* may be passed as a formal parameter value to the procedures *SetWindowPlace*, *CloseWindow*, *GetWindowPlace*, *DisableWindow* and *EnableWindow*; but has no effect if passed to any of the remaining procedures operating on ModelWorks windows; also, at resetting of these windows its size and position will actually not be changed.

```
PROCEDURE SetWindowPlace(mww: MWWindow; x,y,w,h: INTEGER);
```

*SetWindowPlace* places the window *mww* with its lower left corner at the position *x,y* and resizes it to the width *w* and height *h* (size of outer frame including title bar, frame, and shadows). The point [*x,y*] is given in pixel coordinates with an origin at the lower left corner of the main computer screen. If this procedure is called in case the window should not already be open, it will open the window in the proper size at the specified location. Calls to *SetWindowPlace* for the *TimeW* will be discarded, if the simulation environment is not in the state *Running*.

```
PROCEDURE CloseWindow(w: MWWindow);
```

*CloseWindow* closes the window *w* and remembers the location plus size for the next reopening.

```
PROCEDURE GetWindowPlace(mww: MWWindow; VAR x,y,w,h: INTEGER;
  VAR isOpen: BOOLEAN);
```

*GetWindowPlace* returns the current position of the window *mww* and whether it is currently open or not. Since the simulation environment remembers the location and size of a window when it was open the last time, this procedure returns meaningful values even if *isOpen* should be FALSE.

```
PROCEDURE SetDeflWindowPlace(mww: MWindow; x,y,w,h: INTEGER);
```

*SetDeflWindowPlace* sets the default size and position of the window *mww*.

```
PROCEDURE GetDeflWindowPlace(mww: MWindow; VAR x,y,w,h: INTEGER;
                             VAR enabled: BOOLEAN );
```

*GetDeflWindowPlace* returns the default size and position of the window *mww* plus the current IO-window status. If *enabled* is TRUE, it means that the editing functions of the IO-window are currently available to the simulationist. This is the case in the state *No simulation* or partially in the state *Pause*, but editing is disabled in the state *Simulating* (s.a. part II *Theory*, Fig. T15, in particular the title bars with horizontal lines vs. dimmed bars in Fig. T24)

To control the format in which information is displayed in a particular IO-window use the following data structure:

```
TYPE
  IOWColsDisplay = RECORD
    descrCol, identCol : BOOLEAN;
    CASE iow: MWindow OF
      MIOW : m : RECORD
        integMethCol: BOOLEAN;
        END(*RECORD*);
      | SVIOW : sv: RECORD
        unitCol, svInitCol: BOOLEAN;
        fw,dec: INTEGER;
        END(*RECORD*);
      | PIOW : p : RECORD
        unitCol, pValCol, pRtcCol: BOOLEAN;
        fw,dec: INTEGER;
        END(*RECORD*);
      | MVIOW : mv: RECORD
        unitCol, scaleMinCol, scaleMaxCol, mVMonSetCol: BOOLEAN;
        fw,dec: INTEGER;
        END(*RECORD*);
    END(*CASE*)
  END(*RECORD*);
```

The Booleans determine whether a column is to be displayed or not; they correspond to the check boxes which may be set by the simulationist in the entry form which is activated when the IO-window button *Set Up* is clicked. The integers specify the format in which to display real numbers, where *fw* is the field width and *dec* is the number of decimal digits.

```
PROCEDURE SetIOWColDisplay( mww: MWindow; wd: IOWColsDisplay );
```

*SetIOWColDisplay* allows to set a new setup of the columns and new display formats in the IO-window *mww*. The predefined default of the simulation environment is 3 decimal digits to display or parameter values; this procedure allows to alter this format to any other value.

```
PROCEDURE GetIOWColDisplay( mww: MWindow; VAR wd: IOWColsDisplay );
```

*GetIOWColDisplay* returns the setup of the columns and the display formats currently in use by the IO-window *mww*.

Instead of the current values, the following two procedures affect the default values; otherwise they function the same way as the previous two procedures:

```
PROCEDURE SetDeflIOWColDisplay( mww: MWindow; wd: IOWColDisplay );
PROCEDURE GetDeflIOWColDisplay( mww: MWindow; VAR wd: IOWColDisplay );
```

The following procedures allow the modeller to customize the simulation environment even a step further; she may even completely disallow or allow any usage of an IO-window. This control is only available to the modeller but not to the simulationist.

```
PROCEDURE DisableWindow(w: MWindow);
```

*DisableWindow* disables the ModelWorks window *w* for any usage, i.e. neither the opening nor the editing (IO-windows) by the simulationist is any more possible. In case the window *w* should be currently open, it is closed. Menu commands possibly associated with the window (IO-windows, graph and table window) are disabled (dimmed). Note that in case the *AboutMW* is disabled, a model's *about* procedure will still be executed when the simulationist clicks into the question mark button available in the models IO-window. Thus, disabling of the *AboutMW* allows the modeller to entirely replace the predefined ModelWorks help mechanism by her own procedures. Similarly, responsibility for the display of the current simulation time may entirely be taken over by the modeller by disabling the *TimeW*.

```
PROCEDURE EnableWindow (w: MWindow);
```

*EnableWindow* reverses the effect of *DisableWindow* and enables the subsequent usage of the window *w* by the simulationist to its normal and full functionality. However, the window will not be opened until the simulationist executes the corresponding menu command (only possible for IO-windows, graph and table window) or the modeller explicitly calls *SetWindowPlace*.

```
TYPE
  MWindowArrangement = (current, stacked, tiled);

PROCEDURE SetDeflWindowArrangement(a: MWindowArrangement);
```

The procedure *SetDeflWindowArrangement* allows to set the default positions, sizes and columns-display settings (IO-windows only) of all ModelWorks windows to the values corresponding to stacked or tiled windows, or according to the current settings. Typical usage of this procedure may look as follows:

```
SetWindowPlace(MIOW,...);
SetWindowPlace(SVIOW,...);
SetWindowPlace(GraphW,...);
...
SetDeflWindowArrangement( current );
```

This solution is more convenient than having to specify first the current values and then the defaults with exactly the same values.

```
PROCEDURE ResetWindows;
```

copies the default window positions, sizes and column-display settings (IO-windows only) to the current values. It has exactly the same effect as execution by the simulationist of the menu command *Settings /Reset Windows* available at the standard user interface and will close and (re)open all windows, which are not already in their default states.

## 7.5.2 GENERAL MONITORING

After having called *SuppressMonitoring*, all subsequent monitoring will be suppressed.

```
PROCEDURE SuppressMonitoring;
```

The procedure

```
PROCEDURE ResumeMonitoring;
```

resumes all monitoring exactly as it was before procedure *SuppressMonitoring* was called.

```
PROCEDURE InstallClientMonitoring(initClientMon, doClientMon, termClientMon: PROC);
```

Installs in ModelWorks a client provided monitoring mechanism. During the simulation run the monitoring procedure *doClientMon* is called every time ModelWorks does its standard monitoring once. Hereby *doClientMon* will be called as the last monitoring procedure, i.e. after ModelWorks calls the stash file, the tabulation, and the graph monitoring procedures. This allows for instance to draw into the ModelWorks graph window from within the *doClientMon* (to this end you might wish to use the module *SimGraphUtils* from the optional client interface). At the begin respectively the end of every simulation run the procedures *initClientMon* respectively *termClientMon* are called (to locate these events in more details see the calling sequence in Fig. T19, part II *Theory*). Note that *initClientMon* is called for  $t_0$  respectively  $k_0$  only, and that for all subsequent monitoring, including the very last one for  $t_{\text{end}}$  respectively  $k_f$ , the procedure *doClientMon* is used. Typically *initClientMon* does some initial preparations such as opening a file or initializing a data structure, and then it calls *doClientMon*. Note also that the procedure *termClientMon* is called at the very end of the simulation run, in particular even after all model's *terminate* procedures have been executed. Usually the model's *terminate* procedures are used to analyze the run, e.g. to compute a mean, and *termClientMon* is typically only used to do some house-keeping such as closing a file or discarding no longer needed global data structures.

### 7.5.3 STASH FILING

```
PROCEDURE SetStashFileName      (      sfn: ARRAY OF CHAR);
PROCEDURE GetStashFileName     (VAR  sfn: ARRAY OF CHAR);
```

These procedures allow to set or get the current name of the stash file (may contain a path, e.g. MyDisk:Folder:TheFile.DAT). The call to *SetStashFileName* will have no effect until the stash file is actually opened during a subsequent simulation. Calling this procedure in the middle of a simulation run (state *Simulating*) will rename the current stash file.

The current stash file may be changed at any time by calling

```
PROCEDURE SwitchStashFile      (newsfn: ARRAY OF CHAR);
```

If *SwitchStashFile* is called in the state *Simulating*, the current stash file is closed, and monitoring is continued in the file with name *newsfn*. Since this may be done in the middle of a simulation run, as well as in the substate *No run*, *SwitchStashFile* allows to distribute the documentation of individual, very long simulations, as well as of different simulation runs belonging to one and the same experiment among different files. If *SwitchStashFile* is called in the state *No Simulation* it will have the same effect as *SetStashFileName*.

**I M P O R T A N T N O T I C E:** If a file with the name specified in the procedure *SetStashFileName* or *SwitchStashFile* should already exist, it will be overwritten without any warning!! This behaviour contrasts with the setting of the name via the user interface (menu command *Settings/Select stash file ...*).

```
PROCEDURE SetStashFileType     (      filetype, creator: ARRAY OF CHAR);
PROCEDURE GetStashFileType     (VAR  filetype, creator: ARRAY OF CHAR);
```

On the Macintosh, any file is of a particular type and is associated with a particular application characterized by the creator, each given by a 4 character long string. The purpose and timing of the effects by these routines is exactly the same as that described for the routines affecting the name of the stash file. The predefined defaults are those inherited from the "Dialog Machine".

```
PROCEDURE SetDefltStashFileName( dsfn: ARRAY OF CHAR);
PROCEDURE GetDefltStashFileName(VAR dsfn: ARRAY OF CHAR);

PROCEDURE SetDefltStashFileType( dFileType,dCreator: ARRAY OF CHAR);
PROCEDURE GetDefltStashFileType(VAR dFileType,dCreator: ARRAY OF CHAR);
```

The above procedures allow to get or set the default name, type and creator of the stash file. Setting of the defaults will not show any effects until a reset is performed by the modeler using

```
PROCEDURE ResetStashFileNameAndType;
```

or by the simulationist, when executing the equivalent menu command *Settings/Reset Stash File*. Calling *ResetStashFileNameAndType* has the same effect as calling the corresponding *Set...* procedures with the current defaults.

```
PROCEDURE Message(m: ARRAY OF CHAR);
```

Writes the text *m* onto the stash file and inserts it in the table. Hereby, the string *m* is surrounded with quotation marks "" and preceded with the reserved word MESSAGE. This procedure allows to bring state events to the user's attention, which would otherwise slip by undetected or it helps the user to locate particular events while viewing large stash files.

```
PROCEDURE DumpGraph;
```

If the stash file is currently open (currently *stashFiling* attribute (F) for at least one monitorable variable, or the simulation environment mode *Always document run on stash file* is active), *DumpGraph* writes the current graph onto the stash file. The data are written in the so-called RTF-Format which can be opened by several, commercially available document processing software (s.a. previous section on recording flags in the entry form *Project description...* under menu *Settings*). [Not available in Reflex and PC version]

#### 7.5.4 GRAPHICAL MONITORING

The following objects allow to control the curve attributes used by ModelWorks for display of individual monitorable variables in the graph.

```
TYPE
  Stain =
    (coal, snow, ruby, emerald, sapphire, turquoise, pink, gold, autoDefCol);
 LineStyle =
    (unbroken, broken, dashSpotted, spotted, invisible, purge, autoDefStyle);

CONST
  autoDefSym = 200C;
```

Stains and colour variables from module *DMWindowIO* correspond to each other. They can be paired following this sequence:

```
black, white, red, green, blue, cyan, magenta, yellow
```

Stain *coal* is *black*, *snow* is *white*, *ruby* is *red* etc. The following line styles are available to connect points in the graph:



<i>unbroken</i>	_____
<i>broken</i>	-----
<i>dashSpotted</i>	-·-·-·-·-·-
<i>spotted</i>	.....
<i>invisible</i>	no drawing at all, may be used to stop drawing of a particular curve, while others are still drawn
<i>purge</i>	used to erase already drawn curves
<i>autoDefStyle</i>	line style will be determined by ModelWorks according to the automatic definition mechanism of curve attributes

To set or get defaults respectively current curve attributes for the monitorable variable *mv* belonging to model *m* use the following procedures:

```
PROCEDURE SetCurveAttrForMV(m: Model; VAR mv: REAL;
                             st: Stain; ls: LineStyle;
                             sym: CHAR);
PROCEDURE GetCurveAttrForMV(m: Model; VAR mv: REAL;
                             VAR st: Stain; VAR ls: LineStyle;
                             VAR sym: CHAR);

PROCEDURE SetDefltCurveAttrForMV(m: Model; VAR mv: REAL;
                                  st: Stain; ls: LineStyle;
                                  sym: CHAR);
PROCEDURE GetDefltCurveAttrForMV(m: Model; VAR mv: REAL;
                                  VAR st: Stain; VAR ls: LineStyle;
                                  VAR sym: CHAR);
```

Where:

<i>st</i>	<i>Stain</i> (color) is used to draw the lines and/or plotting symbols of a curve
<i>ls</i>	Style of the connecting lines drawn between monitoring points.
<i>sym</i>	Plotting symbol drawn at monitoring points

The latter two procedures which affect the defaults require a reset before becoming effective. This is not the case for the first two procedures, which take effect immediately. [Colours are not available in the PC version]. Resetting of the curve attributes is possible by means of

```
PROCEDURE ResetAllCurveAttributes;
```

which corresponds to the menu command *Settings /Reset all model's curve attributes* available to the simulationist at the standard ModelWorks user interface.

Note that if either *autoDefCol*, or *autoDefStyle*, or *autoDefSym* is used, the automatic definition mechanism for colours, line styles, and for symbols as provided by the simulation environment becomes active (see part II *Theory*, Tab. T1). Hence if you wish to really set a curve attribute, make sure that all(!) attributes are set different from an *autoDef* value.

In particular note, that the procedures affecting current values function also in the middle of a simulation. This behaviour may be useful, for instance to make a portion of a curve for a certain time invisible. A typical application is the simultaneous display of a measured time series and the monitoring of solutions of a system of differential equations; if some measurements are missing there arises the need to suppress partially the monitoring, i.e. to display nothing for the measurements but to monitor the behaviour of the model equations. Using procedure *SetCurveAttrForMV* with the line style *invisible* will allow to achieve the desired effect. The same effect can be achieved even with a much more convenient technique:

Assign the value *UndefREAL* from module *DMConversions* to the monitorable variable. Anytime ModelWorks encounters this value while plotting a monitorable variable, the corresponding curve is automatically interrupted and drawing resumed nicely as soon as the values are defined again. Note, however, this value may lead to program aborts if it is encountered as operand in calculations. Thus, make sure that this value is only assigned just for drawing purposes.

Note that if curve attributes are changed dynamically there may appear inconsistencies between the curve attributes used for the curves themselves and those used to draw the legend. This is because the legend shows only those curve attributes which are currently active while it is drawn. Unfortunately the simulation environment draws the legend in many situations for different reasons and the modeller can not directly control this drawing. However if the modeller follows the following guidelines there should result a satisfying behaviour: The model which changes curve attributes dynamically during the course of a simulation must set all curve attributes exactly as they should appear in the legend at the end of the procedure *Output* if current time  $t = t_0$  resp.  $k = k_0$  and always at the end of the procedure *Terminate* (for an example see the research sample model *LBM* module *LBMObs* in the *Appendix*).

```
PROCEDURE ClearGraph;
PROCEDURE ClearTable;
```

Clear the panel of the graph or thetable window, respectively.

#### 7.5.5 SIMULATION ENVIRONMENT MODES

The following four procedures allow to define the so-called simulation environment modes. They can be used to set under program control the preferences available to the simulationist under the menu command *File /Preferences*.

```
PROCEDURE SetDocumentRunAlwaysMode( dra: BOOLEAN);
PROCEDURE GetDocumentRunAlwaysMode(VAR dra: BOOLEAN);
```

If the mode «document run always» is activated, every execution of a simulation run will be documented onto stash file according to the current settings of the project descriptors. Note that the stash file gets rewritten with every new run.

```
PROCEDURE SetAskStashFileTypeMode( asft: BOOLEAN);
PROCEDURE GetAskStashFileTypeMode(VAR asft: BOOLEAN);
```

If the mode «ask for stash file type» is activated, every time the simulationist has selected a new stash file a dialogue is displayed allowing to specify the file's type and creator.

```
PROCEDURE SetRedrawTableAlwaysMode( rta: BOOLEAN);
PROCEDURE GetRedrawTableAlwaysMode(VAR rta: BOOLEAN);
```

The mode «redraw table always» describes the behaviour of the table window with respect to modifications of the tabulation monitoring settings. For further explanations see mode «redraw graph always» below.

```
PROCEDURE SetCommonPageUpRows( rows: CARDINAL);
PROCEDURE GetCommonPageUpRows(VAR rows: CARDINAL);
```

This mode controls the number of common rows between page ups in the table window. A page up occurs when the table window is full but more rows should be written; then ModelWorks attempts to erase most of the table and restarts tabulating from the top again. The number *rows* specifies how many rows at the bottom are not erased but scrolled to the top of the next page. The rest of the table window is then used to add the rows of the new page. Thus *rows* specifies how many rows are common to two consecutive pages.

```
PROCEDURE SetRedrawGraphAlwaysMode( rga: BOOLEAN);
PROCEDURE GetRedrawGraphAlwaysMode(VAR rga: BOOLEAN);
```

If the mode *RedrawGraphAlways* is activated, each modification of the graphing settings in the state *No Simulation* will be displayed immediately, not only at the begin of the next simulation run. This implies an immediate loss of all simulation results eventually currently visible in the graph as soon the simulationist edits any graphing settings. If this mode is not active, the current graph will not be touched unless the user starts another simulation; at its begin the whole graph will be redrawn.

```
PROCEDURE SetColorVectorGraphSaveMode( cvgs: BOOLEAN);
PROCEDURE GetColorVectorGraphSaveMode(VAR cvgs: BOOLEAN);
```

Above procedures allow to control the mode of graph restoration, graph printing, and transfer of graph into clipboard. If the mode «color and vector graph saving» is activated ( *cvgs* is TRUE), each time the graph window needs to be redrawn the graph will be reconstructed in colours. Restoration is necessary after some parts of it become visible again after they have been covered by another window (see also description of restore or update mechanism in module *DMWindows* of the "Dialog Machine"). Deactivation of this mode results in storing graphical output in a hidden bitmap without colours, with a coarser resolution and more modest memory requirements. Note that this mode won't affect the very first drawing of the graph, i.e. on a color screen you may still get coloured curves, even if this mode should be turned off. Since the full reconstruction in colours for complicated graphs may be slow, especially on monochrome monitors it may be preferable to deactivate this mode (trade-off between colours and speed). In addition to the colours all graphical output is stored as vectored objects. This allows printing and copying to the clipboard of graphs in high resolution quality, but requires a corresponding amount of memory. [Not available in Reflex and PC version]

#### 7.5.6 SETTING OF PREDEFINED DEFAULTS AND GLOBAL RESETTING

```
PROCEDURE SetPredefinitions;
```

Sets the defaults for the global simulation parameters, project description, stash file (name, type, creator) and the windows (positions, columns displays) to the ModelWorks-predefined values.

```
PROCEDURE ResetAll;
```

Resets all global simulation parameters, project description, stash file (name, type, creator), windows (positions, columns displays) as well as all declared models, state variables, parameters, monitorable variables (filing, tabulation, graphing, scaling, curve attributes) to their current defaults. This procedure corresponds to the menu command *Settings /Reset all above* available to the simulationist in the ModelWorks standard simulation environment.

#### 7.5.7 CUSTOMIZATION OF KEYBOARD SHORTCUTS FOR MENU COMMANDS

```
TYPE
  MWMenuCommand =
    (pageSetUpCmd, printGraphCmd, preferencesCmd, customizeCmd,
 (*core m.c.*) setGlobSimParsCmd, setProjDescrCmd, selectStashFileCmd,
  resetGlobSimParsCmd, resetProjDescrCmd, resetStashFileCmd,
  resetWindowsCmd, resetAllIntegrMethodsCmd,
  resetAllInitialValuesCmd, resetAllParametersCmd,
  resetAllStashFilingCmd, resetAllTabulationCmd,
  resetAllGraphingCmd, resetAllScalingCmd, resetAllCurveAttrsCmd,
  resetAllCmd, defineSimEnvCmd,
 (*core m.c.*) tileWindowsCmd, stackWindowsCmd, modelsCmd, stateVarsCmd,
```

```
(*core m.c.*) modelParamsCmd, monitorableVarsCmd, tableCmd, clearTableCmd,
(*core m.c.*) graphCmd, clearGraphCmd,
(*core m.c.*) startRunCmd, haltOrResumeRunCmd, stopCmd, startExperimentCmd);
```

Alias characters associated with ModelWorks menu-commands may be customized according to the needs of the simulationist either interactively (see menu command *File/Customize...* of the standard simulation environment) or by means of the following procedures. While an interactive specification is only possible for the most important commands, the so-called “core” menu commands (“core m.c.”), the client interface allows to modify the keyboard equivalents for all commands available in the standard ModelWorks user interface, except the ones listed under the menu *Edit*. The newly set alias characters for all ModelWorks menu commands are immediately used and remembered by the simulation environment when it is started the next time.

```
PROCEDURE SetMenuCmdAliasChar(cmd: MWMenuCommand; alias: CHAR);
PROCEDURE GetMenuCmdAliasChar(cmd: MWMenuCommand; VAR alias: CHAR);
```

Get, respectively set, an alias character (i.e. keyboard equivalent or shortcut) associated with a particular ModelWorks menu-command.

```
PROCEDURE ResetCoreMenuCmdsAliasChars;
PROCEDURE ResetAllMenuCmdsAliasChars;
```

Allow to reset the interactively specifiable alias characters and the alias characters of all menu commands, respectively, to their default values as described in *Part III, Reference: User Interface* of this manual. Note that setting or resetting of menu command alias characters is only possible in the states *No Model* or *No Simulation* .

# Appendix

The *Appendix* contains sample models, all given in complete source form, to demonstrate the possibilities of ModelWorks and of the auxiliary library modules. They have been carefully selected to be useful for many modellers, in particular for the beginner as well as the advanced modeller. Moreover, it contains technical information on ModelWorks which are needed in the daily use of ModelWorks. Finally, especially the material towards the end serves frequent reference purposes.

The *Appendix* contains the following chapters:

The chapter *Sample Models* explains the working of selected sample models which cover most of ModelWorks' more important features and lists their source code.

The chapter *Literature* lists the references of all literature cited throughout this text.

The chapter *ModelWorks Versions and Implementation* explains the main features of all available ModelWorks versions plus the availability of the RAMSES software.

The chapter *Use and Definitions of ModelWorks and Library Modules* explains the functioning and use of all the modules which are likely to be relevant for the modeler's work. It lists first the definition modules of the optional client interface and secondly of selected auxiliary modules often used in the context of modeling and simulation.

For the modeler the chapter *Quick References* serves as a quick reference for all the objects exported by the RAMSES software. In order to allow for quick access and better overview, these listings omit any comments and explanatory texts.

Any serious modeling with ModelWorks requires to consult the *Appendix* regularly and to carefully study at least those model definition programs of the chapter *Sample Models* which are similar to the ones the reader is working with.

Note, the *Appendix* contains no details on the installation and internal implementational aspects of the software architecture. To learn more about those topics, please consult the separate booklet "*Installation Guide and Technical Reference of the RAMSES Software*" distributed together with the RAMSES software package or read some of the publications listed in the part *Literature*

**Reading Hint:** For easier orientation, the pages, figures and tables of the *Appendix* are prefixed with the letter A. Within this part figures and tables are numbered separately, e.g. Fig. A1.

## A Sample Models

The following sample model definition programs have all been implemented and tested with the ModelWorks Macintosh versions V2.2 and V2.2/II, the IBM PC Windows-Version V2.2/PC, and some with the V2.0/Reflex and the PC Gem-Version 1.1/PC. Except for *GrassAphids*, they are distributed in source form, eventually even with some additional sample models not listed in this chapter. For the Macintosh all source files reside in the folder *Sample Models*, for the IBM PC in the directory `\MW\SAMPLES`. All distributed sample models are ready to be run.

The sample models have been selected according to the following criteria: a) They have been referenced in some parts of this text such as the part I *Tutorial* or part II *Theory*; b) they illustrate a typical use of ModelWorks to implement any of the fundamental elementary or structured model types; c) they demonstrate useful implementation techniques, or d) represent more advanced applications such as sensitivity analysis or parameter identification, which are often essential in the context of modeling and simulation of non-linear systems. It is recommended to study the sample models carefully, in particular those which appear to be similar to the applications in which the reader is interested; because of the open nature of ModelWorks it is important to understand the design principles behind ModelWorks not only in an abstract, but also in the specific ways for which the design strives to provide optimal solutions. Not only pictures can tell more than 1000 words!

## A.1 THE CONTINUOUS TIME SAMPLE MODELS (DESS) OF THE TUTORIAL

### A.1.1 The Sample Model “Logistic Grass Growth” - *Logistic*

The following listing defines the sample model described in the part I *Tutorial* in the section *Getting started with the simulation environment*. For explanations see the subsection *The sample model*.

```

MODULE Logistic;

  (*****
  (* MODEL: Logistic grass growth *)
  (* Author: mu, 9.4.88, ETHZ      *)
  (*****)

  FROM SimBase IMPORT
    Model, IntegrationMethod, DeclM, DeclSV, DeclP, RTCType,
    StashFiling, Tabulation, Graphing, DeclMV, SetSimTime,
    NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout,
    StateVar, Derivative, Parameter;

  FROM SimMaster IMPORT RunSimEnvironment;

  VAR
    m:      Model;
    grass:  StateVar;
    grassDot: Derivative;
    c1, c2: Parameter;

  PROCEDURE Dynamic;
  BEGIN
    grassDot:= c1*grass - c2*grass*grass;
  END Dynamic;

  PROCEDURE ModelObjects;
  BEGIN
    DeclSV(grass, grassDot,1.0, 0.0, 10000.0,
           "Grass", "G", "g dry weight/m^2");

    DeclMV(grass, 0.0,1000.0, "Grass", "G", "g dry weight/m^2",
           notOnFile, writeInTable, isY);
    DeclMV(grassDot, 0.0,500.0, "Grass derivative", "dG/dt", "g dry weight/m^2/day",
           notOnFile, notInTable, notInGraph);

    DeclP(c1, 0.7, 0.0, 10.0, rtc,
          "c1 (growth rate of grass)", "c1", "/day");
    DeclP(c2, 0.001, 0.0, 1.0, rtc,
          "c2 (self inhibition coefficient of grass)", "c2", "m^2/g dw/day");
  END ModelObjects;

  PROCEDURE ModelDefinitions;
  BEGIN
    DeclM(m, Euler, NoInitialize, NoInput, NoOutput, Dynamic,
          NoTerminate, ModelObjects, "Logistic grass growth model",
          "LogGrowth", NoAbout);
    SetSimTime(0.0,30.0);
  END ModelDefinitions;

  BEGIN
    RunSimEnvironment(ModelDefinitions);
  END Logistic

```

A.1.2 The New Model - *GrassAphids*

The following listing defines the sample model described in the manual part I *Tutorial* in the section *Getting started with modeling*. For explanations see the subsection *The new model*.

```

MODULE GrassAphids;

  (*****

MODEL: GrassAphids, Lotka-Volterra grass and aphids model

Frank Thommen, 29.11.91, ETHZ

*****)

FROM SimBase IMPORT
  Model, IntegrationMethod, DeclM, DeclSV, DeclP, RTCType,
  StashFiling, Tabulation, Graphing, DeclMV, SetSimTime,
  NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout,
  StateVar, Derivative, Parameter;

FROM SimMaster IMPORT RunSimEnvironment;

VAR
  m:           Model;
  grass, aphids: StateVar;
  grassDot, aphidsDot: Derivative;
  c1, c2, c3, c4, c5: Parameter;

PROCEDURE Dynamic;
BEGIN
  grassDot := c1*grass - c2*grass*grass - c3*grass*aphids;
  aphidsDot:= c3*c4*grass*aphids - c5*aphids;
END Dynamic;

PROCEDURE ModelObjects;
BEGIN
  DeclSV(grass, grassDot,200.0, 0.0, 10000.0,
    "Grass", "G", "g dry weight/m^2");
  DeclSV(aphids, aphidsDot,20.0, 0.0, 1000.0,
    "Aphids", "A", "g dry weight/m^2");

  DeclMV(grass, 0.0,10000.0, "Grass", "G", "g dry weight/m^2",
    notOnFile, writeInTable, isY);
  DeclMV(grassDot, 0.0,500.0, "Grass derivative", "dG/dt", "g dry weight/m^2/day",
    notOnFile, notInTable, notInGraph);
  DeclMV(aphids, 0.0, 1500.0,"Aphids", "A","g dry weight/m^2",
    notOnFile, writeInTable, isY);

  DeclP(c1, 0.4, 0.0, 10.0, rtc,
    "c1 (growth rate of grass)", "c1", "/day");
  DeclP(c2, 8.0E-5, 0.0, 1.0, rtc,
    "c2 (self inhibition coefficient of grass)", "c2", "m^2/g dw/day");
  DeclP(c3, 1.5E-3, 0.0, 1.0, rtc,
    "c3 (coupling parameter)", "c3", "m^2/g dw/day");
  DeclP(c4, 0.1, 0.0, 10.0, rtc,
    "c4 (ratio of grass net use by aphids)", "c4", "-");
  DeclP(c5, 0.2, 0.0, 10.0, rtc,
    "c5 (death rate of aphids)", "c5", "/day");
END ModelObjects;

PROCEDURE ModelDefinitions;
BEGIN

```



```
DeclM(m, Heun, NoInitialize, NoInput, NoOutput, Dynamic,  
      NoTerminate, ModelObjects, "Aphid-grass model (Lotka-Volterra)",  
      "GrassAphids", NoAbout);  
SetSimTime(0.0, 100.0);  
END ModelDefinitions;
```

```
BEGIN  
  RunSimEnvironment(ModelDefinitions);  
END GrassAphids
```

### A.2 A DISCRETE TIME MODEL (SQM) - *INSECT*

Insect populations often reproduce in distinct steps, which is particularly conspicuous for many univoltine insects. Should the growth of their population follow a logistic pattern, it could be easily modeled as a discrete time analogon (SQM) of the logistic growth model presented in part I *Tutorial* (s.a. above, *Logistic*). From the continuous time logistic growth equation (1)

$$dx(t)/dt = c_1x(t) - c_2x(t)^2 = c_1\left[1 - \frac{c_2}{c_1}x(t)\right]x(t) = c_1 \frac{\frac{c_1}{c_2} - x(t)}{\frac{c_1}{c_2}} x(t) = r \frac{K - x(t)}{K} x(t) \quad (1)$$

and the general relationship (2)

$$\frac{x(t+\Delta t) - x(t)}{\Delta t} \approx dx(t)/dt \quad (2)$$

we can derive with  $\Delta t = 1$  and  $t = k$  the following nonlinear difference equation (3), which describes a discrete time model for insect growth:

$$x(k+1) = x(k) + r \frac{K - x(k)}{K} x(k) = \left\{ 1 + r \left[ 1 - \frac{x(k)}{K} \right] \right\} x(k) \quad (3)$$

where:

State variable:	
insect density or # per ha grassland:	$x(t)$
Initial amount of insects/initial value:	$x(0) = 2.0 \text{ #/ha}$
Model parameters:	
intrinsic growth rate (year <sup>-1</sup> ):	$r = 0.7 \text{ year}^{-1}$
carrying capacity (#/ha):	$K = 7000 \text{ #/ha}$

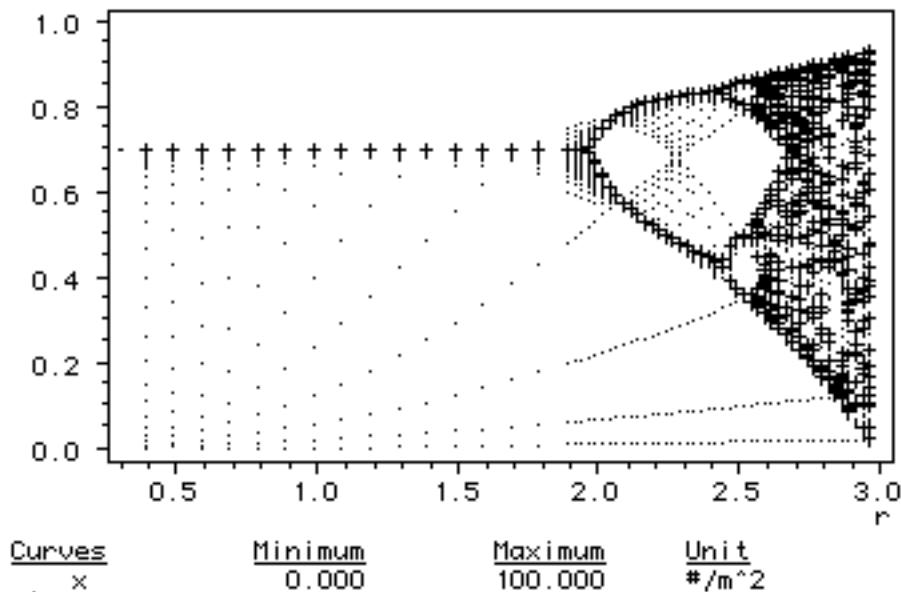


Fig. A1: Bifurcation plot drawn by the structured simulation (experiment) of the sample model *Insect*. Model behavior is shown in function of growth parameter  $r$ .

The following listing shows the corresponding model definition program *Insect*. In general it is very similar to the sample model *Logistic* (see above). However, instead of the derivative  $xDot$  we declare  $xNew$  of type *NewState*. It represents  $x(k+1)$  whereas  $x(k)$  is represented by  $x$ .

```

MODULE Insect;

( *****

MODEL: Insect      af, 21/Dez/93, ETHZ
Discrete time logistic growth, e.g. modeling the growth
of an insect population with non-overlapping generations

Remark: The installed experiment draws a bifurcation plot

***** )

FROM DMWindIO  IMPORT Write, WriteString, WriteLn;
FROM DMMessages IMPORT Ask;

FROM SimBase IMPORT
  Model, StateVar, NewState, Parameter, AuxVar, Derivative,
  IntegrationMethod, DeclM, DeclSV, DeclP, RTCType, StashFiling,
  Tabulation, Graphing, DeclMV, SetSimTime,
  NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout,
  PDeclared, SetMV, GetMV, SetP, SetRedrawGraphAlwaysMode,
  SetCurveAttrForMV, GetCurveAttrForMV, Stain, LineStyle;

FROM SimMaster IMPORT RunSimEnvironment, InstallExperiment,
  SimRun, CurrentStep, ExperimentAborted, ExperimentRunning;

VAR
  m : Model;
  x : StateVar;      xNew: NewState;      r,K : Parameter;
  rMin, rMax, rDeltaBig, rDelta: Parameter;

PROCEDURE Dynamic;
BEGIN
  xNew:= (1.0 + r*(1.0 - x/K))*x;
END Dynamic;

PROCEDURE About;
BEGIN
  WriteString("Difference equation form (SQM) of logistic population growth:"); WriteLn;
  WriteLn;
  WriteString("  x(k+1) = [ 1 + r * 1 - x(k)/K ] * x(k)"); WriteLn;
  WriteLn;
  WriteString("  where"); WriteLn;
  WriteString("    x - population density (# of insects per m^2)"); WriteLn;
  WriteString("    r - per capita growth rate"); WriteLn;
  WriteString("    K - carrying capacity (max. population density)"); WriteLn;
  WriteString("    k - discrete time k");
END About;

PROCEDURE Objects;
BEGIN
  DeclSV(x, xNew, 1.0, 0.0, 100.0, "Insect population density", "x", "#/m^2");

  DeclP(r, 0.3, 0.0, 10.0, rtc, "Growth rate of insect population", "r", "year^-1");
  DeclP(K, 70.0, 0.0, 100.0, rtc, "Carrying capacity", "K", "#/m^2");

  DeclMV(x, 0.0, 100.0, "Insect population density", "x", "#/m^2",
    notOnFile, writeInTable, isY);
END Objects;

```

```

PROCEDURE Output;
BEGIN
  IF ExperimentRunning() AND (CurrentStep())=70) THEN (* no more transient behavior *)
    SetCurveAttrForMV(m,x, sapphire, invisible, "+")
  END;
END Output;

```

```

PROCEDURE DrawBifurcationPlot;
  CONST yes = 1;
  VAR i,answer: INTEGER; curScaleMin, curScaleMax: REAL;
  curSF: StashFiling; curT,tbT: Tabulation; curG: Graphing;
  curStain: Stain; curLStyle: LineStyle; curSym, tbSym: CHAR;
BEGIN
  (* Add some model objects to support zooming into bifurcation plot *)
  IF NOT PDeclared(m,rDelta) THEN
    DeclMV(r, 0.0, 3.0, "Intrinsic growth rate of insects", "r", "year^-1",
      notOnFile, notInTable, notInGraph);
    DeclP(rMin, 0.3, 0.0, 3.0, rtc,
      "Begin of range of r to plot bifurcations", "rMin", "year^-1");
    DeclP(rMax, 3.0, 0.0, 3.0, rtc,
      "End of range of r to plot bifurcations", "rMax", "year^-1");
    DeclP(rDeltaBig, 0.1, 0.0, 10.0, rtc,
      "Increment (big) of r (before 1st bifurcation)", "rDeltaBig", "year^-1");
    DeclP(rDelta, 0.025, 0.0, 10.0, rtc,
      "Increment (small) of r (after 1st bifurcation)", "rDelta", "year^-1");
  END(*IF*);
  (* Prepare monitoring for bifurcation plot *)
  GetMV(m,r, curScaleMin, curScaleMax, curSF, curT, curG);
  SetMV(m,r, rMin, rMax, curSF, curT, isX);
  GetMV(m,x, curScaleMin, curScaleMax, curSF, curT, curG);
  GetCurveAttrForMV(m,x, curStain, curLStyle, curSym);
  Ask("Draw also transient behaviour?", "Yes|No", 8, answer);
  IF answer=yes THEN tbSym:= "."; tbT:= curT ELSE tbSym:= 0C; tbT:= notInTable END;
  SetMV(m,x, curScaleMin, curScaleMax, curSF, tbT, isY);
  r := rMin;
  WHILE (r<rMax) AND NOT ExperimentAborted() DO
    SetCurveAttrForMV(m,x, emerald, invisible, tbSym);
    SimRun;
    IF r<(2.0-rDeltaBig) THEN (* before very first bifurcation *)
      SetP(m,r, r + rDeltaBig)
    ELSE
      SetP(m,r, r + rDelta)
    END;
    r := r + rDelta;
  END(*WHILE*);
  (* Restore previous monitoring settings but avoid immediate
  clearing of bifurcation plot: *)
  SetRedrawGraphAlwaysMode(FALSE); (* defers implicit clearing *)
  SetMV(m,x, curScaleMin, curScaleMax, curSF, curT, curG);
  GetMV(m,r, curScaleMin, curScaleMax, curSF, curT, curG);
  SetMV(m,r, curScaleMin, curScaleMax, curSF, curT, notInGraph);
  SetCurveAttrForMV(m,x, curStain, curLStyle, curSym);
  SetRedrawGraphAlwaysMode(TRUE);
END DrawBifurcationPlot;

```

```

PROCEDURE ModelDefinitions;
BEGIN
  DeclM(m, discreteTime, NoInitialize, NoInput, Output, Dynamic,
    NoTerminate, Objects, "Logistic insect population dynamics",
    "LogGrowth", About);
  SetSimTime(0.0,100.0);
  InstallExperiment(DrawBifurcationPlot);
END ModelDefinitions;

```

BEGIN

```
RunSimEnvironment(ModelDefinitions);  
END Insect.
```

Note, in contrast to the continuous time logistic equation, this model exhibits an astonishingly wide array of behaviors, which depend on the value of parameter  $r$ : The equilibrium  $x(k) = K$  is asymptotically stable if  $r < 2$ , in particular the equilibrium is reached without any oscillations if  $r \leq 1$  and damped oscillations result if  $1 < r < 2$ ; neutrally stable oscillations are produced if  $r = 2$ ; if  $r > 2$  oscillations result, their amplitude increases with  $r$  and they eventually give way to deterministic chaotic behavior: In the range of  $2 \leq r \leq 2.449$  results a stable two-point cycle, between  $2.449 \leq r \leq 2.544$  a stable four-point cycle etc. till chaos ( $r > 2.57$ ). For instance observe the behaviour for  $r = 0.7$ ,  $1.99$  ( $t_{\text{end}} = 1'000$ ),  $2.0$  ( $t_{\text{end}} = 5'000$ ), and  $3.0$  ( $t_{\text{end}} = 100$ ) or execute the installed experiment (Fig. A1). For more details on this topic you may wish to read MAY (1974, 1975, 1976, 1981), for the mathematical background MAY & OSTER (1976), or on the relevance of chaotic models for ecological systems BERRYMAN & MILLSTEIN (1989).

### A.3 A DISCRETE EVENT MODEL (DEVS) - DIVERSITY

Assume we have an island which has been hit by a volcano eruption and all life has been wiped out. But there still exist  $n$  species on the continent. After how many years will the same diversity be reestablished on the island as on the continent?

To keep things as simple as possible let us assume that the island is very big, i.e. in our model we can ignore any subsequent reextinction. Given these assumptions, the distance of the island to the continent, and assuming per year a constant mean probability  $\lambda$  that an individual arrives on the island, we can model the dynamics of the number of species as the result of a Poisson process, which describes the arrival of individuals on the island.

For each species  $j$  we need to know whether it is present on the island or not. Thus the state vector of this model consists of elements which denote the number of individuals  $x_j$  which live on the island and belongs to species  $j$ . Since only arrivals can change the state, it follows the following instantaneous state transition function:

$$x_j(t) = \begin{cases} x_j(t^-)+1 & \text{an individual of species } j \text{ invades the island} \\ x_j(t^-) & \text{no individuals arriving on the island } \forall j \end{cases} \quad (5)$$

where

$x_j$     Number of individuals of species  $j$  living on the island    [#]  
 $t^-$     Continuous left-hand side of time before and up to the discrete event arrival

It models any changes which may occur in the state vector  $\underline{x}$ . In order to characterize the reestablishment of species on the island we use an evenness index  $E$  (6) based on the Shannon-Weaver diversity index (7a):

$$E = H / H_{\max} \quad (6)$$

where

$$H = \sum_{j=1}^n d_j \cdot \text{ld } d_j \quad (7a)$$

and if diversity maximal (even distribution,  $d_j = 1/n$ )

$$H_{\max} = \sum_{j=1}^n 1/n \cdot \text{ld } 1/n = \text{ld } n \quad (7b)$$

where

$\text{ld}$     logarithmus dualis  
 $n$     maximum number of species  
 $d_j$     relative density of species  $j$

In order to compute  $E$  we need for all species their relative densities  $d_j = x_j / \sum x_j$ . Finally, since we are interested in the time  $t^*$  required to restore the maximal diversity, i.e. when the diversity

index  $H$  according to equation (7a) comes close to  $H_{\max}$  as given by equation (7b), we have to know whenever the output variable  $E$  approaches approximately 1; this is the case if the following condition holds

$$1 - E \leq \epsilon \tag{7}$$

where

$\epsilon$  a small number

when  $t^*$  is set equal to the current time  $t$ . The goal of the simulation is to determine  $t^*$  in function of  $\lambda$  and  $n$ .

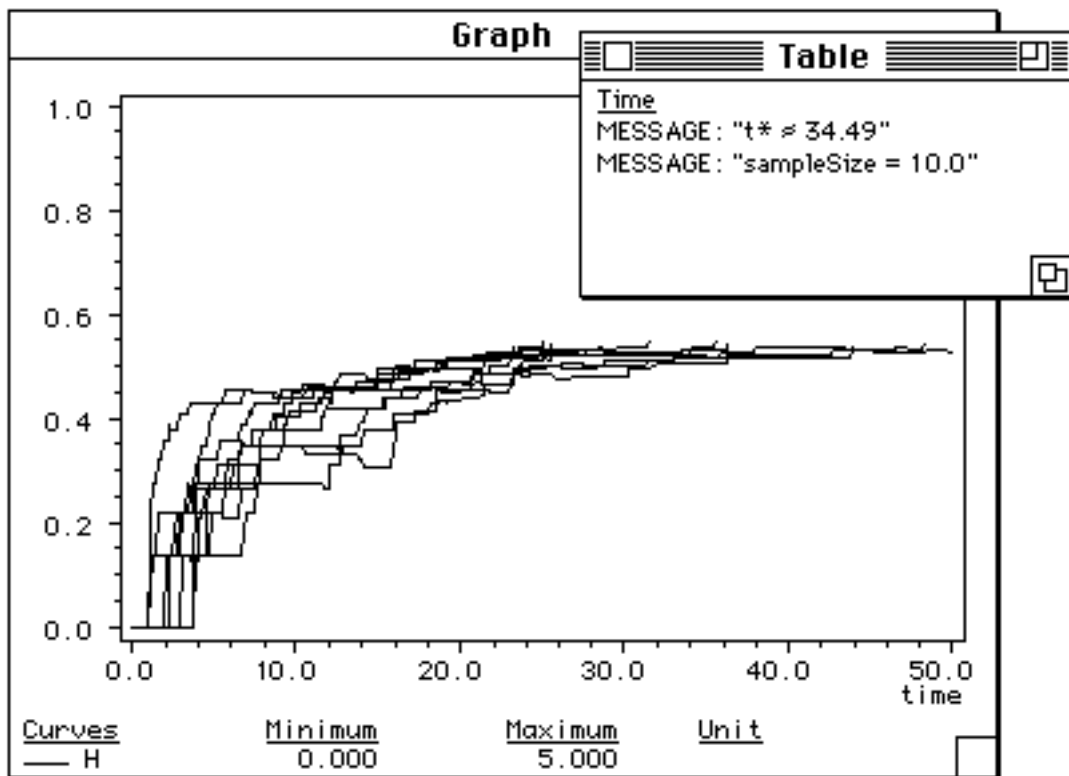


Fig. A2: Result of a stochastic simulation experiment made with the DEVS (Discrete event system specification) sample model *Diversity*. This model simulates the fate of the diversity on an island after it has been hit by a catastrophic vulcano eruption. Initially all species from the island have been wiped out. The reinvasion of individuals from the intact continent is simulated as a Poisson process with the probability  $\lambda$  of arrival of an individual of species  $j$  per time unit (with  $n=20$  species and a sample size of 10 simulation runs the average number of years needed to restore maximal diversity ( $\epsilon = 0.1$ ) has been estimated as  $t^* = 34.49$  years).

MODULE Diversity;

(\*\*\*\*\*)

MODEL: Diversity Restoration of diversity on an island  
after a vulcano eruption (a DEVS)

af, 21.Dec.93, ETHZ

```

*****
FROM DMConversions IMPORT IntToString, RealToString, RealFormat;
FROM DMStrings IMPORT Concatenate, Append;

FROM SimBase IMPORT
  Model, StateVar, NewState, Parameter, AuxVar, Derivative,
  IntegrationMethod, DeclM, DeclSV, DeclP, RTCType, StashFiling,
  Tabulation, Graphing, DeclMV, SetSimTime, NoInitialize,
  NoInput, NoOutput, NoTerminate, NoAbout, RemoveSV,
  notDeclaredModel, MDeclared, ClearTable, Message;
FROM SimMaster IMPORT RunSimEnvironment, SimRun,
  InstallExperiment, CurrentTime, StopRun, ExperimentAborted;
FROM SimEvents IMPORT nilTransaction, Transaction, StateTransition,
  ScheduleEvent, DeclDEVM;

FROM RandGen IMPORT U, Randomize;
FROM RandGen0 IMPORT InstallU0, SetJPar, J, SetNegExpPar, NegExp;
FROM MathLib IMPORT Ln;

CONST
  arrival = 1; (* EventClass *)
  nMax = 100; (* max. possible nr of species *)
  maxPopSize = 1000.0;

VAR
  m: Model;
  n: INTEGER; (* actual number of species on continent *)
  nPar: Parameter; (* used to determine n interactively via IO-window *)
  x : ARRAY [1..nMax] OF StateVar; (* population sizes *)
  lambda: Parameter;
  eps: Parameter;
  sampleSize: Parameter;
  H: AuxVar;
  Hmax: AuxVar;
  tStar: AuxVar;

PROCEDURE Initialize;
  VAR j: INTEGER; dummyNewState: NewState; descr: ARRAY [0..63] OF CHAR;
  ident, jStr: ARRAY [0..15] OF CHAR;
BEGIN
  n := TRUNC(nPar);
  FOR j:= 1 TO n DO
    IntToString(j, jStr, 0);
    Concatenate("Size of population of species ", jStr, descr);
    Concatenate("x[" , jStr, ident); Append(ident, "]");
    DeclSV(x[j], dummyNewState, 0.0, 0.0, maxPopSize/nPar, descr, "#");
  END(*FOR*);
  Hmax := Ln(nPar);
  SetJPar(1, n);
  SetNegExpPar(lambda);
  ScheduleEvent(arrival, NegExp(), nilTransaction);
END Initialize;

PROCEDURE Arrival(ta: Transaction);
  VAR j: INTEGER;
BEGIN
  j:= J(); x[j] := x[j] + 1.0;
  ScheduleEvent(arrival, NegExp(), nilTransaction);
END Arrival;

PROCEDURE Output;
  VAR sumX, dj: REAL; j: INTEGER;
BEGIN
  sumX:= 0.0; FOR j:= 1 TO n DO sumX:= sumX + x[j] END;
  H := 0.0;

```



```

FOR j:= 1 TO n DO IF x[j]>0.0 THEN dj := x[j]/sumX; H := H + Ln(dj)*dj END END(*FOR*);
H := -H;
IF (1.0-H/Hmax)<=eps THEN StopRun END(*IF*);
END Output;

```

```

PROCEDURE Terminate;
  VAR j: INTEGER;
BEGIN
  tStar := CurrentTime();
  n := TRUNC(nPar);
  FOR j:= 1 TO n DO RemoveSV(m,x[j]) END(*FOR*);
END Terminate;

```

```

PROCEDURE EstimateTStarHat;
  VAR tStarHat,tStarSum: REAL; k: INTEGER;
  PROCEDURE Report(m: ARRAY OF CHAR; x: REAL);
    VAR rStr: ARRAY [0..15] OF CHAR; msg: ARRAY [0..127] OF CHAR;
  BEGIN (*Report*)
    RealToString(x,rStr,0,2,FixedFormat);
    Concatenate(m, rStr, msg);
    Message(msg);
  END Report;
BEGIN (*EstimateTStarHat*)
  k := 0; tStarSum := 0.0;
  WHILE NOT ExperimentAborted() AND (FLOAT(k)<sampleSize) DO
    SimRun;
    IF NOT ExperimentAborted() THEN tStarSum := tStarSum + tStar; INC(k) END;
  END(*WHILE*);
  IF k>0 THEN
    tStarHat := tStarSum/FLOAT(k);
    ClearTable;
    Report("t* ≈ ", tStarHat);
    Report("sampleSize = ", FLOAT(k));
  END(*IF*);
END EstimateTStarHat;

```

```

PROCEDURE ModelObjects;
BEGIN (*ModelObjects*)
  DeclMV(H, 0.0, 5.0, "Shannon-Weaver diversity index", "H", "",
    notOnFile, notInTable, isY);

  DeclP(nPar, 20.0, 0.0, FLOAT(nMax), noRtc,
    "# of species on continent", "nPar", "# spec.");
  (* actual state vector declaration deferred to Initialize *)

  DeclP(lambda, 1.0, 0.0, 10000.0, noRtc,
    "Mean # individuals arriving on island per Δt", "lambda", "#/year");
  DeclP(eps, 0.1, 0.0, 1.0, rtc,
    "Relative tolerance between H and Hmax", "eps", "%");
  DeclP(sampleSize, 10.0, 0.0, 10000.0, noRtc,
    "# of runs in experiment", "sampleSize", "#");
END ModelObjects;

```

```

PROCEDURE ModelDefinitions;
  VAR stf: ARRAY [arrival..arrival] OF StateTransition;
BEGIN
  stf[arrival].ec := arrival; stf[arrival].fct := Arrival;
  DeclDEV(m, Initialize, NoInput, Output, stf, Terminate, ModelObjects,
    "Restoration of diversity after vulcano eruption", "Diversity", NoAbout);
  SetSimTime(0.0,50.0);
  InstallExperiment(EstimateTStarHat);
END ModelDefinitions;

```

```

BEGIN
  InstallU0(U);

```

```

RunSimEnvironment(ModelDefinitions);
END Diversity.

```

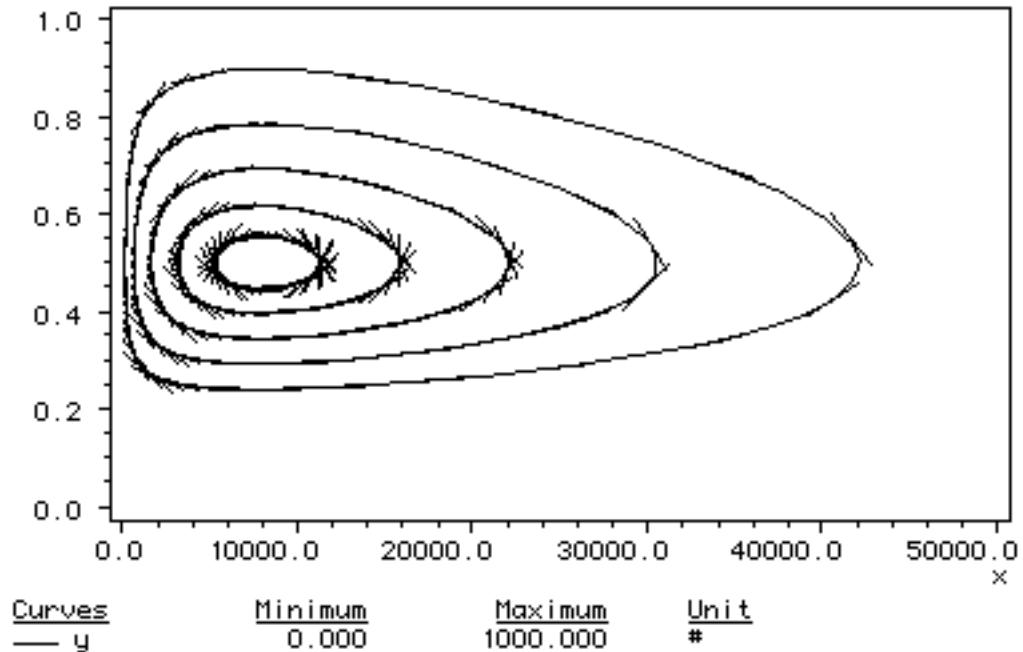
The instantaneous state transition function (5) is implemented in form of a separate procedure *Arrival*, which is passed as actual argument instead of procedure *dynamic* while declaring the DEVS model (*DeclDEVM*). This procedure is called by ModelWorks whenever an event of type *arrival*, i.e. the arrival of an individual on the island, is encountered.

Such events are best scheduled by the procedure *Arrival*, since the time interval to be elapsed between two events, can be determined according to following reasons: From theory follows that the distribution of the time intervals  $\tau$  between the elementary events follows a negative exponential model, i.e. the probability that  $\lambda$  events occur in a time interval of length  $\tau$  follows the cumulative distribution function  $F(\tau) = 1 - e^{-\lambda\tau}$ . Variates from such a distribution can be produced by using the random number generator *NegExp* from module *RandGen*. Hence, in order to simulate a series of events of type *arrival* is produced by calling procedure *ScheduleEvent(arrival, tau, nilTransaction)* after having executed the instantaneous state transition function (5). Note that by definition instantaneous state transition functions may update the state vector immediately. To get the sequence of event scheduling started, procedure *Initialize* calls *ScheduleEvent* at least once. To set the parameter  $\lambda$ , the *SetNegExpPar* is called at the begin of each run (in procedure *Initialize*) and consequently  $\lambda$  is a parameter which may not be changed while a simulation is running (*noRtc*). As soon as  $t^*$  can be determined, any further simulation would be useless; in this situation the current run can be stopped, e.g. by calling *StopRun* from *SimMaster*.

Since  $t^*$  is actually the result of a stochastic process, many simulation runs should be executed before  $\hat{t}^*$  is determined (Fig. A2). Therefore the experiment procedure *EstimateTStarHat* is installed as an experiment and performs several simulation runs before the mean  $\hat{t}^*$  is computed at the end of the experiment *EstimateTStarHat* (s.a. below section *Stochastic Simulations*) and *Message* from *SimBase* is used to display the result (Fig. A2).

The dimension of the state vector is variable, since it depends on the parameter *nPar*. The latter is implemented like any other model parameter, i.e. it can be changed via the IO-window *Model Parameters*. As a consequence, the state variables are not declared within procedure *ModelObjects*. Instead procedure *Initialize* declares the state vector at the begin of a simulation run and procedure *Terminated* discards it immediately after completion of a run. Therefore, the state vector should not be changed during simulations, hence *nPar* is of type *noRtc*.

## A.4 TYPICAL APPLICATIONS

A.4.1 Batch Phase Portrait of Lotka-Volterra - *LVPhasePlot*

The following model definition program allows to simulate the famous Lotka-Volterra predator-prey model and to produce a phase portrait in the state space (Fig. A3) by means of a preprogrammed experiment, i.e. procedure *PhasePortrait*

```

MODULE LVPhasePlot; (* af 15/01/88, dg 06/03/93, dg 25/04/96 *)

(*****
(* Lotka-Volterra prey-predator model *)
*****)

FROM DMWindIO IMPORT
  SetPos, WriteString, SetPen, LineTo, SetColor, cyan,
  SetClipping, RemoveClipping;

FROM DMWindows IMPORT RectArea;

FROM SimBase IMPORT
  DeclM, IntegrationMethod, DeclSV, StashFiling, Tabulation,
  Graphing, DeclMV, DeclP, RTCType, Model, SetSimTime,
  InstallClientMonitoring, TileWindows, DoNothing, SetSV,
  SetDefltWindowArrangement, MWWindowArrangement, GetMV, NoInput,
  NoOutput, NoTerminate, StateVar, Derivative, Parameter;

FROM SimMaster IMPORT
  RunSimEnvironment, SimRun, InstallExperiment, CurrentTime;
  
```

```

FROM SimGraphUtils IMPORT
  SelectForOutputGraph, GraphToWindowPoint, WindowToGraphPoint;

VAR
  m: Model;
  x, y: StateVar;
  xDot, yDot: Derivative;
  c1,c2,c3,c4,c5: Parameter;
  runNo: INTEGER; withVectors: Parameter;
  monIntCount: INTEGER; phaseSpaceGraph: BOOLEAN;
  curSMinx, curSMaxx, curSMiny, curSMaxy: REAL;
  panelr: RectArea; vectorInt: Parameter;

PROCEDURE ShowEqus;
  CONST lm = 3;
BEGIN
  SetPos(3,lm); WriteString("Lotka-Volterra prey (x) - ");
  WriteString("predator (y) model");
  SetPos(5,lm); WriteString(" dx/dt = c1*x - c2*x*x - c3*x*y");
  SetPos(6,lm); WriteString(" dy/dt = c3*c4*x*y - c5*y");
END ShowEqus;

PROCEDURE Initialize;
BEGIN
  CASE runNo OF
    1: SetSV(m,x,4.0E3); SetSV(m,y,250.0);
    | 2: SetSV(m,x,5.0E3); SetSV(m,y,300.0);
    | 3: SetSV(m,x,6.0E3); SetSV(m,y,350.0);
    | 4: SetSV(m,x,7.0E3); SetSV(m,y,400.0);
    | 5: SetSV(m,x,8.0E3); SetSV(m,y,450.0);
    | 6: SetSV(m,x,8.0E3); SetSV(m,y,475.0);
  ELSE
    END(*CASE*);
END Initialize;

PROCEDURE PhasePortrait;
BEGIN
  FOR runNo:= 1 TO 5 DO SimRun END(*FOR*); runNo:= 0;
END PhasePortrait;

PROCEDURE Dynamic;
BEGIN
  xDot:= c1*x - c2*x*x - c3*x*y;
  yDot:= c3*c4*x*y - c5*y;
END Dynamic;

PROCEDURE InitClientMonit;
  VAR curSF: StashFiling; curT: Tabulation; curGx,curGy: Graphing;
BEGIN
  GetMV(m,x, curSMinx,curSMaxx, curSF, curT, curGx);
  GetMV(m,y, curSMiny,curSMaxy, curSF, curT, curGy);
  phaseSpaceGraph := ((curGx = isX) AND (curGy = isY));
  IF phaseSpaceGraph THEN
    WITH panelr DO
      GraphToWindowPoint(curSMinx,0.0,x,y);
      GraphToWindowPoint(curSMaxx,0.0,w,y); w := w-x;
      GraphToWindowPoint(curSMinx,1.0,x,h); h := h-y;
      INC(x); INC(y); DEC(w,2); DEC(h,2);
    END(*WITH*);
    monIntCount := 0;
  END(*IF*);
END InitClientMonit;

```

```

PROCEDURE DrawVectors;
  CONST vele = 10;
  VAR xx,yy: INTEGER; dx,dy, slope, x1,x2,y1,y2: REAL; clipr: RectArea;

  PROCEDURE Min(x,y: INTEGER): INTEGER;
  BEGIN
    IF x<y THEN RETURN x ELSE RETURN y END;
  END Min;

  PROCEDURE Max(x,y: INTEGER): INTEGER;
  BEGIN
    IF x>y THEN RETURN x ELSE RETURN y END;
  END Max;

BEGIN (*. DrawVectors .*)
  IF phaseSpaceGraph AND (withVectors>0.0)
    AND ((monIntCount MOD TRUNC(vectorInt)) = 0)
  THEN
    SelectForOutputGraph; SetColor(cyan);
    slope := yDot/xDot;
    GraphToWindowPoint(x,(y-curSMiny)/(curSMaxy-curSMiny),xx,yy);
    WITH clipr DO
      x := Max(xx-vele,panelr.x); y := Max(yy-vele,panelr.y);
      w := Min(xx+vele,panelr.x+panelr.w); w := w - x;
      h := Min(yy+vele,panelr.y+panelr.h); h := h - y;
    END(*WITH*);
    WindowToGraphPoint(xx-vele,yy,x1,y1); dx := x1-x; y1 := y+slope*dx;
    WindowToGraphPoint(xx+vele,yy,x2,y2); dx := x2-x; y2 := y+slope*dx;
    GraphToWindowPoint(x1,(y1-curSMiny)/(curSMaxy-curSMiny),xx,yy);
    SetPen(xx,yy);
    GraphToWindowPoint(x2,(y2-curSMiny)/(curSMaxy-curSMiny),xx,yy);
    SetClipping(clipr);
    LineTo(xx,yy);
    RemoveClipping;
  END(*IF*);
  INC(monIntCount);
END DrawVectors;

PROCEDURE ModelObjects;
BEGIN
  DeclSV(x, xDot,4.0E3, 0.0, 1.0E5,
    "Prey population (density)", "x", "#");
  DeclSV(y, yDot,250.0, 0.0, 1.0E4,
    "Predator population (density)", "y", "#");

  DeclMV(x, 0.0,50000.0, "Prey population (density)", "x",
    "#", notOnFile, writeInTable, isX);
  DeclMV(y, 0.0, 1000.0,"Predator population (density)",
    "y", "#", notOnFile, writeInTable, isY);

  DeclP(c1, 1.0, 0.0, 100.0, rtc,
    "c1 (birth rate of x)", "c1", "/time");
  DeclP(c2, 0.0, 0.0, 1.0, rtc,
    "c2 (self inhibition coefficient of x)", "c2", "/#/time");
  DeclP(c3, 2.0E-3, 0.0, 1.0, rtc,
    "c3 (coupling parameter)", "c3", "/#/time");
  DeclP(c4, 0.5E-2, 0.0, 10.0, rtc,
    "c4 (ratio of x net use by y)", "c4", "---");
  DeclP(c5, 0.08, 0.0, 10.0, rtc,
    "c5 (death rate of y)", "c5", "/time");
  DeclP(vectorInt, 3.0, 1.0, 1000.0, rtc,
    "# monitoring intervals when to draw vectors", "vectorInt",
    "monitoring intervals");
  DeclP(withVectors, 1.0, 0.0, 1.0, rtc,
    "Draw vectors (0-no/1-yes)", "withVectors", "");
  runNo:= 0;

```

```
END ModelObjects;
```

```
PROCEDURE ModelDeclaration;
```

```
BEGIN
```

```
  DeclM(m, Heun, Initialize, NoInput, NoOutput, Dynamic, NoTerminate,  
        ModelObjects,
```

```
        "Lotka-Volterra prey-predator model", "LV-model", ShowEqus);
```

```
  SetSimTime(0.0,30.0);
```

```
  InstallExperiment(PhasePortrait);
```

```
  InstallClientMonitoring(InitClientMonit,DrawVectors,DoNothing);
```

```
  TileWindows; SetDeflWindowArrangement(current);
```

```
END ModelDeclaration;
```

```
BEGIN
```

```
  RunSimEnvironment(ModelDeclaration);
```

```
END LVPhasePlot.
```

#### A.4.2 Interactive Phase Portrait of the Van-der-Pol Oscillator - *VDPol*

The following model definition program allows to simulate the famous Van-der-Pol oscillator and to determine interactively, i.e. by a mouse-click, the starting point of trajectories in the state space.

The Van-der-Pol oscillator is given by this autonomous, non-linear, second order differential equation

$$\ddot{z} - \mu(1 - z^2)\dot{z} + z = 0$$

This equation can be brought into canonical form by introducing the state variables  $x$  and  $y$  defined as follows:

$$x = z$$

$$y = \dot{z} \Rightarrow \dot{y} = \ddot{z} = \mu(1 - z^2)\dot{z} - z$$

and we obtain a second order system of ordinary, first order differential equations (canonical form):

$$\dot{x} = y$$

$$\dot{y} = \mu(1 - x^2)y - x$$

This system has an unstable singularity in the origin of the state space and an asymptotically stable limit cycle (Fig. A4).

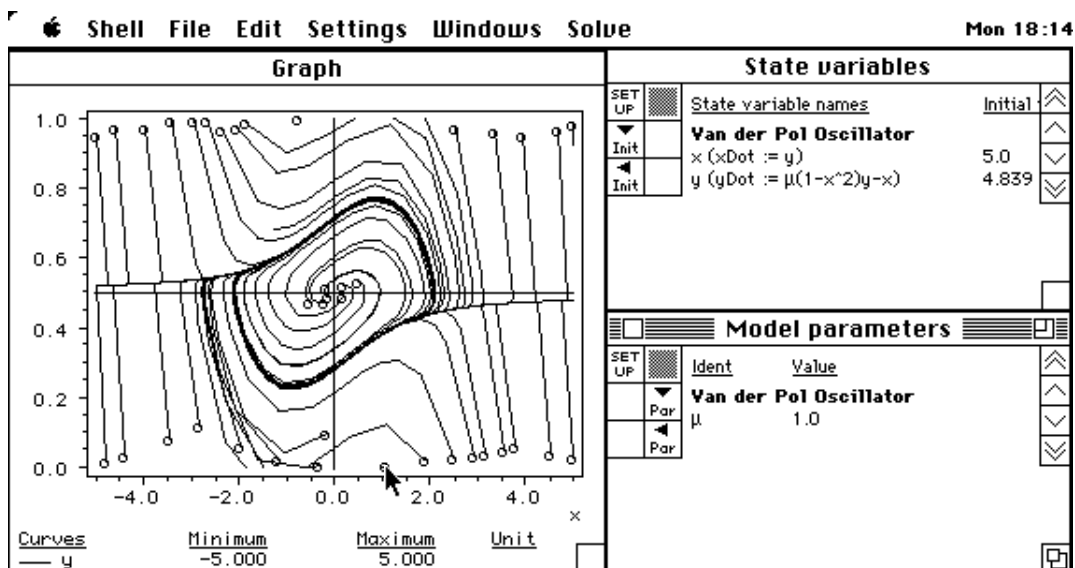


Fig. A4: Phase portrait of the Van-der-Pol oscillator, a second order, autonomous non-linear system of ordinary differential equations. This sample model definition program demonstrates the use of the "Dialog Machine" in conjunction with the auxiliary library module *SimGraphUtils*. It allows the simulationist to set interactively the new initial conditions in the state space shown in the window *Graph* (mouse was clicked at points marked with o). Once a new initial state vector has been defined, the next run will produce a trajectory starting at the clicked point.

This implementation demonstrates the use of the "Dialog Machine" and *SimGraphUtils* to set interactively the initial values of the state vector of a second order model system. Given the simulationist has set the monitoring currently such, that the window *Graph* displays the state space, a mouse click into the graph window's content first halts any eventually still running simulation, then interprets the point where the mouse has been clicked as the new initial values of the state vector, and marks this point with the character 'o'. As soon as the simulationist now starts another simulation run by choosing the menu command *Solve/Start run* (for instance with the keyboard equivalent *R*) the next trajectory drawn starts at the clicked point. This behavior allows to construct interactively a phase portrait as depicted in Fig. A4.

This program behavior is achieved by installing for the window *Graph* a handler, i.e. procedure *GetAndSetNewInits*. To install *GetAndSetNewInits* use procedure *InstallGraphClickHandler* from module *SimGraphUtils* from the optional client interface of ModelWorks. ModelWorks installs then the handler into the "Dialog Machine", which calls *GetAndSetNewInits* each time the user clicks into the content of the window *Graph*.

ModelWorks' client monitoring mechanism is used to draw an extra horizontal line through the origin, normally not shown by ModelWorks. *DrawAbscissa* is installed as the client monitoring initialization routine, i.e. as actual argument for formal parameter *initClientMon* of procedure *InstallClientMonitoring* from module *SimBase*. Remember that the ordinate is always scaled to interval [0..1] only, because the graph must be able to display more than just one curve.

These implementations of *GetAndSetNewInits* as well as *DrawAbscissa* function for both state space representations, i.e. for the graph *x* vs. *y* as well as *y* vs. *x*; the procedures *IsMVOnOrdinate* respectively *IsMVOnAbscissa* allow to determine which monitorable variable, i.e. *x* or *y*, is currently displayed as the ordinate resp. abscissa. All algorithms will then adjust accordingly, regardless which way the simulationist happens to have specified the state space representation.

MODULE *VDPol*;

(\*\*\*\*\*)

Model: *VDPol*

Copyright (c) 1989 by Andreas Fischlin and Harald Bugmann  
 & Swiss Federal Institute of Technology Zurich ETHZ  
 Systems Ecology Group  
 ETH-Zentrum  
 CH-8092 Zurich  
 Switzerland

Purpose:

Simulation of the Van-der-Pol oscillator with an unstable singularity in the origin of the state space and a asymptotically stable limit cycle. This implementation, a ModelWorks model definition program, allows to determine interactively the initial conditions by clicking with the mouse into the window *Graph*, given it displays currently the state space.

Implementation and Revisions:

=====

Author	Date	Description
-----	----	-----
af	29/07/87	First implementation
af	26/01/93	Clicking into state space to



```

                                set new initial condition added
dg      04/03/93      Import lists cleaned up
af      18/03/93      New support by SimGraphUtils used
dg      25/04/96      Cleaned up for PC compatibility

*****

FROM DMWindIO IMPORT
  SetColor, black, SetPen, LineTo, GetLastMouseClicked, ClickKind,
  DrawSym, Color;
FROM SimGraphUtils IMPORT
  Abscissa, CurrentAbscissa, SelectForOutputGraph, PointToMVVal,
  MVValToPoint, InstallGraphClickHandler, StainToColor;
FROM SimBase IMPORT
  StateVar, Derivative, Parameter, DeclM, IntegrationMethod,
  DeclSV, GetMV, GetDefltSV, StashFiling, Tabulation, Graphing,
  DeclMV, DeclP, RTCType, Model, SetSimTime, SetMonInterval,
  SetIntegrationStep, SetDefltCurveAttrForMV, Stain, LineStyle,
  GetCurveAttrForMV, autoDefSym, SetSV, InstallClientMonitoring,
  NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout,
  DoNothing;
FROM SimMaster IMPORT RunSimEnvironment, StopRun;

(*****

VAR
  m:          Model;
  x, y:       StateVar;
  xDot, yDot: Derivative;
  mu:        Parameter;

PROCEDURE GetAndSetNewInits;
  VAR xr,yr: REAL; curG: Graphing; curStain: Stain; curCol: Color;
      curLineStyle: LineStyle; curSym: CHAR; xi,yi: INTEGER; click: ClickKind;
BEGIN
  GetLastMouseClicked(xi,yi,click);
  xr := PointToMVVal(xi,yi,m,x,curG);
  IF (curG=isX) OR (curG=isY) THEN SetSV(m,x,xr) END;
  IF (curG=isY) THEN GetCurveAttrForMV(m,x,curStain, curLineStyle, curSym) END;
  yr := PointToMVVal(xi,yi,m,y,curG);
  IF (curG=isX) OR (curG=isY) THEN SetSV(m,y,yr) END;
  IF (curG=isY) THEN GetCurveAttrForMV(m,y,curStain, curLineStyle, curSym) END;
  SelectForOutputGraph;
  StainToColor(curStain,curCol); SetColor(curCol);
  SetPen(xi,yi); DrawSym('o');
  StopRun;
END GetAndSetNewInits;

PROCEDURE DrawAbscissa; (* is initClientMon procedure *)
  VAR xx,yy: INTEGER; curX: Abscissa; curGx,curGy: Graphing;
BEGIN
  SelectForOutputGraph; SetColor(black);
  CurrentAbscissa(curX);
  xx := MVValToPoint(0.0,m,x,curGx);
  yy := MVValToPoint(0.0,m,y,curGy);
  IF (curGx=isX) AND (curGy=isY) THEN
    SetPen(MVValToPoint(curX.xMin,m,x,curGx),yy);
    LineTo(MVValToPoint(curX.xMax,m,x,curGx),yy);
  ELSIF (curGy=isX) AND (curGx=isY) THEN
    SetPen(MVValToPoint(curX.xMin,m,y,curGy),xx);
    LineTo(MVValToPoint(curX.xMax,m,y,curGy),xx);
  END(*IF*);
END DrawAbscissa;

PROCEDURE Dynamic;
```

```

BEGIN
  xDot:=y;
  yDot:= mu*(1.0-x*x)*y-x;
END Dynamic;

PROCEDURE ModelObjects;
BEGIN
  DeclSV(x, xDot,1.0, -5.0, +5.0, "x (xDot := y)", "x", "");
  DeclSV(y, yDot,1.0, -5.0, +5.0, "y (yDot :=  $\mu(1-x^2)y-x$ ", "y", "");

  DeclMV(x,-5.0,5.0,"x (abscissa)", "x", "",notOnFile,notInTable,isX);
  SetDefltCurveAttrForMV(m,x,ruby,unbroken,0C);
  DeclMV(y,-5.0,5.0,"y (ordinate)", "y", "",notOnFile,notInTable,isY);
  SetDefltCurveAttrForMV(m,y,emerald,unbroken,0C);

  DeclP(mu, 1.0, -10.0, 10.0, rtc, " $\mu$  (oscillator parameter)", "mu", "");
END ModelObjects;

PROCEDURE ModelDefinitions;
BEGIN
  DeclM(m, Euler, NoInitialize, NoInput, NoOutput, Dynamic, NoTerminate, ModelObjects,
    "Van der Pol Oscillator", "VDPol", NoAbout);
  SetSimTime(0.0,20.0); SetMonInterval(0.2); SetIntegrationStep(0.05);
  InstallClientMonitoring(DrawAbscissa,DoNothing,DoNothing);
  InstallGraphClickHandler(GetAndSetNewInits);
END ModelDefinitions;

BEGIN
  ModelDefinitions;
  RunSimEnvironment(DoNothing);
END VDPol.

```

A.4.3 Animation of the Age Pyramid of the Swiss - *SwissPop*

The following sample model simulates the Swiss human population starting from the demographic state and properties in the year 1988. The model considers only the age structure ( $n = 100$  age classes) differentiated for sex but neither immigration nor emigration. Hence the model has been formulated as an autonomous, linear, discrete time<sup>1</sup> Leslie matrix model with the following equations:

$$\underline{x}(k+1) = L \underline{x}(k)$$

where

$$L = \begin{pmatrix} f_0 & f_1 & f_2 & \dots & f_{a-3} & f_{a-2} & f_{a-1} \\ s_{01} & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & s_{12} & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & s_{23} & \dots & 0 & 0 & 0 \\ \dots & & & & & & \\ 0 & 0 & 0 & \dots & s_{n-3n-2} & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & s_{n-2n-1} & 0 \end{pmatrix}$$

All demographic parameters such as fecundity  $f_i$  and survival  $s_{ij}$  are assumed to remain constant and were derived from official statistics valid for the year  $k_0 = 1988$ . This sample model definition program also demonstrates the reading of data, i.e. the initial state vector, from a file by means of the auxiliary library module *ReadData*

*ReadInitialStateVectorFromFile* first to open the data file *SwissPop88.DAT* automatically by calling procedure *OpenDataFile* from *ReadData*. If *OpenDataFile* can't open the data file, it will display first a message informing the simulationist about the problem and then ask her to locate the data file via the standard file opening dialog (*GetExistingFile* from *DMFiles*).

*ReadInitialStateVectorFromFile* is installed in the simulation environment via procedure *InstallDefSimEnv* from *SimMaster*. Therefore *ReadInitialStateVectorFromFile* will be called automatically during the initialization of the simulation environment (see part II *Theory*, section *Initialization of the simulation environment*), but can be called again as many times the simulationist wishes, e.g. to use another file or after having edited the file. *SwissPop88.DAT*. Therefore, if *stateVectorInitialized* is true *ReadInitialStateVectorFromFile* uses *OpenDataFile* which will always open the data file via the standard file opening dialog.

The following excerpt from the data file *SwissPop88.DAT* shows what data the procedure *ReadInitialStateVectorFromFile* expects:

age class	size
0	79700
1	75700
2	74900
3	74000
...	
...	
97	1484
98	1115
99	746
100	377

<sup>1</sup>This example represents the special case where the discrete time  $\kappa$  is restricted to integer numbers  $k$  only.

The data are read free-format, however *ReadInitialStateVectorFromFile* expects first a header line, which it will skip, and then exactly 101 data pairs, each consisting of the index and the size of the age class. *ReadData* will test for each expected number its syntax and the plausibility of the read value by comparing it with an interval (see *GetInt* or *GetReal* from *ReadData*). Whenever *ReadData* detects an error condition, it will inform the user about the cause and location at which the error was detected and asks the user whether she wishes to debug, continue or abort the reading process.

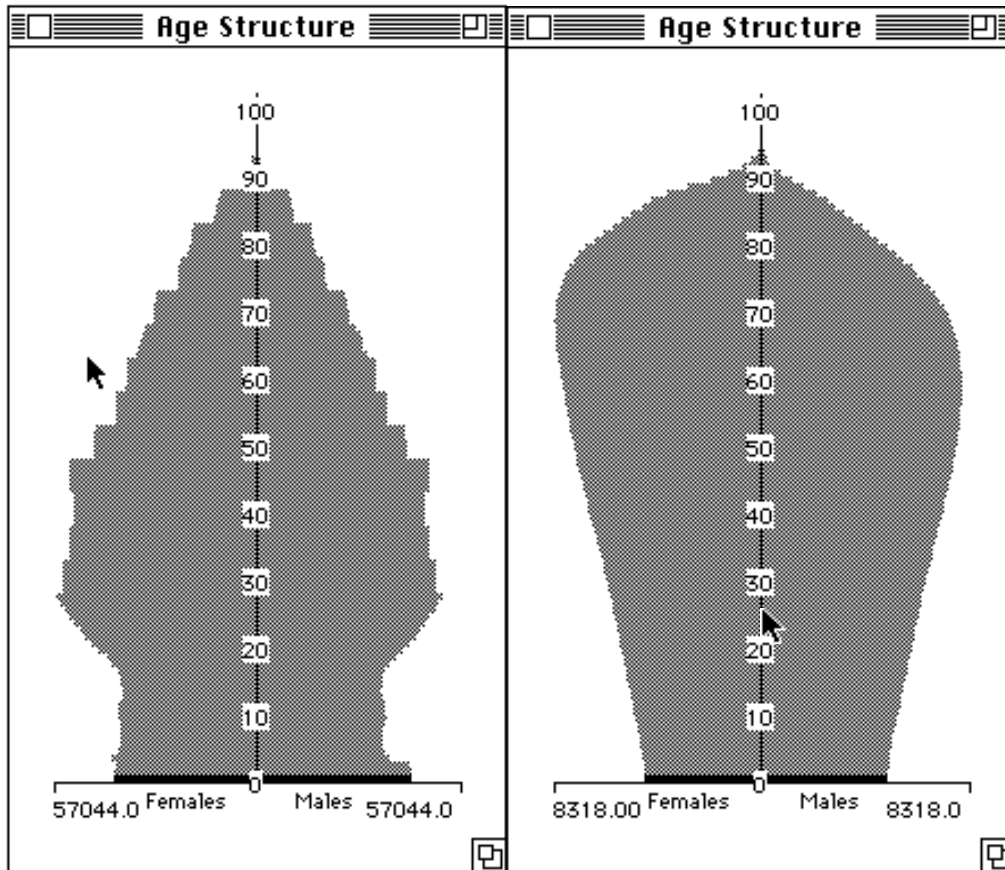


Fig. A5: Client monitoring of the age structure of a Leslie matrix model of the human population of Switzerland. On the left the age structure of the Swiss in 1988, on the right at the end of the simulation in year 2188, i.e. when the population has reached approximately a steady-state age structure.

Note that simulations are not possible if the data file containing the initial state vector could not be successfully opened or processed, since procedure *StateVectorInitialized*, which is installed via *InstallStartConsistency* into the simulation environment, will return FALSE if the reading of the data from the file should have failed for whichever reason (file could not be found, opened, contains data different from the needed respectively expected ones etc.).

Moreover this sample model demonstrates the use of table functions to interpolate for every age class the fecundity and the survival (auxiliary library module *TabFunc*) and a typical client monitoring of the age structure by using auxiliary library module *DrawAgePyramid* (Fig. A5). The latter allows to watch the evolving age structure continuously and to detect when the shape of the age pyramid does no longer change, i.e. the stationary age structure has been reached.

MODULE SwissPop;

(\*\*\*\*\*)

Model: SwissPop

Copyright (c) 1989 by Harald Bugmann, Andreas Fischlin  
 & Swiss Federal Institute of Technology Zurich ETHZ  
 Systems Ecology Group  
 ETH-Zentrum  
 CH-8092 Zurich  
 Switzerland

Purpose:

Population dynamics of Switzerland with an age structure (100 age classes) Leslie model with sex differentiation. The initial state vector for the year 1988 is read from a data file (SwissPop88.DAT), which reads the size for every age class into the state vector. Age and sex specific fecundity and survival are formulated by interpolating within table functions for age specific mortality function (one for both sexes) and an age specific fecundity function for the women.

References:

Statistisches Jahrbuch der Schweiz - 1989. Verlag Neue Zürcher Zeitung und Bundesamt für Statistik.  
 ISBN 3 85823 250 5.

p.24 Table 1.4 (Population)  
 p.36 and 39 (Fecundity, mortality).

Implementation and Revisions:

=====

Author	Date	Description
-----	----	-----
hb	18.06.90	First implementation (DM 2.0, MacMETH 2.6+, ModelWorks 2.0)
hb	22.06.90	Minor improvements & bug fixes
af	25/01/93	Fixing bugs in state vector initialization and complete overhaul to include this module as a sample model in the ModelWorks Manual Appendix; uses now ReadData
dg	06/03/93	Import lists cleaned up

(\*\*\*\*\*)

```
FROM DMConversions IMPORT IntToString;
FROM DMStrings IMPORT Append;
FROM DMMessages IMPORT Warn;
FROM SimBase IMPORT
    Model, DeclM, IntegrationMethod,
    StateVar, NewState, DeclSV, SetSV, Parameter, DeclP, RTCType,
    StashFiling, Tabulation, Graphing, DeclMV, AuxVar,
    SetSimTime, NoInitialize, NoInput,
    NoOutput, NoTerminate, NoAbout, DoNothing,
    InstallClientMonitoring, SetMonInterval;
FROM SimMaster IMPORT
    InstallStartConsistency, InstallDefSimEnv,
    RunSimEnvironment;
```

## ModelWorks 2.2 - Appendix (Sample Models)

```

FROM TabFunc IMPORT  TabFUNC, DeclTabF, Yie, Yi, SetTabF;

FROM ReadData IMPORT
  OpenDataFile, OpenDataFile, CloseDataFile, GetReal, GetInt, TestEOF,
  SkipHeaderLine, readingAborted;

FROM DrawAgePyram IMPORT
  SetPyramidParameters, GetPyramidParameters, MakePyramid,
  ShowPyramidWindow, DiscardPyramid, DrawPyramid, AgePyramid,
  ResetPyramid, HidePyramidWindow;

(*****)

CONST
  maxAge = 100;
  defltBirthSexRatio = 0.48;
  sexRatio = 0.52;

TYPE
  StateVector = ARRAY [0..maxAge] OF StateVar;
  NewStateVector = ARRAY [0..maxAge] OF NewState;

VAR
  m : Model;
  men, women : ARRAY [0..maxAge] OF StateVar;
  menNew, womenNew : NewStateVector;
  ageClass, mortWomen, mortMen : ARRAY [0..11] OF Parameter;
  ageClassFec, fecundity : ARRAY [0..8] OF Parameter;
  offspring, birthSexRatio,
  totalWomen, totalMen, totalPopulation : AuxVar;
  agePyramid: AgePyramid;
  mortMenT, mortWomenT, fecT : TabFUNC;
  stateVectorInitialized: BOOLEAN;

PROCEDURE ReadInitialStateVectorFromFile;
  CONST maxTolerated = 1.0E+8/FLOAT(maxAge);
  VAR fn: ARRAY [0..127] OF CHAR;  VAR opened: BOOLEAN;
      k,ageClass: INTEGER; ageClassSize: AuxVar;
BEGIN
  fn := "SwissPop88.DAT";
  IF stateVectorInitialized THEN
    OpenDataFile(fn,opened);
  ELSE
    OpenDataFile(fn,opened);
  END(*IF*);
  IF opened THEN
    SkipHeaderLine;
    FOR k:= 0 TO maxAge DO
      TestEOF; GetInt("age class",k, ageClass, k,k); (* ageClass = k expected *)
      TestEOF; GetReal("size of age class",k, ageClassSize, 0.0,maxTolerated);
      IF NOT readingAborted THEN
        women[ageClass] := sexRatio * ageClassSize;
        men[ageClass] := ageClassSize - women[ageClass];
        SetSV(m,women[ageClass],women[ageClass]);
        SetSV(m,men[ageClass],men[ageClass]);
        stateVectorInitialized := TRUE;
      ELSE
        stateVectorInitialized := FALSE; (* only partially assigned *)
        RETURN
      END(*IF*);
    END(*FOR*);
    CloseDataFile;
  END(*IF*);
END ReadInitialStateVectorFromFile;

PROCEDURE StateVectorInitialized(): BOOLEAN;
BEGIN

```

```

RETURN stateVectorInitialized
END StateVectorInitialized;

PROCEDURE Dynamic;
  VAR i : INTEGER;
BEGIN
  (* calculate surviving people in each age class *)
  FOR i := 1 TO maxAge DO
    womenNew[i] := ( 1.0 - Yie(mortWomenT, FLOAT(i-1)) ) * women[i-1];
    menNew[i]   := ( 1.0 - Yie(mortMenT,   FLOAT(i-1)) ) * men[i-1];
  END; (* FOR *)

  (* sum all offspring for the age classes 15 to 49 *)
  offspring := 0.0;
  FOR i:= 15 TO 49 DO
    offspring := offspring + Yi(fecT, FLOAT(i))*women[i];
  END; (* FOR *)

  (* calculate babies born *)
  womenNew[0] := birthSexRatio * offspring;
  menNew[0]   := (1.0 - birthSexRatio) * offspring;
END Dynamic;

PROCEDURE Output;
  VAR i: INTEGER;
BEGIN
  totalWomen := 0.0;
  totalMen   := 0.0;
  FOR i:= 0 TO maxAge DO
    totalWomen := totalWomen + women[i];
    totalMen   := totalMen   + men[i];
  END; (* FOR *)
  totalPopulation := totalMen + totalWomen;
END Output;

PROCEDURE PyramidMonitoring;
BEGIN
  ShowPyramidWindow(agePyramid);
  DrawPyramid (agePyramid, women, men);
END PyramidMonitoring;

PROCEDURE ModelObjects;
  VAR i : INTEGER;
      string : ARRAY [1..3] OF CHAR;
      womenDescr, womenIdent, menDescr, menIdent : ARRAY [0..30] OF CHAR;
BEGIN
  DeclMV( totalWomen, 0.0, 7.0E6, "Total women", "Σ women",
          "#", notOnFile, writeInTable, isY);
  DeclMV( totalMen, 0.0, 7.0E6, "Total men", "Σ men",
          "#", notOnFile, writeInTable, isY);
  DeclMV( totalPopulation, 0.0, 7.0E6, "Total population", "Σ pop",
          "#", notOnFile, writeInTable, isY);

  FOR i:= 0 TO maxAge DO
    IntToString(i, string, 3);
    womenDescr := "Women of age "; Append(womenDescr, string);
    womenIdent := "f "; Append(womenIdent, string);
    DeclSV( women[i], womenNew[i], 0.0, 0.0, 1.0E6,
            womenDescr, womenIdent, "#");
    DeclMV( women[i], 0.0, 150000.0, womenDescr, womenIdent,
            "#", notOnFile, notInTable, notInGraph);
  END(*FOR*);

  FOR i:= 0 TO maxAge DO
    IntToString(i, string, 3);

```

## ModelWorks 2.2 - Appendix (Sample Models)

```

menDescr := "men of age "; Append(menDescr, string);
menIdent := "m "; Append(menIdent, string);
DeclSV( men[i], menNew[i], 0.0, 0.0, 1.0E6,
        menDescr, menIdent, "#");
DeclMV( men[i], 0.0, 150000.0, menDescr, menIdent,
        "#", notOnFile, notInTable, notInGraph);
END(*FOR*);

```

```
MakePyramid( agePyramid );
```

```
DeclP( birthSexRatio, defltBirthSexRatio, 0.0, 1.0, rtc,
       "Percentage of female babies", "% fem. babies", "%");
```

```

ageClass[0] := 0.0; mortMen[0] := 0.0077;
ageClass[1] := 5.5; mortMen[1] := 0.0003;
ageClass[2] := 15.0; mortMen[2] := 0.0006;
ageClass[3] := 25.0; mortMen[3] := 0.0015;
ageClass[4] := 35.0; mortMen[4] := 0.0015;
ageClass[5] := 45.0; mortMen[5] := 0.0027;
ageClass[6] := 55.0; mortMen[6] := 0.0076;
ageClass[7] := 62.5; mortMen[7] := 0.0156;
ageClass[8] := 67.5; mortMen[8] := 0.0254;
ageClass[9] := 75.0; mortMen[9] := 0.0520;
ageClass[10]:= 85.0; mortMen[10]:= 0.1139;
ageClass[11]:= 95.0; mortMen[11]:= 0.7482;
DeclTabF(mortMenT, ageClass, mortMen, 12, TRUE,
         "Mortality of men",
         "age", "mortality", "years", "/year",
         0.0, 100.0, 0.0, 1.0);

```

```

ageClass[0] := 0.0; mortWomen[0] := 0.0059;
ageClass[1] := 5.5; mortWomen[1] := 0.0003;
ageClass[2] := 15.0; mortWomen[2] := 0.0003;
ageClass[3] := 25.0; mortWomen[3] := 0.0005;
ageClass[4] := 35.0; mortWomen[4] := 0.0007;
ageClass[5] := 45.0; mortWomen[5] := 0.0016;
ageClass[6] := 55.0; mortWomen[6] := 0.0036;
ageClass[7] := 62.5; mortWomen[7] := 0.0067;
ageClass[8] := 67.5; mortWomen[8] := 0.0114;
ageClass[9] := 75.0; mortWomen[9] := 0.0270;
ageClass[10]:= 85.0; mortWomen[10]:= 0.0923;
ageClass[11]:= 95.0; mortWomen[11]:= 0.7703;
DeclTabF(mortWomenT, ageClass, mortWomen, 12, TRUE,
         "Mortality of women",
         "age", "mortality", "years", "/year",
         0.0, 100.0, 0.0, 1.0);

```

```

ageClassFec[0] := 15.0; fecundity[0] := 0.0;
ageClassFec[1] := 17.0; fecundity[1] := 0.0045;
ageClassFec[2] := 22.0; fecundity[2] := 0.0512;
ageClassFec[3] := 27.0; fecundity[3] := 0.1283;
ageClassFec[4] := 32.0; fecundity[4] := 0.0946;
ageClassFec[5] := 37.0; fecundity[5] := 0.0305;
ageClassFec[6] := 42.0; fecundity[6] := 0.0043;
ageClassFec[7] := 47.0; fecundity[7] := 0.0003;
ageClassFec[8] := 50.0; fecundity[8] := 0.0000;
DeclTabF(fecT, ageClassFec, fecundity, 9, TRUE,
         "Fecundity",
         "age", "Fecundity", "years", "ch/fem/yr",
         0.0, 100.0, 0.0, 1.0);

```

```
END ModelObjects;
```

```
PROCEDURE ModelDefinition;
```

```
BEGIN
```

```

DeclM(m, discreteTime, NoInitialize, NoInput, Output, Dynamic,
      NoTerminate, ModelObjects, "Population model for Switzerland (CH)", "CH Pop",
      NoAbout);

```



```
SetSimTime(1988.0, 2188.0);
SetMonInterval(5.0);
InstallClientMonitoring(DoNothing, PyramidMonitoring, DoNothing);
InstallStartConsistency(StateVectorInitialized);
stateVectorInitialized := FALSE;
InstallDefSimEnv(ReadInitialStateVectorFromFile);
END ModelDefinition;

BEGIN
  RunSimEnvironment(ModelDefinition);
END SwissPop.
```

#### A.4.4 Sensitivity Analysis - *Sensitivity*

To perform a sensitivity analysis we use a structured simulation experiment: Given a set of  $n$  model parameters and for each parameter a triple of values, such as the lower boundary of a confidence interval, the mean, and the upper boundary of the confidence interval (e.g.  $\alpha = 5\%$ ), we are interested in the sensitivity of the model behavior in respect to the changes in the parameters as given by these triplets. Thus it is necessary to execute a simulation run for every combination of parameter values.

First the parameter values can be stored on a text file similar to a format like this one:

```

min p1      mean p1      max p1      descriptor of p1    p1      unit p1
min p2      mean p2      max p2      descriptor of p2    p2      unit p2
...
...
...
min pn      mean pn      max pn      descriptor of pn    pn      unit pn

```

For instance this parameter file might contain these data:

```

0.109      0.234      0.472      Growth ratel      r      day^-1
35.6       42.3       49.8       Half-saturation c. Ks      µg/l
1.0E5      2.5E5      5.0E5      Initial algal dens. x0      cells/ml

```

A full sensitivity analysis can then be realized in form of the following program code:

```

CONST n = 3;
TYPE PVal = (cur, min, mean, max);
PType = RECORD
  v: ARRAY [cur..max] OF REAL;
  descr,ident,unit: ARRAY [0..64] OF CHAR;
END;
VAR p: ARRAY [1..n] OF PType;

PROCEDURE DeclModelObjects;
  VAR i: [1..n]; j: [min..max]; parFile: TextFile2;
BEGIN
  GetExistingFile(parFile, "Open parameter file");
  FOR i:= 1 TO n DO
    FOR j:= min TO max DO GetReal(parFile,p[i].v[j]) END;
    SkipGap(parFile); ReadChars(parFile,p[i].descr);
    SkipGap(parFile); ReadChars(parFile,p[i].ident);
    SkipGap(parFile); ReadChars(parFile,p[i].unit);
  END;
  Close(parFile);
  FOR i:= 1 TO n DO WITH p[i] DO
    DeclP(v[cur],v[mean],0.0,MAX(REAL),noRtc,descr,ident,unit)
  END END;

  DeclSV(...
  ...
END DeclModelObjects;

PROCEDURE MyExperiment;
  VAR i,j,k: [min..max];
BEGIN
  FOR i:= min TO max DO
    FOR j:= min TO max DO
      FOR k:= min TO max DO
        SetP(m,p[1].v[cur], p[1].v[i]);
        SetP(m,p[2].v[cur], p[2].v[j]);
        SetP(m,p[3].v[cur], p[3].v[k]);
      SimRun
    END
  END
END;
END MyExperiment;

```

<sup>1</sup>In order to be able to use blanks in the middle of a descriptor and still be able to write the data onto the text file in a free format use so-called hard spaces (on the Macintosh Option<sup>^</sup>space-bar) or underline within a descriptor.

<sup>2</sup>The objects *TextFile*, *GetExistingFile*, *GetReal*, *SkipGap*, *ReadChars*, and *Close* are to be imported from the "Dialog Machine" module *DMFiles*.

or alternatively the procedure *MyExperiment* may be programmed in the general recursive variant, which works for any n:

```
PROCEDURE MyExperiment;
  PROCEDURE Sensitivity(i: CARDINAL);
    VAR j: [min..max];
  BEGIN (*Sensitivity*)
    FOR j:= min TO max DO SetP(m,p[i].v[cur], p[i].v[j]);
      IF i<n THEN Sensitivity(i+1) ELSE SimRun END;
    END(*FOR*);
  END Sensitivity;
  BEGIN (*MyExperiment*)
    Sensitivity(1);
  END MyExperiment;
```

Note however, such an experiment may quickly grow to an enormous task! Given n model parameters, each with k values and each combination to be tested, the number of simulation runs becomes  $k^n$ . Above simple example with n = 3 parameters, each with k = 3 values (min, mean, max), requires for a full sensitivity analysis already  $3^3 = 27$  simulation runs.

```
MODULE Sensitivity;
```

```
(*****
```

```
ModelWorks model: Sensitivity
```

```
Copyright ©1989 by Andreas Fischlin and Swiss
Federal Institute of Technology Zurich ETHZ
Department of Environmental Sciences
Systems Ecology Group
ETH-Zentrum
CH-8092 Zurich
Switzerland
```

```
Version written for:
  'Dialog Machine' V1.0 (User interface)
  MacMETH V2.6 (1-Pass Modula-2 implementation)
  ModelWorks V1.2 (Modelling & Simulation)
```

```
Purpose Demonstrates a model parameter sensitivity using
ModelWorks
```

```
Remarks Illustrates a manual example (Theory programming
structured simulations)
```

```
Implementation and Revisions:
=====
```

Author	Date	Description
-----	----	-----
af	05/06/89	First implementation (DM 1.0, MacMETH 2.6, ModelWorks 1.2)
dg	25/04/96	Cleaned up for PC compatibility

```
*****)
```

```
FROM DMFiles IMPORT
  GetExistingFile, TextFile, GetReal, SkipGap, ReadChars, Close,
  Response;
```

```
FROM DMWindows IMPORT
  RectArea, Window, WindowKind, ScrollBars, CloseAttr, ZoomAttr,
  WFFixPoint, WindowFrame, CreateWindow, AutoRestoreProc,
  WindowExists, PutOnTop;
```

```
FROM DMWindIO IMPORT
  SelectForOutput, EraseContent, SetPos, Write, WriteLn,
  WriteString, WriteReal, WriteInt, SetWindowFont, WindowFont,
```

```

FontStyle;

FROM SimBase IMPORT
  Model, IntegrationMethod, DeclM, StateVar, Derivative, DeclSV,
  AuxVar, Parameter, RTCType, DeclP, SetP, SetDefltp,
  ResetAllParameters, StashFiling, Tabulation, Graphing, DeclMV,
  SetSimTime, SetMonInterval, SetIntegrationStep, NoInput,
  NoOutput, NoTerminate, NoAbout, CloseWindow, MWWindow,
  GetWindowPlace;

FROM SimMaster IMPORT RunSimEnvironment, SimRun, InstallExperiment,
  CurrentSimNr, InstallDefSimEnv;

CONST n = 3;
TYPE PVal = (cur, min, mean, max); SensiRange = [min..max];
  PType = RECORD
    v: ARRAY [cur..max] OF Parameter;
    descr,ident,unit: ARRAY [0..64] OF CHAR;
  END;
VAR p: ARRAY [1..n] OF PType;
  m: Model;
  x: StateVar;   xDot: Derivative;
  s: StateVar;   sDot: Derivative;
  mu: AuxVar;
  Y: Parameter;

  parSetW: Window;

PROCEDURE Initialize;
BEGIN (*Initialize*)
  SelectForOutput(parSetW);
  WriteString('Run Nr. = '); WriteInt(CurrentSimNr(),2);
  x:= p[3].v[cur];
END Initialize;

PROCEDURE Dynamic;
BEGIN
  mu:= p[1].v[cur]*s/(p[2].v[cur]+s);
  xDot:= mu*x;
  sDot:= - mu/Y;
END Dynamic;

PROCEDURE DeclModelObjects;
BEGIN
  DeclSV(x, xDot,1.0E5, 0.0, 1.0E8,
    'Algal density', 'x', 'cells/ml');
  DeclSV(s, sDot,100.0, 0.0, MAX(REAL),
    'Orthophosphate PO4-P', 's', 'µg/l');

  DeclMV(x,0.0,1.0E7,
    'Algal density', 'x', 'cells/ml',
    notOnFile,notInTable,isY);
  DeclMV(s,0.0,110.0,
    'Orthophosphate PO4-P', 's', 'µg/l',
    notOnFile,notInTable,isY);
  DeclMV(mu,0.0,110.0,
    'Relative growth rate of algae', 'µ', 'day^-1',
    notOnFile,notInTable,notInGraph);
  DeclMV(xDot,0.0,110.0,
    'Growth rate of algae', 'dx/dt', 'cells/ml/day',
    notOnFile,notInTable,notInGraph);
  DeclMV(sDot,-10.0,0.0,
    'Consumption rate of PO2-P by algae', 'ds/dt', 'µg/l/day',
    notOnFile,notInTable,notInGraph);

```

```

DeclP(p[1].v[cur],0.0,0.0,MAX(REAL),noRtc,"parameter 1","p1","u1");
DeclP(p[2].v[cur],0.0,0.0,MAX(REAL),noRtc,"parameter 2","p2","u2");
DeclP(p[3].v[cur],0.0,0.0,MAX(REAL),noRtc,"parameter 3","p3","u3");
DeclP(Y,0.04,0.0,MAX(REAL),noRtc,'Yield','Y','cells/ml/µg/l');

END DeclModelObjects;

PROCEDURE ReadAndSetParameters;
  VAR i: [1..n]; j: SensiRange; parFile: TextFile ;
  PROCEDURE ShowOrOpenParameterSpace;
    VAR parSetWf: WindowFrame; isOpen: BOOLEAN;
  BEGIN
    WITH parSetWf DO GetWindowPlace(TableW,x,y,w,h,isOpen) END;
    IF isOpen THEN CloseWindow(TableW) END(*IF*);
    IF WindowExists(parSetW) THEN
      PutOnTop(parSetW);
    ELSE
      CreateWindow(parSetW,GrowOrShrinkOrDrag,WithoutScrollBars,
        WithCloseBox,WithZoomBox,bottomLeft, parSetWf,
        'Parameter Sets', AutoRestoreProc);
      SetWindowFont(Monaco,9,FontStyle);
    END(*IF*);
    SetPos(1,1);
  END ShowOrOpenParameterSpace;
BEGIN
  GetExistingFile(parFile, 'Open parameter file "Sensitivity.DAT"');
  IF parFile.res=done THEN
    FOR i:= 1 TO n DO
      FOR j:= min TO max DO GetReal(parFile,p[i].v[j]) END;
      SkipGap(parFile); ReadChars(parFile,p[i].descr);
      SkipGap(parFile); ReadChars(parFile,p[i].ident);
      SkipGap(parFile); ReadChars(parFile,p[i].unit);
    END;
    Close(parFile);
    FOR i:= 1 TO n DO WITH p[i] DO
      SetDefltP(m,v[cur],v[mean],0.0,MAX(REAL),noRtc,descr,ident,unit);
      ResetAllParameters;
    END END;
    ShowOrOpenParameterSpace;
  END(*IF*);
END ReadAndSetParameters;

PROCEDURE RecordParSet(i: INTEGER; j,k: SensiRange);
  VAR l: INTEGER;
BEGIN (*RecordParSet*)
  SelectForOutput(parSetW);
  SetPos(CurrentSimNr(),1);
  WriteInt(i,2); Write(' '); WriteInt(ORD(j),2); Write(' '); WriteInt(ORD(k),2);
  WriteString(' ');
  FOR l:= 1 TO n DO
    WriteString(p[l].ident); WriteString(' =');
    WriteReal(p[l].v[cur],8,3); WriteString(' ');
  END(*FOR*);
END RecordParSet;

(*.
PROCEDURE MyExperiment;
  VAR i,j,k: SensiRange;
BEGIN
  FOR i:= min TO max DO
    FOR j:= min TO max DO
      FOR k:= min TO max DO
        SetP(m,p[1].v[cur], p[1].v[i]);
        SetP(m,p[2].v[cur], p[2].v[j]);
        SetP(m,p[3].v[cur], p[3].v[k]);
        SimRun
      END
    END
  END
END;

```

```

END MyExperiment;
.*)

PROCEDURE MyExperiment;
  PROCEDURE Sensitivity(i: INTEGER; k: SensiRange);
    VAR j: SensiRange;
  BEGIN
    FOR j:= min TO max DO
      SetP(m,p[i].v[cur], p[i].v[j]);
      IF i<n THEN Sensitivity(i+1,j) ELSE
        RecordParSet(i,j,k); SimRun;
      END;
    END(*FOR*);
  END Sensitivity;
BEGIN
  CloseWindow(TableW);
  SelectForOutput(parSetW); EraseContent; SetPos(1,1);
  Sensitivity(1,min);
END MyExperiment;

PROCEDURE ModelDefinitions;
BEGIN
  DeclM(m, Euler, Initialize, NoInput, NoOutput, Dynamic, NoTerminate, DeclModelObjects,
    'Sensitivity of Michaelis-Menthen algal growth', 'm', NoAbout);
  SetSimTime(0.0,8.0); SetMonInterval(0.5); SetIntegrationStep(0.1);
  InstallExperiment(MyExperiment);
  InstallDefSimEnv(ReadAndSetParameters);
END ModelDefinitions;

BEGIN
  RunSimEnvironment(ModelDefinitions);
END Sensitivity.

```

#### A.4.5 Parameter Identification - *GauseIdentif*

GAUSE (1934) has pursued the question whether classical population dynamics models such as the Verhulst (logistic) or the Lotka-Volterra equations actually represent true population processes. He reared several populations of microorganisms such as the ciliate *Paramecium caudatum* in the laboratory. In one experiment he tried to fit the observed population densities

Day t	x(t)	Day t	x(t)	Day t	x(t)	Day t	x(t)
0	2	4	39	8	50	12	57
1	5	5	52	9	76	13	70
2	22	6	54	10	69	14	55
3	16	7	47	11	51	15	59

with the logistic equation

$$dx(t)/dt = r \frac{K - x(t)}{K} x(t)$$

where

- x(t) the density of ciliates in number of animals per 0.5 ml
- K carrying capacity  $\approx$  maximum density of ciliates
- r per capita growth rate

The following program module *GauseIdentif* demonstrates the use of a ModelWorks structured simulation (routine *Identify* installed as an experiment via *DeclExperiment* from *SimMaster*) to identify the unknown model parameters K and r by means of the optimization module *IdentifyPars*. The minimization routines of *IdentifyPars* require that *GauseIdentif* computes a performance index which expresses the goodness of fit between the simulated and observed population densities. If the model equations would be analytically unsolvable, which is often the case, the performance index can only be computed by a full simulation run, given some estimates for the unknown parameters exist. The identification algorithm will then try to adjust the parameters such, that the performance index improves, i.e. is minimized. Although the logistic equation can be solved analytically, the following program code follows an architecture which is applicable to any ModelWorks model.

```

MODULE GauseIdentif; (* af 23/01/93, dg 06/03/93, dg 25/04/96 *)

(*****

MODEL: GauseIdentif - Identifies model parameters
      for logistic growth equation
      applied to the experiment by
      Gause (1934) rearing the ciliate
      Paramecium caudatum.

Reference: Gause, G.F., 1934. The struggle for existence.
          Baltimore: Williams and Wilkins. 163pp.

A. Fischlin, 23/Jan/93, Systems Ecology ETHZ

*****

FROM DMSystem IMPORT SuperScreen, MainScreen, TitleBarHeight;
FROM DMStrings IMPORT Concatenate, Append, AppendCh;
FROM DMConversions IMPORT RealToString, RealFormat, IntToString;
FROM DMMessages IMPORT Warn;

```

```

FROM DMWindIO IMPORT
  DisplayPredefinedPicture, BackgroundWidth, BackgroundHeight,
  ScaleUC, UCFrame, EraseContent, SetUCPen, UCLineTo, DrawSym,
  SelectForOutput, Area, pat, GreyContent, SetPen, WriteString,
  WriteReal, CellHeight, CellWidth, StringWidth, SetWindowFont,
  WindowFont, FontStyle;
FROM DMWindows IMPORT
  RectArea, Window, WindowKind, ScrollBars, CloseAttr, ZoomAttr,
  WFFixPoint, WindowFrame, CreateWindow, AutoRestoreProc,
  GetWindowFrame, PutOnTop, WindowExists, RemoveWindow,
  AddWindowHandler, WindowHandlers;

FROM SimBase IMPORT
  Model, StateVar, NewState, Parameter, AuxVar,
  Derivative, IntegrationMethod, DeclM, DeclSV, DeclP, RTCType,
  StashFiling, GetP, SetP, GetMV, SetMV, Message,
  SetDefltCurveAttrForMV, Stain, LineStyle, MWindow,
  GetDefltWindowPlace, GetWindowPlace, SetDefltWindowPlace,
  SetWindowPlace, MWindowArrangement, SetDefltWindowArrangement,
  Tabulation, Graphing, DeclMV, SetSimTime, NoInitialize, NoInput,
  NoOutput, NoTerminate, NoAbout, GetDefltP, SetDefltP;

FROM SimMaster IMPORT
  RunSimEnvironment, InstallStartConsistency, InstallExperiment,
  SimRun, CurrentSimNr, CurrentTime;

FROM SimGraphUtils IMPORT
  DeclDispData, timeIsIndep, DisplayTime;

FROM SimDeltaCalc IMPORT
  DeltaVar, InitDeltaStat, AccuDelta, GetDeltaStat,
  WriteDeltaStatMsg;

FROM IdentifyPars IMPORT
  MinimizeAfterDialog, UnmarkAllParsForIdentification,
  MarkParForIdentification;

CONST
  fstDay = 0; lastDay = 16;
  KMin = 0.0; KMax = 100.0; (* Plausible range for K *)
  rMin = 0.0; rMax = 3.0; (* Plausible range for r *)

VAR
  m: Model;
  x: StateVar;
  xDot: Derivative;
  K,r: Parameter;
  parSpace: RECORD
    w: Window;
    wf: WindowFrame;
    oldK, oldr: Parameter;
  END;
  gauseExp: RECORD
    day,ciliateCount,dummy: ARRAY [fstDay..lastDay] OF REAL;
    xMeasured: REAL; (* monitoring variable for ciliate counts *)
    delta: DeltaVar; (* Δ = x - xMeasured *)
  END;

PROCEDURE GausesMeasurements; (* Gause, 1934, p. 145, Table 4 *)
BEGIN
  gauseExp.day[ 0]:= 0.; gauseExp.ciliateCount[ 0]:= 2.;
  gauseExp.day[ 1]:= 1.; gauseExp.ciliateCount[ 1]:= 5.;
  gauseExp.day[ 2]:= 2.; gauseExp.ciliateCount[ 2]:=22.;
  gauseExp.day[ 3]:= 3.; gauseExp.ciliateCount[ 3]:=16.;
  gauseExp.day[ 4]:= 4.; gauseExp.ciliateCount[ 4]:=39.;

```



```

gauseExp.day[ 5]:= 5.; gauseExp.ciliateCount[ 5]:=52.;
gauseExp.day[ 6]:= 6.; gauseExp.ciliateCount[ 6]:=54.;
gauseExp.day[ 7]:= 7.; gauseExp.ciliateCount[ 7]:=47.;
gauseExp.day[ 8]:= 8.; gauseExp.ciliateCount[ 8]:=50.;
gauseExp.day[ 9]:= 9.; gauseExp.ciliateCount[ 9]:=76.;
gauseExp.day[10]:= 10.; gauseExp.ciliateCount[10]:=69.;
gauseExp.day[11]:= 11.; gauseExp.ciliateCount[11]:=51.;
gauseExp.day[12]:= 12.; gauseExp.ciliateCount[12]:=57.;
gauseExp.day[13]:= 13.; gauseExp.ciliateCount[13]:=70.;
gauseExp.day[14]:= 14.; gauseExp.ciliateCount[14]:=53.;
gauseExp.day[15]:= 15.; gauseExp.ciliateCount[15]:=59.;
gauseExp.day[16]:= 16.; gauseExp.ciliateCount[16]:=57.;
END GausesMeasurements;

PROCEDURE Initialize;
BEGIN
  InitDeltaStat(gauseExp.xMeasured,CurrentTime(),x,gauseExp.delta);
END Initialize;

PROCEDURE Output;
BEGIN
  AccuDelta(gauseExp.delta,CurrentTime(),x);
END Output;

PROCEDURE Dynamic;
BEGIN
  xDot:= r*((K-x)/K)*x;
END Dynamic;

PROCEDURE ParametersPlausible(): BOOLEAN;
  VAR plausible: BOOLEAN; str1,str2,rStr: ARRAY [0..31] OF CHAR;
BEGIN
  plausible := (K>KMin) AND (K<=KMax) AND (r>rMin) AND (r<=rMax);
  IF NOT plausible THEN
    RealToString(KMin,str1,0,1,FixedFormat); Append(str1," < K <= ");
    RealToString(KMax,rStr,0,1,FixedFormat); Append(str1,rStr);
    RealToString(rMin,str2,0,1,FixedFormat); Append(str2," < r <= ");
    RealToString(rMax,rStr,0,1,FixedFormat); Append(str2,rStr);
    Warn("The parameters have become implausible! Valid ranges are:",str1,str2);
  END(*IF*);
  RETURN plausible
END ParametersPlausible;

PROCEDURE ShowNewParameters;
  VAR bottom: RectArea; (*. debugging .*)
BEGIN
  SelectForOutput(parSpace.w);
  SetUCPen(parSpace.oldK,parSpace.oldr); UCLineTo(K,r); DrawSym("•");
  parSpace.oldK := K; parSpace.oldr := r;
END ShowNewParameters;

PROCEDURE ShowPerformance( perfIndex: REAL );
  VAR bottom: RectArea;
BEGIN
  SelectForOutput(parSpace.w);
  bottom.x := 0; bottom.y := 0;
  bottom.h := 20+CellHeight(); bottom.w := 100*CellWidth();
  Area(bottom,pat[light]);
  SetPen(20,20); WriteString("Performance index ( $\sum \Delta^2$ ) = ");
  WriteReal(perfIndex,0,3);
  WriteString(" with K,r = ");
  WriteReal(K,0,3); WriteString(", "); WriteReal(r,0,3);
END ShowPerformance;

```

```

PROCEDURE RedrawParameterSpace(u: Window);
  CONST fw = 6; dec = 3;
  VAR curMinr, curMaxr, curMinK, curMaxK: REAL;
      curSF: StashFiling; curT: Tabulation; curG: Graphing;
      rstr: ARRAY [0..31] OF CHAR; label: ARRAY [0..127] OF CHAR;
BEGIN
  SelectForOutput(parSpace.w);
  GetWindowFrame(u, parSpace.wf);
  GetMV(m, K, curMinK, curMaxK, curSF, curT, curG);
  GetMV(m, r, curMinr, curMaxr, curSF, curT, curG);
  parSpace.wf.x := 20; parSpace.wf.y := 50;
  parSpace.wf.w := parSpace.wf.w - 2*parSpace.wf.x;
  parSpace.wf.h := parSpace.wf.h - parSpace.wf.x - parSpace.wf.y;
  ScaleUC(parSpace.wf, curMinK, curMaxK, curMinr, curMaxr);
  EraseContent; UCFrame;
  label := "K - carrying capacity (Min = ";
  RealToString(curMinK, rstr, fw, dec, FixedFormat);
  Append(label, rstr); Append(label, " / Max = ");
  RealToString(curMaxK, rstr, fw, dec, FixedFormat);
  Append(label, rstr); AppendCh(label, ")");
  SetPen(parSpace.wf.x+parSpace.wf.w-StringWidth(label), parSpace.wf.y-CellHeight());
  WriteString(label);
  SetPen(parSpace.wf.x, parSpace.wf.y+parSpace.wf.h+CellHeight() DIV 2);
  WriteString("r - per capita growth rate");
  WriteString(" (Min = "); WriteReal(curMinr, fw, dec);
  WriteString(" / Max = "); WriteReal(curMaxr, fw, dec);
  WriteString(")");
  SetUCPen(K, r); parSpace.oldK := K; parSpace.older := r; DrawSym("•");
END RedrawParameterSpace;

```

```

PROCEDURE ShowOrOpenParameterSpace;
BEGIN
  IF WindowExists(parSpace.w) THEN
    PutOnTop(parSpace.w);
  ELSE
    CreateWindow(parSpace.w, GrowOrShrinkOrDrag, WithoutScrollBars,
      WithCloseBox, WithZoomBox, bottomLeft, parSpace.wf,
      'Parameter space', AutoRestoreProc);
    SetWindowFont(Geneva, 9, FontStyle);
    AddWindowHandler(parSpace.w, redefined, RedrawParameterSpace, 0);
  END(*IF*);
  RedrawParameterSpace(parSpace.w);
END ShowOrOpenParameterSpace;

```

```

PROCEDURE PutGraphOnTop;
  VAR x, y, w, h: INTEGER; isOpen: BOOLEAN;
BEGIN
  GetWindowPlace(GraphW, x, y, w, h, isOpen);
  SetWindowPlace(GraphW, x, y, w, h);
END PutGraphOnTop;

```

```

PROCEDURE PerformanceIndex(): REAL;
  VAR sumY, sumY2, sumAbsY: REAL; count: INTEGER;
BEGIN
  IF CurrentSimNr()=1 THEN ShowOrOpenParameterSpace END(*IF*);
  ShowNewParameters;
  SimRun;
  GetDeltaStat(gauseExp.xMeasured, sumY, sumY2, sumAbsY, count);
  WriteDeltaStatMsg(gauseExp.xMeasured);
  ShowPerformance(sumY2);
  RETURN sumY2;
END PerformanceIndex;

```

```

PROCEDURE Identify;

```

```

VAR oldPerfInd,newPerfInd: REAL;  str: ARRAY [0..127] OF CHAR;
    neededRuns: INTEGER;
    oldK,oldr, newK,newr: Parameter;
    oldMinK,oldMaxK,oldMinr,oldMaxr,
    oldDfltK, oldDfltr, newDfltK, newDfltr,
    newMinK,newMaxK,newMinr,newMaxr: Parameter;
    descrK,identK,unitK,descrr,identr,unitr: ARRAY [0..127] OF CHAR;
    runTimeChangeK, runTimeChanger: RTCType;

PROCEDURE SaveOldParVals;
BEGIN (* SaveOldParVals *)
    GetP(m,K,oldK);  GetP(m,r,oldr);
    GetDefltp(m,K,oldDfltK,oldMinK,oldMaxK,runTimeChangeK,
        descrK, identK, unitK);
    GetDefltp(m,r,oldDfltr,oldMinr,oldMaxr,runTimeChanger,
        descrr, identr, unitr);
END SaveOldParVals;

PROCEDURE SaveNewParVals;
BEGIN (* SaveNewParVals *)
    GetP(m,K,newK);  GetP(m,r,newr);
    GetDefltp(m,K,newDfltK,newMinK,newMaxK,runTimeChangeK,
        descrK, identK, unitK);
    GetDefltp(m,r,newDfltr,newMinr,newMaxr,runTimeChanger,
        descrr, identr, unitr);
END SaveNewParVals;

PROCEDURE RestoreOldParVals;
BEGIN (* RestoreOldParVals *)
    SetDefltp(m,K,oldDfltK,oldMinK,oldMaxK,runTimeChangeK,
        descrK, identK, unitK);
    SetDefltp(m,r,oldDfltr,oldMinr,oldMaxr,runTimeChanger,
        descrr, identr, unitr);
    SetP(m,K,oldK);  SetP(m,r,oldr);
END RestoreOldParVals;

PROCEDURE RestoreNewParVals;
BEGIN (* RestoreNewParVals *)
    SetDefltp(m,K,newDfltK,newMinK,newMaxK,runTimeChangeK,
        descrK, identK, unitK);
    SetDefltp(m,r,newDfltr,newMinr,newMaxr,runTimeChanger,
        descrr, identr, unitr);
    SetP(m,K,newK);  SetP(m,r,newr);
END RestoreNewParVals;

BEGIN (*Identify*)
    SaveOldParVals;
    UnmarkAllParsForIdentification;
    MarkParForIdentification( K );  MarkParForIdentification( r );
    ShowOrOpenParameterSpace;
    oldPerfInd := PerformanceIndex();

    MinimizeAfterDialog(PerformanceIndex);
    PutGraphOnTop;
    neededRuns := CurrentSimNr()-1 (*determination of oldPerfInd does not count*);

    (* Since graph window has been closed, redraw initial simulation run *)
    SaveNewParVals; RestoreOldParVals;
    SimRun;

    (* Calculate new performance index and display results *)
    RestoreNewParVals;
    newPerfInd := PerformanceIndex();
    RealToString(oldPerfInd,str,0,5,ScientificNotation);
    Concatenate("Before  $\sum \Delta^2 =$ ",str,str);
    Message(str);
    RealToString(newPerfInd,str,0,5,ScientificNotation);
    Concatenate("After  $\sum \Delta^2 =$ ",str,str);
    Message(str);

```

## ModelWorks 2.2 - Appendix (Sample Models)

```

IntToString(neededRuns,str,0);
Concatenate("Optimization required ",str,str);
Append(str," runs");
Message(str);
END Identify;

PROCEDURE ModelObjects;
BEGIN
  DeclSV(x, xDot,2.0, 0.0, 100.0,
    "Population density Paramecium caudatum", "x", "#/0.5ml");

  DeclMV(x, 0.0,KMax, "Ciliate density (Paramecium caudatum)", "x", "#/0.5ml",
    notOnFile, writeInTable, isY);
  SetDefltCurveAttrForMV(m,x,ruby,unbroken,0C);
  DeclMV(xDot, 0.0,5.0, "Density derivative", "dx/dt", "#/0.5ml/day",
    notOnFile, notInTable, notInGraph);
  DeclMV(gauseExp.xMeasured, 0.0,100.0, "Measured ciliate density", "xMeasured",
"#/0.5ml/day",
    notOnFile, notInTable, isY);
  SetDefltCurveAttrForMV(m,gauseExp.xMeasured,ruby,invisible,"");
  DeclMV(K, KMin, KMax, "Carrying capacity", "K", "#/0.5ml",
    notOnFile, notInTable, notInGraph);
  DeclMV(r, rMin, rMax, "Per capita growth rate", "r", "/day",
    notOnFile, notInTable, notInGraph);

  DeclP(K, 10.0, KMin, KMax, rtc,
    "K (carrying capacity of x)", "K", "#/0.5ml");
  DeclP(r, 1.0, rMin, rMax, rtc,
    "r (growth rate of x)", "r", "/day");
END ModelObjects;

PROCEDURE About;
  VAR pictRect: RectArea;
BEGIN
  WITH pictRect DO x:=2; y:=-290; w:=498; h:=290 END(*WITH*);
  DisplayPredefinedPicture("GauseIdentif.R",23011,pictRect);
END About;

PROCEDURE ModelDefinitions;
  VAR supScr,x,y,w,h,nc: INTEGER; enabled: BOOLEAN;
BEGIN
  DeclM(m, Euler, Initialize, NoInput, Output, Dynamic,
    NoTerminate, ModelObjects, "Gause experiment & logistic growth model",
    "Gause", About);
  SetSimTime(0.0,16.0);
  InstallStartConsistency(ParametersPlausible);
  SetDefltWindowArrangement(tiled);

  GetWindowPlace(GraphW,parSpace.wf.x,parSpace.wf.y,parSpace.wf.w,parSpace.wf.h,enabled);
  SuperScreen(supScr,x,y,w,h,nc,TRUE(*color priority*));
  IF supScr<>MainScreen() THEN
    SetDefltWindowPlace(GraphW,x+2,y+3,w-6,h-TitleBarHeight()-5);
  END(*IF*);
  w:=506; h:=297;
  x := (BackgroundWidth()-w) DIV 2; y := (BackgroundHeight()-h) DIV 2;
  SetDefltWindowPlace(AboutMW,x,y,506,297);
  DeclDispData(m,gauseExp.xMeasured,m,timeIsIndep,gauseExp.day,gauseExp.ciliateCount,
    gauseExp.dummy,gauseExp.dummy,17, FALSE(*withErrBars*),showAtInit);
  InstallExperiment(Identify);
END ModelDefinitions;

BEGIN
  GausesMeasurements;
  RunSimEnvironment(ModelDefinitions);
END GauseIdentif.

```

The module *IdentifyPars* allows to determine interactively which parameters are to be identified, to select a minimization algorithm, and to launch the minimization (procedure *MinimizeAfterDialog*). To this purpose *IdentifyPars* uses the ModelWorks modules *SimBase* and *SimObjects*. Then procedure *DeclDispDatafrom SimGraphUtils* is used to enter and display the measurements obtained by Gause. The actual measurements are assigned initially by procedure *GausesMeasurements*. See procedure *PerformanceIndex* which computes the performance index by calling *SimRun* and then returns the accumulated sum of squares. Note that procedure *Initialize*, which is called at the begin of every simulation run, sets first the sum of squares to 0 by a call to routine *InitDeltaStat* from *SimDeltaCalc*. The routines *AccuDelta*, called in procedure *Output* during every integration step, and *GetDeltaStat* from *SimDeltaCalc*, called after *SimRun*, are used to actually compute the sum of squares of the differences between the discrete-time observations and the simulated continuous-time population densities, only whenever measurements are available.

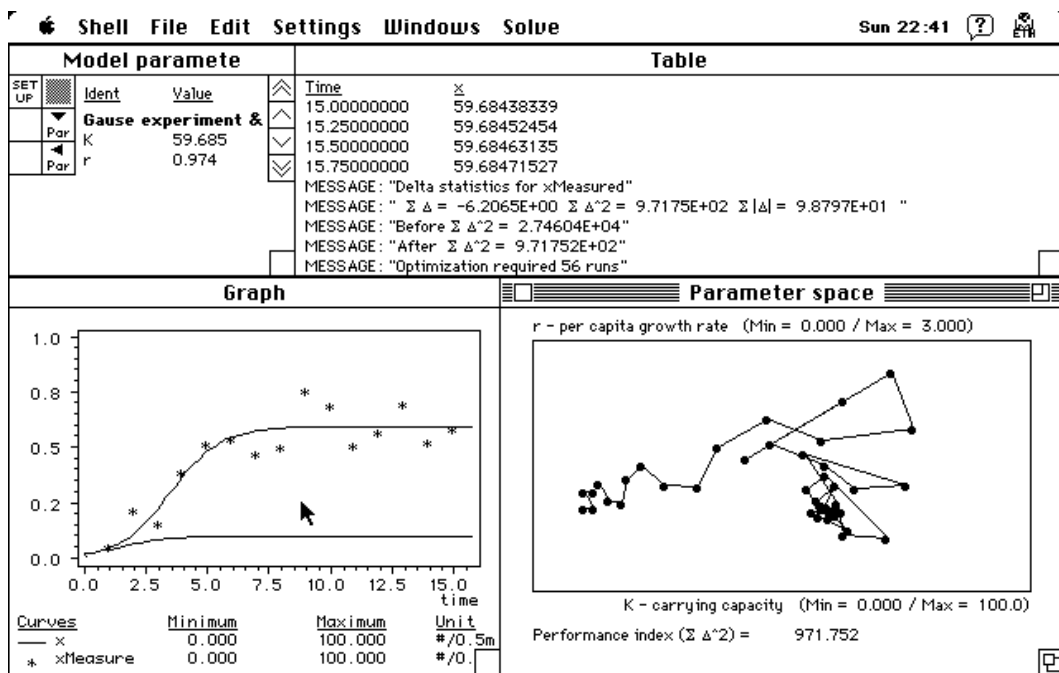


Fig. A6: Result of a parameter identification: The experiment by GAUSE (1934) rearing the ciliate *Paramecium caudatum* (\* observed densities) was fitted with the logistic equation (— simulated densities). Before the optimization  $r = 1.0$ ,  $K = 10.0$  (lower curve in the window *Graph*), after the optimization  $r = 0.974$ ,  $K = 59.7$  (upper curve in the window *Graph*). The optimized performance index was the sum of squares of the differences between simulated and observed population densities at the points where measurements were available. The optimization algorithm was *Amoeba*, which required 56 simulation runs to achieve this result.

*GauseIdentif* allows also to observe the progress of the identification in the parameter space  $r$  vs.  $K$  (Fig. A6 lower right window). The routine *ShowOrOpenParameterSpace* is called at the begin of the experiment *Identify*, which creates an additional window *Parameterspace*, using procedure *CreateWindow* from *DMWindows*. *ShowOrOpenParameterSpace* hereby calls procedure *RedrawParameterSpace*. *RedrawParameterSpace* uses *DMWindowIO*'s user coordinate plotting mechanism to draw the graph's panel and to show the initial parameter combination. *RedrawParameterSpace* is also called, each time the simulationist resizes the window, so that the graph  $r$  vs.  $K$  adjusts automatically to the window's current size. Then

after each simulation run the new parameter combination is drawn by means of procedure *ShowNewParameters* called again from within *PerformanceIndex* and displayed in the parameter IO-window by calling for each parameter the routine *SetP* from *SimBase*.

Finally the experiment *Identify* documents the results by executing two additional simulation runs, first with the original parameters (lower curve in Fig. A6 lower left window *Graph*) and then with the newly identified parameter values (upper curve in Fig. A6 lower left window *Graph*). Both runs together with the measurements (shown as scattergraph with symbols '\*' in Fig. A6 lower left window *Graph*) are displayed and procedure *Message* from *SimBase* is used to display the over-all improvement of the performance index (Fig. A6, upper right window *Table*). Note that some optimization algorithms are often not robust; hence, the procedure *ParametersPlausibles* installed by means of procedure *InstallStartConsistency* from *SimBase* to test the plausibility of the current parameter values before actually starting any simulation. In case *r* or *K* should be out of the ranges [*rMin*..*rMax*] respectively [*KMin*..*KMax*] the simulationist is warned and the identification can even be aborted.

Fig. A6 shows the results of the parameter identification starting with the way-off parameter estimates  $r = 1.0$  and  $K = 10.0$ . With these parameters the logistic equation produces the lower curve as shown in the window *Graph* (Fig. A6, lower left corner). After the optimization with the parameter identification algorithm *Amoeba* the parameters were  $r = 0.974$  and  $K = 59.7$  which results in the upper curve in the window *Graph*. The performance index was improved from  $2.746 \cdot 10^4$  to 971.752 hereby requiring 56 simulation runs, each run shown as a point on the curve in the window *Parameter space* (Fig. A6 lower right corner).

## A.4.6 Stochastic Simulations

Stochastic simulations require usually to run elaborate simulations, i.e. a structured simulation run or experiment, followed by a statistical analysis of the results. The random nature of the individual runs is produced by means of so-called pseudo random number generators (see e.g. library modules *RandGen*, *RandGen0*, and *RandNormal*).

### A.4.6.1 Third Order Finite Markov Chain - *Markov*

The following model definition program *Markov* demonstrates the typical use of the pseudo random number generator *U* for sampling uniformly distributed variates in the auxiliary library module *RandGen* and serves as an example for stochastic simulations. The program simulates a discrete time finite, 3rd-order Markov chain process. By default it models a population where each individual can be in one of the following states: healthy, ill, and dead; but this model can be adapted interactively to any other 3rd-order Markov chain process. Moreover, the program has been written such, that it can be easily adapted to simulate Markov chain processes of a different order.

The program accesses frequently the "Dialog Machine"; for instance it extends the standard user interface by installing the menu *Markov* with the command *Define...*. This menu command allows the simulationist to alter the meaning of the 3 states (procedure *AssignStatesNames*) and to set the coefficients of the Markov matrix (procedure *DefineMarkov*) in a more convenient way than this is possible with the IO-window *Model Parameters*

The model does not compute the temporal evolution of probability vectors, but rather simulates the fate of a vector of individuals  $x$  ( $x$ : *ARRAY [firstIndiv..lastIndiv] OF State*). The initial state of these individuals is sampled according to the initial state probabilities *initp* within procedure *Initialize*. Depending on the parameter *randomize* the random number generator is either randomized (*randomize*= 1) to get for each run a different result or reset (*randomize* = 0) to allow to repeat the simulation, hereby using exactly the same pseudo random number series. Procedure *Initialize* also initializes several auxiliary and statistical variables *pacc*, *fs*, *c*, *n*, *F* to allow for the computation of frequencies. *pacc* holds the accumulated transition probabilities from the Markov matrix *P*; thus allowing for a more efficient calculation of transitions during the stochastic simulation. The vector *fs* contains the accumulated state frequencies as the main monitorable variables, computed from the current states of the individuals (s.a. procedure *Dynamic*). The matrix *F* contains the frequencies of all transitions occurred since the begin of the simulation run, hereby using the counts *c* and *n*; the matrix *c*, contains the counts of all transitions, and vector *n*, the counts of the transitions starting from a particular state regardless of the destination state.

The coefficients of any probability vector such as *initp* or of a row of the Markov matrix *P* must add up to the sum 1. The Markov matrix *P* can not only be edited by means of the additional menu command *Define...*, but also via the IO-window *Model Parameters*. Moreover, the latter is also true for the initial state probabilities *initp*. Hence, the simulationist may easily specify illegal parameters violating any of the aforementioned conditions, leading to meaningless simulation results. ModelWorks allows to suppress any such illegal simulations by providing a mechanism for the installation of a consistency testing function procedure. The listed program installs the function procedure *TestConsistency* by calling procedure *InstallStartConsistency* from *SimBase*. *TestConsistency* is called at the very begin of each simulation run or after a pause and returns only TRUE if none of the aforementioned conditions are violated. If it should return FALSE the simulation will be aborted and the simulationist will be informed about this fact.

Each element of the vector of individuals  $x$  is of the finite enumeration type *State*, not of type *StateVar*. Since the mapping during every access of the real constants 1.0, 2.0, and 3.0 to the constants *one*, *two*, and *three* of the enumeration type *State* via a type conversion would be

quite inefficient, this model definition program does not declare any state variables in the simulation environment. Instead procedure *Dynamic* maintains the state vector  $x$  by first computing the new state  $xDash$  and then assigning  $xDash$  to  $x$ , similar to the way ModelWorks updates the state of discrete time models.

This program also demonstrates the use of state events. Since the default process contains an absorbing state, further computations beyond the state "all dead", i.e. probability vector [healthy,ill,dead] becomes [0,0,1], are superfluous. The terminate condition testing mechanism of ModelWorks (see function procedure *AllDead* installed via *InstallTerminateCondition*) will stop a running simulation anytime this state event is encountered.

As it holds in general for stochastic models, structured simulations are particularly important, for instance to estimate means or distributions. The procedure *TheExperiment* allows to assess statistics such as the expected mean  $E[f_{[j,k]}]$  of the frequencies  $f_{[j,k]}$  of all transitions for the time interval  $[\kappa_0, \kappa_f]$ . It first asks the simulationist for the experiment's sample size *maxRuns*, i.e. how many runs the experiment shall encompass, by calling the function procedure *NrRunsGiven*. If the simulationist has actually entered *maxRuns*, the experiment continues by suppressing all stash filing, since the transient behavior is of no interest for the final state distribution. Then the experiment forces the parameter *randomize* to 1.0, thus ensuring that true samples can be collected, and asks by a call to procedure *CreateNewFile* from *DMFiles* the simulationist to specify the file *recordF* (default name *Markov.DAT*) onto which the simulation results shall be recorded.

If the file *recordF* has been created successfully, a header line is written and the actual experiment starts. Note that the simulationist can abort the experiment any time by a call to menu command *Solve/Stop (Kill) run*, which will have the effect that the function procedure *ExperimentAborted* from *SimMaster* returns TRUE. At the end of every run the elements of the sampled matrix  $f_i$  are written onto the file *recordF*. The content of this file might look similar to the following excerpt:

Run	F[1,1]	F[1,2]	F[1,3]	F[2,1]	...	F[3,2]	F[3,3]	n
1	0.7510	0.1980	0.0510	0.5718	...	0.0000	1.0000	77
2	0.7459	0.1987	0.0555	0.5965	...	0.0000	1.0000	90
3	0.7435	0.2082	0.0483	0.6110	...	0.0000	1.0000	101
4	0.7575	0.1914	0.0511	0.5802	...	0.0000	1.0000	91
5	0.7549	0.1859	0.0592	0.5928	...	0.0000	1.0000	66

Legend:

- Run - Number of simulation run within experiment
- F[i,j] - Relative frequency of transition from state i to state j
- n - Number of transitions used to estimate F[i,j]

Experiment started on 01/Feb/1993 at 09:52:37

Experiment ended on 01/Feb/1993 at 09:53:09

The mean  $\mu_{[j,k]} = 1/n \sum f_{i[j,k]}$  ( $n \leq \text{maxRuns}$ ) is a reestimate of the coefficients of the Markov matrix  $p_{[j,k]}$ . To actually compute  $\mu_{[j,k]}$ , variances or other statistics use a statistical data analysis application such as *StatView*<sup>1</sup> to analyze the file *recordF*. *StatView* could be used successfully to read the simulation results directly by choosing the menu command *Import...* and selecting file *Markov.DAT* plus to compute and analyze the estimates such as  $\mu_{[j,k]}$ .

MODULE Markov;

(\*\*\*\*\*)

ModelWorks model: Markov

<sup>1</sup>StatView 512+™ is an interactive statistics & graphics package from Abacus Concepts, Inc., published by Brainpower, Inc., 24009 Ventura Blvd., Suite 250, Calabasas, CA 91302



Copyright 1989 by Andreas Fischlin and Swiss  
 Federal Institute of Technology Zurich ETHZ  
 Department of Environmental Sciences  
 Systems Ecology Group  
 ETH-Zentrum  
 CH-8092 Zurich  
 Switzerland

Version written for:  
 ModelWorks V2.0 (Modelling & Simulation)

Purpose Simulates a stochastic process defined by a given  
 Markov matrix in order to estimate the matrix  
 from the statistics collected during the simulations.

Implementation and Revisions:

=====

Author	Date	Description
-----	----	-----
af	18/10/89	First implementation (DM 2.0, MacMETH 2.6+, ModelWorks 1.3)
af	21/10/89	Extended to interactive renaming of states
af	03/05/90	Refining of experiment for direct import by StatView 512+ statistics program
af	25/11/90	Adaption for DM 2.02, i.e. uses DMClock.Now and DMClock.Today instead of DateAndTime.GetDateAndTime
ft	29/11/90	Adaption for MW 2.031, i.e. uses Types Parameter and AuxVar
dg	06/03/93	Import lists cleaned up
af	23/03/92	Adaptation to new MW 2.2
dg	25/04/96	Cleaned up for PC compatibility

\*\*\*\*\*)

```

FROM DMSystem IMPORT
    SuperScreen, MainScreen, MenuBarHeight, TitleBarHeight;
FROM DMStrings IMPORT
    Append, AppendCh, AssignString;
FROM DMConversions IMPORT
    IntToString;
FROM DMWindIO IMPORT
    BackgroundHeight, BackgroundWidth;
FROM DMMenus IMPORT
    Menu, Command, AccessStatus, Marking, InstallMenu,
    InstallCommand, InstallAliasChar;
FROM DMFiles IMPORT
    TextFile, CreateNewFile, Close, WriteEOL, PutReal, Response,
    WriteChar, WriteChars, PutInteger;
FROM DMEnterForms IMPORT
    FormFrame, WriteLabel, DefltUse, CharField, StringField,
    CardField, IntField, RealField, PushButton, RadioButtonID,
    DefineRadioButtonSet, RadioButton, CheckBox, UseEntryForm;
FROM DMClock IMPORT
    Today, Now;

FROM RandGen IMPORT
    U, ResetSeeds, Randomize;
FROM WriteDatTim IMPORT
    DateAndTimeRec, DateFormat, TimeFormat, WriteDate, WriteTime;

FROM SimBase IMPORT
    Model, IntegrationMethod, DeclM, RTCType, DeclP, StashFiling,
    Tabulation, Graphing, DeclMV, SetMonInterval,
    SetIntegrationStep, SetMV, GetMV, SetP, SetDefltP, SetDefltMV,
    GetDefltMV, LineStyle, GetCurveAttrForMV, SetCurveAttrForMV,
    
```

```

ClearGraph, GetDefltCurveAttrForMV, SetDefltCurveAttrForMV,
Stain, MWWindowArrangement, SetDefltWindowArrangement,
MWWindow, GetDefltWindowPlace, SetDefltWindowPlace, NoInput,
NoOutput, NoTerminate, NoAbout, Parameter, AuxVar;

FROM SimMaster IMPORT
  RunSimEnvironment, SimRun, InstallDefSimEnv,
  InstallStartConsistency, InstallTerminateCondition,
  ExperimentAborted, InstallExperiment, CurrentStep;

CONST
  firstIndiv = 1; lastIndiv = 100;
TYPE
  Individuals = [firstIndiv..lastIndiv];

TYPE
  State = (one, two, three);
CONST
  firstState = MIN(State); lastState = MAX(State);

VAR
  m: Model;
  x,xDash: ARRAY [firstIndiv..lastIndiv] OF State;
             (* pseudo state vars, i.e. not declared to ModelWorks *)
  P: ARRAY [firstState..lastState],[firstState..lastState] OF Parameter;
             (*Markov matrix*)
  Pacc: ARRAY [firstState..lastState],[firstState..lastState] OF REAL;
             (*Matrix containing accumulated transition probabilities*)
  initp: ARRAY [firstState..lastState] OF Parameter;
             (*Probabilities used to compute initial states*)
  fs: ARRAY [firstState..lastState] OF AuxVar;
             (*frequencies of states*)
  F: ARRAY [firstState..lastState],[firstState..lastState] OF AuxVar;
             (*frequencies of transitions*)
  C: ARRAY [firstState..lastState],[firstState..lastState] OF INTEGER;
             (*counting of transitions*)
  n: ARRAY [firstState..lastState] OF INTEGER;
             (*number of transitions starting from a state*)
  randomize: Parameter; (* controls seed randomization *)

PROCEDURE PRED(s: State): State;
BEGIN
  DEC(s); RETURN s;
END PRED;

PROCEDURE SUCC(s: State): State;
BEGIN
  INC(s); RETURN s;
END SUCC;

PROCEDURE Initialize;
  VAR l: Individuals; is,js: State; u : REAL;
BEGIN (*Initialize*)
  IF randomize>0.0 THEN Randomize ELSE ResetSeeds END;
  FOR is:= firstState TO lastState DO fs[is] := 0.0 END(*FOR*);
  FOR l:= firstIndiv TO lastIndiv DO
    u := U();
    IF u<=initp[one] THEN
      x[l] := one
    ELSIF u<=(initp[one]+initp[two]) THEN
      x[l] := two
    ELSE
      x[l] := three
    END
  END
END

```

```

    END(*IF*);
    fs[x[l]] := fs[x[l]] + 1.0;
END(*FOR*);
FOR is:= firstState TO lastState DO
    FOR js:= firstState TO lastState DO
        C[is,js] := 0;
        F[is,js] := 0.0;
    END(*FOR*);
    n[is] := 0;
END(*FOR*);
FOR is:= firstState TO lastState DO
    Pacc[is, firstState] := P[is,firstState];
    FOR js:= SUCC(firstState) TO lastState DO
        Pacc[is,js] := Pacc[is,PRED(js)] + P[is,js];
    END(*FOR*);
END(*FOR*);
fs[firstState] := fs[firstState]/FLOAT(lastIndiv-firstIndiv+1);
FOR is:= SUCC(firstState) TO lastState DO
    fs[is] := fs[PRED(is)] + fs[is]/FLOAT(lastIndiv-firstIndiv+1);
END(*FOR*);
END Initialize;

PROCEDURE InitSimSess;
    VAR is,js: State; curMin, curMax: REAL;
        curStain: Stain; curStyle: LineStyle; curSym: CHAR;
        curFiling: StashFiling; curTabul: Tabulation; curGraphing: Graphing;
BEGIN
    FOR is:= firstState TO lastState DO
        FOR js:= firstState TO lastState DO
            IF is<>three THEN
                GetCurveAttrForMV( m, F[is,js], curStain, curStyle, curSym );
                SetCurveAttrForMV( m, F[is,js], gold, spotted, 0C );
            ELSE
                GetMV( m, F[is,js], curMin, curMax,
                    curFiling, curTabul, curGraphing);
                SetMV( m, F[is,js], curMin, curMax,
                    curFiling, curTabul, notInGraph);
            END(*IF*);
        END(*FOR*);
    END(*FOR*);
    ClearGraph;
END InitSimSess;

PROCEDURE Dynamic;
    VAR l: Individuals; is,js: State;
    PROCEDURE Transition(oldS: State; VAR newS: State);
        VAR u: REAL;
    BEGIN (*Transition*)
        u := U();
        IF u<=Pacc[oldS,one] THEN
            newS := one
        ELSIF u<=Pacc[oldS,two] THEN
            newS := two
        ELSE
            newS := three
        END(*IF*);
    END Transition;
BEGIN (*Dynamic*)
    (* init state frequencies *)
    FOR is:= firstState TO lastState DO fs[is] := 0.0 END;
    (* compute new state vars & compute statistics *)
    FOR l:= firstIndiv TO lastIndiv DO
        Transition(x[l],xDash[l]);
        INC(C[x[l],xDash[l]]); INC(n[x[l]]); fs[x[l]] := fs[x[l]] + 1.0;
    END(*FOR*);

```

```

(* update pseudo state vars *)
FOR l:= firstIndiv TO lastIndiv DO
  x[l] := xDash[l];
END(*FOR*);
FOR is:= firstState TO lastState DO
  FOR js:= firstState TO lastState DO
    IF n[is]<>0 THEN
      F[is,js] := FLOAT(C[is,js])/FLOAT(n[is]);
    ELSE
      F[is,js] := 0.0;
    END(*IF*);
  END(*FOR*);
END(*FOR*);
fs[firstState] := fs[firstState]/FLOAT(lastIndiv-firstIndiv+1);
FOR is:= SUCC(firstState) TO lastState DO
  fs[is] := fs[PRED(is)] + fs[is]/FLOAT(lastIndiv-firstIndiv+1);
END(*FOR*);
END Dynamic;

PROCEDURE CircaEqual(x,y,eps: REAL): BOOLEAN;
BEGIN (*CircaEqual*)
  RETURN ((x-eps)<=y) AND (y<=(x+eps))
END CircaEqual;

PROCEDURE TestConsistency(): BOOLEAN;
  VAR is,js: State; sum: REAL; sofarOk: BOOLEAN;
BEGIN (*TestConsistency*)
  sum := 0.0;
  FOR is:= firstState TO lastState DO
    sum := sum + initp[is];
  END(*FOR*);
  sofarOk := CircaEqual(sum,1.0,1.0E-3);
  FOR is:= firstState TO lastState DO
    sum := 0.0;
    FOR js:= firstState TO lastState DO
      sum := sum + P[is,js];
    END(*FOR*);
    sofarOk := sofarOk AND CircaEqual(sum,1.0,1.0E-3);
  END(*FOR*);
  RETURN sofarOk
END TestConsistency;

PROCEDURE AllDead(): BOOLEAN;
BEGIN
  RETURN CircaEqual(fs[one],0.0,1.0E-3)
    AND CircaEqual(fs[two]-fs[one],0.0,1.0E-3);
END AllDead;

VAR
  myMenu: Menu; defMarkovCmd: Command;
  nameStateOne, nameStateTwo, nameStateThree: ARRAY [0..40] OF CHAR;

PROCEDURE AssignStatesNames(n1,n2,n3: ARRAY OF CHAR);
  VAR l: Individuals; is,js: State;
  istr, descr,ident: ARRAY [0..40] OF CHAR;

PROCEDURE StateToString (s: State; VAR str: ARRAY OF CHAR);
BEGIN (*StateToString*)
  CASE s OF
    one : AssignString(nameStateOne,str);
    | two : AssignString(nameStateTwo,str);
    | three : AssignString(nameStateThree,str);
  END(*CASE*);
END StateToString;

```

```

BEGIN (*. AssignStatesNames .*)
  AssignString(n1,nameStateOne);
  AssignString(n2,nameStateTwo);
  AssignString(n3,nameStateThree);
  FOR is:= firstState TO lastState DO
    AssignString("Initial prob. of state ",descr);
    StateToString(is,istr); Append(descr,istr);
    ident := "initp[";
    IntToString(ORD(is)+1,istr,0); Append(ident,istr);
    AppendCh(ident,"]");
    SetDefltP(m,initp[is],initp[is],0.0,1.0,
      rtc,descr,ident,"");
  END(*FOR*);
  FOR is:= firstState TO lastState DO
    FOR js:= firstState TO lastState DO
      AssignString("Prob. Transition ",descr);
      StateToString(is,istr); Append(descr,istr);
      Append(descr,"->");
      StateToString(js,istr); Append(descr,istr);
      ident := "P[";
      IntToString(ORD(is)+1,istr,0); Append(ident,istr);
      AppendCh(ident,",");
      IntToString(ORD(js)+1,istr,0); Append(ident,istr);
      AppendCh(ident,"]");
      SetDefltP(m,P[is,js],P[is,js],0.0,1.0, rtc, descr,ident,"");
    END(*FOR*);
  END(*FOR*);
  (* declaration of monitorable variables *)
  FOR is:= firstState TO lastState DO
    AssignString("State freq. ",descr);
    StateToString(is,istr); Append(descr,istr);
    ident := "fs[";
    IntToString(ORD(is)+1,istr,0); Append(ident,istr);
    AppendCh(ident,"]");
    SetDefltMV(m,fs[is],0.0,1.0, descr,ident,"",
      notOnFile,notInTable,isY);
  END(*FOR*);
  FOR is:= firstState TO lastState DO
    FOR js:= firstState TO lastState DO
      AssignString("Freq. transition ",descr);
      StateToString(is,istr); Append(descr,istr);
      Append(descr,"->");
      StateToString(js,istr); Append(descr,istr);
      ident := "F[";
      IntToString(ORD(is)+1,istr,0); Append(ident,istr);
      AppendCh(ident,",");
      IntToString(ORD(js)+1,istr,0); Append(ident,istr);
      AppendCh(ident,"]");
      SetDefltMV(m,F[is,js],0.0,1.0, descr,ident,"",
        writeOnFile,writeInTable,isY);
    END(*FOR*);
  END(*FOR*);
END AssignStatesNames;

PROCEDURE DefineMarkov;
  CONST lem = 3; tab1 = 25; tab2 = 35; tab3 = 45;
  VAR ef: FormFrame; ok: BOOLEAN; cl: INTEGER;
BEGIN
  cl := 2;
  WriteLabel(cl,lem,"Define Markov Process:"); INC(cl);
  WriteLabel(cl,lem,"State");
  WriteLabel(cl,tab1,"Transition probabilities"); INC(cl);
  StringField(cl,lem,15,nameStateOne,useAsDeflt);
  RealField(cl,tab1,7,P[one,one],useAsDeflt,0.0,1.0);
  RealField(cl,tab2,7,P[one,two],useAsDeflt,0.0,1.0);
  RealField(cl,tab3,7,P[one,three],useAsDeflt,0.0,1.0);
  INC(cl);
  StringField(cl,lem,15,nameStateTwo,useAsDeflt);

```

ModelWorks 2.2 - Appendix (Sample Models)

```

RealField(cl,tab1,7,P[two,one],useAsDeflt,0.0,1.0);
RealField(cl,tab2,7,P[two,two],useAsDeflt,0.0,1.0);
RealField(cl,tab3,7,P[two,three],useAsDeflt,0.0,1.0);
INC(cl);
StringField(cl,lem,15,nameStateThree,useAsDeflt);
RealField(cl,tab1,7,P[three,one],useAsDeflt,0.0,1.0);
RealField(cl,tab2,7,P[three,two],useAsDeflt,0.0,1.0);
RealField(cl,tab3,7,P[three,three],useAsDeflt,0.0,1.0);
INC(cl);
ef.x:= 0; ef.y:= -1 (*display entry form in middle of screen*);
ef.lines:= cl+1; ef.columns:= 55;
UseEntryForm(ef,ok);
IF ok THEN AssignStatesNames(nameStateOne,nameStateTwo,nameStateThree) END;
END DefineMarkov;

```

```

PROCEDURE ModelObjects;
  VAR l: Individuals; is,js: State;
  istr, descr,ident: ARRAY [0..40] OF CHAR;
BEGIN (*Objects*)
  (* declaration of parameters *)
  FOR is:= firstState TO lastState DO
    DeclP(initp[is],1.0/FLOAT(ORD(lastState)+1),0.0,1.0,
      rtc,"","","");
  END(*FOR*);
  DeclP(randomize,0.0,0.0,1.0,
    rtc,"Randomize option (= 0 don't, = 1 do randomize)",
    "randomize","[0..1]");
  P[one,two] := 0.2;
  P[one,three] := 0.05;
  P[one,one] := 1.0 - P[one,two] - P[one,three];
  P[two,two] := 0.3;
  P[two,three] := 0.1;
  P[two,one] := 1.0 - P[two,two] - P[two,three];
  P[three,two] := 0.0;
  P[three,one] := 0.0;
  P[three,three] := 1.0;
  FOR is:= firstState TO lastState DO
    FOR js:= firstState TO lastState DO
      DeclP(P[is,js],P[is,js],0.0,1.0, rtc, "", "", "");
    END(*FOR*);
  END(*FOR*);

  (* compute initial states *)
  Initialize;

  (* declaration of monitorable variables *)
  FOR is:= firstState TO lastState DO
    DeclMV(fs[is],0.0,1.0, "", "", "",
      notOnFile,notInTable,isY);
  END(*FOR*);
  SetDefltCurveAttrForMV(m, fs[one], emerald, unbroken, "." );
  SetDefltCurveAttrForMV(m, fs[two], ruby, unbroken, "o" );
  SetDefltCurveAttrForMV(m, fs[three], coal, unbroken, "+" );
  FOR is:= firstState TO lastState DO
    FOR js:= firstState TO lastState DO
      DeclMV(F[is,js],0.0,1.0, "", "", "",
        writeOnFile,writeInTable,isY);
      SetDefltCurveAttrForMV( m, F[is,js], autoDefCol, autoDefStyle, 0C );
    END(*FOR*);
  END(*FOR*);
  AssignStatesNames("healthy","ill","dead");
END ModelObjects;

```

```

VAR
  recordF: TextFile;

```

```

PROCEDURE WriteOnFile(ch: CHAR); (* Needed by RecordDateTime *)

```

```

BEGIN
  WriteChar(recordF,ch);
END WriteOnFile;

PROCEDURE TheExperiment;
  CONST TAB = 11C;
  VAR is,js: State;
      dt: DateAndTimeRec; curMin, curMax: REAL;
      curFiling: StashFiling; curTabul: Tabulation; curGraphing: Graphing;
      dummyStr, ident: ARRAY [0..63] OF CHAR;i, maxRuns: CARDINAL;

  PROCEDURE NrRunsGiven(): BOOLEAN;
    CONST lem = 5; tab = 35; VAR ef: FormFrame; ok: BOOLEAN; cl: INTEGER;
  BEGIN (*NrRunsGiven*)
    cl := 2;
    WriteLabel(cl,lem,"How many runs:");
    CardField(cl,tab,7,maxRuns,useAsDeflt,1,MAX(CARDINAL)); INC(cl);
    ef.x:= 0; ef.y:= -1 (* display entry form in middle of screen *);
    ef.lines:= cl+1; ef.columns:= 55;
    UseEntryForm(ef,ok);
    RETURN ok
  END NrRunsGiven;

  PROCEDURE GetDateAndTime(VAR dt: DateAndTimeRec);
  BEGIN
    Today (dt.year,dt.month,dt.day, dt.dayOfWeek);
    Now (dt.hour,dt.minute, dt.second);
  END GetDateAndTime;

  PROCEDURE RecordDateTime(s: ARRAY OF CHAR; dt: DateAndTimeRec);
  BEGIN
    WriteChars(recordF,s);
    WriteDate(dt,WriteOnFile,letMonth); WriteChars(recordF," at ");
    WriteTime(dt,WriteOnFile,brief24hSecs); WriteEOL(recordF);
  END RecordDateTime;

  BEGIN (*TheExperiment*)
    maxRuns := 100;
    IF NrRunsGiven() THEN
      FOR is:= firstState TO lastState DO
        FOR js:= firstState TO lastState DO
          GetMV( m, F[is,js], curMin, curMax,
                curFiling, curTabul, curGraphing);
          SetMV( m, F[is,js], curMin, curMax,
                notOnFile, curTabul, curGraphing);
        END(*FOR*);
        GetMV( m, fs[is], curMin, curMax,
              curFiling, curTabul, curGraphing);
        SetMV( m, fs[is], curMin, curMax,
              notOnFile, curTabul, curGraphing);
        SetP( m, initp[is], 0.0);
      END(*FOR*);
      SetP(m,initp[one], 1.0);
      SetP(m,randomize,1.0);
      CreateNewFile(recordF,"Record results on file","Markov.DAT");
      IF recordF.res=done THEN
        GetDateAndTime(dt);
        WriteChars(recordF,"Run"); WriteChar (recordF,TAB);
        FOR is:= firstState TO lastState DO
          FOR js:= firstState TO lastState DO
            GetDefltMV(m,F[is,js],curMin, curMax,
                      dummyStr,ident,dummyStr,
                      curFiling, curTabul, curGraphing);
            WriteChars(recordF,ident); WriteChar (recordF,TAB);
          END(*FOR*);
        END(*FOR*);
        WriteChars(recordF,"n");
        WriteEOL(recordF);
      END
    END
  END

```

```

i := 1;
WHILE (i <= maxRuns) AND NOT ExperimentAborted() DO
  SimRun;
  PutInteger(recordF,i,0); WriteChar (recordF,TAB);
  FOR is:= firstState TO lastState DO
    FOR js:= firstState TO lastState DO
      PutReal(recordF,F[is,js],8,4); WriteChar (recordF,TAB);
    END(*FOR*);
  END(*FOR*);
  PutInteger(recordF,CurrentStep(),0);
  WriteEOL(recordF);
  INC(i);
END(*WHILE*);
WriteChars(recordF, "Legend:"); WriteEOL(recordF);
WriteChars(recordF, "  Run    - Number of simulation run within experiment");
WriteEOL(recordF);
WriteChars(recordF, "  F[i,j] - Relative frequency of transition from state i to state j");
WriteEOL(recordF);
WriteChars(recordF, "  n      - Number of transitions used to estimate F[i,j]");
WriteEOL(recordF);
RecordDateTime("Experiment started on ",dt);
GetDateAndTime(dt);
RecordDateTime("Experiment ended on ",dt);
Close(recordF);
END(*IF*);
END(*IF*);
END TheExperiment;

```

PROCEDURE Definitions;

```

CONST marg = 2;
VAR supScr,x,y,w,h,nc: INTEGER; enabl: BOOLEAN;
BEGIN
  DeclM(m, discreteTime, Initialize, NoInput, NoOutput, Dynamic, NoTerminate,
    ModelObjects, 'Markov chain simulated', 'm', NoAbout);
  SetIntegrationStep(1.0);
  SetMonInterval(1.0);
  InstallDefSimEnv(InitSimSess);
  InstallStartConsistency(TestConsistency);
  InstallTerminateCondition(AllDead);
  InstallExperiment(TheExperiment);
  SetDeflWindowArrangement(tiled);
  SuperScreen(supScr,x,y,w,h,nc,TRUE(*color priority*));
  IF supScr<>MainScreen() THEN
    SetDeflWindowPlace(GraphW,x+2,y+3,w-6,h-TitleBarHeight()-5);
    GetDeflWindowPlace(TableW,x,y,w,h,enabl);
    SetDeflWindowPlace(TableW,x,y,BackgroundWidth(),h);
  ELSE
    SetDeflWindowPlace(GraphW,marg,
      TitleBarHeight()+marg+(BackgroundHeight()-2*marg) DIV 3,
      BackgroundWidth()-2*marg,
      (BackgroundHeight()-MenuBarHeight()-2*marg)*2 DIV 3);
    SetDeflWindowPlace(TableW,marg,
      marg,
      BackgroundWidth()-2*marg,
      (BackgroundHeight()-2*marg) DIV 3);
  END(*IF*);
  InstallMenu(myMenu,'Markov',enabled);
  InstallCommand(myMenu,defMarkovCmd,"Define...",
    DefineMarkov,enabled, unchecked);
  InstallAliasChar(myMenu,defMarkovCmd, "W");
END Definitions;

```

BEGIN

```

  RunSimEnvironment(Definitions);
END Markov.

```



A.4.6.2 Statistical Analysis of Simulation Results - *StochLogGrow*

Repeating many individual simulation runs allows to estimate by means of the so-called Monte-Carlo simulation technique the properties such as the means or the variances of the resulting random variables which the model generates. Sample model *StochLogGrow* is based on a continuous time logistic growth model (s.a. sample model *Logistic*) where the model parameters vary stochastically during every integratio step. In order to analyze the stochastic properties of the mean behavior of such a model, several runs are executed and iteratively analyzed by means of the auxiliary module *StochStat*. This allows to estimate a confidence interval of mean behavior as estimated by the Monte-Carlo technique.

```
MODULE StochLogGrow;
```

```
(*
```

```
Module      StochLogGrow
```

```
Purpose
```

```
Demonstration of the simulation of a stochastic growth process
and the statistical analysis of the simulation results.
```

```
The model simulates a logistic growth of the biomass of a
grass species. The growth parameters r and K are normally
distributed random variables, which are sampled anew during
each time step of the numerical integration. This procedure
is to simulate the variability of the environment. A Monte
Carlo experiment (procedure MonteCarloExperiment) allows to
sample multiple realisations of this stochastic process in
order to estimate the expected value of the state variable
plus an interval of confidence by means of module StochStat.
```

```
Revision history:
```

```
=====
```

Author	Date	Description
-----	----	-----
tn	24/04/90	First implementation demonstrating use of StochStat
AF	14/12/93	Preparation as a sample model
dg	25/04/96	Cleaned up for PC compatibility

```
*)
```

```
FROM DMWindIO IMPORT SetMode, PaintMode;
FROM DMMessages IMPORT Inform;
```

```
FROM RandGen IMPORT U, Randomize;
FROM RandNormal IMPORT Np, InstallU;
FROM StochStat IMPORT
  StatArray, Prob2Tail, DeclStatArray, RemoveStatArray,
  PutValue, DeclDispMV, DisplayArray;
```

```
FROM SimBase IMPORT
  Model, IntegrationMethod, DeclM, DeclSV, DeclP, RTCType,
  StashFiling, Tabulation, Graphing, DeclMV, SetSimTime,
  NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout,
  DoNothing, StateVar, Derivative, AuxVar, Parameter,
  SetDeflCurveAttrForMV, Stain, LineStyle, GetCurveAttrForMV,
  SetCurveAttrForMV, GetGlobSimPars, ClearGraph,
  InstallClientMonitoring;
```

```
FROM SimMaster IMPORT
  SimRun, RunSimEnvironment, CurrentTime, InstallExperiment,
  ExperimentRunning, ExperimentAborted;
```

```
FROM SimGraphUtils IMPORT timeIsIndep;
```

```

CONST
  grass0 = 1.0;
  grassMax = 10000.0;

VAR
  m: Model;
  grass:      StateVar;
  grassDot:   Derivative;
  expectedGrass,
  r, K:       AuxVar;
  mur, muK,
  sigmar, sigmaK: Parameter;

  maxRuns, nRun,
  clrAll, rand: REAL;
  statArray:  StatArray;
  jt:         INTEGER;

PROCEDURE Dynamic;
  PROCEDURE AvoidOverflow(VAR grass: StateVar);
  BEGIN
    IF grass < grass0 THEN (*force grass=grass0*) grass := grass0 END;
  END AvoidOverflow;
  BEGIN
    AvoidOverflow(grass);
    r := Np(mur, sigmar);
    K := Np(muK, sigmaK);
    grassDot := r * ((K - grass) / K) * grass;
  END Dynamic;

PROCEDURE DoMonit;
  BEGIN
    IF ExperimentRunning() THEN
      INC(jt); PutValue(statArray, jt, CurrentTime(), grass);
    END (*IF*);
  END DoMonit;

PROCEDURE ModelObjects;
  BEGIN
    DeclSV(grass, grassDot, grass0, 0.0, grassMax,
      "Grass", "G", "g dry weight/m^2");

    DeclMV(grass, 0.0, 1000.0, "Grass", "G", "g dry weight/m^2",
      notOnFile, notInTable, isY);
    SetDefltCurveAttrForMV(m, grass, emerald, unbroken, 0C );
    DeclMV(expectedGrass, 0.0, 1000.0, "Expected value of grass", "E[G]", "g dry weight/m^2",
      notOnFile, notInTable, isY);
    SetDefltCurveAttrForMV(m, expectedGrass, turquoise, unbroken, 0C );

    DeclP(mur, 0.7, 0.0, 10.0, rtc,
      "Mean of r (=growth rate of grass)", "μ(r)", "day^-1");
    DeclP(muK, mur/0.001, 0.0, grassMax, rtc,
      "Mean of K (=carrying capacity)", "μ(K)", "m^2/g dw/day");
    DeclP(sigmar, 0.5, 0.0, 10.0, rtc,
      "Standard deviation of r (=growth rate of grass)", "s(r)", "day^-1");
    DeclP(sigmaK, muK*0.3, 0.0, grassMax, rtc,
      "Standard deviation of K (=carrying capacity)", "s(K)", "m^2/g dw/day");

    DeclP(maxRuns, 20.0, 0.0, 100.0, rtc,
      "Number of runs in experiment", "# runs", "#");
    DeclP(nRun, maxRuns/5.0, 0.0, 100.0, rtc,
      "Show statistics each nRun'th run", "nRun", "#");
    DeclP(clrAll, 0.0, 0.0, 1.0, rtc,

```

```

"Clear graph regularly (1=yes/0=no)", "clrAll", "");
DeclP(rand, 1.0, 0.0, 1.0, rtc,
"Randomize experiment (1=yes/0=no)", "rand", "");
END ModelObjects;

PROCEDURE MonteCarloExperiment;
  VAR i : INTEGER;
      t0, tend, h, er, c, hm: REAL;
      curSt: Stain; curLS: LineStyle; curCh: CHAR;
  PROCEDURE ShowStatistics(stErrBar, stMue: Stain; mueCh: CHAR);
  BEGIN
    SetCurveAttrForMV(m, expectedGrass, stErrBar, unbroken, 0C );
    DisplayArray(statArray, TRUE, prob950);
    SetCurveAttrForMV(m, expectedGrass, stMue, unbroken, mueCh );
    DisplayArray(statArray, FALSE, prob950);
  END ShowStatistics;
  PROCEDURE ROUND(x: REAL): INTEGER;
  BEGIN
    RETURN TRUNC(x+0.5)
  END ROUND;
  BEGIN (*MonteCarloExperiment*)
    GetGlobSimPars(t0, tend, h, er, c, hm);
    DeclStatArray(statArray, ROUND( (tend-t0)/hm + 1.0 ) );
    DeclDispMV(statArray, m, expectedGrass, m, timeIsIndep);
    GetCurveAttrForMV(m, grass, curSt, curLS, curCh );
    IF ROUND(rand)=1 THEN Randomize END;
    i := 0;
  LOOP
    jt := 0; INC(i);
    SimRun;
    IF (i=ROUND(maxRuns)) OR ExperimentAborted() THEN EXIT END;
    IF (i MOD ROUND(nRun)) = 0 THEN (* show statistics *)
      IF ODD(i DIV ROUND(nRun) ) THEN
        IF (ROUND(clrAll)=0) AND (i=ROUND(nRun)) THEN (* very first time *)
          (* hide from now on individual run results to keep graph simpler *)
          Inform("From now on individual run results will be hidden.",
            "But the converging statistics will be shown instead.",
            "");
          SetCurveAttrForMV(m, grass, curSt, invisible, 0C );
          ClearGraph;
          END(*IF*);
          ShowStatistics(turquoise,turquoise,0C);
        ELSE
          IF ROUND(clrAll)=1 THEN
            Inform("Attention: graph will now be cleared to avoid",
              "clutter! However, the statistics shown next",
              "summarizes the whole simulation history.");
            ClearGraph;
            END;
            ShowStatistics(sapphire,sapphire,0C);
          END(*IF*);
        END(*IF*);
      END(*LOOP*);
    IF i>0 THEN
      ShowStatistics(pink,ruby,"•");
    END(*IF*);
    RemoveStatArray(statArray);
    SetCurveAttrForMV(m, grass, curSt, curLS, curCh );
  END MonteCarloExperiment;

PROCEDURE ModelDefinitions;
  BEGIN
    DeclM(m, Euler, NoInitialize, NoInput, NoOutput, Dynamic,
      NoTerminate, ModelObjects, "Stochastic logistic grass growth",
      "StochGrass", NoAbout);
    SetSimTime(0.0,30.0);
    InstallExperiment(MonteCarloExperiment);
  END

```

## ModelWorks 2.2 - Appendix (Sample Models)

```
    InstallClientMonitoring(DoNothing, DoMonit, DoNothing);  
    InstallU(U);  
END ModelDefinitions;
```

```
BEGIN  
    RunSimEnvironment(ModelDefinitions);  
END StochLogGrow.
```

#### A.4.7 Modular Modeling - *GreenHouse*

There are two basic techniques of modular modeling: First, declaring several ModelWorks models within the same model definition program; second, splitting a structured model system into several submodels, where each module contains a single submodel (s.a. *Theory* chapter *Structured Model Definition Programs (Modular Modeling)*)

The sample model *GreenHouse* demonstrates the second technique, i.e. the splitting of a structured model system into several submodels implemented as independent modules. It simulates the green-house effect based on a simplified version of the global carbon cycle (Fig. A7) between atmosphere and terrestrial biosphere. The two compartments atmosphere and biosphere are modeled separately, so that the behavior of each component can be studied independently or in a combined way (Fig. A8).

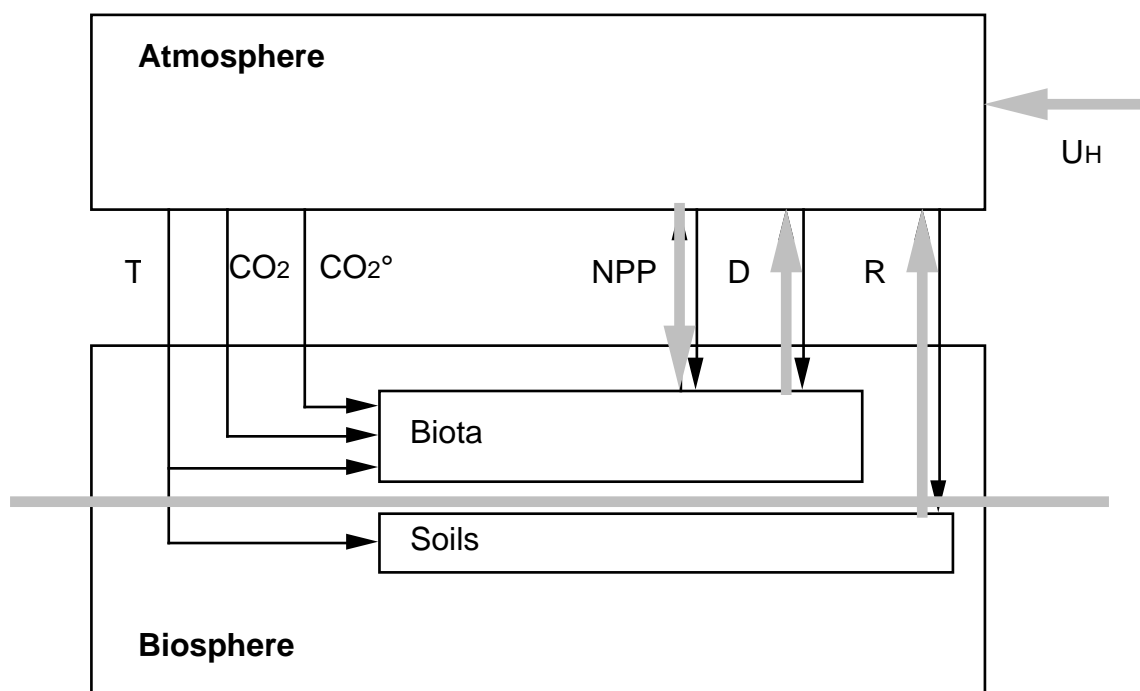


Fig. A7: Main components and interactions modeled by the sample model *GreenHouse*. Atmosphere and biosphere form a structured model system, which can be modeled in a modular way where both spheres are represented as a submodel. In the corresponding model definition program, each sphere is implemented in form of a separate module.

It is intuitively appealing to separate the whole system into two components or submodels, i.e. the atmosphere and the biosphere. The submodel atmosphere is defined by module *GHAtmosphere* (GH stands for Green House), the biosphere including the soils by module *GHBiosphere*. Since we need to make some overall observations on the global C-cycle, we can add an additional module called *GHObserver* (Fig. A8). It can also be formulated as a submodel, which is convenient, since it observes total carbon injected as an integral of annual anthropogenic fluxes. Yet, note that such an observer, despite its internal dynamics, has no influence on the dynamics of the global carbon cycle modelled by the two other submodels; it only receives inputs which are output by these two other submodels. Finally we require a program module to bind all parts together, i.e. module *GHMaster* (Fig. A8).

Since this structure offers much flexibility I recommd to implement structured models according to this technique (see part II *Theory* chapter *Formalisms*). Moreover, since this structure is the same for other systems (see e.g. research sample model *LBM* below), it can be supported by a general auxiliary module, i.e. *StructModAux*. The latter allows to announce via a simple interface the dynamic instantiation (declaration) of submodels independently from each other. Thus, any master module is built similar to *GHMaster* and becomes very simple. *StructModAux* offers the simulationist a mechanism to activate or deactivate submodels dynamically via menu commands from within the simulation environment (see part II *Theory*, chapter *Functions*, section *User Interface Customization*). If the modeller wishes to prevent the removal of a particular submodel during simulations, the procedures *DeactivatexyzModel* can remove the submodel xyz only conditionally, i.e. after inspecting the current state of the simulation environment by a call to procedure *GetMWState* from module *SimMaster*.

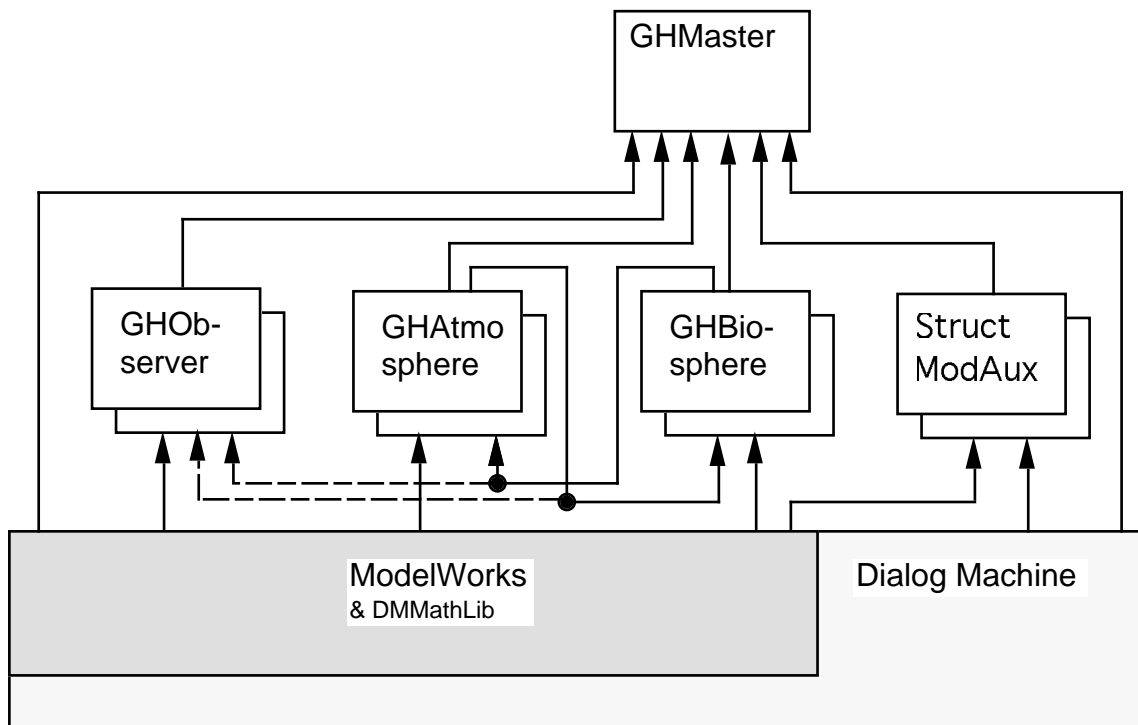


Fig. A8: Module structure of the sample model *GreenHouse*.

All these modules form a so-called RAMSES<sup>1</sup> project. A list of all the files which hold the involved modules is contained in a project description file, here called *GHMaster.PRJ*:

```
GHAtmosphere.DEF
GHBiosphere.DEF
GHObserver.DEF
GHAtmosphere.MOD
GHBiosphere.MOD
GHObserver.MOD
GHMaster.MOD
GHMaster.R
```

<sup>1</sup>On the IBM PC use the "Make" facility of the used development environment to achieve a similar result.

The master module *GHMaster* imports from every submodel module and installs the submodel activating routines in *StructModAux*'s mechanism:

```

MODULE GHMaster;

(*
  Module GHMaster (Master module of structured model Green-House)

  Purpose: Demonstration of modular modeling using RAMSES
           and ModelWorks software.

  The structured model simulates the global carbon cycle; in
  particular the interaction between the compartment atmosphere
  and terrestrial biosphere under an anthropogenetic
  green-house gas forcing. The model has been derived from data
  and some model equations described in the following references.

  References :

    Schneider, S.H., 1989. The greenhouse effect: science and
    policy. Science 243: 771-81.

    Kohlmaier,G.H., Janecek, A. & Kindermann, J., 1990. In: Bouwman,
    A.F. (ed.), Soils and the greenhouse effect. John Wiley &
    Sons: 415-422.

    Bolin, B., 1986. How much CO2 will remain in the atmopshere?. In:
    Bolin, B., Döös, B.R., Jäger, J. & Warrick, R.A. (eds.), The
    greenhouse effect, climatic change and ecosystems. Wiley,
    Chichester a.o. (SCOPE Vol. 29): 93-156.

  Revision history:
  =====

    Author  Date      Description
    -----  ----      -
    AF       3/11/93    First implementation

*)

FROM DMMenus IMPORT Command, InstallCommand, Separator, InstallSeparator,
  AccessStatus, Marking;

(* Imports from ModelWorks (Sim) *)
FROM SimBase IMPORT SetDefltGlobSimPars, MWWindowArrangement;
FROM SimMaster IMPORT RunSimEnvironment;
FROM SimGraphUtils IMPORT PlaceGraphOnSuperScreen;

FROM StructModAux IMPORT InstallCustomMenu, SetSimEnv, AssignSubModel,
  InstallMyGlobPreferences, customM;
FROM Help IMPORT ShowHelpWindow, SetHelpFileName, SetResourceFileName;

(* Imports from modular model definitions (GH-modules) *)
FROM GHAtmosphere IMPORT atmosModelDescr,
  ActivateAtmosModel, DeactivateAtmosModel, AtmosModelIsActive;

FROM GHBiosphere IMPORT biosModelDescr,
  ActivateBiosModel, DeactivateBiosModel, BiosModelIsActive;

FROM GHObserver IMPORT obsModelDescr,
  ActivateObsModel, DeactivateObsModel, ObsModelIsActive;

VAR
  atmos, bios, bios2, obs: INTEGER;

```

```

helpCmd: Command;

PROCEDURE GiveHelp;
BEGIN
  SetHelpFileName("GreenHouse Help");
  SetResourceFileName("GreenHouse Help");
  ShowHelpWindow;
END GiveHelp;

PROCEDURE InitSimEnv;
BEGIN
  InstallCustomMenu("Models","Activation...","L");
  SetSimEnv(atmos,bios,obs);
  InstallSeparator(customM,line);
  InstallCommand(customM,helpCmd, "On the model...", GiveHelp, enabled,unchecked);
END InitSimEnv;

PROCEDURE SetMyGlobPreferences;
BEGIN
  SetDefltGlobSimPars(1900.0, 2300.0, 0.5, 0.0001, 1.0, 10.0);
  PlaceGraphOnSuperScreen(tiled);
END SetMyGlobPreferences;

BEGIN
  InstallMyGlobPreferences(SetMyGlobPreferences);
  AssignSubModel(atmos, atmosModelDescr,
    ActivateAtmosModel, DeactivateAtmosModel, AtmosModelIsActive);
  AssignSubModel(bios, biosModelDescr,
    ActivateBiosModel, DeactivateBiosModel, BiosModelIsActive);
  AssignSubModel(obs, obsModelDescr,
    ActivateObsModel, DeactivateObsModel, ObsModelIsActive);
  RunSimEnvironment( InitSimEnv );
END GHMaster.

```

Each submodel has to provide a similar interface: First, it has to export its output variables on behalf of the modules which need the input (see also part II *Theory*, section *Structured Model Definition Programs (Modular Modeling)* in particular Fig. T28). Second, it has to provide procedures which allow *GHMaster* to announce the submodel to the auxiliary module *StructModAux*. The following two definition modules for the submodels atmosphere and biosphere illustrate this technique:

```

DEFINITION MODULE GHAtmosphere;

(*****

Module GHAtmosphere      (Version 1.0)

    Copyright (c) 1993 by Andreas Fischlin and Swiss
    Federal Institute of Technology Zürich ETHZ

    Purpose Submodel modeling the C-dynamics of the atmosphere

    Remark This module is the submodel Atmosphere of the
    structured model Green-House (GH)

    Programming

    o Design and Implementation
      A. Fischlin      15/12/93

```



Systems Ecology  
 Institute of Terrestrial Ecology  
 Department of Environmental Sciences  
 Swiss Federal Institute of Technology Zurich ETHZ  
 Grabenstr. 3  
 CH-8952 Schlieren/Zurich  
 Switzerland

Last revision of definition: 15/12/93 AF

\*\*\*\*\*)

FROM SimBase IMPORT AuxVar, Parameter, OutVar;

(\* exported for submodel Biosphere \*)

VAR

CO2: OutVar; (\* CO2-concentration in the atmosphere \*)  
 CO20: Parameter; (\* Initial CO2-concentration in the atmosphere \*)  
 deltaT: OutVar; (\* change in global annual mean near surface temperature,  
 e.g. global warming caused by CO2-increase \*)

(\* exported for submodel Observer only: \*)

cInAnthros: OutVar; (\* Anthropogenic CO2 C-input-flux into atmosphere  
 from fossil fuel burning \*)  
 cDeltaInAtmos: OutVar; (\* Change of carbon stored in atmosphere \*)  
 fRemInA: Parameter; (\* Fraction of the net C-input-flux into atmosphere  
 which remains there, i.e. which is not absorbed  
 by oceans \*)

(\* exported for GHMaster only: \*)

CONST

atmosModelDescr = "Atmosphere";

PROCEDURE ActivateAtmosModel;  
 PROCEDURE DeactivateAtmosModel;  
 PROCEDURE AtmosModelIsActive(): BOOLEAN;

END GHAtmosphere.

DEFINITION MODULE GHBiosphere;

\*\*\*\*\*

Module GHBiosphere (Version 1.0)

Copyright (c) 1993 by Andreas Fischlin and Swiss  
 Federal Institute of Technology Zürich ETHZ

Purpose Submodel modeling the C-dynamics of the biosphere  
 composed of the biota and soils

Remark This module is the submodel Biosphere of the  
 structured model Green-House (GH)

Programming

o Design and Implementation  
 A. Fischlin 15/12/93

Systems Ecology  
 Institute of Terrestrial Ecology  
 Department of Environmental Sciences  
 Swiss Federal Institute of Technology Zurich ETHZ  
 Grabenstr. 3  
 CH-8952 Schlieren/Zurich  
 Switzerland

Last revision of definition: 15/12/93 AF

\*\*\*\*\*)

FROM SimBase IMPORT AuxVar, Parameter, OutVar;

(\* exported for submodel Atmosphere: \*)

VAR

prodBio: OutVar; (\* productive world biota (NPP) \*)  
 Q10AB: Parameter; (\* Q10-value for atmosphere -> biosphere C-flux,  
 i.e. C-fixation by photosynthesis \*)  
 decSOM: OutVar; (\* C in Soil Organic Matter (SOM) exposed to  
 oxidation \*)  
 Q10SA: Parameter; (\* Q10-value for soils -> atmosphere C-flux,  
 i.e. soil respiration \*)  
 deforestation: OutVar; (\* C flux biosphere -> atmosphere due to  
 land use changes, i.e. deforestation \*)

PROCEDURE TEffect(Q10ij: Parameter; deltaT: AuxVar): AuxVar;  
 (\* Change of the C-flux from compartment i to j, caused by a  
 temperature change deltaT (assumes a change of Q10-value of Q10ij  
 per 10° temperature change) \*)

PROCEDURE CO2Fertilization(CO2conc: AuxVar): AuxVar;  
 (\* Increase of the atmosphere -> biosphere C-flux caused by the  
 photosynthesis, a fertilization effect due to an increased  
 ambient CO2-concentration in the biosphere \*)

(\* exported for submodel Observer only: \*)

VAR

cDeltaInBios: OutVar; (\* Total change of carbon stored in biosphere  
 (biota and soils) \*)  
 cBiof: OutVar; (\* Fraction (%) of biota from total carbon pools \*)

(\* exported for GHMaster only: \*)

CONST

biosModelDescr = "Terrestrial biosphere";

PROCEDURE ActivateBiosModel;  
 PROCEDURE DeactivateBiosModel;  
 PROCEDURE BiosModelIsActive(): BOOLEAN;

END GHBiosphere.

The model equations of the two submodels are then provided by the implementation modules, which resemble in their structure that of a simple, unstructured model definition program (compare e.g. with sample model *Logistic*). The following two implementation modules for the submodels atmosphere and biosphere illustrate this technique:

```
IMPLEMENTATION MODULE GHAtmosphere;
```

```
(*
```

```
  Implementation and Revisions:
  =====
```

```
  Author   Date       Description
  -----   -
  AF       21/12/93   First implementation
```

```
*)
```

```
FROM DMMathLib IMPORT Ln;
```

```
FROM SimBase IMPORT
```

```
  Model, IntegrationMethod, DeclM, DeclSV, DeclP, RTCType,
  StashFiling, Tabulation, Graphing, DeclMV, SetSimTime,
  NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout,
  StateVar, Derivative, Parameter, AuxVar, InVar,
  MDeclared, RemoveM, GetSV, notDeclaredModel;
```

```
FROM SimMaster IMPORT CurrentTime;
```

```
FROM GHBiosphere IMPORT prodBio, Q10SA, Q10AB, decsOM, deforestation,
  TEffect, CO2Fertilization;
```

```
VAR
```

```
  atmosM: Model;
  cAtmo: StateVar; (* C content of atmosphere *)
  cAtmoDot: Derivative;
  ffB: StateVar; (* Annual fossil fuel burning *)
  ffBDot: Derivative;
```

```
  releaseBySoil: InVar; (* C-input-flux from soils into atmosphere *)
  uptakeByBiota: InVar; (* C-output-flux from atmosphere into biosphere,
  C-fixation by photosynthesis *)
  netGainBios: InVar; (* net gain of C from biosphere *)
```

```
CONST
```

```
  cAtmo00 = 700.0; (* Default initial C content of atmosphere *)
  CO200 = 330.0; (* Default initial CO2-concentration in the atmosphere *)
  deltaT0 = 0.0; (* Initial temperature change *)
  fRemInA0 = 0.45; (* Default fraction of net C-input-flux remaining in
  the atmosphere, i.e. not *)
  deltaT2xCO20 = 3.7; (* default of parameter deltaT2xCO2,
  IPCC 1990, Tab.3.2, p.87 *)
  alpha0 = 2.13; (* default of parameter alpha *)
  ffB0 = 5.0; (* default of initial fossil fuel burning *)
  begffB0 = 2000.0; (* default first year of fossil fuel burning *)
  endffB0 = 2240.0; (* default last year of fossil fuel burning *)
```

```
VAR
```

```
  cAtmo0: Parameter; (* Initial C content of atmosphere *)
  ffBGrR: Parameter; (* Growth rate of fossil fuel burning *)
  begffB: Parameter; (* First year of fossil fuel burning *)
  endffB: Parameter; (* Last year of fossil fuel burning *)
  deltaT2xCO2: Parameter; (* equilibrium temperature change for
  CO2-doubling (2 x CO2).*)
  alpha: Parameter; (* Factor by which global warming is increased
  if not only CO2 but all other green house
  gases are considered to have also an
  effect (hereby assuming a constant ratio
  between all greenhouse gases) *)
  sensT: Parameter; (* sensitivity of temperature to CO2-doubling *)
```

```

PROCEDURE Initialize;
BEGIN
  IF MDeclared(atmosM) THEN
    GetSV(atmosM,cAtmo,cAtmo0);
    END(*IF*);
    sensT := deltaT2xC02/Ln(2.0);
  END Initialize;

PROCEDURE Input;
BEGIN
  releaseBySoil := TEffect(Q10SA,deltaT)*decSOM;
  uptakeByBiota := prodBio*TEffect(Q10AB,deltaT)*CO2Fertilization(CO2);
  netGainBios := releaseBySoil - uptakeByBiota + deforestation;
END Input;

PROCEDURE EffectOfFossilFuelBurning(t: REAL; x: AuxVar): Derivative;
BEGIN
  IF (t>=begffb) AND (t<endffb) THEN RETURN x ELSE RETURN 0.0 END;
END EffectOfFossilFuelBurning;

PROCEDURE Dynamic;
BEGIN
  ffBDot := EffectOfFossilFuelBurning(CurrentTime(), ffBGrR * ffB );
  cAtmoDot := (cInAnthros + netGainBios) * fRemInA;
END Dynamic;

PROCEDURE Output;
BEGIN
  cInAnthros := EffectOfFossilFuelBurning(CurrentTime(), ffB );
  cDeltaInAtmos := cAtmo-cAtmo0;
  CO2 := cAtmo * CO20/cAtmo0;
  deltaT:= sensT*Ln(1.0+ (alpha*(CO2-CO20)/CO20) );
END Output;

PROCEDURE ModelObjects;
BEGIN
  (* state variables *)
  DeclSV(cAtmo, cAtmoDot, cAtmo0, 100.0, MAX(REAL),
    'C content of atmosphere', 'cAtmo', 'Gt C');
  DeclSV(ffB, ffBDot, ffB0, 0.0, 20.0,
    'Fossil fuel burning', 'ffB', 'Gt C/a');

  DeclMV(cAtmo,650.0,20000.0,
    'C content of atmosphere', 'cAtmo', 'Gt C',
    notOnFile, writeInTable, notInGraph);
  DeclMV(cAtmoDot,-100.0,100.0,
    'Change in C content of the atmosphere', 'cAtmoDot', 'Gt C/a',
    notOnFile, notInTable, notInGraph);

  (* input variables *)
  DeclMV(releaseBySoil, 250.0, 1000.0,
    'C-flux from soils to atmosphere', 'releaseBySoil', 'Gt C/a',
    notOnFile, notInTable, notInGraph);
  DeclMV(uptakeByBiota, 250.0, 1000.0,
    'C-flux from atmosphere to biosphere', 'uptakeByBiota', 'Gt C/a',
    notOnFile, notInTable, notInGraph);

  (* internal auxiliary variables *)
  DeclMV(netGainBios, 250.0, 1000.0,
    'Net gain of C from biosphere', 'netGainBios', 'Gt C/a',
    notOnFile, notInTable, notInGraph);

  (* output variables *)
  DeclMV(CO2, 250.0, 10000.0,
    'CO2-concentration in atmosphere', 'CO2', 'ppm',
    notOnFile, writeInTable, isY);
  DeclMV(cDeltaInAtmos,0.0,15.0,

```

```

    'Change of carbon stored in atmosphere', 'ΔC', 'Gt C',
    notOnFile, notInTable, notInGraph);
DeclMV(deltaT,-1.0,15.0,
'Temperature change (global warming)', 'ΔT', '°C',
notOnFile, writeInTable, isY);

(* parameters also exported *)
DeclP(CO20, CO200, 200.0, 400.0, rtc,
'Initial CO2 conc. in atmosphere', 'CO20', 'ppm');

(* internal parameters *)
DeclP(begffb, begffb0, 1800.0, 2500.0, rtc,
'First year of fossil fuel burning', 'begffb', 'a');
DeclP(endffb, endffb0, 1800.0, 2500.0, rtc,
'Last year of fossil fuel burning', 'endffb', 'a');
DeclP(ffBGrR, 0.01, -0.5, 0.5, rtc,
'Relative growth rate of fossil fuel burning', 'ffBGrR', '/a');
DeclP(deltaT2xCO2, deltaT2xCO20, 0.0, 5.0, rtc,
'Change in temperature by 2xCO2', 'ΔT2xCO2', '°C');
DeclP(fRemInA, fRemInA0, 0.0, 1.0, rtc,
'Fraction of CO2-input-flux remaining in atmosphere', 'fRemInA', '%');
DeclP(alpha, alpha0, 0.0, 3.0, rtc,
'Ratio from other GHG to CO2 on warming', 'alpha', '-');
END ModelObjects;

PROCEDURE AssignDefaultOutputs;
BEGIN
    (* overwrite any eventual changes with defaults to parametrize
    submodel Atmosphere *)
    cAtmo0 := cAtmo00;
    cAtmo := cAtmo0;
    CO20 := CO200;
    ffB := ffB0;
    begffb := begffb0;
    endffb := endffb0;
    deltaT := deltaT0;
    fRemInA := fRemInA0;
    deltaT2xCO2 := deltaT2xCO20;
    alpha := alpha0;

    Initialize;
    Output;
END AssignDefaultOutputs;

PROCEDURE ActivateAtmosModel;
BEGIN
    IF NOT MDeclared(atmosM) THEN
        DeclM(atmosM, Heun, Initialize, Input, Output, Dynamic, NoTerminate, ModelObjects,
            "Atmosphere submodel",
            "atmosM", NoAbout);
    END(*IF*);
END ActivateAtmosModel;

PROCEDURE DeactivateAtmosModel;
BEGIN
    IF MDeclared(atmosM) THEN RemoveM(atmosM); AssignDefaultOutputs END(*IF*);
END DeactivateAtmosModel;

PROCEDURE AtmosModelIsActive(): BOOLEAN;
BEGIN
    RETURN MDeclared(atmosM)
END AtmosModelIsActive;

BEGIN
    atmosM := notDeclaredModel;
    AssignDefaultOutputs
END GHAtmosphere.

```

```

IMPLEMENTATION MODULE GHBIosphere;

(*
  Implementation and Revisions:
  =====

  Author   Date       Description
  -----   -
  AF       21/12/93   First implementation

*)

FROM DMMathLib IMPORT Ln;
FROM SimBase IMPORT
  Model, IntegrationMethod, DeclM, DeclSV, DeclP, RTCType,
  StashFiling, Tabulation, Graphing, DeclMV, SetSimTime,
  NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout,
  StateVar, Derivative, Parameter, AuxVar, InVar,
  MDeclared, RemoveM, GetSV, notDeclaredModel;

FROM GHAtmosphere IMPORT CO2, CO20, deltaT;

VAR
  biosM: Model;
  cBio: StateVar; (* C in biomass *)
  cBioDot: Derivative;
  cSOM: StateVar; (* C in Soil Organic Matter (SOM) *)
  cSOMDot: Derivative;

  CO2Amb: InVar; (* Ambient CO2-concentration in biosphere *)
  changeTAMB: InVar; (* Change of ambient temperature in biosphere *)

  litter: AuxVar; (* C-flux Biomass -> Soil *)
  soilResp: AuxVar; (* C-flux Soil -> Atmosphere *)
  NPP: AuxVar; (* C-flux Atmosphere -> Biosphere, i.e. net
  primary production *)

CONST
  cBio00 = 700.0; (* Default initial C in biomass *)
  cSOM00 = 1320.0; (* Default initial C in soils (SOM) *)
  NPP00 = 117.5; (* Default initial net primary production *)
  Q10SA0 = 1.0; (* Default Q10-value for soil respiration,
  C-flux Soils -> Atmosphere *)
  Q10AB0 = 0.3; (* Default Q10-value for photosynthesis,
  C-flux Atmosphere -> Biosphere *)
  defR00 = 2.0; (* Default initial deforestation rate *)

VAR
  cBio0: Parameter; (* Initial C in biomass *)
  cSOM0: Parameter; (* Initial C in soils (SOM) *)
  NPP0: Parameter; (* initial net primary production (NPP) of world
  biota *)

  beta: Parameter; (* CO2 fertilization-parameter *)
  defR: Parameter; (* relative deforestation rate *)
  defR0: Parameter; (* Initial deforestation rate *)

PROCEDURE Initialize;
BEGIN
  IF MDeclared(biosM) THEN
    GetSV(biosM, cBio, cBio0);

```

```

    GetSV(biosM,cSOM,cSOM0);
  END(*IF*);
  defR := defR0 / cBio0;
END Initialize;

PROCEDURE Input;
BEGIN
  CO2Amb:= CO2;
  changeTAmb := deltaT;
END Input;

PROCEDURE TEffect(Q10ij: Parameter; changeTAmb: AuxVar): AuxVar;
BEGIN
  IF MDeclared(biosM) THEN
    RETURN 1.0+(Q10ij/10.0)*changeTAmb
  ELSE
    RETURN 1.0
  END(*IF*);
END TEffect;

PROCEDURE CO2Fertilization(CO2conc: AuxVar): AuxVar;
BEGIN
  IF MDeclared(biosM) THEN
    RETURN 1.0+beta*Ln(CO2conc/CO20)
  ELSE
    RETURN 1.0
  END(*IF*);
END CO2Fertilization;

PROCEDURE Dynamic;
BEGIN
  NPP:= prodBio*TEffect(Q10AB,changeTAmb)*CO2Fertilization(CO2Amb);
  litter:= prodBio*(cBio/cBio0);
  soilResp:= TEffect(Q10SA,changeTAmb)*decSOM;
  cBioDot := NPP - litter - deforestation;
  cSOMDot := litter - soilResp;
END Dynamic;

PROCEDURE Output;
BEGIN
  prodBio := NPP0*(cBio/cBio0);
  decSOM := NPP0*cSOM/cSOM0;
  deforestation := defR * cBio;
  cDeltaInBios := cBio-cBio0 + cSOM-cSOM0;
  cBiof := cBio/(cBio+cSOM);
END Output;

PROCEDURE ModelObjects;
BEGIN
  (* state variables *)
  DeclSV(cBio, cBioDot, cBio0, 100.0, MAX(REAL),
    'C content of biota', 'cBio', 'Gt C');
  DeclSV(cSOM, cSOMDot, cSOM0, 200.0, MAX(REAL),
    'C content of soils (SOM = Soil Organic Matter)', 'cSOM', 'Gt C');

  DeclMV(cBio, 500.0, 2000.0,
    'C content of biota', 'cBio', 'Gt C',
    notOnFile, writeInTable, notInGraph);
  DeclMV(cBioDot, 0.0, 4.0,
    'Change in C content of biota', 'cBioDot', 'Gt C/a',
    notOnFile, notInTable, notInGraph);
  DeclMV(cSOM, 500.0, 2000.0,
    'C content of soils (SOM = Soil Organic Matter)', 'cSOM', 'Gt C',
    notOnFile, writeInTable, notInGraph);
  DeclMV(cSOMDot, -0.5, 0.5,
    'Change in C content of soils (SOM)', 'cSOMDot', 'Gt C/a',
    notOnFile, notInTable, notInGraph);

```

```

(* input variables *)
DeclMV(CO2Amb, 250.0, 1000.0,
  'Ambient CO2-conc. in biosphere', 'CO2Amb', 'ppm',
  notOnFile, notInTable, notInGraph);
DeclMV(changeTAmb, 250.0, 1000.0,
  'Change of ambient T in biosphere', 'ΔTamb', '°C',
  notOnFile, notInTable, notInGraph);

(* internal auxiliary variables *)
DeclMV(NPP, 110.0, 160.0,
  'Net Primary Production (net photosynthesis)', 'NPP', 'Gt C/a',
  notOnFile, notInTable, notInGraph);
DeclMV(soilResp, 110.0, 160.0,
  'Soil respiration', 'soilResp', 'Gt C/a',
  notOnFile, notInTable, notInGraph);

(* output variables *)
DeclMV(decSOM, 110.0, 160.0,
  'Decaying Soil Organic Matter (SOM)', 'decSOM', 'Gt C',
  notOnFile, notInTable, notInGraph);

(* parameters also exported *)
DeclP(NPP0, NPP00, 0.0, 200.0, rtc,
  'Initial Net Primary Production', 'NPP0', 'Gt C/a');
DeclP(Q10AB, Q10AB0, 0.0, 0.5, rtc,
  'Q10-value for photosynthesis', 'Q10AB', '/°C');
DeclP(Q10SA, Q10SA0, 0.0, 5.0, rtc,
  'Q10-value for soil respiration', 'Q10SA', '/°C');

(* internal parameters *)
DeclP(beta, 0.3, 0.0, 2.0, rtc,
  'CO2-fertilization parameter', 'beta', '-');
DeclP(defR0, defR00, 0.0, 100.0, rtc,
  'Initial deforestation rate', 'defR0', 'Gt C/a');
END ModelObjects;

PROCEDURE AssignDefaultOutputs;
BEGIN
  (* overwrite any eventual changes with defaults to parametrize
  submodel Biosphere *)
  cBio0 := cBio00;
  cSOM0 := cSOM00;
  NPP0 := NPP00;
  cBio := cBio0;
  cSOM := cSOM0;
  Q10SA := Q10SA0;
  Q10AB := Q10AB0;
  defR0 := defR00;
  Initialize;
  Output;
END AssignDefaultOutputs;

PROCEDURE ActivateBiosModel;
BEGIN
  IF NOT MDeclared(biosM) THEN
    DeclM(biosM, Heun, Initialize, Input, Output, Dynamic, NoTerminate, ModelObjects,
      "Biosphere submodel", "biosM", NoAbout);
  END(*IF*);
END ActivateBiosModel;

PROCEDURE DeactivateBiosModel;
BEGIN
  IF MDeclared(biosM) THEN RemoveM(biosM); AssignDefaultOutputs END(*IF*);
END DeactivateBiosModel;

PROCEDURE BiosModelIsActive(): BOOLEAN;

```



```

BEGIN
  RETURN MDeclared(biosM)
END BiosModelIsActive;

BEGIN
  biosM := notDeclaredModel;
  AssignDefaultOutputs
END GHBiosphere.

```

The interface of the submodel *GHObserver* resembles that of any other submodel except that it does not export any outputs; it only has to export the procedures which allow *GHMaster* to announce the submodel to the auxiliary module *StructModAux*.

```

DEFINITION MODULE GHObserver;

(*****

Module GHObserver      (Version 1.0)

    Copyright (c) 1993 by Andreas Fischlin and Swiss
    Federal Institute of Technology Zürich ETHZ

    Purpose Observes overall system behavior of the structured
    model Green-House

    Remark This module is a submodel of the structured
    model Green-House (GH)

    Programming

        o Design and Implementation
          A. Fischlin      3/1/94

    Systems Ecology
    Institute of Terrestrial Ecology
    Department of Environmental Sciences
    Swiss Federal Institute of Technology Zurich ETHZ
    Grabenstr. 3
    CH-8952 Schlieren/Zurich
    Switzerland

    Last revision of definition:  3/1/94  AF

***** )

(* exported for GHMaster only: *)

CONST
  obsModelDescr = "Observer";

PROCEDURE ActivateObsModel;
PROCEDURE DeactivateObsModel;
PROCEDURE ObsModelIsActive(): BOOLEAN;

END GHObserver.

```

The implementation of *GHObserver* is again similar to that of any other model definition program, since it does not only collect data from the other submodels but does also integrate some of these inputs to compute the total fossil fuels burnt.

```

IMPLEMENTATION MODULE GHObserver;

( *
  Implementation and Revisions:
  =====

  Author   Date       Description
  -----  ----       -
  AF       3/1/94     First implementation

*)

FROM DMConversions IMPORT RealToString, RealFormat;
FROM DMStrings IMPORT Concatenate, Append;

FROM SimBase IMPORT
  Model, IntegrationMethod, DeclM, DeclSV, DeclP, RTCType,
  StashFiling, Tabulation, Graphing, DeclMV, SetSimTime,
  NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout, NoDynamic,
  StateVar, Derivative, Parameter, AuxVar,
  MDeclared, RemoveM, GetSV, Message, notDeclaredModel;
FROM SimMaster IMPORT RunSimEnvironment;

FROM GHAtmosphere IMPORT deltaT, CO2, cInAnthros, cDeltaInAtmos, fRemInA;

FROM GHBiosphere IMPORT cDeltaInBios, cBiof;

VAR
  obsM: Model;
  cffBTot: StateVar; (* Total fossil fuel burnt *)
  cffBTotDot: Derivative;

PROCEDURE Dynamic;
BEGIN
  cffBTotDot := cInAnthros;
END Dynamic;

PROCEDURE Terminate;
  PROCEDURE MakeMsgForX(descr: ARRAY OF CHAR; x: REAL; unit: ARRAY OF CHAR);
    VAR msg: ARRAY [0..127] OF CHAR;
    BEGIN (*MakeMsgForX*)
      RealToString(x,msg,0,3,FixedFormat);
      Concatenate(descr,msg,msg); Append(msg,unit);
      Message(msg);
    END MakeMsgForX;
  BEGIN (*Terminate*)
    MakeMsgForX("Global warming = ",deltaT," [°C]");
    MakeMsgForX("CO2-concentration = ",CO2," [ppm]");
    MakeMsgForX("∑ fossil fuel burnt = ",cffBTot," [Gt]");
    MakeMsgForX("ΔC in atmosphere = ",cDeltaInAtmos," [Gt]");
    MakeMsgForX("ΔC in oceans = ",cDeltaInAtmos/fRemInA*(1.0-fRemInA)," [Gt]");
    MakeMsgForX("ΔC in biosphere = ",cDeltaInBios," [Gt]");
    MakeMsgForX(" - hereof in biota = ",cDeltaInBios*cBiof," [Gt]");
    MakeMsgForX(" - hereof in soils = ",cDeltaInBios*(1.0-cBiof)," [Gt]");
  END Terminate;

PROCEDURE ModelObjects;
BEGIN
  DeclSV(cffBTot, cffBTotDot, 0.0, 0.0, 0.0,
    'Total fossil fuel burnt', 'cffBTot', 'Gt C');
  DeclMV(cffBTot,1000.0,2000.0,
    'Total fossil fuel burnt', 'cffBTot', 'Gt C',
    notOnFile, writeInTable, notInGraph);

```

```

END ModelObjects;

PROCEDURE ActivateObsModel;
BEGIN
  IF NOT MDeclared(obsM) THEN
    DeclM(obsM, Heun, NoInitialize, NoInput, NoOutput, Dynamic, Terminate, ModelObjects,
      "Observer submodel", "obsM", NoAbout);
  END(*IF*);
END ActivateObsModel;

PROCEDURE DeactivateObsModel;
BEGIN
  IF MDeclared(obsM) THEN RemoveM(obsM) END(*IF*);
END DeactivateObsModel;

PROCEDURE ObsModelIsActive(): BOOLEAN;
BEGIN
  RETURN MDeclared(obsM);
END ObsModelIsActive;

BEGIN
  obsM := notDeclaredModel;
END GHobserver.

```

This technique of module modeling offers many advantages, such as discarding or reactivation of modules depending on the current needs, e.g. with or without the observer submodel, as well as to expand the submodels, e.g. by another sphere like the oceans.

In particular note that it is also possible even to deactivate a submodel which produces outputs needed as input by another submodel. The removed submodel will then lose its dynamic character, but still provide an output, since an output variable does not cease to exist only because the submodel has been removed from ModelWorks's model base and may be freely used by any other submodel still active. In order to avoid artefact outputs, submodels should be implemented such, that they produce a constant output if the corresponding submodel is no longer active. This behavior can then be physically interpreted as a parametrization of the submodels dynamics. The sample model GreenHouse has exactly been implemented that way: The procedures *AssignDefaultOutputs* in both submodels *GHA*tmosphere and *GH*Biosphere serve exactly this purpose and define the parametrized submodel when it is not dynamically active.

## A.5 MIXED TYPE STRUCTURED MODELS

### A.5.1 Mixing Continuous (DESS) and Discrete Time Models (SQM)

The following listing defines a model demonstrating the mixing of a continuous time submodel with a discrete time submodel. Both models form together a structured model of mixed type (see also in the manual part II *Theory* in the section *Model formalisms* especially the subsection *Structured models (Coupling of submodels)*):

```

MODULE Combined;

( ***** )
( *
  Structured model built from a continuous and discrete time submodel
  The continuous time submodel consists of a simple linear differential
  equation whereby its parameter depends on an input which has been
  coupled with the output from the discrete time submodel. The discrete
  time submodel contains a simple step function. Every submodel is
  modelled as a local module.

  af    29/Mai/1988
                                           *)
( ***** )

IMPORT SimMaster;
FROM SimBase  IMPORT  SetSimTime, SetMonInterval;
IMPORT SimBase;
FROM SimMaster  IMPORT  RunSimEnvironment;

MODULE SubModDisc; ( ***** )

  FROM SimBase  IMPORT  DeclM, IntegrationMethod, DeclSV, StashFiling,
    Tabulation, Graphing, DeclMV, DeclP, RTCType,
    Model, SetSimTime, SetMonInterval,
    NoInitialize, NoInput, NoTerminate, NoAbout,
    StateVar, NewState, Parameter, AuxVar;

  EXPORT DeclSubModDisc, y;

  VAR
    discM:  Model;
    step:   StateVar;
    newStep: NewState;
    a, flip: Parameter;
    y:      AuxVar;

  PROCEDURE Dd;
  BEGIN
    newStep:= flip*step;
  END Dd;

  PROCEDURE Od;
  BEGIN
    y:= a*step;
  END Od;

  PROCEDURE ModelObjectsDisc;

```

```

BEGIN
  DeclSV(step, newStep,1.0, -1.0E3, 1.0E3,
    "Step of discrete time submodel", "Step", "-");

  DeclMV(step, -5.0, 2.0,
    "Step of discrete time submodel",
    "Step", "-", notOnFile, writeInTable, isY);

  DeclP(a, 1.0, -100.0, 100.0, rtc,
    "Amplitude of step function", "a", "---");

  DeclP(flip, -1.0, -1.0, 1.0, rtc,
    "Factor to reverse sign of step function", "f", "---");
END ModelObjectsDisc;

PROCEDURE DeclSubModDisc;
BEGIN
  DeclM(discM, discreteTime, NoInitialize, NoInput, Od, Dd,
    NoTerminate, ModelObjectsDisc,
    "Discrete time submodel",
    "DiscSubMod", NoAbout);
END DeclSubModDisc;

END SubModDisc; (*****

MODULE SubModCont; (*****

  FROM SimBase   IMPORT  DeclM, IntegrationMethod,DeclSV, StashFiling,
    Tabulation, Graphing, DeclMV, DeclP, RTCType,
    Model, SetSimTime, SetMonInterval,
    NoInitialize, NoOutput, NoTerminate, NoAbout,
    StateVar, Derivative, Parameter, AuxVar;

  IMPORT y;
  EXPORT DeclSubModCont;

  VAR
    contM: Model;
    x:     StateVar;
    xDot: Derivative;
    r:     Parameter;
    u:     AuxVar;

  PROCEDURE Ic;
  BEGIN
    u:= y;
  END Ic;

  PROCEDURE Dc;
  BEGIN
    xDot:= r*u*x;
  END Dc;

  PROCEDURE ModelObjectsCont;
  BEGIN
    DeclSV(x, xDot,1.0, -1.0E3, 1.0E3,
      "State variable of continuous time submodel", "x", "-");

    DeclMV(x, 0.0, 5.0,
      "State variable of continuous time submodel", "x", "-",
      notOnFile, writeInTable, isY);

    DeclP(r, 1.0, -100.0, 100.0, rtc,
      "Intrinsic rate of change for continous time submodel",

```

```
    "r", "time^-1");  
END ModelObjectsCont;
```

```
PROCEDURE DeclSubModCont;  
BEGIN  
    DeclM(contM, Euler, NoInitialize, Ic, NoOutput, Dc,  
        NoTerminate, ModelObjectsCont,  
        "Continuous time submodel",  
        "ContSubMod", NoAbout);  
END DeclSubModCont;
```

```
END SubModCont; (*****)
```

```
PROCEDURE StructuredModelDef;  
BEGIN  
    DeclSubModCont; DeclSubModDisc;  
    SetSimTime(0.0,10.0); SetMonInterval(0.25);  
END StructuredModelDef;
```

```
BEGIN  
    RunSimEnvironment(StructuredModelDef);  
END Combined
```

## A.5.2 Mixing a Discrete Event System (DEVS) With a Continuous Time Model (DESS) - CarPollution

### A.5.2.1 The Discrete Event System - Traffic(DEVS)

DEFINITION MODULE CPTraffic;

(\*\*\*\*\*)

Module CPTraffic (Version 1.0)

Copyright (c) 1993 by Andreas Fischlin and Swiss  
Federal Institute of Technology Zürich ETHZ

Purpose Submodel modeling the dynamics of cars

Remarks This module is used by the ModelWorks research  
sample model CarPollution (CP).

Programming

o Design and Implementation  
A. Fischlin 15/12/93

Systems Ecology  
Institute of Terrestrial Ecology  
Department of Environmental Sciences  
Swiss Federal Institute of Technology Zurich ETHZ  
Grabenstr. 3  
CH-8952 Schlieren/Zurich  
Switzerland

Last revision of definition: 15/12/93 AF

(\*\*\*\*\*)

PROCEDURE ActivateTrafficModel;  
PROCEDURE DeactivateTrafficModel;  
PROCEDURE TrafficModelIsActive(): BOOLEAN;

END CPTraffic.

IMPLEMENTATION MODULE CPTraffic;

(\*

Implementation and Revisions:  
=====

Author	Date	Description
-----	----	-----
AF	15/12/93	First implementation (MacMETH_V3.2.1)
dg	25/04/96	Cleaned up for PC compatibility

\*)

FROM DMConversions IMPORT IntToString;

## ModelWorks 2.2 - Appendix (Sample Models)

```

FROM DMStrings      IMPORT AssignString, Append;
FROM DMMessages     IMPORT Warn;

FROM SimBase IMPORT
  Model, DeclP, RTCType, MDeclared, RemoveM, notDeclaredModel,
  StashFiling, Tabulation, Graphing, DeclMV, SetSimTime, NoInput,
  NoOutput, NoTerminate, NoAbout, Parameter, Message,
  DoNothing, ClearTable, InstallClientMonitoring;
FROM SimEvents IMPORT
  nilTransaction, StateTransition, ScheduleEvent,
  DeclDEVM;
FROM SimMaster IMPORT
  CurrentTime;

FROM RandGen IMPORT U, ResetSeeds, Randomize;
FROM RandGen0 IMPORT InstallU0, NegExpP;
FROM Queues IMPORT
  EmptyFIFOQueue, FileIntoFIFOQueue, FIFOQueueLength,
  Take1stFromFIFOQueue, FirstInFIFOQueue, IsFIFOQueueFull;

FROM CPObjects IMPORT TrafficLight, crossRoad, VehicleKind,
  Vehicle, RecognizeVehicle, ForgetVehicle, traffic;
FROM CPCrossRoad IMPORT EnableAnimation, DisableAnimation,
  ShowCrossRoad, ClearCrossRoad, AnimateTrafficLight,
  AnimateArrivingVehicle, AnimatePassingVehicle,
  AnimateLeavingVehicle, AnimateQueueAdvancement;

CONST
  (* EventClasses: *)
  arrival = 1;
  leaving = 2;
  switchLight = 3;

VAR
  trafficM: Model;
  randomize,
  animate,
  stepWise: Parameter;

PROCEDURE ReportEvent( txt1: ARRAY OF CHAR; v: Vehicle; txt2: ARRAY OF CHAR;
                      writeOnFile: BOOLEAN );
  VAR mssg: ARRAY [0..63] OF CHAR; istr: ARRAY [0..7] OF CHAR;
BEGIN
  AssignString(txt1,mssg);
  IF v<>NIL THEN
    CASE v^.kind OF
      | truck   : Append(mssg,"Truck ");
      | car     : Append(mssg,"Car ");
    END(*CASE*);
    IntToString(v^.licensePlate,istr,0); Append(mssg,istr);
  END(*IF*);
  Append(mssg,txt2);
  Message(mssg);
  IF stepWise>0.0 THEN Warn("ReportEvent:",mssg,"") END(*IF*);
END ReportEvent;

PROCEDURE VehicleArrival( ta: Transaction );
BEGIN
  ta := RecognizeVehicle();
  IF (crossRoad.trafficLight=red) THEN
    IF NOT IsFIFOQueueFull(crossRoad.fifoQ) THEN

```



```

    FileIntoFIFOQueue(crossRoad.fifoQ,ta);
    AnimateArrivingVehicle(ta);
    ReportEvent("Red: ",ta," stops to join queue", TRUE);
ELSE
    ReportEvent("Red: ",ta," arrives, Queue overflow! cross road blocked", FALSE);
END(*IF*);
crossRoad.qLe := FLOAT(FIFOQueueLength(crossRoad.fifoQ));
ELSIF (FIFOQueueLength(crossRoad.fifoQ)>0) THEN
    IF NOT IsFIFOQueueFull(crossRoad.fifoQ) THEN
        FileIntoFIFOQueue(crossRoad.fifoQ,ta);
        AnimateArrivingVehicle(ta);
        ReportEvent("Green: ",ta," arrives, but cross road blocked", TRUE);
    ELSE
        ReportEvent("Green: ",ta," arrives, Queue overflow! cross road blocked", FALSE);
    END(*IF*);
    crossRoad.qLe := FLOAT(FIFOQueueLength(crossRoad.fifoQ));
ELSE
    AnimatePassingVehicle(ta);
    ReportEvent("Green: ",ta," passes cross road", FALSE);
    ForgetVehicle(ta);
END(*IF*);
ScheduleEvent(arrival,NegExpP(traffic.muNrVeh),nilTransaction);
END VehicleArrival;

PROCEDURE VehicleLeave( ta: Transaction );
    VAR v: Vehicle;
BEGIN
    IF (FIFOQueueLength(crossRoad.fifoQ)>0) AND (crossRoad.trafficLight=green) THEN
        (* there is at least a vehicle waiting *)
        ta := Take1stFromFIFOQueue(crossRoad.fifoQ);
        ReportEvent("",ta," starts engine and leaves", TRUE);
        AnimateLeavingVehicle(ta);
        ForgetVehicle(ta);
        AnimateQueueAdvancement;
        IF FIFOQueueLength(crossRoad.fifoQ)>0 THEN
            (* there are some more vehicles waiting *)
            v := FirstInFIFOQueue(crossRoad.fifoQ);
            ScheduleEvent(leaving,NegExpP(1.0/traffic.TmuSE[v^.kind]),nilTransaction);
        END(*IF*);
        crossRoad.qLe := FLOAT(FIFOQueueLength(crossRoad.fifoQ));
    END(*IF*);
END VehicleLeave;

PROCEDURE SwitchTrafficLight( ta: Transaction );
    VAR v: Vehicle;
BEGIN
    IF crossRoad.trafficLight=red THEN
        ReportEvent("Switching from red to green",nilTransaction,"", FALSE);
        AnimateTrafficLight(green);
        crossRoad.trafficLight := green;
        IF crossRoad.qLe>0.0 THEN
            IF FIFOQueueLength(crossRoad.fifoQ)<=0 THEN
                Warn("SwitchTrafficLight: Attempt to schedule nonexisting transaction","", "");
            END(*IF*);
            ReportEvent("Post: ",nilTransaction," Scheduling VehicleLeave", FALSE);
            v := FirstInFIFOQueue(crossRoad.fifoQ);
            ScheduleEvent(leaving,NegExpP(1.0/traffic.TmuSE[v^.kind]),nilTransaction);
            crossRoad.qLe := FLOAT(FIFOQueueLength(crossRoad.fifoQ));
        END(*IF*);
    ELSIF crossRoad.trafficLight=green THEN
        ReportEvent("Switching from green to red",nilTransaction,"", FALSE);
        AnimateTrafficLight(red);
        crossRoad.trafficLight := red;
    END(*IF*);
    ScheduleEvent(switchLight,crossRoad.Tls,nilTransaction);
END SwitchTrafficLight;

```

```

PROCEDURE Initialize;
BEGIN
  IF randomize<=0.0 THEN ResetSeeds END(*IF*);
  EmptyFIFOQueue(crossRoad.fifoQ);
  crossRoad.qLe := 0.0;
  traffic.vehicleNr := 0;
  crossRoad.trafficLight := red;
  ScheduleEvent(arrival,CurrentTime(),nilTransaction);
  ScheduleEvent(switchLight,CurrentTime(),nilTransaction);
  ClearTable;
  IF animate=0.0 THEN DisableAnimation ELSE EnableAnimation END;
  ClearCrossRoad;
END Initialize;

PROCEDURE Terminate;
BEGIN
  IF randomize>0.0 THEN Randomize END(*IF*);
END Terminate;

PROCEDURE TrafficModelObjects;
  VAR v: VehicleKind;
BEGIN
  DeclMV(crossRoad.qLe, 0.0,10.0,
    "Cars waiting in queue before red light", "qLe", "#",
    notOnFile, writeInTable, isY);

  DeclP(traffic.truckFrac, 0.2, 0.0, 1.0, rtc,
    "Fraction of trucks among all vehicles", "truckFrac", "%");

  DeclP(crossRoad.Tls, 0.5, 0.0, 10.0, rtc,
    "Traffic control light switch time", "Tls", "hour");
  FOR v:= fstVK TO lstVK DO
    IF v=truck THEN
      DeclP(traffic.TmuSE[v], 0.4, 0.0, 100.0/60.00, rtc,
        "Mean time needed to start truck engine", "TmuSEtruck", "hour");
    ELSE
      DeclP(traffic.TmuSE[v], 0.2, 0.0, 100.0/60.00, rtc,
        "Mean time needed to start non-truck engine", "TmuSE", "hour");
    END(*IF*);
  END(*FOR*);
  DeclP(traffic.muNrVeh, 2.0, 0.0, 10.0, rtc,
    "Mean # vehicles arriving per Δt", "muNrVeh", "hour^-1");
  DeclP(traffic.rhPeak, 0.0, 0.0, 10.0, rtc,
    "Rush hour peak amplitude", "rhPeak", "");

  DeclP(randomize, 1.0, 0.0, 1.0, noRtc,
    "Randomize (TRUE=1.0/FALSE=0.0)", "randomize", "");
  DeclP(animate, 1.0, 0.0, 1.0, noRtc,
    "Animate (TRUE=1.0/FALSE=0.0)", "animate", "");
  DeclP(stepWise, 0.0, 0.0, 1.0, rtc,
    "Step through simulation (TRUE=1.0/FALSE=0.0)", "stepWise", "");

END TrafficModelObjects;

PROCEDURE ActivateTrafficModel;
  VAR stf: ARRAY [arrival..switchLight] OF StateTransition;
BEGIN
  IF NOT MDeclared(trafficM) THEN
    stf[arrival].ec := arrival; stf[arrival].fct := VehicleArrival;
    stf[leaving].ec := leaving; stf[leaving].fct := VehicleLeave;
    stf[switchLight].ec := switchLight; stf[switchLight].fct := SwitchTrafficLight;
    DeclDEVM(trafficM, Initialize, NoInput, NoOutput, stf, Terminate,

```

```

        TrafficModelObjects, "Traffic at a crossroad", "trafficM", NoAbout);
    SetSimTime(0.0,30.0);
    InstallClientMonitoring(ShowCrossRoad, DoNothing, DoNothing);
END(*IF*);
END ActivateTrafficModel;

PROCEDURE DeactivateTrafficModel;
BEGIN
    IF MDeclared(trafficM) THEN Initialize; RemoveM(trafficM) END(*IF*);
END DeactivateTrafficModel;

PROCEDURE TrafficModelIsActive(): BOOLEAN;
BEGIN
    RETURN MDeclared(trafficM)
END TrafficModelIsActive;

BEGIN
    trafficM := notDeclaredModel;
    InstallUO(U);
END CPTraffic.

```

### A.5.2.2 The Crossroad and the Traffic

```

DEFINITION MODULE CPCrossRoad;

(*****

Module CPCrossRoad      (Version 1.0)

    Copyright (c) 1992 by Andreas Fischlin and Swiss
    Federal Institute of Technology Zürich ETHZ

    Purpose Algorithms used to simulate
    and animate traffic jams at a cross road.

    Remarks This module is used by the ModelWorks research
    sample model CarPollution (CP).

    Programming

        o Design
          A. Fischlin      17/Mar/93

        o Implementation
          A. Fischlin      17/Mar/93

    Swiss Federal Institute of Technology Zurich ETHZ
    CH-8092 Zurich
    Switzerland

    Last revision of definition: 16/Sep/93  AF

*****
)

FROM CPObjects IMPORT TrafficLight, Vehicle;

PROCEDURE EnableAnimation;      (* default *)
PROCEDURE DisableAnimation;    (* after calling DisableAnimation, any call
                                to one of the subsequent procedures will
                                have no effect and it closes also the animation
                                window *)

```

ModelWorks 2.2 - Appendix (Sample Models)

```
PROCEDURE ShowCrossRoad;
PROCEDURE ClearCrossRoad;

PROCEDURE AnimateTrafficLight(newTL: TrafficLight);
PROCEDURE AnimateArrivingVehicle(v: Vehicle);
PROCEDURE AnimatePassingVehicle(v: Vehicle);
PROCEDURE AnimateLeavingVehicle(v: Vehicle);
PROCEDURE AnimateQueueAdvancement;
```

END CPCrossRoad.

DEFINITION MODULE CPObjects;

(\*\*\*\*\*)

Module CPObjects (Version 1.0)

Copyright (c) 1992 by Andreas Fischlin and Swiss  
Federal Institute of Technology Zürich ETHZ

Version written for:  
MacMETH\_V3.2 (1-Pass Modula-2 implementation)

Purpose Data structures used to simulate the air pollution  
caused by traffic jams at a cross road.

Remarks This module is used by the ModelWorks research  
sample model CarPollution (CP).

Programming

- o Design  
A. Fischlin 17/Mar/93
- o Implementation  
A. Fischlin 17/Mar/93

Swiss Federal Institute of Technology Zurich ETHZ  
CH-8092 Zurich  
Switzerland

Last revision of definition: 13/Dec/93 AF

(\*\*\*\*\*)

```
FROM SimBase IMPORT Parameter, AuxVar;
FROM Queues IMPORT FIFOQueue;
```

TYPE

```
TrafficLight = (green, red);
CrossRoad = RECORD
    trafficLight: TrafficLight;
    Tls: Parameter; (* Switching time of traffic light *)
    fifoQ: FIFOQueue; (* FIFO queue of waiting vehicles *)
    qLe: AuxVar; (* Current length of queue of waiting
    vehicles. *)
END;
```

VAR

```
crossRoad: CrossRoad;
```

```

TYPE
  VehicleKind = (car, truck);

CONST
  fstVK = MIN(VehicleKind); lstVK = MAX(VehicleKind);

TYPE
  Vehicle = POINTER TO VehicleDescr;
  VehicleDescr = RECORD
    licensePlate: INTEGER;
    kind: VehicleKind;
  END;

PROCEDURE RecognizeVehicle(): Vehicle;
PROCEDURE ForgetVehicle(v: Vehicle);

TYPE
  Traffic = RECORD
    vehicleNr: INTEGER;
    truckFrac: Parameter; (* fraction of trucks among vehicles *)
    muNrVeh: Parameter; (* Mean number of vehicles arriving at
                          cross road per unit of time *)
    rhPeak: Parameter; (* Amplitude of diurnal fluctuation of
                        muNrVeh relative to annual mean. If this
parameter is
                                0, vehicles arrives at the cross road evenly
                                distributed, i.e. without any rush hours. *)
    TmuSE: ARRAY [fstVK..lstVK] OF Parameter;
                                (* Mean time required by a vehicle
                                to starts its engine and to leave
                                the cross road. *)
  END;

VAR
  traffic: Traffic;

END CPObjects.

```

```
IMPLEMENTATION MODULE CPCrossRoad;
```

```

(*)
  Implementation and Revisions:
  =====

  Author   Date       Description
  -----   ----       -
  AF       17/03/93   First implementation (MacMETH_V3.2)
  af       15/12/93   pictIDexhaust added
  dg       25/04/96   Cleaned up for PC compatibility

```

```
*)
```

```

FROM DMSystem IMPORT
  SuperScreen, MainScreen;
FROM DMwindIO IMPORT
  DisplayPredefinedPicture, GetPredefinedPictureFrame,
  BackgroundWidth, LineTo, BackgroundHeight, EraseContent,
  SelectForOutput, Area, pat, GreyContent, SetPen, WriteString,
  WriteInt, CellHeight, CellWidth, SetWindowFont, WindowFont,
  FontStyle, Color, SetColor, DrawAndFillPoly;

```

```

IMPORT DMWindIO;
FROM DMWindows IMPORT
  RectArea, notExistingWindow, Window, WindowKind, ScrollBars,
  CloseAttr, ZoomAttr, WFFixPoint, WindowFrame, CreateWindow,
  GetWindowFrame, PutOnTop, WindowExists, RemoveWindow,
  AddWindowHandler, WindowHandlers;
FROM DMMaster IMPORT PlayPredefinedMusic;

FROM SimEvents IMPORT Transaction;
FROM SimBase IMPORT MWindowArrangement, SetDefltWindowArrangement;

FROM Queues IMPORT FIFOQueueLength,
  DoForAllInFIFOQueue;
FROM CPObjects IMPORT
  TrafficLight, VehicleKind, Vehicle, crossRoad;

TYPE
  CrossRoadW = RECORD
    w: Window;
    wf: WindowFrame;
    pictTrLght: INTEGER;
    pictID: ARRAY [fstVK..lstVK] OF INTEGER;
    pictIDexhaust: INTEGER;
    tlr,bmr, groundr: RectArea;
    vr: ARRAY [fstVK..lstVK] OF RectArea;
    exhcr: RectArea;
    stopLn, vehicleW, vehicleH: INTEGER;
  END;

VAR
  crW: CrossRoadW;
  doAnimate: BOOLEAN;
  supScr,spcw,spch: INTEGER;

PROCEDURE EnableAnimation;
BEGIN
  doAnimate := TRUE;
END EnableAnimation;

PROCEDURE DisableAnimation;
BEGIN
  IF doAnimate THEN RemoveWindow(crW.w) END;
  doAnimate := FALSE;
END DisableAnimation;

PROCEDURE NoAnimation(): BOOLEAN;
BEGIN
  IF NOT WindowExists(crW.w) THEN RETURN TRUE END;
  IF doAnimate THEN SelectForOutput(crW.w) (*important side effect!*) END;
  RETURN NOT doAnimate
END NoAnimation;

PROCEDURE RedrawCrossRoad(u: Window);
  VAR yy: INTEGER;
BEGIN
  SelectForOutput(crW.w); (* needed to handle also DM-event redefined *)
  GetWindowFrame(crW.w,crW.wf);
  crW.tlr.x := crW.wf.w-crW.tlr.w;  crW.tlr.y := (9*crW.wf.h DIV 10) -crW.tlr.h;
  yy := crW.tlr.y+crW.tlr.h; SetPen(0,yy); LineTo(crW.wf.w,yy);
  crW.bmr.x := 0; crW.bmr.y := crW.wf.h DIV 2;
  crW.bmr.w := crW.tlr.x+112;  crW.bmr.h := crW.tlr.y+crW.tlr.h-28-crW.bmr.y;
  DisplayPredefinedPicture(' ',crW.pictTrLght,crW.tlr);
  crW.stopLn := crW.bmr.x + crW.bmr.w - crW.wf.w DIV 10;
  crW.groundr := crW.wf;  crW.groundr.x := 0; crW.groundr.y := 0;
  crW.groundr.h := crW.groundr.h DIV 10;

```

```

    Area(crW.groundr,pat[grey]);
    AnimateTrafficLight(crossRoad.trafficLight);
END RedrawCrossRoad;

PROCEDURE ClearCrossRoad;
BEGIN
    IF NoAnimation() THEN RETURN END;
    EraseContent;
    RedrawCrossRoad(crW.w);
END ClearCrossRoad;

PROCEDURE ShowCrossRoad;
BEGIN
    IF doAnimate THEN (* force window creation or front positioning *)
        IF WindowExists(crW.w) THEN
            PutOnTop(crW.w);
        ELSE
            CreateWindow(crW.w, GrowOrShrinkOrDrag, WithoutScrollBars,
                WithCloseBox,WithZoomBox, bottomLeft, crW.wf,
                'Cross road', RedrawCrossRoad);
            SetWindowFont(Geneva,9,FontStyle);
            AddWindowHandler(crW.w, redefined, RedrawCrossRoad, 0);
        END(*IF*);
    END(*IF*);
END ShowCrossRoad;

PROCEDURE AnimateTrafficLight(newTL: TrafficLight);
PROCEDURE DisplayBeam(x0,y0,w,h,n: INTEGER; r: REAL; c: Color; cl: ARRAY OF CHAR);
    VAR i,k: INTEGER; x,y: ARRAY [0..3] OF INTEGER;
        we: ARRAY [0..3] OF BOOLEAN; ec: ARRAY [0..3] OF Color;
BEGIN (*DisplayBeam*)
    FOR k:= 0 TO 3 DO we[k]:= FALSE; ec[k]:= c END;
    i := n;
    REPEAT
        x[0] := x0;   y[0] := y0;
        x[1] := x0-w; y[1] := y0;
        x[2] := x[1]; y[2] := y0-TRUNC(FLOAT(h)*r);
        x[3] := x0;   y[3] := y0-h;
        DEC(c.saturation,10);
        DrawAndFillPoly(4,x,y,we,ec,TRUE(*isFilled*),c,pat[(*VAL*)GreyContent(i)]);
        h := y0-y[2]; DEC(x0,w); DEC(i);
    UNTIL i=0;
    SetColor(c); SetPen(x0+(w DIV 4),y0-13); WriteString(cl); SetColor(DMWindIO.black);
END DisplayBeam;
PROCEDURE EraseBeams;
BEGIN
    Area(crW.bmr,pat[light]);
END EraseBeams;
BEGIN
    IF NoAnimation() THEN RETURN END;
    IF newTL<>crossRoad.trafficLight THEN
        EraseBeams
    END(*IF*);
    IF newTL=red THEN
        DisplayBeam(crW.bmr.x+crW.bmr.w,crW.bmr.y+crW.bmr.h,
            64, 5,4, 1.5, DMWindIO.red, "red");
    ELSIF newTL=green THEN
        DisplayBeam(crW.bmr.x+crW.bmr.w,crW.bmr.y+crW.bmr.h-16,
            64, 5,4, 1.5, DMWindIO.green, "green");
    END(*IF*);
END AnimateTrafficLight;

PROCEDURE DrawVehicle(v: Vehicle; inRect: RectArea);
BEGIN
    IF ODD(v^.licensePlate) THEN
        DisplayPredefinedPicture('',crW.pictID[v^.kind],inRect);
    END;
END;

```

```

ELSE
  DisplayPredefinedPicture('', crW.pictID[v^.kind]+1, inRect);
END(*IF*);
END DrawVehicle;

PROCEDURE CalcVehiclePlace( v: Vehicle; nrInQ: INTEGER; VAR plc: RectArea );
BEGIN
  plc := crW.vr[v^.kind];
  plc.x := crW.stopLn - nrInQ*crW.vehicleW
    + (crW.vehicleW-plc.w) DIV 2;
  plc.y := crW.groundr.y+crW.groundr.h;
END CalcVehiclePlace;

PROCEDURE CalcExhaustionPlace(plc: RectArea; VAR exh: RectArea);
BEGIN
  exh := crW.exhcr;
  exh.x := plc.x-crW.exhcr.w;  exh.y := plc.y+3;
END CalcExhaustionPlace;

PROCEDURE DrawHaltingVehicle(v: Vehicle; atPos: INTEGER);
  VAR plc, exh: RectArea;
BEGIN
  CalcVehiclePlace(v, atPos, plc);
  DrawVehicle(v, plc);
  CalcExhaustionPlace(plc, exh);
  DisplayPredefinedPicture('', crW.pictIDexhaust, exh);
  SetPen(plc.x+plc.w DIV 2-CellWidth(), crW.groundr.y+crW.groundr.h-CellHeight());
  WriteInt(v^.licensePlate, 2);
END DrawHaltingVehicle;

PROCEDURE EraseVehicle(plc: RectArea);
BEGIN
  plc.h := crW.vehicleH;
  DEC(plc.x, crW.exhcr.w); INC(plc.w, crW.exhcr.w);
  Area(plc, pat[light]);
  (*
  SetMode(invert);
  DisplayPredefinedPicture('', crW.pictID[v^.kind], plc);
  SetMode(replace);
  .*)
  DEC(plc.y, 3*CellHeight() DIV 2); plc.h := crW.groundr.h-plc.y;
  Area(plc, pat[grey]);
END EraseVehicle;

VAR
  carPlace: INTEGER;

PROCEDURE Shift1PlaceForward(ta: Transaction);
  VAR plcNew, plcOld: RectArea;
BEGIN
  CalcVehiclePlace(ta, carPlace, plcNew);
  EraseVehicle(plcNew);
  INC(carPlace);
  CalcVehiclePlace(ta, carPlace, plcOld);
  EraseVehicle(plcOld);
  DrawHaltingVehicle(ta, carPlace-1);
END Shift1PlaceForward;

PROCEDURE AnimateLeavingVehicle(v: Vehicle);
  VAR plc, plcOld: RectArea; b: INTEGER;
BEGIN
  IF NoAnimation() THEN RETURN END;
  CalcVehiclePlace(v, 1, plc);
  b := 1;
  plcOld := plc;
  WHILE plc.x < crW.wf.w DO
    INC(plc.x, 20+b);
    DrawVehicle(v, plc);
  
```



```

    IF plc.x>=(plcOld.x+plcOld.w) THEN
        EraseVehicle(plcOld);
        INC(plcOld.x,plcOld.w);
    END(*IF*);
    b := b*2;
END(*WHILE*);
IF plcOld.x<crW.wf.w THEN
    EraseVehicle(plcOld);
    END(*IF*);
END AnimateLeavingVehicle;

PROCEDURE AnimateQueueAdvancement;
BEGIN
    IF NoAnimation() THEN RETURN END;
    carPlace := 1;
    DoForAllInFIFOQueue(crossRoad.fifoQ,Shift1PlaceForward);
END AnimateQueueAdvancement;

PROCEDURE AnimateArrivingVehicle(v: Vehicle);
BEGIN
    IF NoAnimation() THEN RETURN END;
    DrawHaltingVehicle(v,FIFOQueueLength(crossRoad.fifoQ));
END AnimateArrivingVehicle;

PROCEDURE AnimatePassingVehicle(v: Vehicle);
    VAR plc: RectArea;
BEGIN
    IF NoAnimation() THEN RETURN END;
    CalcVehiclePlace(v,1,plc);
    plc.x := 0;
    WHILE plc.x<crW.wf.w DO
        DrawVehicle(v,plc);
        EraseVehicle(plc);
        INC(plc.x,20);
    END(*WHILE*);
END AnimatePassingVehicle;

BEGIN
    crW.w := notExistingWindow;
    crW.pictTrLght := 3132;
    crW.pictID[truck] := 3130; crW.pictID[car] := 3128;
    crW.pictIDexhaust := 3133;
    GetPredefinedPictureFrame('',crW.pictTrLght,crW.tlr);
    GetPredefinedPictureFrame('',crW.pictID[truck],crW.vr[truck]);
    GetPredefinedPictureFrame('',crW.pictID[car],crW.vr[car]);
    crW.vr[car].x := 0; crW.vr[car].y := 0;
    crW.vr[car].w := 70; crW.vr[car].h := 19;
    IF crW.vr[truck].w>crW.vr[car].w THEN
        crW.vehicleW := crW.vr[truck].w
    ELSE
        crW.vehicleW := crW.vr[car].w
    END;
    INC(crW.vehicleW,15);
    IF crW.vr[truck].h>crW.vr[car].h THEN
        crW.vehicleH := crW.vr[truck].h
    ELSE
        crW.vehicleH := crW.vr[car].h
    END;
    INC(crW.vehicleH,10);
    GetPredefinedPictureFrame('',crW.pictIDexhaust,crW.exhcr);
    SetDefltWindowArrangement(tiled);
    SuperScreen(supScr,crW.wf.x,crW.wf.y,spcw,spch,
        crW.wf.h(*dummy*),TRUE(*color priority*));
    crW.wf.h := 200;
    IF supScr=MainScreen() THEN
        crW.wf.w := BackgroundWidth()-8;
        crW.wf.x := (BackgroundWidth()-crW.wf.w) DIV 2;

```

```

    crW.wf.y := (BackgroundHeight()-crW.wf.h) DIV 2;
ELSE
    crW.wf.w := 4*spcw DIV 5;
    crW.wf.x := crW.wf.x + (spcw-crW.wf.w) DIV 2;
    crW.wf.y := crW.wf.y + (spch-crW.wf.h) DIV 2;
END(*IF*);
END CPCrossRoad.

```

IMPLEMENTATION MODULE CPObjects;

```

(*)
    Implementation and Revisions:
    =====

    Author   Date       Description
    -----  ----       -
    AF       3/17/93    First implementation (MacMETH_V3.2)

*)

FROM DMStorage IMPORT Allocate, Deallocate;
FROM DMSystem  IMPORT CurrentDMLLevel, InstallTermProc;
FROM DMMessages IMPORT Warn, Abort;

FROM Queues    IMPORT CreateFIFOQueue;
FROM RandGen   IMPORT U;

TYPE
    VehiclePtr = POINTER TO VehicleItem;
    VehicleItem = RECORD v: Vehicle; next,prev: VehiclePtr END;

VAR
    vroot: VehiclePtr;
    installed: BOOLEAN; loadLev: CARDINAL;

PROCEDURE RecognizeVehicle(): Vehicle;
    VAR vi: VehiclePtr; ok: BOOLEAN;
BEGIN
    ok := FALSE;
    Allocate(vi, SIZE(VehicleItem));
    IF vi<>NIL THEN
        vi^.v := NIL;
        Allocate(vi^.v, SIZE(Vehicle));
        IF vi^.v<>NIL THEN
            ok := TRUE;
            INC(traffic.vehicleNr);
            vi^.v^.licensePlate := traffic.vehicleNr;
            IF U() <= traffic.truckFrac THEN
                vi^.v^.kind := truck;
            ELSE
                vi^.v^.kind := car;
            END(*IF*);
        END(*IF*);
        (* insert at begin *)
        vi^.next := vroot;
        vi^.prev := NIL;
        IF vroot<>NIL THEN vroot^.prev := vi END;
        vroot := vi;
    END(*IF*);
    IF ok THEN
        RETURN vi^.v
    ELSE
        Abort("Can't instantiate more vehicles - insufficient memory","", "")
    END

```

```

    END(*IF*);
END RecognizeVehicle;

PROCEDURE Find(v: Vehicle): VehiclePtr;
  VAR p: VehiclePtr;
BEGIN
  p := vroot;
  WHILE (p<>NIL) AND (p^.v<>v) DO
    p := p^.next
  END(*WHILE*);
  IF (p<>NIL) AND (p^.v=v) THEN RETURN p ELSE RETURN NIL END;
END Find;

PROCEDURE Discard(vi: VehiclePtr); (* assumes vi exists *)
BEGIN
  IF vi=vroot THEN
    vroot := vi^.next;
    vi^.prev := NIL;
  ELSE
    IF vi^.prev<>NIL THEN vi^.prev^.next := vi^.next END;
    IF vi^.next<>NIL THEN vi^.next^.prev := vi^.prev END;
  END(*IF*);
  Deallocate(vi^.v);
  Deallocate(vi);
END Discard;

PROCEDURE ForgetVehicle(v: Vehicle);
  VAR vi: VehiclePtr;
BEGIN
  vi := Find(v);
  IF vi<>NIL THEN Discard(vi) ELSE Warn("Can't forget unknown vehicle","", "") END;
END ForgetVehicle;

PROCEDURE ForgetAllVehicles;
BEGIN
  IF CurrentDMLevel()=loadLev THEN
    WHILE vroot<>NIL DO Discard(vroot) END(*WHILE*);
  END(*IF*);
END ForgetAllVehicles;

BEGIN
  crossRoad.trafficLight := red;
  CreateFIFOQueue(crossRoad.fifoQ,10 (*maxLength* ));
  traffic.vehicleNr := 0;
  vroot := NIL; loadLev := CurrentDMLevel();
  InstallTermProc(ForgetAllVehicles,installed);
END CPObjects.

```

### A.5.2.3 Adding Traffic's Air Pollution - Pollutants (DESS)

```

DEFINITION MODULE CPPollutants;

(*****

Module CPPollutants      (Version 1.0)

    Copyright (c) 1993 by Andreas Fischlin and Swiss
    Federal Institute of Technology Zürich ETHZ

    Purpose Submodel for the dynamics of air pollutants

    Remarks This module is used by the ModelWorks research

```

sample model CarPollution (CP).

Programming

o Design and Implementation  
A. Fischlin 15/12/93

Systems Ecology  
Institute of Terrestrial Ecology  
Department of Environmental Sciences  
Swiss Federal Institute of Technology Zurich ETHZ  
Grabenstr. 3  
CH-8952 Schlieren/Zurich  
Switzerland

Last revision of definition: 15/12/93 AF

\*\*\*\*\*)

PROCEDURE ActivatePollutantsModel;  
PROCEDURE DeactivatePollutantsModel;  
PROCEDURE PollutantsModelIsActive(): BOOLEAN;

END CPPollutants.

IMPLEMENTATION MODULE CPPollutants;

(\*

Implementation and Revisions:  
=====

Author	Date	Description
-----	----	-----

AF	15/12/93	First implementation (MacMETH_V3.2.1)
----	----------	---------------------------------------

\*)

FROM SimBase IMPORT  
Model, DeclM, MDeclared, RemoveM, notDeclaredModel,  
IntegrationMethod, DeclSV, StashFiling, Tabulation, Graphing,  
DeclMV, DeclP, RTCType, NoInitialize, NoInput, NoOutput,  
NoTerminate, NoAbout, SetSimTime, StateVar, Derivative,  
Parameter;

FROM CPObjects IMPORT crossRoad;

VAR  
pollM: Model;  
pollutants: StateVar; pollutantsDot: Derivative;  
emissionRate, decayRate: Parameter;

PROCEDURE DiffEquations;  
BEGIN  
pollutantsDot := emissionRate\*crossRoad.qLe - decayRate\*pollutants;  
END DiffEquations;

PROCEDURE PollutionModelObjects;

```

BEGIN
  DeclSV(pollutants, pollutantsDot,0.0, 0.0, 10000.0,
    "Air pollutants (aggregated)", "pollutants", "mg/m^3");

  DeclMV(pollutants, 0.0,10.0,
    "Air pollutants (aggregated)", "pollutants", "mg/m^3",
    notOnFile, writeInTable, isY);

  DeclP(emissionRate, 0.1, 0.0, 10.0, rtc,
    "Relative emission rate of air pollutants", "emissionRate", "/vehicle/hour");
  DeclP(decayRate, 0.2, 0.0, 10.0, rtc,
    "Decay rate of air pollutants", "decayRate", "/hour");

END PollutionModelObjects;

PROCEDURE ActivatePollutantsModel;
BEGIN
  IF NOT MDeclared(pollM) THEN
    DeclM(pollM, Euler, NoInitialize, NoInput, NoOutput, DiffEquations,
      NoTerminate, PollutionModelObjects, "Air pollution at a crossroad",
      "pollM", NoAbout);
    SetSimTime(0.0,30.0);
  END(*IF*);
END ActivatePollutantsModel;

PROCEDURE DeactivatePollutantsModel;
BEGIN
  IF MDeclared(pollM) THEN RemoveM(pollM) END(*IF*);
END DeactivatePollutantsModel;

PROCEDURE PollutantsModelIsActive(): BOOLEAN;
BEGIN
  RETURN MDeclared(pollM)
END PollutantsModelIsActive;

BEGIN
  pollM := notDeclaredModel;
END CPPollutants.

```

#### A.5.2.4 Putting All Together

```

MODULE CPMaster;

(*
  Module CPMaster

  Purpose: Demonstration of the combination of a DEVS (discrete event
  system) for traffic with a DESS (Differential equation system
  specification) for air pollutants.

  The model simulates the arrival, queue formation in case of
  a red traffic light, and leaving of vehicles at a crossroad (a
  DEVS). Moreover the model simulates also the accumulation
  resp. decay of hereby exhausted air pollutants (DESS). Traffic
  parameters determine the frequencies of trucks and cars, the
  time needed to restart an engine, the switching time of the
  traffic light etc. This allows to study the effect of crossroad
  policies and behavior recommendations for drivers. Pollutant
  parameters determine the production and fate of aggregated air
  pollutants in order to simulate scenarios of different
  regulations for vehicle emissions.

```

Revision history:

=====

Author	Date	Description
-----	-----	-----
AF	03/11/93	First implementation

\*)

(\* Imports from 'Dialog Machine' (DM) \*)

```
FROM DMMenu IMPORT Menu, Command, AccessStatus, Marking,
  InstallMenu, InstallCommand, InstallAliasChar;
FROM DMEnterForms IMPORT FormFrame, WriteLabel,
  CheckBox, UseEntryForm;
```

(\* Imports from ModelWorks (Sim) \*)

```
FROM SimMaster IMPORT RunSimEnvironment;
```

(\* Imports from CPMaster modular model definition (CP) \*)

```
FROM CPTraffic IMPORT ActivateTrafficModel, DeactivateTrafficModel,
  TrafficModelIsActive;
FROM CPPollutants IMPORT ActivatePollutantsModel, DeactivatePollutantsModel,
  PollutantsModelIsActive;
```

VAR

```
  modM: Menu; modActCmd: Command;
```

PROCEDURE Choose;

```
  CONST lm = 6; VAR bf: FormFrame; ok, carCB, pollCB: BOOLEAN; cl: INTEGER;
BEGIN
  cl := 2; WriteLabel(cl,lm-1,"Check models to be activated:"); INC(cl);
  carCB := TrafficModelIsActive();
  pollCB := PollutantsModelIsActive();
  CheckBox(cl,lm,"Traffic sub model (DEVS)",carCB); INC(cl);
  CheckBox(cl,lm,"Pollutants sub model (DESS)",pollCB); INC(cl);
  bf.x:= 0; bf.y:= -1 (*display dialog window in middle of screen*);
  bf.lines:= cl+1; bf.columns:= 50;
  UseEntryForm(bf,ok);
  IF ok THEN
    IF carCB THEN ActivateTrafficModel ELSE DeactivateTrafficModel END;
    IF pollCB THEN ActivatePollutantsModel ELSE DeactivatePollutantsModel END;
  END(*IF*);
END Choose;
```

PROCEDURE InstallMenus;

```
BEGIN
  ActivateTrafficModel; (* by default active *)
  ActivatePollutantsModel; (* by default active *)
  InstallMenu(modM,"Models", enabled);
  InstallCommand(modM, modActCmd,"Activation...", Choose, enabled, unchecked);
  InstallAliasChar(modM, modActCmd,"L");
END InstallMenus;
```

BEGIN

```
  RunSimEnvironment( InstallMenus );
END CPMaster.
```

## A.6 RESEARCH SAMPLE MODELS

The following research sample models demonstrate the typical use of various aspects of ModelWorks in research applications. However, these model definition programs list not the full original source code, but a slightly streamlined one. This should make the underlying principles more visible, than this would be the case for the original source code, somewhat cluttering the essence. Yet all these programs are fully functional.

All these model definition programs use ModelWorks' standard user interface, some extend it by installing additional menus, entry forms, and windows. Most demonstrate the use of structured simulations or experiments and make use of either the "Dialog Machine", ModelWorks' optional client interface, and auxiliary library modules.

### A.6.1 Population Dynamics of Larch Bud Moth - *LBM*

The following program code contains a sample model demonstrating the use of ModelWorks in a research project. This model definition program demonstrates also modular modeling, in particular the use of a so-called parallel model in order to allow the simulationist to compare simulation results with measured data, dynamic setting of curve attributes during simulation runs, and dynamic activation respectively deactivation of models during a simulation session.

The model system is a structured system consisting of two submodels (Fig. A9).

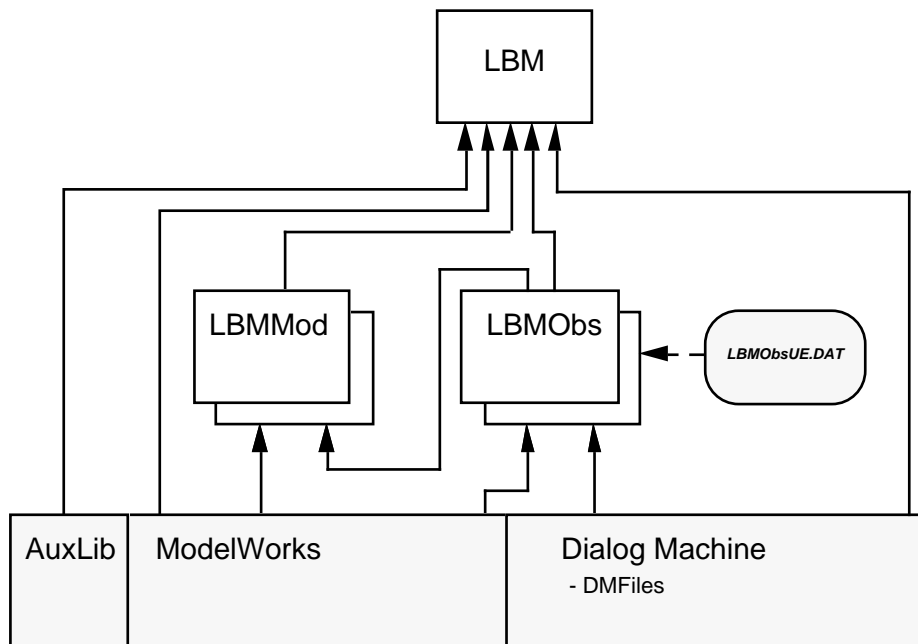


Fig. A9: Module structure of the research sample model.

The first submodel, module *LBMMod*, describes the ecological interaction of the host plant larch *Larix decidua* MILLER with the herbivorous insect larch bud moth *Zeiraphera diniand* GN. (*Lep.*, *Tortricida*) (FISCHLIN, 1982; BALTEUSWEILER & FISCHLIN, 1988). The second submodel, module *LBMObs*, is a parallel model formulated like any other ModelWorks model, except that its dynamic part has only some output equations but no state variables nor dynamic equations. It mimics the real system by reading and outputting field data in function of time

(CELLIER & FISCHLIN, 1980; FISCHLIN, 1991). A master module, the program module *LBM*, combines all modules to a model definition program (Fig. A9).

The next two listings show the definition and the implementation parts of the module *LBMMod* containing the discrete time submodel describing the relationship between the host plant and the insect:

```
DEFINITION MODULE LBMMod;
```

```
(*
```

```
  Purpose      Simulates Larch Bud Moth population dynamics for the
                Upper Engadine valley from 1949 till 1977. Model b:
                local dynamics: larch - larch bud moth interaction
```

```
  Reference    Fischlin 1982, "Analyse eines Wald-Insekten Systemes:
                Der subalpine Lärchen-Arvenwald und der Graue
                Lärchenwickler Zeiraphera diniana Gn. (Lep.,
                Tortricidae)", Diss ETHZ No. 6977.
```

```
  Remark      This program module contains the model which runs
                under the simulation environment ModelWorks V0.5
```

```
  Programming  A.Fischlin, Systems Ecology, ETHZ, Dez. 1986
```

```
*)
```

```
FROM SimBase IMPORT AuxVar;
```

```
VAR
```

```
  yt: AuxVar; (* output: simulated larval density for whole valley *)
  ytLn: AuxVar; (* output: ln of simulated larval density for whole valley *)
```

```
PROCEDURE ActivateLarchLBMMModel;
PROCEDURE DeactivateLarchLBMMModel;
PROCEDURE LarchLBMMModelIsActive(): BOOLEAN;
```

```
END LBMMod.
```

```
IMPLEMENTATION MODULE LBMMod;
```

```
(*
```

```
  Revision history:
  =====
```

Author	Date	Description
-----	----	-----
af	Dez.86	First implementation
af	12/05/90	ModelWorks 2.0 adaptation, now dynamic model activation and de- activation supported
dg	05/12/91	Now imports from DMMathLib instead MathLib

```
*)
```

```
FROM DMMathLib IMPORT Exp, Ln;
FROM SimMaster IMPORT RunSimEnvironment;
```

```
FROM SimBase IMPORT Model, DeclM, MDeclared, RemoveM, notDeclaredModel,
  IntegrationMethod, DeclSV, StashFiling, Tabulation, Graphing,
```



```

DeclMV, DeclP, RTCType, NoInput, NoTerminate, NoAbout,
SetSimTime, SetMonInterval, SetDeflCurveAttrForMV, Stain,
LineStyle, StateVar, NewState, Parameter, AuxVar;

FROM LBMObs IMPORT negLogDelta, yLL, yUL, kmin, kmax (* time domain *);

VAR
  m: Model;
  c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16,c17,nrt: Parameter;
  p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14: REAL;
  rt,et: StateVar;
  rt1,et1: NewState;
  def, springEggs: AuxVar;

PROCEDURE Initialize;

  PROCEDURE Parameters;
  BEGIN
    p1:=c4;
    p2:=c5;
    p3:=-c2*c6*(1.0-c1);
    p4:=c6*(1.0-c1)*(1.0 -c3);
    p5:=c2*c7*c9*c10*(1.0-c1);
    p6:=c9*(1.0-c1)*(c2*c7*c11-c10*(c2*(1.0-c8)+c7*(1.0-c3)));
    p7:=c9*(1.0-c1)*(c10*(1.0-c3)*(1.0-c8)-c11*(c2*(1.0-c8)+c7*(1.0-c3)));
    p8:=c9*c11*(1.0-c1)*(1.0-c3)*(1.0-c8);
    p9:=c12;
    p10:=c13;
    p11:=c14;
    p12:=c15;
    p13:=c16;
    p14:=-c6*c17*nrt;
  END Parameters;

  BEGIN (*Initialize*)
    Parameters;
  END Initialize;

  PROCEDURE Output;
  BEGIN
    yt:= (p3*rt+p4)*et/p14;
    ytLn:= Ln(negLogDelta+yt);
    springEggs:= (1.0 - c1) * et;
  END Output;

  PROCEDURE Dynamic;

  PROCEDURE gmstarv(x1,x2: REAL): REAL;
  BEGIN
    IF x2=0.0 THEN RETURN 0.0 END;
    IF x2>0.0 THEN RETURN Exp(-x1/x2) END;
  END gmstarv;

  PROCEDURE grecr(def,rt: REAL): REAL;
  CONST eps = 0.00001;
  VAR
    zrt: REAL;
  BEGIN (*grecr*)
    IF (def < p12) THEN
      IF (rt >= p9-eps) AND (rt <= p9) (* rt = p9 *) THEN
        RETURN 1.0
      ELSIF rt > p9 THEN
        zrt:= p10+ABS((p11-rt)/(rt-p9));

```

```

    IF zrt > rt-p9 THEN
      RETURN p9/rt
    ELSE (*zrt <= rt-p9*)
      RETURN 1.0-zrt/rt
    END(*IF*);
  ELSE
    (* " --- warning: rt < p9" *)
    HALT
  END(*IF*);
ELSE (*def >= p12*)
  IF def < p13 THEN
    RETURN 1.0+(def-p12)*(p11-rt)/(p13-p12)/rt
  ELSIF (def > p12) (*AND (def >= p13)*) THEN
    RETURN p11/rt
  ELSE (*(def = p12) AND (def >= p13)*)
    HALT
  END(*IF*);
END(*IF*);
END grecr;

BEGIN (*Dynamic*)
  def:= (1.0-gmstarv(p1*rt+p2,p3*rt*et+p4*et))*(p3*rt*et+p4*et)/(p1*rt+p2);
  rt1:=gregr(def,rt)*rt;
  et1:=(1.0-gmstarv(p1*rt+p2,p3*rt*et+p4*et))*
    (p5*rt*rt*rt+p6*rt*rt+p7*rt+p8)*et;
END Dynamic;

PROCEDURE ModelObjects;
BEGIN
  DeclSV(rt, rt1, 15.0, 11.99, 18.5,
    "Raw fiber content (% fresh weight)", "rf", "%");
  DeclSV(et, et1, 4765975.0, 0.0, 1.0E12,
    "Larch bud moth eggs (individuals)", "eggs", "numbers");

  DeclMV(rt, 10.0, 20.0, "Raw fiber content (% fresh weight)", "rf",
    "%", notOnFile, writeInTable,notInGraph);
  SetDefltCurveAttrForMV (m, rt,sapphire,dashSpotted,".");
  DeclMV(springEggs, 0.0, 1.0E12,"Larch bud moth eggs in spring (individuals)",
    "eggs", "lbn", notOnFile, notInTable, notInGraph);
  DeclMV(yt, yLL, yUL,"Larval density (larvae/kg branches)",
    "Y", "lbn/kg", notOnFile, writeInTable, notInGraph);
  SetDefltCurveAttrForMV (m, yt,ruby,unbroken,"");
  DeclMV(ytLn, Ln(negLogDelta), Ln(negLogDelta+yUL),
    "Ln of larval density (larvae/kg branches)",
    "Ln(Y)", "lbn/kg", notOnFile, notInTable, isY);
  SetDefltCurveAttrForMV (m, ytLn,ruby,unbroken,"");
  DeclMV(def, 0.0, 1.0,"Defoliation",
    "def", "", notOnFile, notInTable, notInGraph);
  SetDefltCurveAttrForMV (m, def,emerald,broken,0C);

  DeclP(nrt, 511147.0, 511147.0, 511147.0, noRtc,
    "nrt (number of trees)", "trees", "trees");
  DeclP(c1, 0.5728, 0.4841, 0.6538, noRtc,
    "c1 (egg winter mortality)", "c1", "lbn");
  DeclP(c2, 0.05112, 0.016, 0.087, noRtc,
    "c2 (slope of small larvae mortality vs. rf)", "c2", "/%");
  DeclP(c3, -0.17932, -0.565, 0.206, noRtc,
    "c3 (y-intercept of small larvae mortality vs. rf)", "c3", "");
  DeclP(c4, -2.25933*nrt, -2.4129*nrt, -2.1057*nrt, noRtc,
    "c4 (slope of needle biomass vs. rf)", "c4", "/%");
  DeclP(c5, 67.38939*nrt, 62.8076*nrt, 71.9712*nrt, noRtc,
    "c5 (y-intercept of needle biomass vs. rf)", "c5", "");
  DeclP(c6, 0.005472, 0.0027, 0.0106, noRtc,
    "c6 (food demand of a large larvae)", "c6", "kg/lbn");
  DeclP(c7, 0.124017, 0.1070, 0.1410, noRtc,
    "c7 (slope of large larvae mortality vs. rf)", "c7", "/%");
  DeclP(c8, -1.435284, -1.685, -1.1855, noRtc,

```

```

    "c8 (y-intercept of large larvae mortality vs. rf)", "c8", "");
DeclP(c9, 0.44, 0.363, 0.517, noRtc,
    "c9 (sex ratio)", "c9", "");
DeclP(c10, -18.475457, -24.7217, -12.2294, noRtc,
    "c10 (slope of fecundity vs. rf)", "c10", "lbm/%");
DeclP(c11, 356.72636, 264.9847, 448.4680, noRtc,
    "c11 (y-intercept of fecundity vs. rf)", "c11", "lbm");
DeclP(c12, 11.99, 11.79, 12.19, noRtc,
    "c12 (minimum rf)", "c12", "%");
DeclP(c13, 0.425, 0.4, 0.5, noRtc,
    "c13 (minimum decrement of rf)", "c13", "%");
DeclP(c14, 18.0, 17.5, 18.5, noRtc,
    "c14 (maximum rf)", "c14", "%");
DeclP(c15, 0.4, 0.35, 0.6, noRtc,
    "c15 (defoliation threshold)", "c15", "");
DeclP(c16, 0.8, 0.7, 1.0, noRtc,
    "c16 (defoliation threshold of maximum stress)", "c16", "");
DeclP(c17, 91.3, 91.3, 91.3, noRtc,
    "c17 (branches per tree)", "c17", "kg");

END ModelObjects;

PROCEDURE ActivateLarchLBMMModel;
BEGIN
    IF NOT MDeclared(m) THEN
        DeclM(m, discreteTime, Initialize, NoInput, Output, Dynamic, NoTerminate,
ModelObjects,
            "Larch Bud Moth model b1 V3.0 (Larch-Larch bud moth relationship)",
            "LWMod3 b1", NoAbout);
        END(*IF*);
    END ActivateLarchLBMMModel;

PROCEDURE DeactivateLarchLBMMModel;
BEGIN
    IF MDeclared(m) THEN RemoveM(m) END(*IF*);
    END DeactivateLarchLBMMModel;

PROCEDURE LarchLBMMModelIsActive(): BOOLEAN;
BEGIN
    RETURN MDeclared(m)
    END LarchLBMMModelIsActive;

BEGIN
    m := notDeclaredModel;
    END LBMMMod.

```

The module *LBMObs* provides a parallel submodel of the measured larval densities of the larch bud moth (observations) made in the field while studying the larch bud moth system in the Upper Engadine valley in Switzerland from 1949 till the present (BALTENSWEILER & FISCHLIN, 1988). This allows to compare the observations with simulated values. At the beginning of the simulation session this parallel model simply reads the observations stored in the data file into an array and will assign the measured values during any simulations to a monitoring variable, which the simulationist can display from within the simulation environment.

In case the simulationist should set the global simulation time such that it lies outside the range 1949 and 1988, the values produced by this module are no longer valid. The module has been programmed such that it visualizes missing values in the graph by letting portions of the curve(s) disappear. This is accomplished by setting the curve attribute to invisible as soon as values have become undefined, yet the legend is drawn with the attributes normally used if values are available.

The next three listings show the definition and the implementation parts of the module *LBMObs* which reads the data from the text file *LBMObsUE.DAT*:

```

DEFINITION MODULE LBMObs;

(*
  Module LBMObs

  Purpose Simulates the real larch bud moth system in the
           Upper Engadine Valley as a parallel model.

  Method  Observed larval densities in larvae/kg larch
           branches as sampled from the Upper Engadine Valley
           are simulated by means of a ModelWorks submodel.
           Data from Fischlin, A. 1982. Analyse
           eines Wald-Insekten-Systems: Der subalpine
           Lärchen-Arvenwald und der graue Lärchenwickler
           Zeiraphera diniana Gn. (Lep., Tortricidae).
           Diss. ETH Nr. 6977. Swiss Federal Institute of
           Technology Zürich, Switzerland, 294pp, page 90,
           Table 10 and from Baltensweiler, W. and Fischlin, A.
           1987, The larch bud moth in the European Alps, In
           Berryman, A.A. (ed.), Population Dynamics of Forest-
           Insect Systems, Plenum Press, in print.

  Remark  The data are read from a file only once at model
           declaration and are loaded into memory for subsequent
           usage.
           This program module contains the model which runs
           under the simulation environment ModelWorks V0.5

  Programming A.Fischlin, Systems Ecology, ETHZ, 01/05/87

*)

CONST
  kmin = 1949; (*first year sampled*)
  kmax = 1986; (*last year sampled*)
  limkmax = 1977; (* beyond limkmax yminDash, ymaxDash no longer available *)
  yLL = 0.0; (*minimum used on graph scale for larval densities *)
  yUL = 600.0; (*maximum used on graph scale for larval densities *)
  negLogDelta = 0.01; (*offset used to plot log scale if values <= 0*)

(* The following variables may be freely used in another submodel,
   typically to compare simulation results of a simulation model
   with the observed values *)

VAR
  yminDash: REAL; (* minimum annual value found in anyone site *)
  ymeanDash: REAL; (* average annual value for whole valley *)
  ymaxDash: REAL; (* maximum annual value found in anyone site *)
  yminDashLn: REAL; (* ln of minimum annual value found in anyone site *)
  ymeanDashLn: REAL; (* ln of average annual value for whole valley *)
  ymaxDashLn: REAL; (* ln of maximum annual value found in anyone site *)

PROCEDURE ActivateLBMObsModel;
PROCEDURE DeactivateLBMObsModel;
PROCEDURE LBMObsModelIsActive(): BOOLEAN;

END LBMObs.

```

```
IMPLEMENTATION MODULE LBMObs;
```

```
(*
```

```
Revision history:
=====
```

Author	Date	Description
-----	----	-----
af	01/05/87	First implementation
af	12/05/90	- ModelWorks 2.0 adaptation, now dynamic model activation and de- activation supported - Curve attributes set, in particular if no observations available lineStyle is set to invisible
dg	05/12/91	Now imports from DMMathLib instead MathLib
dg	06/03/93	Import lists cleaned up
af	06/04/93	Prepared for ModelWorks 2.2 release

```
*)
```

```
FROM DMFiles IMPORT
  Response, TextFile, Lookup, Reset, Close, EOF, EOL, ReadChars,
  SkipGap, ReadChar, GetCardinal, legalNum, GetExistingFile;
FROM DMMessages IMPORT Inform, Warn;
FROM DMStrings IMPORT Concatenate;
FROM DMConversions IMPORT CardToString, StringToReal, UndefREAL,
  IsUndefREAL;
```

```
FROM DMMathLib IMPORT Ln;
```

```
FROM SimBase IMPORT
  Model, DeclM, IntegrationMethod, DeclSV, DeclMV, MDeclared,
  notDeclaredModel, RemoveM, StashFiling, Tabulation, Graphing,
  SetSimTime, SetMonInterval, NoInitialize,
  NoInput, NoOutput, NoDynamic, NoTerminate, NoAbout,
  SetDefltCurveAttrForMV, Stain, LineStyle;
```

```
FROM SimMaster IMPORT CurrentTime, InstallDefSimEnv;
```

```
VAR
```

```
(*storage for observations*)
yminD, ymeanD, ymaxD: ARRAY [kmin..kmax] OF REAL;

obsMod: Model;
```

```
PROCEDURE InitData;
```

```
VAR f: TextFile; r: Response; year,k: CARDINAL; ch: CHAR;
```

```
PROCEDURE TestEOF;
```

```
BEGIN
```

```
IF EOF(f) THEN Warn("Not enough data in observation file","", "") END;
```

```
END TestEOF;
```

```
PROCEDURE ReadReal(VAR f: TextFile; VAR r: REAL);
```

```
VAR numStr: ARRAY [0..31] OF CHAR;
```

```
BEGIN (*ReadReal*)
```

```
SkipGap(f); ReadChars(f,numStr);
```

```
IF (CAP(numStr[0])='N') AND (numStr[1]=0C) THEN
```

```
  r := UndefREAL(); legalNum := TRUE;
```

```
ELSE
```

```
  StringToReal(numStr,r,legalNum);
```

```
END(*IF*);
```

```

END ReadReal;
PROCEDURE CheckNum(k: CARDINAL; curVar: ARRAY OF CHAR);
  VAR msg1,msg2: ARRAY [0..127] OF CHAR; numStr: ARRAY [0..3] OF CHAR;
BEGIN (*CheckNum*)
  IF NOT legalNum OR (k<>year) THEN
    CardToString(k,numStr,0);
    Concatenate("Illegal number encountered: year = ",numStr,msg1);
    Concatenate("while attempting to read ",curVar,msg2);
    Warn(msg1,msg2,"");
  END(*IF*);
END CheckNum;
BEGIN (*InitData*)
  f.filename := "LBMObsUE.DAT";
  Lookup(f,f.filename,FALSE);
  IF f.res<>done THEN
    Inform("Couldn't open file 'LBMObsUE.DAT' containing observations", "", "");
    GetExistingFile(f,"Please locate 'LBMObsUE.DAT' with observations");
  END(*IF*);
  IF f.res=done THEN
    ReadChar(f,ch);
    WHILE ch<>EOL DO ReadChar(f,ch) END;
    FOR k:= kmin TO kmax DO
      TestEOF; GetCardinal(f,year); CheckNum(k,"year");
      TestEOF; ReadReal(f,ymeanD[k]); CheckNum(k,"Ymean'");
      TestEOF; ReadReal(f,yminD[k]); CheckNum(k,"Ymin'");
      TestEOF; ReadReal(f,ymaxD[k]); CheckNum(k,"Ymax'");
    END(*FOR*);
    Close(f);
  ELSE
    FOR k:= kmin TO kmax DO
      ymeanD[k] := UndefREAL();
      yminD[k] := UndefREAL();
      ymaxD[k] := UndefREAL();
    END(*FOR*);
  END(*IF*);
END InitData;

PROCEDURE Output;
  VAR k: INTEGER;
BEGIN
  k:= TRUNC(CurrentTime()+0.1)(*ensures correct rounding*);
  IF (k>=kmin) AND (k<=kmax) THEN
    ymeanDash := ymeanD[k];
    IF NOT IsUndefREAL(ymeanDash) THEN ymeanDashLn := Ln(negLogDelta+ymeanDash) END;
  ELSE
    ymeanDash:= UndefREAL();
    ymeanDashLn:= UndefREAL();
  END(*IF*);
  IF (k>=kmin) AND (k<=limkmax) THEN
    yminDash := yminD[k];
    ymaxDash := ymaxD[k];
    IF NOT IsUndefREAL(yminDash) THEN yminDashLn := Ln(negLogDelta+yminDash) END;
    IF NOT IsUndefREAL(ymaxDash) THEN ymaxDashLn := Ln(negLogDelta+ymaxDash) END;
  ELSE
    yminDash:= UndefREAL();
    ymaxDash:= UndefREAL();
    yminDashLn:= UndefREAL();
    ymaxDashLn:= UndefREAL();
  END(*IF*);
END Output;

PROCEDURE ModelObjects;
BEGIN
  DeclMV(yminDash, yLL, yUL,
    "Minimum larval density per site", "Ymin'",
    "larvae/kg branches",

```

```

    notOnFile, notInTable, notInGraph);
SetDefltCurveAttrForMV (obsMod, yminDash,turquoise,spotted,0C);
DeclMV(ymeanDash, yLL, yUL,
    "Average larval density in valley", "Y'",
    "larvae/kg branches",
    notOnFile, writeInTable, notInGraph);
SetDefltCurveAttrForMV (obsMod, ymeanDash,turquoise,unbroken,0C);
DeclMV(ymaxDash, yLL, yUL,
    "Maximum larval density per site", "Ymax'",
    "larvae/kg branches",
    notOnFile, notInTable, notInGraph);
SetDefltCurveAttrForMV (obsMod, ymaxDash,turquoise,spotted,0C);
DeclMV(yminDashLn, Ln(negLogDelta), Ln(yUL),
    "Ln of minimum larval density per site", "Ln(Ymin')",
    "larvae/kg branches",
    notOnFile, notInTable, notInGraph);
SetDefltCurveAttrForMV (obsMod, yminDashLn,turquoise,spotted,0C);
DeclMV(ymeanDashLn, Ln(negLogDelta), Ln(yUL),
    "Ln of average larval density in valley", "Ln(Y')",
    "larvae/kg branches",
    notOnFile, notInTable, isY);
SetDefltCurveAttrForMV (obsMod, ymeanDashLn,turquoise,unbroken,0C);
DeclMV(ymaxDashLn, Ln(negLogDelta), Ln(yUL),
    "Ln of maximum larval density per site", "Ln(Ymax')",
    "larvae/kg branches",
    notOnFile, notInTable, notInGraph);
SetDefltCurveAttrForMV (obsMod, ymaxDashLn,turquoise,spotted,0C);
END ModelObjects;

PROCEDURE ActivateLBMObsModel;
BEGIN
    IF NOT MDeclared(obsMod) THEN
        DeclM(obsMod, discreteTime,
            NoInitialize, NoInput, Output, NoDynamic, NoTerminate, ModelObjects,
            "Observations from the Upper Engadine Valley", "Obs UE",
            NoAbout);
        InstallDefSimEnv(InitData);
    END(*IF*);
END ActivateLBMObsModel;

PROCEDURE DeactivateLBMObsModel;
BEGIN
    IF MDeclared(obsMod) THEN RemoveM(obsMod) END(*IF*);
END DeactivateLBMObsModel;

PROCEDURE LBMObsModelIsActive (): BOOLEAN;
BEGIN
    RETURN MDeclared(obsMod)
END LBMObsModelIsActive;

BEGIN
    obsMod := notDeclaredModel;
END LBMObs.

```

Excerpt (middle portion missing) from data file *LBMObsUE.DAT* accessed by module *LBMObs*:

Year	y'	y'MIN	y'MAX
1949	0.018	0.006	0.041

1950	0.082	0.006	0.232
1951	0.444	0.001	1.266
1952	4.174	0.191	10.464
1953	68.797	16.667	128.490
1954	331.760	163.340	933.524
1955	126.541	25.048	317.868
1956	21.280	9.888	41.974
1957	2.246	1.330	4.538
1958	0.085	0.000	0.359
...			
...			
...			
1985	0.120	N	N
1986	0.690	N	N
1987	2.279	0.445	4.866
1988	39.029	4.149	88.146

Legend  
y' mean observed larval density  
y'MIN minimum observed larval density  
y'MAX maximum observed larval density

The following module is the main program module *LBM*. Its sole purpose is to start the simulation environment (procedure *RunSimEnvironment*) and to install a menu (procedure *InstallMenus*) which gives access to the actual models. The latter menu contains a command which asks the simulationist which submodel(s) she wishes to load (activate) or to remove (de-activate) (procedure *Choose*). The master module imports from the modules *LBMod* (population model) and *LBObs* (exports the parallel observation model) the procedures *ActivateLarchLBMod* and *DeactivateLarchLBMod* resp. *ActivateLBObsModel* and *DeactivateLBObsModel*. These procedures will declare or remove the desired models, thus allowing the simulationist to drop or load a model anytime during the simulation session.

```

MODULE LBM; (* af 1/5/87; 12/5/90 *)

(*
Module LBM (Larch Bud Moth)

Purpose    master module modelling the larch bud moth system
           by means of ModelWorks V0.3 simulating the system
           behavior for the Upper Engadine Valley

References Fischlin, A. 1982. Analyse eines Wald-Insekten-
           Systems: Der subalpine Laerchen-Arvenwald und der
           graue Laerchenwickler Zeiraphera diniana Gn. (Lep.,
           Tortricidae). Diss. ETH Nr 6977. Swiss Federal
           Institute of Technology Zuerich, Switzerland, 294pp.
*)

(* Imports from ModelWorks (Sim) *)
FROM SimBase IMPORT SetDefltGlobSimPars, MWWindowArrangement;
FROM SimMaster IMPORT RunSimEnvironment;
FROM SimGraphUtils IMPORT PlaceGraphOnSuperScreen;
FROM StructModAux IMPORT InstallCustomMenu, SetSimEnv, AssignSubModel,
InstallMyGlobPreferences;

(* Imports from Larch Bud Moth modular model definition (LBM) *)
FROM LBObs IMPORT ActivateLBObsModel, DeactivateLBObsModel,
LBObsModelIsActive, kmin, kmax;
FROM LBMod IMPORT ActivateLarchLBMod, DeactivateLarchLBMod,
LarchLBModIsActive;

VAR
obs, larchLBM: INTEGER;

```



```

PROCEDURE InitSimEnv;
BEGIN
  InstallCustomMenu("Models","Activation...","L");
  SetSimEnv(obs);
END InitSimEnv;

PROCEDURE SetMyGlobPreferences;
  CONST dummy = 0.1;
BEGIN
  SetDefltGlobSimPars(FLOAT(kmin), FLOAT(kmax), dummy, dummy, 1.0, 1.0);
  PlaceGraphOnSuperScreen(tiled);
END SetMyGlobPreferences;

BEGIN
  InstallMyGlobPreferences(SetMyGlobPreferences);
  AssignSubModel(obs,"Observations - Parallel Model Upper Engadine",
    ActivateLBMObsModel, DeactivateLBMObsModel, LBMObsModelIsActive);
  AssignSubModel(larchLBM,"Larch - Larch Bud Moth Model (b1)",
    ActivateLarchLBMMModel, DeactivateLarchLBMMModel, LarchLBMMModelIsActive);
  RunSimEnvironment( InitSimEnv );
END LBM.

```

### A.6.2 Discrete Event Harvesting In a Continuously Growing Forest - *ForestYield*

Forests can fix or release previously fixed carbon, hereby affecting the CO<sub>2</sub>-concentration of the atmosphere. Forest growth, e.g. in form of aforestations, is the essential process, which is potentially able to fix additional carbon. It is more or less proportional to the carbon fixing capability of a forest. Growth of tree biomass  $Q$  can be roughly modeled by the pattern of logistic growth (PEARL, 1927), i.e. growth is the slower the less biomass is already present, reaches then a maximum while biomass accumulates, and slows down again in a mature forest (FISCHLIN & BUGMANN, 1993; 1994). In order to account for various growth patterns (FISCHLIN & BUGMANN, 1993; 1994), the model parameters  $r$  and  $K$  can be adjusted according to the studied type of forest  $j$ . Given these assumptions the following continuous time model equations result:

$$\frac{dQ_j(t)}{dt} = r_j \cdot \frac{(K_j - Q_j(t))}{K_j} \cdot Q_j(t) \quad (1)$$

where

$j$	Type of forest, e.g. "Beech forest", "Montane spruce", or "Subalpine spruce"	
$Q_j$	Dry Weight [DW] of above ground biomass (includes wood)	[t DW/ha]
$r_j$	Maximum relative growth rate	[/a]
$K_j$	Carrying capacity	[t DW/ha]

In addition to growth, the model has to simulate harvesting and the fate of the carbon transported out of the forest in form of wood and transferred into long-lived forest products.

A simple model of conventional harvesting as currently practiced in Switzerland is in form of discrete events: In reality forests are cut in steps, hence harvesting can be modelled as a row of 3 cuts with e.g. 8 years in between, initiated as soon as  $Q_j$  amounts to 90% of  $K_j$ . For each type of forest the harvest  $H_j$  can be modeled as a sequence of three state events occurring at the cutting times  $t^- = t_h, t_{h+8},$  and  $t_{h+16}$ , where  $h_i = 30, 50$  respectively 70% of the current  $Q_j$  is cut, i.e. removed out of the forest:

$$H_j(t^-) = \begin{cases} h_i \cdot Q_j(t^-) & t^- = t_h + i \cdot 8 \quad i = 0,1,2 \\ 0 & \text{else} \end{cases} \quad (2)$$

$$t_h = t \mid Q_j(t) = 0.9 \cdot K_j \quad h_0 = 0.3 \quad h_1 = 0.5 \quad h_2 = 0.7 \quad (a)$$

$$Q_j(t) = Q_j(t^-) - H_j(t^-) \quad (3)$$

where

$H_j$	Harvested biomass	[t DW/ha·a]
$h_i$	Fraction of harvested wood in percentages of currently present biomass $Q_j$	
$t_h$	Harvesting time or time of first cut in a sequence of 3 cuts	
$t^-$	Continuous left-hand side of time before and up to the discrete event harvest	

Alternatively a maximum sustainable yield (MSY) can be obtained if forest biomass  $Q_j$  is around  $Q_j^*$ , i.e. a biomass which maximizes  $dQ_j(t)/dt$ . This is the case if  $Q_j^* = K_j/2$  and if harvesting occurs continuously. However, in practice a truly continuous harvesting is not feasible, hence, a MSY harvesting scheme can be modeled approximately as follows: Whenever  $Q_j$  exceeds  $K_j/2 + \epsilon_j$  a biomass of  $2 \cdot \epsilon_j$  is harvested:

$$H_j(t^*) = \begin{cases} 2 \cdot \epsilon_j & t^* = t_h \\ 0 & t^* \neq t_h \end{cases} \quad (2')$$

$$t_h = t \mid Q_j(t) = 0.5 \cdot K_j + \epsilon_j \quad (a')$$

Both types of harvesting transfer a certain fraction  $\mu$ , e.g. 40% (HARMON *et al.*, 1990), of the harvested biomass  $H_j$  to enduring wood products  $P_j$ :

$$P_j(t) = P_j(t^*) + \mu \cdot H_j(t^*) \quad (4)$$

where

$P_j$  Biomass in enduring wood products [t DW/ha]  
 $\mu$  Fraction of harvested biomass ending up in enduring wood products

Finally, the decay of the enduring wood products and the associated release of CO<sub>2</sub> to the atmosphere may be modeled as follows:

$$\frac{dP_j(t)}{dt} = -d_j \cdot P_j(t) \quad (5)$$

where

$d_j$  Relative decay rate [a]

The following table lists all needed model parameters and the model can be solved by using small initial values for the biomass, e.g.  $Q_j(0) = 5$  t/ha.

Parameter	Unit	j = Beech forest	j = Montane spruce forest	j = Subalpine spruce forest
$r_j$	a <sup>-1</sup>	0.04	0.05	0.05
$K_j$	t DW/ha	550	600	170
$d_j$	a <sup>-1</sup>	0.025	0.037	0.037
$\epsilon_j$	t DW/ha	40	80	25

Eq. (1) and (5) are differential equations and form a DESS, called *biomass*, which describes growth and decay of biomass pools. Eq. (2) respectively (2'), (3), and (4) correspond to instantaneous state transition functions and define the dynamics of a discrete event system (DEVS), called *harvest*, since it describes the discrete harvest. What results represents a structured, continuous time system coupling a DESS with a DEVS.

There are two state variables,  $Q_j$  and  $P_j$ , both belonging to the DESS submodel;  $H_j$  is just an auxiliary variable of the DEVS. However, the state variables  $Q_j$  and  $P_j$  are also affected by the dynamics of the DEVS (see Eq. 3 and 4). One solution to model the system is the following:

As a consequence of the state event occurring at condition (a) respectively (a') and at time  $t_h=t$ , the event output function  $g_\theta$  of the DESS *biomass* produces an event output  $\vartheta$  on behalf of the DEVS *harvest* (see chapter *Theory* Eq. 4.2b). Such an event output is of class  $v = h_0$ , its  $\tau = 0$ , and it will pass as the transaction  $\alpha$  the DESS' state vector  $[Q_j, P_j]'$ . The corresponding event input of the DEVS *harvest* will cause the calculation of the auxiliary variable  $H_j$  and according to Eq. (3) and (4) also a change in the state of the DESS, i.e. the first cut of a harvesting sequence. Furthermore, in the case of a conventional harvesting scheme, the event

of class  $h_0$  schedules immediately the subsequent second cut (event of class  $v = h_1$ ). Finally, the event of class  $v = h_1$  schedules the third cut (event of class  $v = h_2$ ). In order to allow for the proper state changes, all events of class  $h_0$  to  $h_2$  pass the received transaction, i.e. the state vector  $[Q_j, P_j]$ , on to the subsequent event, i.e. while scheduling harvesting events they use this vector as the transaction  $\alpha$ .

The following model definition program ForestYield implements the described model and allows to experiment with all three forest types (beech, montane and subalpine spruce) and with various management or silvicultural practices such as (no harvesting at all, clear cutting, and plenter management).

```
MODULE ForestYield;
```

```
(*****
```

```
MODEL: ForestYield
```

```
Purpose:      Simulation of silvicultural management
             strategies for Swiss forestry under the perspective
             of C-sequestration in order to contribute to
             curbing climatic change. The model definition
             program allows to explore strategies of maximum
             sustainable yield vs. conventional management in
             relation with the management of the carbon fluxes
             and pools, in particular the storing of carbon in
             form of durable wood products. For more details
             see the listed references.
```

```
References
```

```
Fischlin, A. & Bugmann, H., 1993. Think globally, act
locally! A small country case study in reducing
net CO2 emissions by carbon fixation policies. In:
Kanninen, M. (ed.), Carbon balance of the world's
forested ecosystems: Towards a global assessment.
Publications of the Academy of Finland, VAPK
Publishing, Helsinki: in print.
```

```
Fischlin, A. & Bugmann, H.K., 1994. Können forstliche
Massnahmen einen Beitrag zur Verminderung der
schweizerischen CO2-Emissionen leisten? Ökologische
Grundlagen und erste Abschätzungen. Schweiz. Z.
Forstw., 145(4): 275-292.
```

```
Authors:    A. Fischlin & H. Bugmann, 21.Nov.93,
             Systems Ecology, ETHZ
```

```
*****)
```

```
FROM DMStrings IMPORT Concatenate, Append, AppendCh, AssignString;
FROM DMStorage IMPORT Allocate, Deallocate;
FROM DMConversions IMPORT IntToString, UndefinedREAL, IsUndefinedREAL,
    RealToString, RealFormat;
FROM DMMessages IMPORT Ask;
FROM DMMenus IMPORT InstallMenu, InstallCommand, Menu, Command,
    CheckCommand, UncheckCommand, IsCommandChecked, AccessStatus, Marking,
    InstallSeparator, Separator;
FROM DMMaster IMPORT DialogMachineTask;
```

```
FROM SimBase IMPORT
    Model, IntegrationMethod, DeclM, DeclSV, DeclP, RTCType,
    StashFiling, Tabulation, Graphing, DeclMV, SelectM,
    SetP, GetP, SetProjDescrs, RemoveM, MDeclared, RemoveP, PDeclared,
    SetSimTime, SetIntegrationStep, SetMonInterval,
    GetDefltGlobSimPars, Message, MWindowArrangement,
    NoInitialize, NoInput, NoOutput, NoTerminate, NoAbout, DoNothing,
```

```

StateVar, Derivative, Parameter, AuxVar,
MWindow, GetWindowPlace, SetWindowPlace, ClearTable;

FROM SimEvents IMPORT
  EventClass, nilTransaction, Transaction, StateTransition,
  AsTransaction, noStateTransition, InitEventScheduler,
  SchedulingOnlyAfter, ScheduleEvent, PendingEvents, NextEventAt,
  DiscardEventsBefore, DiscardEventsAfter, never, DeclDEVM;

FROM SimMaster IMPORT
  RunSimEnvironment, CurrentTime, InstallExperiment, SimRun,
  PauseRun, MWSubState, GetMWSubState;

FROM SimGraphUtils IMPORT PlaceGraphOnSuperScreen;

FROM StateEvents IMPORT
  ExpectStateEvt, StateEvt, IsStateEvt, unexpectedStateEvt,
  StateEvtExpected, IgnoreStateEvt;

(*****
(* Data - parameters and structures of submodels: *)
*****)

TYPE
  Alfa = ARRAY [0..31] OF CHAR;

(* Forest *)
(* ===== *)

TYPE
  ForestType = (Beech, MontaneSpruce, SubalpineSpruce);

  Forest = RECORD
    m: Model;           (* forest model *)
    j: ForestType;     (* type of forest *)
    name,ident: ARRAY [MIN(ForestType)..MAX(ForestType)] OF Alfa;
    Qj: StateVar;      (* dry weight of above-ground biomass in forest *)
    QjDot: Derivative;
    rj: Parameter;     (* maximum relative growth rate *)
    r: ARRAY [MIN(ForestType)..MAX(ForestType)] OF Parameter;
    Kj: Parameter;     (* carrying capacity of forest *)
    K: ARRAY [MIN(ForestType)..MAX(ForestType)] OF Parameter;
  END(*RECORD*);

(* Woodsector *)
(* ===== *)

TYPE
  WoodSector = RECORD
    m: Model;           (* wood sector model *)
    Pj: StateVar;      (* dry weight of endurable forest products *)
    PjDot: Derivative;
    mu: Parameter;     (* fraction of wood harvest ending in endurable forest
products *)
    dj: Parameter;     (* relative decay rate of endurable forest products *)
    d: ARRAY [MIN(ForestType)..MAX(ForestType)] OF Parameter;
  END(*RECORD*);

(* Harvest *)
(* ===== *)

CONST

```

```

clearCutting = 1;      (* EventClasses have to be globally unique *)
plenterHarvesting = 2;
fstSubCut = 0;
lastSubCut = 2;

TYPE
  HarvestType = (unused, clearCut, plenter);
  Harvest = RECORD
    m: Model;          (* harvesting model *)
    hT: HarvestType;   (* type of harvesting *)
    name,ident: ARRAY [MIN(HarvestType)..MAX(HarvestType)] OF Alfa;
    thetaClrCut: Parameter; (* Clear cut threshold (fraction of Qj) at which
harvest takes place *)
    thetaPlent: Parameter; (* Plenter threshold (fraction of Qj) at which harvest
takes place *)
    epsj: Parameter;   (* fraction of harvested wood as well as
tolerance for exceeding thetaPlent before plenter
harvesting *)
    eps: ARRAY [MIN(ForestType)..MAX(ForestType)] OF Parameter;
    i: [fstSubCut..lastSubCut]; (* index of sub cut while clear cutting *)
    h: ARRAY [fstSubCut..lastSubCut] OF Parameter;
    (* fraction of harvested wood in % of Qj *)
    interval: Parameter; (* years between subsequent sub cuts while clear
cutting *)
    hEvt: StateEvt;    (* state event harvesting *)
    Hj: AuxVar;        (* harvested biomass *)
  END(*RECORD*);

(* Observer *)
(* ===== *)

TYPE
  Observer = RECORD
    m: Model;
    accCPool: StateVar; (* total carbon fixed pooled over time, needed to compute
avgTotCFixed *)
    accCPoolDot: Derivative;
    totCFixed: AuxVar;  (* total carbon fixed in forest and enduring forest products
*)
    avgTotCFixed: AuxVar; (* average over time of total carbon fixed (totCFixed) *)
    cDWRatio: Parameter; (* ratio of C to dry weight *)
    thetaDash: Parameter; (* fraction of Kj, as soon as reached by Qj, causing to
start assessing the average total carbon pool (see
procedure StartCPoolAssessment), or in other words
to ignore the transient behavior of the system
before that moment *)
    begAccCEvt: StateEvt; (* state event start of C-pool assessment *)
    startCPoolAssesTime: REAL; (* time when the C-pool assessment started *)
  END(*RECORD*);

(* Forestry *)
(* ===== *)

TYPE
  Forestry = RECORD
    forest: Forest;
    harvest: Harvest;
    woodSector: WoodSector;
    observer: Observer;
  END;

VAR
  f: Forestry;

```

```

(*****
(* Submodels *)
(*****

(* Forest *)
(* ===== *)

PROCEDURE ForestInitialize;
BEGIN
  WITH f.forest DO
    GetP (m, r[j], rj);
    GetP (m, K[j], Kj);
  END(*WITH*);
END ForestInitialize;

PROCEDURE ForestDynamic;
BEGIN
  WITH f.forest DO
    QjDot := rj*(Kj - Qj)/Kj*Qj;          (* Eq. (1) *)
  END(*WITH*);
END ForestDynamic;

PROCEDURE ForestOutput;
BEGIN
  WITH f.forest DO WITH f.harvest DO
    IF (hT=clearCut) AND IsStateEvt(hEvt,Qj) THEN
      i := fstSubCut;
      ScheduleEvent(clearCutting,0.0,AsTransaction(f));
    ELSIF (hT=plenter) AND IsStateEvt(hEvt,Qj) THEN
      ScheduleEvent(plenterHarvesting,0.0,AsTransaction(f));
    END(*IF*);
  END(*WITH*) END(*WITH*);
END ForestOutput;

(* Harvest *)
(* ===== *)

PROCEDURE HarvestInitialize;
BEGIN
  WITH f.forest DO WITH f.harvest DO
    IF (hT=clearCut) THEN
      ExpectStateEvt(hEvt,Qj,thetaClrCut*Kj,MAX(REAL));
    ELSIF (hT=plenter) THEN
      ExpectStateEvt(hEvt,Qj,thetaPlent*Kj+epsj,MAX(REAL));
    END(*IF*);
    IF (hT=plenter) THEN GetP (f.harvest.m, eps[j], epsj) END;
  END(*WITH*) END(*WITH*);
END HarvestInitialize;

PROCEDURE ClearCutEvent(alfa: Transaction);
TYPE ForestryAsTransaction = POINTER TO Forestry;
VAR msg: ARRAY [0..127] OF CHAR; f: ForestryAsTransaction;
BEGIN
  f := alfa;
  WITH f^ DO
    msg := "Clear cut: sub cut "; AppendCh(msg,CHR(ORD('0')+harvest.i));
    Message(msg);
    harvest.Hj := harvest.h[harvest.i]*forest.Qj;          (* Eq. (2) *)
    IF harvest.i<lastSubCut THEN
      INC(harvest.i);
      ScheduleEvent(clearCutting,harvest.interval,alfa);
    END(*IF*);
    forest.Qj := forest.Qj - harvest.Hj;          (* Eq. (3) *)
  END;
END;

```

```

        woodSector.Pj := woodSector.Pj + woodSector.mu*harvest.Hj;      (* Eq. (4) *)
    END(*WITH*);
END ClearCutEvent;

PROCEDURE PlenterHarvestEvent(alfa: Transaction);
    TYPE ForestryAsTransaction = POINTER TO Forestry;
    VAR msg: ARRAY [0..127] OF CHAR; f: ForestryAsTransaction;
BEGIN
    f := alfa;
    WITH f^ DO
        Message("Plenter harvest");
        harvest.Hj := 2.0*harvest.epsj;                                (* Eq. (2') *)
        forest.Qj := forest.Qj - harvest.Hj;                          (* Eq. (3) *)
        woodSector.Pj := woodSector.Pj + woodSector.mu*harvest.Hj;    (* Eq. (4) *)
    END(*WITH*);
END PlenterHarvestEvent;

(* Woodsector *)
(* ===== *)

PROCEDURE WoodSectorInitialize;
BEGIN
    WITH f.forest DO WITH f.woodSector DO
        GetP (f.woodSector.m, d[j], dj);
    END(*WITH*) END(*WITH*);
END WoodSectorInitialize;

PROCEDURE WoodSectorDynamic;
BEGIN
    WITH f.woodSector DO
        PjDot := - dj*Pj;                                             (* Eq. (5) *)
    END(*WITH*);
END WoodSectorDynamic;

(* Observer *)
(* ===== *)

PROCEDURE ObserverInitialize;
BEGIN
    WITH f.observer DO
        startCPoolAssesTime := UndefREAL();
        ExpectStateEvt(begAccCEvt,f.forest.Qj,thetaDash*f.forest.Kj,MAX(REAL));
    END(*WITH*);
END ObserverInitialize;

PROCEDURE ObserverDynamic;
BEGIN
    WITH f.observer DO
        accCPoolDot := totCFixed;
    END(*WITH*);
END ObserverDynamic;

PROCEDURE StartCPoolAssessment;
    VAR msg: ARRAY [0..127] OF CHAR;
BEGIN
    WITH f.observer DO
        startCPoolAssesTime := CurrentTime();
        accCPool := 0.0;
        msg := "Start of assessing average total C-pool size";
        Message(msg);
        IgnoreStateEvt(begAccCEvt); (* subsequently ignore any such event *)
    END(*WITH*);
END StartCPoolAssessment;

```



```

PROCEDURE ObserverOutput;
BEGIN
  WITH f.observer DO
    totCFixed := cDWRatio*(f.forest.Qj + f.woodSector.Pj);
    IF IsUndefREAL(startCPoolAssessTime) THEN
      avgTotCFixed := UndefREAL();
    ELSE
      avgTotCFixed := accCPool/(CurrentTime()-startCPoolAssessTime);
    END(*IF*);
    IF IsStateEvt(begAccCEvt,f.forest.Qj) AND StateEvtExpected(begAccCEvt) THEN
      StartCPoolAssessment;
    END(*IF*);
  END(*WITH*);
END ObserverOutput;

PROCEDURE ObserverTerminate;
  PROCEDURE MakeMsgForX(descr: ARRAY OF CHAR; x: REAL; unit: ARRAY OF CHAR);
    VAR msg: ARRAY [0..127] OF CHAR;
  BEGIN (*MakeMsgForX*)
    RealToString(x,msg,0,3,FixedFormat);
    Concatenate(descr,msg,msg); Append(msg,unit);
    Message(msg);
  END MakeMsgForX;
  BEGIN (*ObserverTerminate*)
    WITH f.observer DO
      MakeMsgForX("Mean total C fixed = ",avgTotCFixed," [t/ha]");
    END(*WITH*);
  END ObserverTerminate;

(* Forestry *)
(* ===== *)

PROCEDURE DeclForestryBase; (* Declare basis of all model variants *)
BEGIN
  (* some objects of model will be declared dynamically by DeclForest *)
  WITH f.forest DO
    f.forest.j := Beech; (* must be initialized once *)
    name[Beech] := "Beech forest";
    name[MontaneSpruce] := "Montane spruce forest";
    name[SubalpineSpruce] := "Subalpine spruce forest";
    ident[Beech] := "Beech";
    ident[MontaneSpruce] := "MtSprc";
    ident[SubalpineSpruce] := "SaSprc";
    r[Beech] := 0.04;
    r[MontaneSpruce] := 0.05;
    r[SubalpineSpruce] := 0.05;
    K[Beech] := 550.0;
    K[MontaneSpruce] := 600.0;
    K[SubalpineSpruce] := 450.0;
    DeclM(m, Heun, ForestInitialize, NoInput,
      ForestOutput, ForestDynamic, NoTerminate, DoNothing,
      "Forest submodel", "forest.m", NoAbout);

    DeclSV(Qj, QjDot, 5.0, 0.0, 800.0,
      "Biomass (dry weight) of forest", "Qj", "t/ha");

    DeclMV(Qj, 0.0, 600.0,
      "Biomass (dry weight) of forest", "Qj", "t/ha",
      notOnFile, writeInTable, isY);
    DeclMV(QjDot, 0.0, 40.0,
      "Biomass derivative", "dQj/dt", "t/ha/a",
      notOnFile, notInTable, notInGraph);
  END(*WITH*);

  (* harvest model and model objects will be declared only dynamically by DeclHarvesting

```

```

*)
WITH f.harvest DO
  hT := unused; (* must be initialized once *)
  name[unused] := "Unused forest";
  name[clearCut] := "Clear cutting";
  name[plenter] := "Plenter management";
  ident[Beech] := "unused";
  ident[MontaneSpruce] := "clrCut";
  ident[SubalpineSpruce] := "plent";
  hEvt := unexpectedStateEvt; (* must be initialized once *)
  thetaClrCut := 0.9;
  thetaPlent := 0.5;
  eps[Beech] := 40.0;
  eps[MontaneSpruce] := 80.0;
  eps[SubalpineSpruce] := 25.0;
  h[fstSubCut] := 0.3;
  h[fstSubCut+1] := 0.5;
  h[lastSubCut] := 0.7;
  interval := 8.0;
END(*WITH*);

WITH f.woodSector DO
  DeclM(m, Heun, WoodSectorInitialize, NoInput,
    NoOutput, WoodSectorDynamic, NoTerminate, DoNothing,
    "Wood sector submodel", "woodSect.m", NoAbout);

  DeclSV(Pj, PjDot, 0.0, 0.0, 800.0,
    "Endurable forest products", "Pj", "t/ha");

  DeclMV(Pj, 0.0, 600.0,
    "Endurable forest products", "Pj", "t/ha",
    notOnFile, writeInTable, isY);

  d[Beech] := 0.025;
  d[MontaneSpruce] := 0.037;
  d[SubalpineSpruce] := 0.037;
  DeclP(mu, 0.4, 0.0, 1.0, rtc,
    "Fraction transferred from harvest to wood sector", "μ", "%");
END(*WITH*);

WITH f.observer DO
  begAccCEvt := unexpectedStateEvt;
  DeclM(m, Heun, ObserverInitialize, NoInput,
    ObserverOutput, ObserverDynamic, ObserverTerminate, DoNothing,
    "Observer submodel", "observer.m", NoAbout);

  DeclSV(accCPool, accCPoolDot, 0.0, 0.0, 0.0,
    "Total carbon ever fixed (pooled over time)", "accCPool", "t/ha");

  DeclMV(accCPool, 0.0, 600.0,
    "Total carbon ever fixed (pooled over time)", "accCPool", "t/ha",
    notOnFile, notInTable, notInGraph);
  DeclMV(totCFixed, 0.0, 600.0,
    "Total carbon fixed", "totCFixed", "t/ha",
    notOnFile, writeInTable, isY);
  DeclMV(avgTotCFixed, 0.0, 600.0,
    "Average total carbon fixed", "avgTotCFixed", "t/ha",
    notOnFile, writeInTable, isY);

  DeclP(cdWRatio, 0.45, 0.3, 0.6, rtc,
    "C in wood (ratio C/dry weight)", "cdWRatio", "%");
  DeclP(thetaDash, 0.3, 0.0, 1.0, rtc,
    "Threshold (% of K) to start assessing C-pool size", "theta'", "%");
END(*WITH*);

END DeclForestryBase;

PROCEDURE DeclPlenterHarvest;

```

```

VAR descr1,descr2: ARRAY [0..63] OF CHAR; id1,id2: ARRAY [0..15] OF CHAR;
  selected: BOOLEAN;
BEGIN
  WITH f.harvest DO
    SelectM(m,selected);
    DeclP(thetaPlent, thetaPlent, 0.0, 1.0, rtc,
      "Threshold (% of K) to intitiate plenter harvesting", "theta[plent]", "%");
    descr1 := "Tolerance of theta";
    id1 := "eps";
    Concatenate(descr1,f.forest.ident[f.forest.j],descr2); Append(descr2," before harvesting");
    Concatenate(id1,f.forest.ident[f.forest.j],id2); Append(id2,"");
    DeclP(eps[f.forest.j], eps[f.forest.j], 0.0, 300.0, rtc, descr2, id2, "%");
    epsj := eps[f.forest.j];
  END(*WITH*);
END DeclPlenterHarvest;

PROCEDURE DeclForest;
  VAR descr1,descr2: ARRAY [0..63] OF CHAR; id1,id2: ARRAY [0..15] OF CHAR;
  selected: BOOLEAN;
BEGIN
  WITH f.forest DO
    SelectM(m,selected);

    descr1 := "Intrinsic growth rate of ";
    id1 := "r";
    Concatenate(descr1,name[j],descr2); Append(descr2," forest");
    Concatenate(id1,ident[j],id2); Append(id2,"");
    DeclP(r[j], r[j], 0.0, 1.0, rtc, descr2, id2, "/a");
    rj := r[j];

    descr1 := "Carrying capacity of ";
    id1 := "K";
    Concatenate(descr1,name[j],descr2); Append(descr2," forest");
    Concatenate(id1,ident[j],id2); Append(id2,"");
    DeclP(K[j], K[j], 0.0, 1000.0, rtc, descr2, id2, "t/ha");
    Kj := K[j];

  END(*WITH*);
  IF MDeclared(f.harvest.m) AND (f.harvest.ht=plenter) THEN DeclPlenterHarvest END;
  WITH f.woodSector DO
    SelectM(m,selected);

    descr1 := "Decay rate of durable ";
    id1 := "d";
    Concatenate(descr1,f.forest.name[f.forest.j],descr2); Append(descr2," forest products");
    Concatenate(id1,f.forest.ident[f.forest.j],id2); Append(id2,"");
    DeclP(d[f.forest.j], d[f.forest.j], 0.0, 1.0, rtc, descr2, id2, "t/ha");
    dj := d[f.forest.j];

  END(*WITH*);
END DeclForest;

PROCEDURE DeclHarvesting;
  VAR stTransFct: ARRAY [0..0] OF StateTransition;
  istr: ARRAY [0..7] OF CHAR; ii: [fstSubCut..lastSubCut];
  descr1,descr2: ARRAY [0..63] OF CHAR; id1,id2: ARRAY [0..15] OF CHAR;
BEGIN
  WITH f.harvest DO

    CASE ht OF
    | clearCut:
      stTransFct[0].ec := clearCutting;
      stTransFct[0].fct := ClearCutEvent;
    | plenter:
      stTransFct[0].ec := plenterHarvesting;
      stTransFct[0].fct := PlenterHarvestEvent;
    END(*CASE*);
  END;
END DeclHarvesting;

```

```

DeclDEVM(m, HarvestInitialize, NoInput, NoOutput, stTransFct, NoTerminate,
  DoNothing, "Harvesting submodel", "harvest.m", NoAbout);

DeclSV(Hj, Hj, 0.0, 0.0, 0.0,
  "Harvested wood", "Hj", "t/ha");

DeclMV(Hj, 0.0, 600.0,
  "Harvested wood", "Hj", "t/ha",
  notOnFile, writeInTable, notInGraph);

CASE hT OF
| clearCut:
  DeclP(thetaClrCut, thetaClrCut, 0.0, 1.0, rtc,
    "Threshold (% of K) to initiate clear cutting", "theta[clrCut]", "%");
  FOR ii:= fstSubCut TO lastSubCut DO
    IntToString(ii,iStr,0);
    Concatenate("Fraction harvested in sub cut ",iStr,descr1);
    Append(descr1," while clear cutting");
    Concatenate("h",iStr,id1);
    DeclP(h[ii], h[ii], 0.0, 1.0, rtc, descr1, id1, "%");
  END(*FOR*);
  i:= fstSubCut;
  DeclP(interval, interval, 0.0, 20.0, rtc,
    "Interval between sub cuts while clear cutting", "interv", "a");
| plenter:
  DeclPlenterHarvest;
END(*CASE*);

END(*WITH*);
END DeclHarvesting;

(*****
(* Interactive specification of model variants via menu commands *)
*****)

VAR
  forMenu : Menu;
  cmdF: ARRAY [MIN(ForestType)..MAX(ForestType)] OF Command;
  cmdH: ARRAY [MIN(HarvestType)..MAX(HarvestType)] OF Command;

PROCEDURE DiscardCurForest;
BEGIN
  WITH f.forest DO
    UncheckCommand(forMenu,cmdF[j]);
    IF MDeclared(m) THEN
      IF PDeclared(m,r[j]) THEN RemoveP(m,r[j]) END;
      IF PDeclared(m,K[j]) THEN RemoveP(m,K[j]) END;
      IF PDeclared(f.woodSector.m,f.woodSector.d[j]) THEN
        RemoveP(f.woodSector.m,f.woodSector.d[j])
      END(*IF*);
      IF MDeclared(f.harvest.m) AND (f.harvest.hT=plenter) THEN
        RemoveP(f.harvest.m,f.harvest.thetaPlent);
        RemoveP(f.harvest.m,f.harvest.eps[j])
      END(*IF*);
    END(*IF*);
  END(*WITH*);
END DiscardCurForest;

PROCEDURE ActivateAForest(ft: ForestType);
BEGIN
  DiscardCurForest;
  WITH f.forest DO
    j := ft;
    CheckCommand(forMenu, cmdF[j]);
  END(*WITH*);
  DeclForest;

```

```

    SetProjDescrs( "Swiss forests and C-sequestration", f.forest.name[ft], "", TRUE, TRUE,
                  TRUE, TRUE, TRUE, TRUE, TRUE, TRUE);
END ActivateAForest;

PROCEDURE ActivateBeechForest;
BEGIN
    ActivateAForest(Beech);
END ActivateBeechForest;

PROCEDURE ActivateMontaneSpruceForest;
BEGIN
    ActivateAForest(MontaneSpruce);
END ActivateMontaneSpruceForest;

PROCEDURE ActivateSubalpineSpruceForest;
BEGIN
    ActivateAForest(SubalpineSpruce);
END ActivateSubalpineSpruceForest;

PROCEDURE DiscardCurHarvesting;
BEGIN
    UncheckCommand(forMenu, cmdH[f.harvest.hT]);
    IF MDeclared(f.harvest.m) THEN
        RemoveM(f.harvest.m);
        IgnoreStateEvt(f.harvest.hEvt);
        (* ignore all eventually pending events *)
        IF f.harvest.hT=clearCut THEN
            DiscardEventsAfter(clearCutting, CurrentTime(), AsTransaction(f));
        ELSIF f.harvest.hT=plenter THEN
            DiscardEventsAfter(plenterHarvesting, CurrentTime(), AsTransaction(f));
        END(*IF*);
    END(*IF*);
END DiscardCurHarvesting;

PROCEDURE ActivateAHarvesting(harv: HarvestType);
BEGIN (*ActivateAHarvesting*)
    WITH f.harvest DO
        DiscardCurHarvesting;
        hT := harv;
        CheckCommand(forMenu, cmdH[harv]);
        IF hT<>unused THEN
            DeclHarvesting
        ELSIF PendingEvents(>0 THEN
            DiscardEventsBefore(never)
        END(*IF*);
    END(*WITH*);
END ActivateAHarvesting;

PROCEDURE ActivateUnused;
BEGIN
    ActivateAHarvesting(unused)
END ActivateUnused;

PROCEDURE ActivateClearCutMgmt;
BEGIN
    ActivateAHarvesting(clearCut)
END ActivateClearCutMgmt;

PROCEDURE ActivatePlenterMgmt;
BEGIN
    ActivateAHarvesting(plenter)
END ActivatePlenterMgmt;

PROCEDURE InstallCustomMenu;
BEGIN
    InstallMenu(forMenu, "Forestry", enabled);

```

```

InstallCommand(forMenu, cmdF[Beech], f.forest.name[Beech],
  ActivateBeechForest, enabled, unchecked);
InstallCommand(forMenu, cmdF[MontaneSpruce], f.forest.name[MontaneSpruce],
  ActivateMontaneSpruceForest, enabled, unchecked);
InstallCommand(forMenu, cmdF[SubalpineSpruce], f.forest.name[SubalpineSpruce],
  ActivateSubalpineSpruceForest, enabled, unchecked);
InstallSeparator(forMenu, line);
InstallCommand(forMenu, cmdH[unused], "Unused forest",
  ActivateUnused, enabled, unchecked);
InstallCommand(forMenu, cmdH[clearCut], "Clear cutting",
  ActivateClearCutMgmt, enabled, unchecked);
InstallCommand(forMenu, cmdH[plenter], "Plenter management",
  ActivatePlenterMgmt, enabled, unchecked);
END InstallCustomMenu;

```

```

(*****
(* Experiment *)
(*****

```

```

PROCEDURE Experiment;
  VAR jj: ForestType; ii: HarvestType;
      z: ARRAY [MIN(ForestType)..MAX(ForestType)],
          [MIN(HarvestType)..MAX(HarvestType)] OF REAL;
      x,y,w,h: INTEGER; isOpen : BOOLEAN;
  PROCEDURE MakeMsgForX(descr: ARRAY OF CHAR;
                        x: REAL; unit: ARRAY OF CHAR;
                        jj: ForestType; ii: HarvestType);
    VAR rStr: ARRAY [0..15] OF CHAR; msg: ARRAY [0..127] OF CHAR;
  BEGIN (*MakeMsgForX*)
    RealToString(x,rStr,0,3,FixedFormat);
    AssignString(descr,msg); Append(msg,rStr); Append(msg,unit);
    Append(msg,f.forest.name[jj]);
    Append(msg," / ");
    Append(msg,f.harvest.name[ii]);
    Message(msg);
  END MakeMsgForX;
BEGIN (*Experiment*)
  FOR jj:= MIN(ForestType) TO MAX(ForestType) DO
    ActivateAForest(jj);
    FOR ii:= MIN(HarvestType) TO MAX(HarvestType) DO
      ActivateAHarvesting(ii);
      SimRun;
      z[jj,ii] := f.observer.avgTotCFixed;
    END(*FOR*);
  END(*FOR*);

  (* Display of results: *)
  GetWindowPlace(TableW,x,y,w,h,isOpen);
  IF isOpen THEN ClearTable ELSE SetWindowPlace(TableW,x,y,w,h) END;
  FOR jj:= MIN(ForestType) TO MAX(ForestType) DO
    FOR ii:= MIN(HarvestType) TO MAX(HarvestType) DO
      MakeMsgForX("Mean total C fixed = ",z[jj,ii]," [t/ha] <-- Run: ",jj,ii);
    END(*FOR*);
  END(*FOR*);
END Experiment;

```

```

(*****
(* Initialization of models and default variants *)
(*****

```

```

PROCEDURE DefineModelAndEnvironment;
BEGIN
  DeclForestryBase;
  InstallCustomMenu;
  ActivateBeechForest;

```

```
    ActivateClearCutMgmt;  
    SetSimTime(0.0,500.0);  
    SetIntegrationStep(0.5);  
    SetMonInterval(1.0);  
    InstallExperiment(Experiment);  
    PlaceGraphOnSuperScreen(tiled);  
END DefineModelAndEnvironment;  
  
BEGIN  
    RunSimEnvironment(DefineModelAndEnvironment);  
END ForestYield.
```





## B Literature

The following list contains references of cited literature as well as references to recommended further reading on the subject of modelling and simulation:

- ATKINSON, L.V. & HARLEY, P.J., 1983. *An Introduction to numerical methods with Pascal*. London: Addison-Wesley, 300pp.
- BALTENSWEILER, W. & FISCHLIN, A., 1988. *The larch bud moth in the Alps*. In: Berryman, A.A. (ed.), *Dynamics of forest insect populations: patterns, causes, implications*. New York a.o.: Plenum Publishing Corporation: 331-351.
- BERRYMAN, A.A. & MILLSTEIN, J.A., 1989. *Are ecological systems chaotic - and if not why not?* TREE **4**: 26-8.
- CELLIER, F.E. & FISCHLIN, A., 1980. *Computer-assisted modelling of ill-defined systems*. In: Trappl, R., Klir, G.J. & Pichler, F.R. (eds.), *General Systems Methodology, Mathematical Systems Theory, Fuzzy Sets, Proc. of the Fifth European Meeting on Cybernetics and Systems Research, Vol. VIII*, 417-429, McGraw-Hill Intern. Book Comp., Washington, New York, 1982, 544pp.
- CODY, W.J., 1981. *Analysis of proposals for the floating-point standard*. IEEE Computer, **14** (3): 63-68.
- ENGELN-MÜLLGES, G. & REUTTER, F., 1988. *Formelsammlung zur Numerischen Mathematik mit MODULA 2-Programmen*. Wissenschaftsverlag, Mannheim a.o., 510pp.
- FISCHLIN, A., 1982. *Analyse eines Wald-Insekten-Systems: Der subalpine Lärchen-Arvenwald und der graue Lärchenwickler *Zeiraphera diniana* Gn. (Lep., Tortricidae)*. Diss. Eidg. Tech. Hochsch. Zürich, No. 6977, 294pp.
- FISCHLIN, A., 1986a. *Simplifying the usage and programming of modern workstations with Modula-2: The Dialog Machine*. Internal report, Project-Centre IDA, Swiss Federal Institute of Technology Zürich (ETHZ), Switzerland, 15pp.
- FISCHLIN, A., 1986b. *The "Dialog Machine" for the Macintosh..* Internal report, Project-Centre IDA, Swiss Federal Institute of Technology Zürich (ETHZ), Switzerland.
- FISCHLIN, A., 1991. *Interactive modeling and simulation of environmental systems on workstations*. In: Möller, D.P.F. (ed.), *Analysis of dynamic systems in medicine, biology, and ecology*. Proc. of the 4th Ebernburger Working Conference, April 5-7, 1990, Ebernburg, Bad Münster am Stein-Ebernburg, BRD, Informatik-Fachberichte 275, Springer, Berlin a.o.: 131-145.
- FISCHLIN, A., 1992. *Modellierung und Computersimulationen in den Umweltwissenschaften [Modelling and computer simulation in the environmental sciences]*. In: Schaufelberger, W. et al. (eds.), *Computer im Unterricht an der ETH Zürich, Bericht über das Projekt IDA (Informatik Dient Allen) 1986-1991*, 197pp., Zürich, Verlag der Fachvereine: 165-178.
- FISCHLIN, A., MANSOUR, M.A., RIMVALL, M. & SCHAUFELBERGER, W., 1987. *Simulation and computer aided control system design in engineering education*. In: Troch, I., Kopacek, P. & Breitenecker, F. (eds.), *Simulation of Control Systems*, Pergamon Press, 459pp., Oxford a.o., 51-60pp.
- FISCHLIN, A. & SCHAUFELBERGER, W., 1987. *Arbeitsplatzrechner im technisch-naturwissenschaftlichen Hochschulunterricht*. Bulletin SEV/VSE, **78** (Januar): 15-21.
- FISCHLIN, A. & ULRICH, M., 1987. *Interaktive Simulation schlecht-definierter Systeme auf modernen Arbeitsplatzrechnern: die Modula-2 Simulationssoftware ModelWorks*. Proceedings, Treffen des GI/ASIM-Arbeitskreises 4.5.2.1 "Simulation in Biologie und Medizin", February, 27-28, 1987, Vieweg, Braunschweig: 1-9.
- FISCHLIN, A. & BUGMANN, H., 1993. *Think globally, act locally! A small country case study in reducing net CO<sub>2</sub> emissions by carbon fixation policies*. In: Kanninen, M. (ed.), *Carbon balance of the world's*

forested ecosystems: Towards a global assessment. Publications of the Academy of Finland, VAPK Publishing, Helsinki: in print.

- FISCHLIN, A. & BUGMANN, H., 1994. *Können forstliche Massnahmen eine Beitrag zur Verminderung der schweizerischen CO<sub>2</sub>-Emissionen leisten?* Schweiz. Z. Forstwes., **145** (4): 275-292.
- FORRESTER, J.R., 1970. *Principles of systems*. Addison Wesley, N.Y.
- IEEE STD 754-1985, 1985. *IEEE standard for binary floating-point arithmetic*. New York: IEEE, Inc. or IEEE TASK P754, 1981. A proposed standard for binary floating-point arithmetic - Draft 8. IEEE Computer, **14** (3): 51-62.
- KELLER, D., 1989. *Introduction to the Dialog Machine*. Interner Bericht Nr. 5 (Nov.), Projekt-Zentrum IDA, Swiss Federal Institute of Technology Zürich (ETHZ), Switzerland, 37pp.
- KORN, G.A. & WAIT, J.V., 1978. *Digital continuous-system simulation*. Prentice-Hall, Englewood Cliffs, N.J., 212pp.
- KREUTZER, W., 1986. *System simulation: programming styles and languages*. Sydney a.o.: Addison-Wesley, 366pp.
- LOTKA, A.J., 1925. *Elements of physical biology*. Baltimore: Williams and Wilkins.
- LUENBERGER, D.G., 1979. *Introduction to dynamic systems - Theory, models, and applications*. Wiley, New York, 446pp.
- MANSOUR, M. & SCHAUFELBERGER, W., 1989. *Software and laboratory experiments using computers in control education*. IEEE Control Systems Magazine, **272** (April): 19-24.
- MAY, R.M. & OSTER, G.F., 1976. *Bifurcations and dynamic complexity in simple ecological models*. Am. Nat., **110**: 573-99.
- MAY, R.M. (ed.), 1981. *Theoretical ecology. Principles and applications*. Blackwell Scientific Publications, Osney Mead, Oxford, 2nd ed., 489pp.
- MAY, R.M., 1974. *Biological populations with nonoverlapping generations: stable points, stable cycles, and chaos*. Science, **186**: 645-7.
- MAY, R.M., 1975. *Biological populations obeying difference equations: stable points, stable cycles, and chaos*. J. Theor. Biol. **51**: 511-24.
- MAY, R.M., 1976. *Simple mathematical models with very complicated dynamics*. Nature **261**: 459-67.
- NEMECEK, T., 1993. *The role of aphid behavior in the epidemiology of potato virus Y: a simulation study*. Diss. ETH Zürich No. 10086, 232pp.
- PEARL, R., 1927. *The growth of populations*. Q. Rev. Biol., **2**: 532-548.
- ROBINSON, S.B., 1986. *STELLA - Modeling and simulation software for use with the Macintosh*, Byte: 277-278
- THOENY, J., FISCHLIN, A. & GYALISTRAS, D., 1994. *RASS<sup>1</sup>: Towards bridging the gap between interactive and off-line simulation*. Halin, J. (ed.), 1995, Proc. CISS 94, Springer, in prep.
- ULRICH, M., 1987. *ModelWorks. An interactive Modula-2 simulation environment*. Post-graduate thesis, Project-Centre IDA, Swiss Federal Institute of Technology Zürich (ETHZ), Switzerland, 53pp.
- VOLTERRA, V., 1926. *Variatione e fluttuazioni del numero d'individui in specie animali conviventi*. Mem. Accad. Nazionale Lincei (ser. 6) **2**: 31-113.
- WIRTH, N., 1985. *Programming in Modula-2, Third, Corrected Edition*. Springer-Verlag, Berlin a.o., 202pp.

---

<sup>1</sup>RASS is an acronym for RAMSES Simulation Server.

- WIRTH, N., 1988. *Programming in Modula-2*. Springer, Berlin a.o., 4th, corrected edition.
- WIRTH, N., GUTKNECHT, J., HEIZ, W., SCHÄR, H., SEILER, H. & VETTERLI, C., 1988. *MacMETH. A fast Modula-2 language system for the Apple Macintosh. User Manual*. 2nd ed. Institut für Informatik ETH Zürich, Switzerland, 100pp.
- WIRTH, N., GUTKNECHT, J., HEIZ, W., SCHÄR, H., SEILER, H., VETTERLI, C. & FISCHLIN, A., 1992. *MacMETH. A fast Modula-2 language system for the Apple Macintosh. User Manual*. 4th. completely revised ed., Departement Informatik ETH Zürich, Switzerland, 116pp.
- WYMORE, A.W., 1984. *Theory of Systems*. In: VICK, C. R., RAMAMOORTHY, C. V.(EDS.): *Handbook of Software Engineering*, Van Nostrand Reinhold Company, New York, 1984
- ZEIGLER, B. P., 1976. *Theory of Modelling and Simulation*, John Wiley & Sons.
- ZEIGLER, B. P., 1984. *System Theoretic Foundations of Modelling and Simulation*. In: Ören, T. I., Zeigler, B. P., Elzas, M. S.(eds): *Simulation and Model-Based Methodologies: An Integrative View*, Springer-Verlag.

## C ModelWorks Versions and Implementations

The ModelWorks version described in this text is version V2.2 finalized in spring 1994. There exist in fact five, slightly differing implementations or versions of ModelWorks:

- 1) The standard Macintosh version V2.2. Runs on all Macintosh computers with at least 1 MBytes of main memory and offers all functions as described in this text without any restrictions.
- 2) The Reflex Macintosh version V2.0/Reflex. It is a reduced subset from the standard version and runs on 512KBytes machines like the Macintosh Reflex (Mac 512KE). The following restrictions apply: no graph printing except screen dumps, no clipboard support, and no dumping of graphs onto the stash file. Colors are available on color screens and on printer systems which support color screen dumps. However the simulation environment mode "restore graph with colors" is not available.
- 3) The IBM PC GEM-Version V1.1/PC. It is also a reduced subset from the standard Macintosh version. Besides the same restrictions which apply to the Reflex Macintosh version, this version can not support colors. This is because of the MS DOS memory limitation of 640 KBytes. Furthermore this version requires static linking.
- 4) The IBM PC Windows-Version V2.2/PC. It is functionally equivalent with the standard Macintosh version and runs on every machine capable of running MS Windows 3.1. This version requires static linking.
- 5) The Macintosh II version V2.2/II. It is functionally identical with the standard version but takes full advantage of the Motorola 68020, 68030 or 68040 32-Bit CPU and the mathematical coprocessors Motorola 68881, 68882. It is faster, however, it runs only on Macintosh II, Quadra, or other similar models, given the machine is equipped with a floating point unit (FPU).

For the Macintosh ModelWorks is distributed as part of the RAMSES<sup>1</sup> software package, for the IBM PC only as ModelWorks alone. In both cases ModelWorks is released together with the "Dialog Machine". All mentioned software can be obtained via anonymous internet file transfer ftp (at no charge) from the host *ftp.ito.umnw.ethz.ch* (current internet address 129.132.80.130) in ftp directory */pub/mac/RAMSES* or */pub/pc/RAMSES*. For details on the installation and the software architecture see the separate booklet "*Installation Guide and Technical Reference of the RAMSES software*" distributed together with the RAMSES software package.

The usage of the software for noncommercial purposes is free and unrestricted as long as the authorship of the used software is stated clearly on any redistributed model or other program, i.e. any product descriptions or labels must state in writing that the "Interactive ModelWorks Simulation Software by A. Fischlin *et al.* from the Swiss Federal Institute of Technology Zürich ETHZ" has been used to develop the product. All copyrights are reserved and are held by the authors and the Swiss Federal Institute of Technology Zürich ETHZ. ModelWorks may not be sold, nor included in any sold product as an incentive, nor otherwise redistributed for a profit without prior written consent by the authors and the Swiss Federal Institute of Technology Zürich ETHZ. Please keep the software and the documentation together!

---

<sup>1</sup>RAMSES is an acronym for Research Aids for Modeling and Simulation of Environmental Systems. For more information on the concepts of RAMSES see FISCHLIN (1991).

## D Use and Definitions of ModelWorks and Library Modules

### D.1 MODEL WORKS MANDATORY CLIENT INTERFACE

The definition modules belonging to the mandatory client interface of ModelWorks, i.e. the modules *SimBase* and *SimMaster*, are not listed here. Please consult chapter *Client Interface* from the part III *Reference* instead, since all objects exported by *SimBase* and *SimMaster* are already fully described there. Note, for the reader's convenience, the chapter *Quick References* lists the modules *SimBase* and *SimMaster* once more fully; thus, in order to gain a good overview over the whole client interface of ModelWorks, consult the ModelWorks' quick reference.

### D.2 MODEL WORKS OPTIONAL CLIENT INTERFACE

#### D.2.1 *SimEvents*

The module *SimEvents* is needed to work with models of the type discrete event system (DEVS) as described in the chapter *Model Formalisms* in part II *Theory*. This module extends the client interface of ModelWorks, is optional, but has to be used whenever the modeler wishes to implement DEVS. For a typical usage see the sample model *Diversity*, the submodel *CPTraffic* of the structured model definition program *CarPollution*, or the research sample model *ForestYield*.

DEFINITION MODULE *SimEvents*;

(\*\*\*\*\*

Module *SimEvents* (MW\_V2.2)

Copyright (c) 1993 by Andreas Fischlin and Swiss  
Federal Institute of Technology Zürich ETHZ

Purpose Support for discrete event simulations (DEVS)  
according to the event scheduling approach

This module is part of the optional client interface of  
"ModelWorks", an interactive Modula-2 modelling  
and simulation environment.

Programming

- o Design
  - A. Fischlin 7/Mar/93
- o Implementation
  - A. Fischlin 7/Mar/93

Systems Ecology  
Institute of Terrestrial Ecology  
Department of Environmental Sciences  
Swiss Federal Institute of Technology Zurich ETHZ  
Grabenstr. 3  
CH-8952 Schlieren/Zurich  
Switzerland

Last revision of definition: 21/Mar/94 AF

```

***** )

FROM SYSTEM IMPORT ADDRESS, BYTE;
FROM SimBase IMPORT Model;

(***** )
(* Discrete event classes *)
(***** )

CONST
  minEventClass = 0;
  maxEventClass = 3000;
  unknownEventClass = maxEventClass;

TYPE
  EventClass = [minEventClass..maxEventClass];
  (*
   Each state transition of a DEVS is characterized by a
   particular discrete event class. A DEVS owns a finite set of
   event classes. Each event class must be positive and unique
   within the simulation environment and must be declared and
   associated with a given state transition function via a data
   structure of type StateTransition. The set of event classes
   respectively state transition functions belonging to a DEVS
   are declared when calling procedure DeclDiscEvtM. Event
   classes are mainly useful if another model wishes to produce
   an event output. Such a model, e.g. a continuous time (DESS)
   or discrete time model (SQM), may do so by scheduling the
   event output together with the appropriate event class (using
   procedure ScheduleEvent). If the simulation time is advanced
   to the time the event is due, ModelWorks will then dispatch
   the event to the appropriate state transition function.
  *)

  Transaction = ADDRESS;
  (*
   Every discrete event may be associated with a particular set
   of data, the transaction. E.g. arriving customers may be
   described by several attributes such as sex, age, demand
   etc. Use nilTransaction to schedule or handle data-less
   events.
  *)

TYPE
  StateTransitionFunction = PROCEDURE (Transaction);
  StateTransition = RECORD
    ec: EventClass;
    fct: StateTransitionFunction;
  END;
  (*
   Associates state transition function fct with the event class
   cl. Typically a state transition function changes
   instantaneously the state of a DEVS if a corresponding event
   is encountered.
  *)

VAR
  nilTransaction: Transaction; (* read only! *)
  noStateTransition: ARRAY [0..0] OF StateTransition; (* read only! *)

PROCEDURE AsTransaction(VAR d: ARRAY OF BYTE): Transaction;
  (*
   Converts any data structure into a Transaction.
   Example:

   ScheduleEvent(ec,tau,AsTransaction(myGlobObject));
  *)

```

schedules an event of class `ec` operating on the transaction `myGlobject`. `myGlobject` has been declared as a global variable and is of type `ObjectDescriptor`, the latter having been declared similar to this:

```
TYPE ObjectDescriptor = RECORD
    x,y: INTEGER;
    r: REAL;
    ...
END;
```

Fields of the transaction may then be accessed from within the state transition function associated with the event class `ec` as follows:

```
PROCEDURE MyStatetransFct (alfa: Transaction);
    VAR theObj: ObjectDescriptor;
BEGIN
    theObj := alfa;
    WITH theObj^ DO
        IF x=y THEN r := ...
        ...
    END(*WITH*);
END MyStatetransFct;
```

Make sure that the transaction exists not only during scheduling, but also when it becomes due; otherwise the state transition function is likely to corrupt your program.

\*)

```
PROCEDURE EventClassExists(ec: EventClass): BOOLEAN;
(*
    Tests whether any DEVS has been declared to ModelWorks which does
    provide state transitions for the event class ec.
*)
```

```
(*****
(* Declaration of discrete event models (DEVS) *)
*****)
```

```
VAR
    dummyDEVChg: REAL;
(*
    Use this dummy variable instead of the formal parameter ds
    (Derivative or NewState) tau declaring state variables
    belonging to a discrete event model (see procedure DeclSV
    from module SimBase.
*)
```

```
PROCEDURE DeclDEVM(VAR m: Model; initialize, input, output: PROC;
    statetransfct: ARRAY OF StateTransition; terminate,
    declModelObjects: PROC; descriptor, identifier: ARRAY OF CHAR;
    about: PROC);
```

(\*

Declares a discrete event model (DEVS). The array `statetransfct` contains for every event class the corresponding state transition function. For all other formal parameters see `DeclM` from module `SimBase`. The integration method will appear as `discreteEvent` (see `IntegrationMethod` from module `SimBase`). StateTransitions remain known to ModelWorks as long as owner model remains declared.

IMPLEMENTATION RESTRICTION: During simulations event classes can't be

reused, e.g. by removing a model (using SimBase.RemoveM) and immediately reusing the same event classes for another model by calling DeclDEVM. This implies that every event class has to be uniquely associated with a single model during the entire course of a simulation run.

\*)

```
PROCEDURE GetDefltdEVM(VAR m: Model; VAR initialize, input, output: PROC;
    VAR statetransfct: ARRAY OF StateTransition; terminate: PROC;
    VAR descriptor, identifier: ARRAY OF CHAR; VAR about: PROC);
PROCEDURE SetDefltdEVM(VAR m: Model; initialize, input, output: PROC;
    statetransfct: ARRAY OF StateTransition; terminate: PROC;
    descriptor, identifier: ARRAY OF CHAR; about: PROC);
```

```
(*****
(* Event scheduling *)
*****)
```

```
CONST
    always = MIN(REAL);
    never = MAX(REAL);
```

```
VAR
    schedulingDone: BOOLEAN;
```

```
PROCEDURE InitEventScheduler;
(*
    Clears the event scheduling mechanism, i.e. the event
    scheduling queue, of the simulation environment. Any
    eventually still pending events will be discarded. Then it
    makes the scheduling mechanism ready to accept events always by
    calling SchedulingOnlyAfter(always).
*)
```

```
PROCEDURE ScheduleEvent(ec: EventClass; tau: REAL; alfa: Transaction);
(*
    Schedules the event of class ec for transaction alfa. Use
    nilTransaction to schedule an event without a transaction, i.e.
    without any data and attributes. The event will be due after
    time tau has elapsed. The event can only be successfully
    scheduled if the following condition is satisfied (t+tau) >=
    tmin. If the scheduling was successful => schedulingDone = TRUE.
*)
```

```
PROCEDURE NextEventAt(): REAL;
(* Returns the time (ts+tau) at which the next pending event is due. *)
```

```
PROCEDURE ProbeNextPendingEvent(VAR ec: EventClass; VAR when: REAL;
    VAR alfa: Transaction);
(*
    Retrieves the characteristics of the next pending event. The
    time when (ts+tau) is the due time.
*)
```

```
PROCEDURE GetNextPendingEvent (VAR ec: EventClass; VAR when: REAL;
    VAR alfa: Transaction);
(*
    Retrieves the characteristics and removes the next pending
    event from the event scheduling queue.
*)
```

```
PROCEDURE PendingEvents(): INTEGER;
(* Returns the total number of currently pending events *)
```



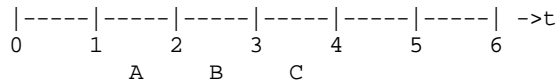
```
PROCEDURE SchedulingOnlyAfter(tmin: REAL);
(*
  Disallows the scheduling of any events with a due time <=
  tmin. Once this routine has been called, ScheduleEvent is only
  succesful, if it schedules events with a due time > tmin.
*)

PROCEDURE DiscardEventsAfter(ec: EventClass; aftert: REAL; alfa: Transaction);
(*
  Discards from the event scheduling queue all events for event
  class ec, due after the time aftert, and which operate on the
  transaction alfa. Note: events with a due time = aftert are
  also discarded. (event is only really discarded if alfa is
  the same as the scheduled transaction!)
*)

PROCEDURE DiscardEventsBefore(beforet: REAL);
(*
  discards from the event scheduling queue all events for which
  the due time < beforet. Note: events with a due time = beforet
  are not discarded)
*)

END SimEvents.
```





Validation statistics example (see fig. above):  
 Y is e.g. a state variable, x = simulated, v = observed or measured; t stands for an independent variable e.g. time.

A series of n (v,t) points has to be declared by DeclDispData from module SimGraphUtils. As a goodness of fit criterion we look for the vertical distances (d) of simulated (linearly interpolated) results (x) and the observed data (v). There may occur 3 cases (see fig. above): A) one observed data point falls into the last simulated time interval; B) no observed data point was encountered during last time interval; C) many observed data points were encountered during the last time interval. The procedures "AccuDelta" and "GetDelta" keep track on the last independent variable and look ahead to the next element in the data array if a new has to be computed. This requires correct sorting of the data array before declaration!

\*)

TYPE

DeltaVar;

```
DeltaProc = PROCEDURE ( (*ySim~*)REAL, (*yData*)REAL ): REAL;
(* ySim~ denotes simulated y interpolated at position xData,
  yData denotes the y value from the installed data series at
  position xData *)
```

VAR defaultDelta: DeltaProc;

```
PROCEDURE InstallDeltaProc( VAR mvDepVar: REAL; compDelta: DeltaProc );
```

```
PROCEDURE InitDeltaStat( VAR mvDepVar: REAL; xSim, ySim: REAL;
  VAR dv: DeltaVar );
```

(\* initializes the internal variables which hold the wanted statistics.  
 - call InitDeltaStat from within your "Initial" procedure,  
 note: set xSim to its actual value at t0 before!  
 mvDepVar should be declared previously with DeclDispData;  
 returns dv for later reference when calling AccuDelta \*)

```
PROCEDURE AccuDelta( dv: DeltaVar; xSim, ySim: REAL );
```

(\* This procedure accumulates simple statistics intermediates such as:  
 $\int y^2 dx$ ;  $\int y dx$ ;  $\int |y| dx$ ; n; where  $\int$  is the difference between  
 the installed data series (DeclDispDat) and the interpolated  
 simulated variable (  $\int$  is computed by means of the installed DeltaProc)  
 and n holds the count of accumulated  $\int$  s.  
 - AccuDelta should be called once for each time step, i.e. normally  
 in procedure "output" of your model;  
 assumes that dv is correct! \*)

```
PROCEDURE GetDeltaStat( VAR mvDepVar: REAL;
```

```
  VAR sumY, sumY2, sumAbsY: REAL;
```

```
  VAR count: INTEGER );
```

(\* allows you to get the stored statistics which were accumulated  
 since the last InitDeltaStat (normally a simulation run, see AccuDelta  
 for more details). \*)

```
PROCEDURE SetDeltaStat( VAR mvDepVar: REAL;  
                        sumY, sumY2, sumAbsY: REAL; count: INTEGER );  
(* allows to set these statistics. This procedure can be usefull  
if you have to resume an interrupted run or ev. for a complete reset *)
```

```
PROCEDURE WriteDeltaStatMsg( VAR mvDepVar: REAL );  
(* writes the stored statistics for each variable to the table window  
and to the stashfile in form of a message. *)
```

```
END SimDeltaCalc.
```

### D.2.3 SimGraphUtils

This module allows first to make output in ModelWorks' window *Graph*, e.g. to draw additional curves or any other graphical elements. This may be useful to show measured time series together with simulated results, to draw error bars or non-standard symbols etc. Thirdly does this module allow to place the window *Graph* always on that screen which has colors and the highest resolution; this feature is particularly useful when running ModelWorks model definition programs on different computer systems, among which some have more than one screen, e.g. one black and white only and one with colors. Thirdly, this module supports window input, i.e. allows to detect mouse clicks in the window *Graph*. The latter can be used to determine points such as an initial state vector in the state space of a 2nd order system, e.g. to explore interactively a phase portrait. For a typical usage of this optional client interface module see the sample models *Lorenz*<sup>1</sup>, *GauseIdentif*, *LVPhasePlot*, *StochLogGrow* for graphical output, plus module *VDPol* for mouse input.

```
DEFINITION MODULE SimGraphUtils;
```

```
(*****
```

```
Module SimGraphUtils      (MW_V2.2)
```

```
    Copyright 1989 by Olivier Roth and Swiss
    Federal Institute of Technology Zuerich ETHZ
```

```
    Purpose: Provides some utilities to make I/O to the graph window and the
             graph of the modelling and simulation environment "ModelWorks".
```

```
    This module is part of the optional client interface of
    "ModelWorks", an interactive Modula-2 modelling
    and simulation environment.
```

```
    Remarks: Most procedures behave similar to those of the module DM2DGraphs
             and may now be combined with many procedures from DMWindIO.
             The window and its associated graph are objects of the ModelWorks
             environment and should therefore not be removed.
```

```
    Programming
```

- o Programming and Implementation
  - O. Roth                    12.09.89
- o Implementation
  - O. Roth                    12.09.89

```
    Systems Ecology
    Institute of Terrestrial Ecology
    Department of Environmental Sciences
    Swiss Federal Institute of Technology Zurich ETHZ
    Grabenstr. 3
    CH-8952 Schlieren/Zurich
    Switzerland
```

```
    Last revision of definition: 22/04/96  af
```

```
*****)
```

```
FROM SimBase IMPORT MWWindowArrangement, Model,
                   Stain,LineStyle, Graphing;
FROM DMWindIO IMPORT Color;
```

---

<sup>1</sup>Only distributed but not listed in this *Appendix*

```

FROM Matrices IMPORT Matrix;

TYPE
  Curve;

VAR
  nonexistent: Curve; (* read only! *)

(* -----
Procedure to arrange the ModelWorks Windows on a multi-screen machine
----- *)

PROCEDURE PlaceGraphOnSuperScreen(deflwa: MWWindowArrangement);
(* Defines the default window arrangement according to 'deflwa'
and places the ModelWorks graph window on the largest color
screen in case the Model Definition Program is running on a
multi-screen machine. *)

(* -----
Procedure to access the ModelWorks 'Graph' WINDOW:
----- *)

PROCEDURE SelectForOutputGraph;
(* This procedure brings the ModelWorks 'Graph' window to front
and makes it the current output window. This allows subsequently
calls to almost all of the I/O procedures of the 'Dialog
Machine' module 'DMWindIO'. *)

(* -----
Procedures to access the GRAPH in the 'Graph' window similar to
the routines exported by module DM2DGraphs (see 'Dialog Machine'):
----- *)

PROCEDURE DefineCurve( VAR c: Curve;
                      col: Stain; style: LineStyle; sym: CHAR );
(* Every curve has its own plotting style and color. This allows
for the simultaneous drawing of an arbitrary number of curves
within the ModelWorks graph. sym specifies a character which is
drawn repeatedly at the data points, they help identifying a
curve (sym = 0C, no mark is plotted).
Use this procedure also if you want to alter an already existing
curve. *)

PROCEDURE RemoveCurve( VAR c: Curve );
(* This procedure removes a curve definition. This procedure sets c
to nonexistent. *)

PROCEDURE DrawLegend( c: Curve; x, y: INTEGER; comment: ARRAY OF CHAR );
(* Draws a portion of curve c with the current attributes at position
x and y and writes the comment to the right of c. After this procedure
the pen location is just to the right of the string "comment", so it's
possible to add for example values of parameters by calling DMWindIO
procedures WriteReal (etc.) just after this procedure. *)

PROCEDURE Plot( c: Curve; newX, newY: REAL );
(* You can plot (draw a curve) from the last (saved) position to the point
specified by the new coordinates newX and newY.
Note: ModelWorks resets the pen position when clearing the graph.
Errors: If the point specified by newX and newY lies outside the integer
(pixel) range DM2DGraphsDone will be set to FALSE. *)

```

```
PROCEDURE Move( c: Curve; newX, newY: REAL );
(* moves the pen to position (x,y). Typically used to draw several curves
with the same attributes to reset the pen position after having drawn a
curve.
Errors: If the point specified by x and y lies outside the integer (pixel)
range DM2DGraphsDone will be set to FALSE. *)
```

```
PROCEDURE PlotSym( x, y: REAL; sym: CHAR );
(* draws the symbol sym at the position (x,y). May be used as an alternate
method to make scatter grams.
Errors: If the point specified by x and y lies outside the integer (pixel)
range DM2DGraphsDone will be set to FALSE. *)
```

```
PROCEDURE PlotCurve( c: Curve; nrOfPoints: CARDINAL; x, y: ARRAY OF REAL );
(* Plots an entire sequence of nrOfPoints coordinate pairs contained within
the two vectors x and y. May also be useful to implement an update mechanism.
Errors: - If the point specified by x and y lies outside the integer (pixel)
range DM2DGraphsDone will be set to FALSE.
- If the maximum number of elements of x or y is less than nrOfPoints,
then only the lower number of elements of either x or y will be
plotted. WARNING: The x and y arrays are value parameters,
hence require sufficient stack size at run time. The design of
this routine is for curves of a rather small dimension. To
plot large data sets use instead of PlotCurve the procedure
DeclDispDataM (see below). *)
```

```
PROCEDURE GraphToWindowPoint( xReal, yReal: REAL;
VAR xInt, yInt: INTEGER );
(* Calculates the pixel coordinates (xInt and yInt) of the
graph's window (see WindowIO) from the specified graph
coordinates (xReal and yReal). Note that the vertical axis of the
ModelWorks graph is transformed to yMin = 0.0 and yMax = 1.0 (see
also procedure MVValToPoint).
Errors: If the point specified by xReal and yReal lies outside
the integer (pixel) range, DM2DGraphsDone will be set to
FALSE and xInt and yInt is set to MIN(INTEGER) or
MAX(INTEGER) respectively. *)
```

```
PROCEDURE WindowToGraphPoint( xInt, yInt: INTEGER;
VAR xReal, yReal: REAL );
(* Calculates graph coordinates (xReal and yReal) from the
specified pixel coordinates (xInt and yInt) of the graph's window
(see WindowIO). Note that the vertical axis of the ModelWorks
graph is transformed to yMin = 0.0 and yMax = 1.0 (see also
procedure PointToMVVal).
Errors: If the point specified by xReal and yReal lies outside the
integer (pixel) range, DM2DGraphsDone will be set to FALSE
and xInt and yInt is set to MIN(INTEGER) or MAX(INTEGER)
respectively. *)
```

```
(* -----
Drawing procedures used in a ModelWorks aware context:
----- *)
```

```
PROCEDURE InstallGraphClickHandler(gch: PROC);
(* Installs the mouse click handler procedure gch into the
ModelWorks simulation environment. After successful
installation, each time the simulationist clicks into the graph
window, gch will be called and a pair of xpixel coordinates [x,y]
where the mouse click occurred, are passed to the handler. Use
```

procedures such as PointToMVVal to interpret the meaning of the point [x,y] in terms of monitorable variables. \*)

VAR

timeIsIndep: REAL;

PROCEDURE PointToMVVal(xInt,yInt: INTEGER; m: Model; VAR mv: REAL;  
VAR curG: Graphing): REAL;

(\* Returns the corresponding value of the monitorable variable mv of the model m from the given pixel coordinates (xInt and yInt) of the ModelWorks graph window. As a side effect the routine returns also the current graphing of the mv. In case the mv should currently not be in display (curG=notInGraph), the value is returned as if curG would have been isY. To denote the independent variable time, use timeIsIndep as the actual parameter for mv (see also procedure WindowToGraphPoint).  
Errors: If m or mv should not be known to ModelWorks' model base, the routine displays an appropriate error message and returns 0.0 and curG=notInGraph.  
If the point specified by xReal and yReal lies outside the integer (pixel) range, DM2DGraphsDone will be set to FALSE and xInt and yInt is set to MIN(INTEGER) or MAX(INTEGER) respectively. \*)

PROCEDURE MVValToPoint(val: REAL; m: Model; VAR mv: REAL;  
VAR curG: Graphing): INTEGER;

(\* Returns the pixel coordinate for the window Graph (see WindowIO) from the specified coordinate val interpreted for the monitorable variable mv of the model m. As a side effect the routine returns also the current graphing of the mv. In case the mv should currently not be in display (curG=notInGraph), the value is returned as if curG would have been isY. To denote the independent variable time, use timeIsIndep as the actual parameter for mv (see also procedure GraphToWindowPoint).  
Errors: If m or mv should not be known to ModelWorks' model base, the routine displays an appropriate error message and returns 0 and curG=notInGraph.  
If the point specified by val lies outside the integer (pixel) range, DM2DGraphsDone will be set to FALSE and the routine returns either MIN(INTEGER) or MAX(INTEGER) respectively. \*)

PROCEDURE TimeIsX() : BOOLEAN;

(\* Above procedure returns whether time is the current abscissa (x axis). \*)

TYPE

Abscissa = RECORD isMV: POINTER TO REAL; xMin,xMax: REAL END;

PROCEDURE CurrentAbscissa(VAR a: Abscissa);

(\* Returns a pointer (isMV) to the monitorable variable currently used as abscissa and its extremes (xMin~curScaleMin,xMax~curScaleMax). In case that time is in use, isMV will point to timeIsIndep \*)

(\* - - - - -  
Procedures to convert different Color Types:  
- - - - - \*)

PROCEDURE StainToColor( stain: Stain; VAR color: Color );  
PROCEDURE ColorToStain( color: Color; VAR stain: Stain );  
(\* Translates Stain from module SimBase to Color from module DMWindIO and vice versa; exception for StainToColor: autoDefCol is translated to black. \*)



```
(* -----
Display data series (e.g. for validation) all at once:
----- *)
```

```
(*
Follow these steps to use the data display feature of that module:
1. Declare an ordinary monitorable variable with the procedure 'DeclMV'
   as a "master" monitorable variable for data arrays to be
   declared later (see next step). Several properties, i.e. descr,
   ident, unit, (and curve attributes as color, linestyle, symbol)
   will be inherited by the later associated data arrays. So if the
   monitorable variable's graphing variable is set 'isY' the data are
   selected to be displayed. (This mv is called "master"-mv in what
   follows)
2. Since the data arrays symbol (CHAR), line style (LineStyle) and
   color (Stain) will be taken from the "master" monitorable variable
   you can call 'SetCurveAttrForMV' and ev. 'SetDeflCurveAttrForMV'.
3. Declare the associated data arrays with the "master" monitorable
   variable, the independent monitorable variable, and all the data
   arrays with a call to 'DeclDispData'.
4. To enable the display mechanism the monitorable variable mvDepVar
   must be isY and mvIndepVar must be isX. If another monitorable
   variable represents the current x axis then nothing can be
   displayed.
5. ModelWorks will display automatically all declared data in the
   normal graph of the "Graph" window at the specified moment,
   i.e. typically at InitMonitoring, or at TermMonitoring. To
   allow for a general control of the moment of display the
   procedure 'DisplayDataNow' and 'DisplayAllDataNow' are also
   exported.
```

Caution:

- Be sure to follow the steps given above in the correct order (1 before 3!) or no data can be declared and displayed.
- Do not assign any values to the "master" monitorable variable to avoid conflicts with the data declaration.
- Setting writeInTable or writeOnFile of the "Master" monitorable variable is not prohibited but makes no sense, since a dummy value NAN(017) and not the data series will be displayed.

```
*)
```

TYPE

```
DisplayTime = ( showAtInit, showAtTerm, noAutoShow );
DispDataProc = PROCEDURE( Model, VAR REAL );
```

```
PROCEDURE DeclDispData( mDepVar      : Model;  VAR mvDepVar  : REAL;
                       mIndepVar    : Model;  VAR mvIndepVar: REAL;
                       x, v,
                       vLo, vUp     : ARRAY OF REAL;
                       n             : INTEGER;
                       withRangeBars: BOOLEAN;
                       dispTime     : DisplayTime );
```

(\* In order to display a data series (e.g. validation data) f.ex. before a simulation run, the necessary data have to be declared beforehand, i.e. normally just at the end of all other ModelWorks objects declarations. The variables are as follows:

```
mDepVar      : model to which belongs the mvDepVar
mvDepVar     : monitorable variable representing the dependent data array
mIndepVar    : model to which belongs the mvIndepVar
mvIndepVar   : monitorable variable representing the independent data array,
              if mvIndepVar is specified
              "timeIsIndep" (or is not a declared monitorable var), then
              "time" is assumed to be the independent variable,
x            : array of independent values,
v            : array of dependent values,
```

```
vLo          : array of lower e.g. confidence or range values,
vUp          : array of upper e.g. confidence or range values,
n            : number of given data,
withRangeBars: flag, if TRUE range bars will be drawn using vLo and vUp,
dispTime     : the time when the data should be displayed,
```

Note:

The curve attributes of the data to display can be set through the procedure 'SetCurvAttrForMV' on the monitorable variable 'mvDepVar' and the default strategy for curve attributes assignments are the same as for ordinary monitorable variables for color and symbol but not for the lineStyle:

the default line style is hidden which means that the connections from [x,v]-point to [x,v]-point are not drawn. In that case and if withRangeBars is set true then the error bars are displayed solidly. All other line styles are applied to the connections from point to point as well as to the error bars themselves.

This procedure allows also to redeclare such data series, i.e. to associate other data to the same mvDepVar and mvIndepVar.

WARNING: The x, v, vLo, vUp arrays are value parameters, hence require sufficient stack size at run time. The design of this routine is for vectors of a rather small dimension. To plot large data sets use instead of this routine the procedure DeclDispDataM (see below).

\*)

```
PROCEDURE DeclDispDataM( mDepVar      : Model;  VAR mvDepVar  : REAL;
                        mIndepVar    : Model;  VAR mvIndepVar: REAL;
                        data         : Matrix;
                        withRangeBars: BOOLEAN;
                        dispTime     : DisplayTime );
(* alternate form of DeclDispData (described above) using type Matrix to pass
 * the data (x = col 1, v = col 2, vLo = col 3, vUp = col 4) *)
```

```
PROCEDURE DisplayDataNow( mDepVar : Model;  VAR mvDepVar  : REAL );
(* This procedure allows to display a series of e.g. validation data
before a simulation run. The previously declared data are displayed
in the current graph window under the following conditions:
+ the data have been declared properly and are valid;
+ the associated monitorable variable is selected to be displayed (isY);
+ the declared indepVar is the currently active independent
  monitorable variable (isX);
+ the declared indepVar is either not a monitorable variable (for
  example 'timeIsIndep' what implies that time is meant) and time is
  the selected independent var;
+ the data fall into the declared scaling range;
*)
```

```
PROCEDURE DisplayAllDataNow;
(* Displays all declared datasets at the specified moments. The same conditions
apply as for 'DisplayDataNow'.
*)
```

```
PROCEDURE DoForAllDispData( p: DispDataProc );
(* Calls procedure p for all DispData currently declared. Be
careful when using this procedure, since it allows to access
also DispData-definitions which may not belong to the caller.
*)
```

```
PROCEDURE RemoveDispData( mDepVar : Model;  VAR mvDepVar  : REAL );
(* This procedure allows to free the memory from the declared data
to display.
```

\*)

```
PROCEDURE SetDispDataM( mDepVar: Model; VAR mvDepVar: REAL; data: Matrix );  
PROCEDURE GetDispDataM( mDepVar: Model; VAR mvDepVar: REAL; VAR data: Matrix );  
(* these procedures allow to set/retrieve the installed data through matrices *)
```

END SimGraphUtils.

### D.2.4 SimIntegrate

The optional client interface module *SimIntegrate* can be used to compute definite integrals, i.e. to solve the equations of a particular model without advancing ModelWorks' global independent variable *t*. Instead a lower and upper boundary of the independent variable is used and such a numerical integration can be called always, e.g. in the client procedure *initialize*. Note however, that this is only possible for autonomous models, since the integration is performed for a single model only.

DEFINITION MODULE SimIntegrate;

(\*\*\*\*\*)

Module SimIntegrate (MW\_V2.2)

Copyright 1989 by Andreas Fischlin and Swiss  
Federal Institute of Technology Zuerich ETHZ

Purpose Provides means to integrate an autonomous  
differential equation system without any  
monitoring

This module is part of the optional client interface of  
"ModelWorks", an interactive Modula-2 modelling  
and simulation environment.

Programming

- o Design  
A. Fischlin 26/06/89
- o Implementation  
A. Fischlin 26/06/89

Systems Ecology  
Institute of Terrestrial Ecology  
Department of Environmental Sciences  
Swiss Federal Institute of Technology Zurich ETHZ  
Grabenstr. 3  
CH-8952 Schlieren/Zurich  
Switzerland

Last revision of definition: 26/06/89 af

(\*\*\*\*\*)

FROM SimBase IMPORT Model;

PROCEDURE Integrate ( m: Model; from, till: REAL);

(\*  
Computes the definite integral of the autonomous model *m*  
within the boundaries *from* and *till*. It integrates the model  
equations with the current integration method associated with  
the model *m*. The integration will be performed for every  
state variable and as initial values ModelWorks will use the  
current initial values associated with the declared state  
variables. Either stopping the simulation permanently (*kill*)  
or encountering the termination condition will stop the  
integration.  
\*)

END SimIntegrate.



### D.2.5 SimObjects

The optional client interface module *SimObjects* supports advanced uses of ModelWorks in the following ways: First it provides mechanisms to attach attributes to any model or model object currently declared in ModelWorks' data base. Typical attributes are a model or state variable index or a pointer to a data structure maintained by the client. Of course this module provides also procedures to access such attributes. Moreover, *SimObjects* allows to access directly ModelWorks' internal data structures of models and model objects. This may be important if the modeler wishes to use ModelWorks' objects in algorithms which use dynamic lists or for efficiency reasons. A typical example of such a functionality supported by this module is the procedure *MinimizeAfterDialog* from the auxiliary library module *IdentifyPars* as used in the sample model *GauseIdentif*.

DEFINITION MODULE SimObjects;

(\*\*\*\*\*

Module SimObjects (MW\_V2.2)

Copyright 1991 by Dimitrios Gyalistras and Swiss  
Federal Institute of Technology Zuerich ETHZ

Purpose Provides an access to the Model- and Model Object- base  
of ModelWorks as well as procedures to attach reference  
attributes to ModelWorks objects.

This module is part of the optional client interface of  
"ModelWorks", an interactive Modula-2 modelling  
and simulation environment.

Programming

- o Design
 

D. Gyalistras	25/07/91
O. Roth	09/10/91
A. Fischlin	27/11/91
- o Implementation
 

D. Gyalistras	25/7/91
O. Roth	09/10/91

Systems Ecology  
Institute of Terrestrial Ecology  
Department of Environmental Sciences  
Swiss Federal Institute of Technology Zurich ETHZ  
Grabenstr. 3  
CH-8952 Schlieren/Zurich  
Switzerland

Last revision of definition: 02/03/93 dg

\*\*\*\*\* )

FROM SYSTEM IMPORT ADDRESS;  
FROM DMStrings IMPORT String;  
FROM SimBase IMPORT Model;

TYPE  
RefAttr;

VAR  
aDetachedRefAttr: RefAttr; (\* read only variable \*)

```

PROCEDURE AttachRefAttrToModel (m: Model; VAR a: RefAttr; val: ADDRESS);
PROCEDURE DetachRefAttrFromModel(m: Model; VAR a: RefAttr );

PROCEDURE AttachRefAttrToObject (m: Model; VAR o: REAL; VAR a: RefAttr; val: ADDRESS);
PROCEDURE DetachRefAttrFromObject(m: Model; VAR o: REAL; VAR a: RefAttr );

PROCEDURE FindModelRefAttr (m: Model; VAR a: RefAttr);
PROCEDURE FindObjectRefAttr(m: Model; VAR o: REAL; VAR a: RefAttr);

PROCEDURE SetRefAttr(a: RefAttr; val: ADDRESS);
PROCEDURE GetRefAttr(a: RefAttr): ADDRESS;
(*
  You may associate with any model or model object an address
  attribute by calling AttachRefAttrToModel respectively
  AttachRefAttrToObject. The attribute's value may then be
  freely used via SetRefAttr for assignments or GetRefAttr for
  retrieval purposes. RefAttrs are particularly useful when using
  one of the SimBase.DoForAllXYZ procedures. Note that in case
  there is currently no attribute attached to a model or object,
  the value aDetachedRefAttr is passed by ModelWorks. It is also
  possible to access an attribute via model respectively model
  plus object by the procedures FindModelRefAttr respectively
  FindObjectRefAttr. Note however, that the latter method is
  less efficient and is therefore not recommended in heavy
  number-crunching simulations. Again aDetachedRefAttr is
  returned in case there is currently no attribute attached.
*)

PROCEDURE CurCalcMRefAttr(): ADDRESS;
(*
  Returns first attribute associated to the model of which the
  initialize, input, output, or dynamic etc. procedure is
  currently calculated. The value NIL is returned if
  (SimMaster.MWState <> simulating) or (SimMaster.MWSubState <>
  running), or if no attribute has been attached to the model.
*)

PROCEDURE CurAboutMRefAttr(): ADDRESS;
(*
  Returns first attribute associated to the model of which the
  about procedure is currently executed. The value NIL is returned
  if this procedure is called outside 'about'.
*)

PROCEDURE ModelLevel(m: Model):CARDINAL;
(*
  Returns the program level at which model m has been
  instanciated if the model exists, otherwise 0.
*)

PROCEDURE ObjectLevel(m: Model; VAR o: REAL):CARDINAL;
(*
  Returns the program level at which object o of model m has been
  instanciated if such an object exists, otherwise 0.
*)

(* ----- *)
(* direct object manipulations: *)

(* the following type and procedures allow for very efficient access
  to the most important ModelWorks objects. These information are
  provided for the advanced client who writes additional, generally

```

usable tools for the ModelWorks environment. The direct access to some of these data can be risky - the programmer ought to understand well what he/she does. \*)

TYPE

```
MWObj = (Mo, SV, Pa, MV, AV );
ExportObjectType = (stateVar, modParam, outAuxVar);
RealPtr = POINTER TO REAL;
PtrToClientObject = ADDRESS;

ObjPtr = POINTER TO ObjectHeader;
ObjectHeader = RECORD
  ident      : String;
  descr      : String;
  unit       : String;
  varAdr     : RealPtr; (* read only; real itself may be altered *)
  min, max   : REAL;
  nrAttr     : INTEGER;
  refAttr    : PtrToClientObject; (* read only *)
  chAttr     : CHAR;
  kind       : MWObj; (* read only *)
  parentM    : Model; (* read only *)
  next       : ObjPtr; (* read only *)
  prev       : ObjPtr; (* read only *)
END(*ObjectHeader*);
```

```
PROCEDURE FirstModel(): ObjPtr;
PROCEDURE FirstSV( m: Model ): ObjPtr;
PROCEDURE FirstP ( m: Model ): ObjPtr;
PROCEDURE FirstMV( m: Model ): ObjPtr;
```

```
PROCEDURE LastModel(): ObjPtr;
PROCEDURE LastSV( m: Model ): ObjPtr;
PROCEDURE LastP ( m: Model ): ObjPtr;
PROCEDURE LastMV( m: Model ): ObjPtr;
```

```
PROCEDURE AllowForRAMSESExport (owner: Model;
                                VAR obj: REAL; ident: ARRAY OF CHAR;
                                eot: ExportObjectType);
```

```
(*
  Once a model object has been passed to this routine, it becomes
  visible for RAMSES model systems, which corresponds to an
  export from the ModelWorks object base to the RAMSES object
  base. Within the RAMSES object base objects can be identified
  via their identifiers.
*)
```

END SimObjects.



### D.3 AUXILIARY LIBRARY

The here listed definitions represent only a few of the modules actually contained in the auxiliary library of the RAMSES software package<sup>1</sup>. They are likely to be of interest in a modeling and simulation context. For the relationship between auxiliary library modules and the "Dialog Machine" and ModelWorks see also part II *Theory* section *Module structure of ModelWorks*.

#### D.3.1 *IdentifyPars*

The module *IdentifyPars* supports the interactive (or batch), nonlinear parameter identification of model parameters. No restrictions apply either to the type of model (any elementary or structured model type may be involved), to the model parameters (linear or nonlinear in the parameters), nor to the type of optimization criteria. The only requirements are that the model definition program is complete in order to run a first simulation and that the user provides some data which specify the desired model behavior. Module *IdentifyPars* allows then to search for other model parameter values, or other initial values which behave eventually closer to the desired model behavior. Of course, neither convergence nor minimal identification time can be warranted for such a general procedure. For a typical usage of this auxiliary library module see the sample model *GauseIdentif*.

DEFINITION MODULE *IdentifyPars*;

```
(*****
Module  IdentifyPars      (Version 1.1)

      Copyright ©1989 by Olivier Roth and Swiss
      Federal Institute of Technology Zürich ETHZ

      Purpose:  Identifies parameters of a "ModelWorks"
                 model implemented as a model definition program.

      Remarks:  Uses internally the "Dialog Machine", the mandatory
                 client interface of ModelWorks, i.e. SimBase and
                 SimMaster, and from the optional client interface the
                 module SimObjects. Moreover the auxiliary library
                 modules Lists, IRand, and Matrices (actually consisting
                 of many modules).

      Note, this module exists in several implementation
      versions, since the more complex identification routines
      such as Powell have been implemented in form of large
      libraries. The simple implementation does not
      import from this package and has therefore the advantage
      of being much smaller; however, as a consequence, the
      method Powell can't be used in this version (On the Macintosh
      check the version text with the Get Info command to verify
      which version you are currently using).

      Implementation restriction:
          A maximum of 1024 parameters can be identified at once.

      Programming

          o Design and Implementation
            O. Roth                      19.05.90
```

---

<sup>1</sup>For availability, installation, and complete list see the separate booklet "Installation Guide and Technical Reference of the RAMSES software".

A. Fischlin 27.01.93

Swiss Federal Institute of Technology Zurich ETHZ  
 Department of Environmental Sciences  
 Systems Ecology Group  
 ETH-Zentrum  
 CH-8092 Zurich  
 Switzerland

Last revision of definition: 27/Jan/93 af

\*\*\*\*\*)

TYPE

RealFct = PROCEDURE (): REAL;  
 MinMethod = (halfDouble, amoeba, price, random, brent, powell, simplex);

(\* Description of the different methods can be found in:  
 + Press, H.W., Flannery, B.P., Teukolsky, S.A. & Vetterling, W.T., 1986,  
 "Numerical Recipes: the Art of Scientific Computing", Cambridge University Press,  
 New York, 818pp.  
 + Price, W.L., 1976, "A controlled random search procedure for global  
 optimisation", The Computer Journal 20(4): 367-370.  
 \*)

PROCEDURE MarkParForIdentification( VAR p: REAL );  
 PROCEDURE UnmarkParForIdentification( VAR p: REAL );  
 PROCEDURE UnmarkAllParsForIdentification;

(\* maintains a list of parameters which are to be identified later  
 with the identification procedures of this module (see  
 below). You may mark or unmark any parameter from that list  
 by means of the 3 procedures above, given the parameters have  
 been previously declared to ModelWorks by SimBase.DeclP. or  
 interactively by a call to procedure MinimizeAfterDialog (see  
 below). \*)

PROCEDURE SetDefltMinim( meth: MinMethod;  
 maxIter: INTEGER;  
 convC: REAL );  
 PROCEDURE GetDefltMinim( VAR meth: MinMethod;  
 VAR maxIter: INTEGER;  
 VAR convC: REAL );

(\* Set/get the default minimization method "meth", the default maximum  
 number of iterations "maxIter", and the default convergence  
 criterion value "convC". These procedures are typically called  
 before "MinimizeAfterDialog" to assure meaningful default  
 settings. \*)

PROCEDURE MinimizeAfterDialog( func: RealFct );  
 (\* Opens a scrollable selector box in which you may choose and select  
 (mark) parameters interactively to be identified. "func" is  
 the procedure computing the performance index (i.e. it calls  
 e.g. SimRun and then SimDeltaCalc.GetDeltaStat). \*)

PROCEDURE Minimize( method: MinMethod; convC: REAL;  
 maxIter: INTEGER; func: RealFct );  
 (\* Executes all necessary runs to perform an identification. "method"  
 specifies one of the above listed identification methods,  
 "convC" stands for a convergence criterion value, "maxIter"  
 denotes the maximum number of iterations, and "func" is a  
 function procedure returning the value of the performance  
 index by calling e.g. SimRun and then  
 SimDeltaCalc.GetDeltaStat. Note: "maxIter" is NOT the exact  
 maximal number of performed SimRun's, since an iteration

consists usually of several runs (depending on the selected identification method). \*)

END IdentifyPars.

### D.3.2 JulianDays

Calendar time is usually the preferred way to associate important characteristics with time, e.g. the seasonal temperature cycle is usually related to months. However, computations such as time intervals from calendar time are a nuisance; e.g. how many hours are between 21st February, 8h23'00", and 2nd September, 14h17'00" in the year 1954? On the other hand if any date is measured in Julian days, i.e. a real number  $t$ , which have an origin or reference point  $t_0$  at a known calendar time of year<sub>0</sub>, month<sub>0</sub>, day<sub>0</sub>, hour<sub>0</sub>, min<sub>0</sub>, sec<sub>0</sub> etc., such computations are reduced to simple real arithmetics. To the end that such a time can still be read by humans, the only thing needed are convenient functions which allow to convert between calendar time and a Julian time at any point in time. In addition we need also means to define the point of origin  $t_0$ . This functionality is exactly the purpose of module *JulianDays*. In the context of dynamic models which have to operate on time, it provides the numerically delicate but easy to use conversion algorithms which allow to use a Julian time scale conveniently.

DEFINITION MODULE *JulianDays*;

(\*\*\*\*\*  
 \*\*\*\*\*  
 \*\*\*\*\*)

Module *JulianDays* (Version 2.0)

Copyright 1989 by Andreas Fischlin and Swiss  
 Federal Institute of Technology Zuerich ETHZ

Purpose Translates back and forth dates into a number of days (Julian days) in order to allow the computing with dates.

Remark This implementation is based on the Gregorian calendar, which is valid after 15.Oct.1582. Note that this date followed immediately after 4.Oct.1582 to correct for accumulated errors in the Julian calendar introduced by Julius Caesar "ab urbe condita", the foundation of Rome, i.e. 753 BC (Gregorian calendar correction by Pope Gregor XIII). The Gregorian calendar will need no corrections for 3333 years.

Note there is also the so-called Julian Period, which is used in astronomy as proposed by Joseph Justus Scaliger (1581): First Julian Date (J.D.) is middle noon, 1. Jan.4713 BC. The Julian time is calculated in days, and is a real defining hours, minutes plus seconds. Note that in this method a day starts at noon of standard world time or Greenwich time. There is a modified Julian Date (M.J.D.) in use today (much used in space travel) which starts at 17.Nov.1858 00h00'00" ~24 00 000.5 J.D.

Programming

- Design
  - A. Fischlin 24/09/89
- Implementation
  - A. Fischlin 24/09/89

Swiss Federal Institute of Technology Zurich ETHZ  
 Department of Environmental Sciences  
 Systems Ecology Group  
 ETH-Zentrum  
 CH-8092 Zurich  
 Switzerland

Last revision of definition: 3/02/94 af

```

*****
CONST
  Jan = 1; Feb = 2; Mar = 3; Apr = 4; Mai = 5; Jun = 6;
  Jul = 7; Aug = 8; Sep = 9; Oct = 10; Nov = 11; Dec = 12;

  Sun = 1; Mon = 2; Tue = 3; Wed = 4; Thur = 5; Fri = 6; Sat = 7;

TYPE
  Month = [Jan..Dec];
  WeekDay = [Sun..Sat];
  DateAndTime = RECORD
    year: INTEGER;          (* e.g. 1582,...,1994,...,2040 etc.*)
    month: Month;
    day: INTEGER;          (* [1..31] (depends on month) *)
    hour,
    min: INTEGER;          (* [0..59] *)
    sec: INTEGER;          (* [0..59] *)
    dayOfWeek: WeekDay;    (* e.g. Sun *)
    secFrac: REAL;         (* fraction of a second,
                           e.g. 0.13 for 13 hundredth of a second *)
  END;

PROCEDURE DateTimeToJulDay(dt: DateAndTime): LONGREAL;
PROCEDURE JulDayToDateTime(jd: LONGREAL; VAR dt: DateAndTime);
(*
  Above two routines allow to convert between a julian day given
  as a real number and an ordinary calendar date plus the time of
  the day.
*)

PROCEDURE DateToJulDay(day,month,year: INTEGER): LONGINT;
PROCEDURE JulDayToDate(jd: LONGINT; VAR day: INTEGER;
  VAR month: Month;
  VAR year: INTEGER;
  VAR dayOfWeek: WeekDay);
(*
  Above two routines allow to convert between a julian day and an
  ordinary calendar date. Hereby ignoring the time of the day.
*)

PROCEDURE IsLeapYear(yr: INTEGER): BOOLEAN;

PROCEDURE SetCalendarRange(firstYear,lastYear,firstSunday: INTEGER);
(*
  This procedure allows to set the calendar range for which the
  algorithms of this module shall work. They work correctly
  from the date 15.Oct.1582 onwards for the next 3333 years and
  given the following restrictions are satisfied: The first
  year must be an year following immediately a leap year. The
  day of the first Sunday in January in the first year
  (firstSunday) must be specified, otherwise weekdays won't be
  computed correctly. If faulty values are specified
  this routine will lead to an error condition.

  The default range is firstYear = 1949, lastYear = 5282,
  firstSunday = 2, since the 2nd January 1949 is a
  Sunday. (Other possibilities: Sunday, 6.Jan.1805).

  Note that calling this procedure may be useful in order to
  use Julian days of type INTEGER instead of LONGINT. Then the
  calendar routines can cover fully 137 years without causing
  an overflow when assigning the LONGINT result of procedure
  DateToJulDay to an INTEGER variable.
*)

```

END JulianDays.

### D.3.3 Queues

Queues of any objects such as persons or parcels are often needed in the context of simulations with DEVS. Module *Queues* provides the instantiation and the management of FIFO-queues (First In, First Out). For a typical usage of this auxiliary library module see the sample model *CarPollution*, in particular the submodels *CPTraffic* and *CPCrossRoad*.

```
DEFINITION MODULE Queues;
```

```
(*****
```

```
Module Queues      (Version 1.0)
```

```
    Copyright (c) 1992 by Andreas Fischlin and Swiss
    Federal Institute of Technology Zürich ETHZ
```

```
Version written for:
    MacMETH_V3.2      (1-Pass Modula-2 implementation)
```

```
Purpose Utilities needed for discrete event simulations
    involving queues
```

```
Programming
```

```
    o Design
      A. Fischlin      17/Mar/93
```

```
    o Implementation
      A. Fischlin      17/Mar/93
```

```
    Swiss Federal Institute of Technology Zurich ETHZ
    CH-8092 Zurich
    Switzerland
```

```
    Last revision of definition: 17/Mar/93  AF
```

```
*****)
```

```
FROM SimEvents IMPORT Transaction;
```

```
TYPE
```

```
    FIFOQueue;
    ItemAction = PROCEDURE (Transaction);
```

```
VAR
```

```
    notExistingFIFOQueue: FIFOQueue; (* read only *)
```

```
PROCEDURE CreateFIFOQueue(VAR q: FIFOQueue; maxLength: INTEGER);
```

```
PROCEDURE EmptyFIFOQueue(q: FIFOQueue);
PROCEDURE FileIntoFIFOQueue(q: FIFOQueue; ta: Transaction);
PROCEDURE FirstInFIFOQueue(q: FIFOQueue): Transaction;
PROCEDURE Take1stFromFIFOQueue(q: FIFOQueue): Transaction;
PROCEDURE FIFOQueueLength(q: FIFOQueue): INTEGER;
PROCEDURE IsFIFOQueueFull(fifoq: FIFOQueue): BOOLEAN;
PROCEDURE IsFIFOQueueEmpty(fifoq: FIFOQueue): BOOLEAN;
PROCEDURE DoForAllInFIFOQueue(q: FIFOQueue; ia: ItemAction);
```

```
PROCEDURE FIFOQueueExists(q: FIFOQueue): BOOLEAN;
```

```
PROCEDURE DiscardFIFOQueue(VAR q: FIFOQueue);
```

```
END Queues.
```





### D.3.4 RandGen

Any stochastic simulation requires the generation of pseudo-random numbers. This module provides the basic algorithms to produce a sequence of random numbers or variates, uniformly distributed in the interval [0..1). The algorithm contained in this module has been carefully selected for maximum period length, maximum randomness, maximum reliability and portability, needed for scientific applications, in contrast to the random number generators often available from the system software. For a typical usage of this auxiliary library module see the sample model *Markov* or any of these sample models: *Diversity*, *StochLogGrow*, *CarPollution* (in particular *CPTraffic* plus *CPOjects*), and *StochLogGrow*.

DEFINITION MODULE RandGen;

(\*\*\*\*\*

Module RandGen (Version 1.0)

Copyright 1988 by Andreas Fischlin and Systems  
Ecology Group ETHZ, Swiss Federal Institute of  
Technology Zuerich ETHZ

Purpose Basic pseudo-random number generator producing  
uniformly distributed variates within interval (0,1).  
The generator is based on a combination of three  
multiplicative linear congruential random number  
generators.

Remarks The generator is highly portable and produces  
very-long-cycle random-number sequences. They  
exceed the usual period length of MAX(INTEGER)  
given by the machine dependent word length. Thus  
the generator produces satisfactory results even on  
a personal computer with a small word length (e.g.  
16-Bit machines) and it is efficient, since it does  
not require double precision arithmetics. On  
32-Bit machines like IBM main-frames or the Apple®  
Macintosh™ PC this means that the slow 64-Bit  
multiplication and division can be  
avoided.

The cycle length of the generator is estimated to  
be > 2.78 E13 so that the sequence will not repeat  
for over 220 years in case that 1000 variates were  
calculated per second (Wichmann & Hill, 1987)

References:

Wichmann, B.A. & Hill, I.D., 1982. An efficient and  
portable pseudo-random number generator. Algorithm  
AS 183. Applied Statistics, 31(2): 188-190.

Wichmann, B. & Hill, D., 1987. Building a random-number  
generator. A Pascal routine for very-long-cycle  
random-number sequences. Byte 1987(March):  
127-28

Programming

- Design  
A. Fischlin (21 Dez 88)
- Implementation  
A.Fischlin/O.Roth (21 Dez 88)

Swiss Federal Institute of Technology Zurich

Systems Ecology  
 Department of Environmental Sciences  
 ETH-Zentrum  
 CH-8092 Zurich  
 Switzerland

Last revision: 31 Jan 89 (A.F.)

\*\*\*\*\*)

```

PROCEDURE SetSeeds(z0,z1,z2: INTEGER);
  (*defaults:  z0 = 1, z1 = 10000, z2 = 3000 *)
PROCEDURE GetSeeds(VAR z0,z1,z2: INTEGER);
PROCEDURE Randomize;
  (*set seeds using seed values depending on a particular, unique
  and non repeatable event in real time, e.g. date and time of
  the clock.  Implies a call to SetSeeds*)
PROCEDURE ResetSeeds;
  (*reset seeds to values defined by last call to SetSeeds*)

PROCEDURE U(): REAL;
  (*returns within (0,1) uniformly distributed variates*)

  (*
  Based on a combination of three multiplicative linear
  congruential random number generators of the form  z(k+1) =
  A*z(k) MOD M  with a prime modulus and a primitive root
  multiplier (=> individual generator full length period). The
  multipliers A are: 171, 172, and 170; the modulus' M are:
  30269, 30307, and 30323.
  *)

END RandGen.
```

### D.3.5 RandGen0

This module provides some generators for often used variates such as uniformly distributed integer or real variates, and negative exponentially distributed variates. Note that this module together with similar modules has been designed for optimal flexibility by allowing to install into it any basic random number generator providing uniformly distributed variates in the interval [0..1) or alternatively (0..1] resp. (0..1). For a typical usage of this auxiliary library module see the research sample models *Diversity* and *CarPollution* (in particular *CPTraffic*). For the use of this auxiliary library module see also auxiliary library module *RandGen*.

```

DEFINITION MODULE RandGen0;

  (*****

  Module  RandGen0      (Version 1.0)

      Copyright (c) 1992 by Andreas Fischlin and Swiss
      Federal Institute of Technology Zürich ETHZ

  Version written for:
      MacMETH_V3.2      (1-Pass Modula-2 implementation)

  Purpose Simple random number generators often used in
      stochastic simulations.
```

Remarks This module is best used in connection with  
module RandGen.

Programming

- o Design
  - A. Fischlin 12/Mar/93
- o Implementation
  - A. Fischlin 12/Mar/93

Swiss Federal Institute of Technology Zurich ETHZ  
CH-8092 Zurich  
Switzerland

Last revision of definition: 12/Mar/93 AF

\*\*\*\*\*)

```

PROCEDURE J(): INTEGER;
PROCEDURE Jp(min,max: INTEGER): INTEGER;
(*
  Return in the range [min..max] uniformly distributed integer
  variates. For J the range [min..max] has to be defined by
  procedure SetJPar. Default: [min..max] = [0..1].
*)

PROCEDURE SetJPar( min,max: INTEGER);
PROCEDURE GetJPar(VAR min,max: INTEGER);
(*
  Setting and retrieval of the range parameters [min..max] used
  by the integer random number generator J.
*)

PROCEDURE R(): REAL;
PROCEDURE Rp(min,max: REAL): REAL;
(*
  Return in the range [min..max] uniformly distributed real
  variates. For R the range [min..max] has to be defined by
  procedure SetRPar. Default: [min..max] = [0.0..1.0].
*)

PROCEDURE SetRPar( min,max: REAL);
PROCEDURE GetRPar(VAR min,max: REAL);
(*
  Setting and retrieval of the range parameters [min..max] used
  by the real random number generator R.
*)

PROCEDURE NegExp(): REAL;
PROCEDURE NegExpP(lambda: REAL): REAL;
(*
  Sampling of negative exponentially distributed variates. For
  NegExp the mean lambda has to be defined by procedure
  SetNegExpPar. Default: lambda = 1, i.e. a Poisson process where
  on average occurs 1 event per time unit.
*)

PROCEDURE SetNegExpPar( lambda: REAL);
PROCEDURE GetNegExpPar(VAR lambda: REAL);
(*
  Setting and retrieval of the mean parameter lambda used by the

```

```

    negative exponential random number generator NegExp.
*)

TYPE
  URandGen = PROCEDURE(): REAL;

(* NOTE: Always call one of the following two procedures before
calling any other random number generator from this module: *)

PROCEDURE InstallU0(u0: URandGen);
(*)
  Allows to install the basic random number generator needed by
  all generators exported by this module. The random number
  generator u0 must sample uniformly distributes variates within
  interval [0..1), i.e. it may generate 0.0, but must not
  generate exactly 1. For instance you may install procedure U
  from module RandGen contained in the auxiliary library of the
  RAMSES software, which satisfies these specifications and
  produces high quality pseudo-random number sequences (See also
  procedure InstallU1).
*)

PROCEDURE InstallU1(u1: URandGen);
(*)
  Allows to install the basic random number generator needed by
  all generators exported by this module. The random number
  generator u1 must sample uniformly distributes variates within
  interval (0..1] or (0..1), i.e. it may or may not generate 1.0,
  but must not generate exactly 0. The installation of a good
  generator u1 satisfying these specifications results in more
  efficient variates sampling by the NegExp generator than when
  installing a basic generator via procedure InstallU0. However,
  the efficiency may be in conflict with the quality of the
  generated pseudo-random number sequences (see also procedure
  InstallU0).
*)

END RandGen0.

```

### D.3.6 RandGen1

More generators. For the use of this auxiliary library module see also auxiliary library modules *RandGen0* and *RandGen*.

```

DEFINITION MODULE RandGen1;

(*****

Module RandGen1 (former RandGens) (Version 2.0)

    Copyright ©1990 by Thomas Nemecek and Swiss
    Federal Institute of Technology Zürich ETHZ

Version written for:
    'Dialog Machine' DM_V2.02 (User interface)
    MacMETH_V2.6.2 (1-Pass Modula-2 implementation)

Purpose provides different random number generators

Programming

```

- o Design
  - T. Nemecek 20.7.90
- o Implementation
  - T. Nemecek 20.7.90

Swiss Federal Institute of Technology Zurich ETHZ  
 Department of Environmental Sciences  
 Systems Ecology Group  
 ETH-Zentrum  
 CH-8092 Zurich  
 Switzerland

Last revision of definition: 9/5/96 ft

\*\*\*\*\*)

```
PROCEDURE Weibull(): REAL;
PROCEDURE WeibullP(alpha, beta: REAL): REAL;
(*
  Provides Weibull distributed random variables. The 2-parametric
  Weibull distribution is used. Prbability density function:
  f(x) = alpha * beta^-alpha * x^(alpha-1) * Exp(-(x/beta)^alpha)

  For Weibull the parameters have to be defined by procedure
  SetWeibullPars (s.b.). Defaults are: alpha = 1.0
                                     beta  = 1.0
*)
```

```
PROCEDURE SetWeibullPars(  alpha, beta: REAL);
PROCEDURE GetWeibullPars(VAR alpha, beta: REAL);
(*
  Setting and retrieval of the parameters alpha and beta
  used by the random number generator Weibull.
*)
```

```
PROCEDURE Triang(): REAL;
PROCEDURE TriangP(min, mode, max: REAL): REAL;
(*
  Provides random numbers following a triangular distribution
  with the parameters min,mode,max, where
  min = lowest value
  max = highest value
  mode = coordinate of maximum.
  For Triang the parameters have to be defined by procedure
  SetTriangPars (s.b.). Defaults are: min = -1.0
                                     max  = 1.0
                                     mode = 0.0
*)
```

```
PROCEDURE SetTriangPars(  min, mode, max: REAL);
PROCEDURE GetTriangPars(VAR min, mode, max: REAL);
(*
  Setting and retrieval of the parameters min, max and mode
  used by the random number generator Triang.
*)
```

```
PROCEDURE VM(): REAL;
PROCEDURE VMP(mean, kappa: REAL): REAL;
(*
  provides random number from the von Mises distribution
  (called also the circular normal distribution). The
  values are in the interval [0, 2 ]
  For VM the parameters have to be defined by procedure
  SetVMPars (s.b.). Defaults are: mean = 0.0
                                   kappa = 1.0
*)
```

```

*)

PROCEDURE SetVMPars( mean, kappa: REAL);
PROCEDURE GetVMPars(VAR mean, kappa: REAL);
  (*
  Setting and retrieval of the parameters mean and kappa
  used by the random number generator VM.
  *)

TYPE
  URandGen = PROCEDURE(): REAL;

(* NOTE: ALWAYS call one of the following two procedures before
calling any other random number generator from this module: *)

PROCEDURE InstallU0(u0: URandGen);
  (* Allows to install the basic random number generator needed by
  all generators exported by this module. The random number
  generator u0 must sample uniformly distributed variates within
  interval [0..1), i.e. it may generate 0.0, but must not
  generate exactly 1. For instance you may install procedure U
  from module RandGen contained in the auxiliary library of the
  RAMSES software, which satisfies these specifications and
  produces high quality pseudo-random number sequences (See also
  procedure InstallU1). *)

PROCEDURE InstallU1(u1: URandGen);
  (* Allows to install the basic random number generator needed by
  all generators exported by this module. The random number
  generator u1 must sample uniformly distributed variates within
  interval (0..1] or (0..1), i.e. it may or may not generate 1.0,
  but must not generate exactly 0. The installation of a good
  generator u1 satisfying these specifications results in more
  efficient variates sampling by the NegExp generator than when
  installing a basic generator via procedure InstallU0. However,
  the efficiency may be in conflict with the quality of the
  generated pseudo-random number sequences (see also procedure
  InstallU0). *)

END RandGen1.

```

### D.3.7 RandNormal

This module provides a generator for normally distributed real variates. Note that this module together with similar modules has been designed for optimal flexibility by allowing to install into it any basic random number generator providing uniformly distributed variates in the interval [0..1), (0..1] or (0..1). For a typical usage of this auxiliary library module see the research sample model *StochLogGrow*. For the use of this auxiliary library module see also auxiliary library module *RandGen*.

```

DEFINITION MODULE RandNormal;

  (*****

  Module RandNormal      (Version 1.0)

      Copyright 1987 by Andreas Fischlin and CELTIA,
      Swiss Federal Institute of Technology Zuerich ETHZ

      Purpose Computation of normally distributed variates

```

References

Bell, J.R. 1968. Normal random deviates. Algorithm 334. Collected Algorithms from CACM (Communications of the Association for Computing Machinery): 334-P 1-R1

Box, G. & Muller, M. 1958. A note on the generation of normal deviates. Ann. Math. Stat. 28: 610.

Von Neumann, J. 1959. Various techniques used in connection with random digits. In: Nat. Bur. Standards Appl. Math. Ser. 12, US Govt. Printing Off., Washington, D.C., p. 36.

Remark This implementation allows to be completely independent from any particular random number generator (see InstallU).  
 NOTE: The module won't crash if InstallU is never called, but it will not be able to produce correct results!

Imported modules: System, MathLib

Programming

- Design  
     A. Fischlin                      (17 Dec 87)
- Implementation  
     A. Fischlin                      (17 Dec 87)

Swiss Federal Institute of Technology Zurich  
 Project Centre IDA  
 Pilot Project CELTIA  
 [Computer-aided Explorative Learning and Teaching  
 with Interactive Animated Simulation]  
 ETH-Zentrum  
 CH-8092 Zurich  
 Switzerland

Last revision: 22/Mar/93      (af)

\*\*\*\*\*)

TYPE

URandGen = PROCEDURE(): REAL;

PROCEDURE InstallU(U: URandGen);

(\*  
 Installs procedure U which returns variates from a random variable uniformly distributed within interval [0..1). (NOTE: Always call this procedure before calling N or Np).  
 \*)

PROCEDURE N(): REAL;

PROCEDURE Np(mu, stdDev: REAL): REAL;

(\*  
 Return a variate from a normally distributed random variable with mean mu and the standard deviation stdDev. For N these parameters have to be set by procedure SetPars, where the default values for mu respectively stdDev are 0 resp. 1.0. The variates are computed by the method Box and Muller and the Von Neumann rejection technique.  
 \*)

Implementation note: Crashing of N() or Np in case where U() returns zero is prevented by calling U() again; however, if zero is an absorbing state for U() this would lead to an infinite loop within N() resp. Np(); hence, the implementation counts the occurrences of U() returning zero and halts program execution after 50000 occurrences.

\*)

```
PROCEDURE SetPars(mu, stdDev: REAL);
```

```
PROCEDURE GetPars(VAR mu, stdDev: REAL);
```

```
(*
```

```
  Set or get the current parameters mu (mean) and the  
  stdDev (standard deviation = SQRT(variance)) for  
  the normally distributed random variable for which  
  procedure N returns variates.
```

```
*)
```

```
PROCEDURE ResetN;
```

```
(*
```

```
  The here adopted method (Box and Muller and the Von Neumann  
  rejection) computes at each second call of N resp. Np two  
  values. Inbetween the already computed and not yet used value  
  is simply returned without any further calculations. In order  
  to produce completely defined results, for instance after  
  setting a new seed value in the basic pseudo-random sequence  
  used by U, call this procedure. Only this will fully reset the  
  internal mode of this module and put it to a state where it  
  always produces the same pseudo random sequence of normally  
  distributed variates.
```

```
*)
```

```
END RandNormal.
```



### D.3.8 ReadData

This module facilitates the reading of data from files, for instance from files storing measurements, at the begin or during simulations. *ReadData* is capable to scan a text file by recognizing numbers, strings, and comments. Numbers are checked for syntax and range, and if an error is detected, the user is informed and asked via a dialog box to correct the error or to abandon the reading process completely. The scanner recognizes strings delimited by blanks (actually any ASCII-ch <= ' ') and comments bracketed by the symbols "(" respectively ")". This module is most useful while implementing and debugging the reading of complex data sets (note, an alternative to this module is to use directly the module *DMFiles* from the "Dialog Machine" as demonstrated by the sample model *Sensitivity* and the research sample model *LBM*). For a typical usage of module *ReadData* see the sample model *SwissPop*.

DEFINITION MODULE ReadData;

(\*\*\*\*\*

Module ReadData ( Version 1.0 )

Copyright 1989 by Andreas Fischlin and CELTIA,  
Swiss Federal Institute of Technology Zuerich ETHZ

Purpose Export of several utilities to read and test data  
while reading from a file with data in columnar form.

Programming

- Design  
A. Fischlin ( 12 Feb 89 )

- Implementation  
A. Fischlin ( 12 Feb 89 )  
T. Nemecek ( 9 Sep 89 )  
O. Roth ( 23 Nov 89 )  
F. Thommen ( 03 Mar 91 )

Swiss Federal Institute of Technology Zurich  
Systems Ecology Group  
ETH-Zentrum  
CH-8092 Zurich  
Switzerland

Last revision: 15 Mar 91 ft

\*\*\*\*\*)

(\* List of all idents exported by this module:

```
FROM ReadData IMPORT
negLogDelta, SkipGapOrComment, ReadCharsUnlessAComment,
SetMissingValCode, GetMissingValCode, SetMissingReal,
GetMissingReal, SetMissingInt, GetMissingInt, dataF,
OpenADataFile, OpenDataFile, ReReadDataFile, CloseDataFile,
SkipHeaderLine, ReadHeaderLine, ReadLn, GetChars, GetStr,
GetInt, GetReal, SetEOSCode, GetEOSCode, FindSegment,
SkipToNextSegment, AtEOL, AtEOS, AtEOF, TestEOF, Relation,
Compare2Strings, ErrorType, NumbType, ErrMsgProc, SetErrMsgP,
GetErrMsgP, UseDefaultErrMsg;
```

\*)

```
FROM DMStrings IMPORT String;
```

```

FROM DMFiles    IMPORT TextFile;

CONST
  negLogDelta = 0.01; (*offset to plot log scale if values <= 0*)

(* File handling: *)
VAR dataF: TextFile;

PROCEDURE OpenADataFile( VAR fn: ARRAY OF CHAR; VAR ok: BOOLEAN );
(* opens a file using the standard open file dialog *)

PROCEDURE OpenDataFile ( VAR fn: ARRAY OF CHAR; VAR ok: BOOLEAN );
(* opens a file specified by fn automatically, and calls OpenADataFile
  * if fn couldn't be found *)

PROCEDURE ReReadDataFile;

PROCEDURE CloseDataFile;

(* Reading and number testing *)

VAR readingAborted: BOOLEAN;
(* Returns wether the file reading has been aborted by pressing the
  * pushButton "Stop reading". It is highly recommended to use this
  * variable to test whether the reading of the data has been
  * successful. If readingAborted = FALSE subsequently avoid any
  * program loop, for instance a simulation; instead make sure you
  * immediately return control to the 'Dialog Machine'. The latter
  * is very important if the user has pressed the button 'Abort
  * prgm', which has signaled to the 'Dialog Machine' to terminate
  * itself (i.e. it actually called QuitDialogMachine from
  * DMMaster). After executing QuitDialogMachine, the 'Dialog
  * Machine' accepts no more user events and any loop under client
  * control can no longer be terminated via ordinary user events
  * such as a menu command 'Stop'. Thus any loop with a termination
  * condition depending on an user event will no longer function,
  * since the current (sub)program level accepts no more user
  * events. *)

PROCEDURE SkipGapOrComment;
(* skips all characters <= " " and all text enclosed in comment
  * brackets as used in Modula-2, i.e. "( * ..... * )"
  * This procedure is used in this module. *)

PROCEDURE ReadCharsUnlessAComment( VAR string: ARRAY OF CHAR );
(* reads a string beginning from the current position until
  * a character <= " " or a comment is encountered. *)

(* Missing values: *)

(* default missingValCode = "N" *)
PROCEDURE SetMissingValCode( missingValCode   : CHAR );
PROCEDURE GetMissingValCode( VAR missingValCode: CHAR );

(* default missingReal = DMConversions.UndefREAL() *)
PROCEDURE SetMissingReal( missingReal       : REAL );
PROCEDURE GetMissingReal( VAR missingReal: REAL );

(* default missingInt = 0 *)
PROCEDURE SetMissingInt( missingInt        : INTEGER );
PROCEDURE GetMissingInt( VAR missingInt: INTEGER );

```

```

PROCEDURE SkipHeaderLine;

PROCEDURE ReadHeaderLine( VAR labels: ARRAY OF String;
                          VAR nrVars: INTEGER );
(* IMPORTANT NOTE: labels must be initialized to NIL before first use! *)

PROCEDURE ReadLn ( VAR txt: ARRAY OF CHAR );

PROCEDURE GetChars( VAR str: ARRAY OF CHAR );

PROCEDURE GetStr ( VAR str: String );

(* In the following procedures the two first parameters desc and
 * loc are only needed for the display of error messages and help
 * the user to identify an erroneous location within the data file:
 * - desc a string describing the kind of data to be read, e.g.
 *       population density or number of individuals
 * - loc   a location number indicating where the error has
 *       been found, e.g. a line number
 *)

PROCEDURE GetInt ( desc : ARRAY OF CHAR;  loc: INTEGER;
                  VAR x: INTEGER;  min, max: INTEGER );

PROCEDURE GetReal( desc : ARRAY OF CHAR;  loc:  INTEGER;
                  VAR x: REAL;    min, max: REAL  );

(* Working with data segments (EOS means End Of Segment): *)

PROCEDURE SetEOSCode( eosCode   : CHAR );

PROCEDURE GetEOSCode( VAR eosCode: CHAR );

PROCEDURE FindSegment( segNr: CARDINAL; VAR found: BOOLEAN );

PROCEDURE SkipToNextSegment( VAR done: BOOLEAN );

(* Testing: *)

PROCEDURE AtEOL(): BOOLEAN;

PROCEDURE AtEOS(): BOOLEAN;

PROCEDURE AtEOF(): BOOLEAN;

PROCEDURE TestEOF; (* use only where you don't yet expect EOF (shows alert) *)

TYPE Relation = ( smaller, equal, greater );

PROCEDURE Compare2Strings( a, b: ARRAY OF CHAR ): Relation;

(* Alerts: *)

TYPE
  ErrorType = (NoInt, NoReal, TooBig, TooSmall,
              NotEqual, EndOfFile, FileNotFound, DataFNotOpen);
(* type of the error:
 * -NoInt      : Integer expected but string or real encountered
 * -NoReal     : Real expected but string or Integer encountered
 * -TooBig     : Number higher than max
 * -TooSmall   : Number smaller than min
 * -NotEqual   : Special case if min=max and number #min resp. max
 * -EndOfFile  : Attempt to read the file over it's end
 *)

```

```
* -FileNotFound : Data file not found
* -DataFNotOpen : Data file could not be opened *)
```

```
NumbType = (Real, Integer);
(* tells weather a real or an integer had to be read *)
```

```
Error =
  RECORD
    errorType : ErrorType;
    strFound  :ARRAY[0..63] OF CHAR;
    CASE numbType :NumbType OF
      Integer : minI, maxI: INTEGER
      | Real   : minR, maxR: REAL
    ELSE
      END;
    desc :ARRAY [0..255] OF CHAR;
    loc  :INTEGER
  END(*RECORD*);
```

```
ErrMsgProc = PROCEDURE(Error);
```

```
PROCEDURE SetErrMsgP(errP: ErrMsgProc);
(* sets the current alert procedure to alert. Useful if working in batch
* mode to avoid program halt *)
```

```
PROCEDURE GetErrMsgP(VAR currErrP: ErrMsgProc);
(* gets the current alert procedure *)
```

```
PROCEDURE UseDefaultErrMsg;
(* re-installs the default alert procedure *)
```

```
END ReadData
```

### D.3.9 *StochStat*

Stochastic simulations (see also section *Stochastic Simulations* in this *Appendix*) often require a statistical analysis of the simulation results. The purpose of module *StochStat* is to support the sampling of a set of trajectories and to calculate and display graphically some basic statistics such as means and confidence intervals, assuming a normal distribution. Typically these trajectories are produced by running the same stochastic model several times (but with different pseudo random numbers) from within an experiment procedure. For a typical usage of this module see the sample model *StochLogGrow*.

DEFINITION MODULE *StochStat*;

(\*\*\*\*\*)

Module *StochStat* (Version 1.1)

Copyright ©1990 by Thomas Nemecek and Swiss  
Federal Institute of Technology Zürich ETHZ

Version written for:

'Dialog Machine' DM\_V2.2 (User interface)  
ModelWorks MW\_V2.2 (Modelling & Simulation)

Purpose

Auxiliary module for stochastic simulation. Calculates means, standard deviation and confidence intervals of n arrays with m observations of a monitorable variable and allows to display the means and the confidence intervals in the graph window, using the module *SimGraphUtils*.

Programming

- o Design  
T. Nemecek 19.4.90
- o Implementation  
T. Nemecek 24.4.90

Swiss Federal Institute of Technology Zurich ETHZ  
Department of Environmental Sciences  
Systems Ecology Group  
ETH-Zentrum  
CH-8092 Zurich  
Switzerland

Last revision of definition: 24.06.91 tn

(\*\*\*\*\*)

```
(* FROM StochStat IMPORT
StatArray, StatArrayExists, Prob2Tail, Str31, notExistingStatArray, DeclStatArray,
RemoveStatArray, RemoveAllStatArrays, ClearStatArray, ClearAllStatArrays,
SetStatArray, SetUndefValue, GetUndefValue, SetTolerance, GetTolerance,
PutValue, GetValue, GetSingleStatistics, GetStatistics, DeclDispMV, DisplayArray,
DisplayAllArrays, RealFileFormat, n, dec, FileOutFormat, indepsFormat,
meansFormat, sumsYFormat, sumsYSquareFormat, stdDevsYFormat, confIntsYFormat,
confProb, meansOnly, meansSDCI, allVals, DumpStatArray, DumpStatArrays;
```

```
FROM DMFiles IMPORT TextFile;
FROM DMConversions IMPORT RealFormat;
FROM SimBase IMPORT Model;
```

```

TYPE
  StatArray;
  Prob2Tail = (prob999, prob990, prob950, prob900, prob800);
  (* 2-tailed probability for confidence intervals the values mean
     promilles. *)
  Str31 = ARRAY [0..31] OF CHAR;

VAR
  notExistingStatArray:      StatArray; (* read only *)

  (*****
  (* StatArray management *)
  (*****

PROCEDURE StatArrayExists(statArray: StatArray): BOOLEAN;

PROCEDURE DeclStatArray(VAR statArray: StatArray; length: INTEGER);
  (* declares an array of data with n=length observation per run.
     Implicitly calls ClearStatArray! *)

PROCEDURE RemoveStatArray(VAR statArray: StatArray);

PROCEDURE RemoveAllStatArrays;

PROCEDURE ClearStatArray(statArray: StatArray);
  (* fills all columns of the array of data with 0.0,
     except the column with the independent variables, which
     is initialized to the undefined value.
     Resets the array to the initial state. *)

PROCEDURE ClearAllStatArrays;

PROCEDURE SetStatArray(statArray: StatArray;
  VAR N, X, sumY, sumYSquare: ARRAY OF REAL);
  (* an initial state of the statArray can be set (var parameters
     only for speed-up reasons). Can be used e.g. to continue an
     experiment, which had to be aborted.
     CAUTION: If any of the values are not known, set undefVal
     for the independent, and 0 for all N, sumY and
     sumYSquare! *)

  (*****
  (* Data storage *)
  (*****

PROCEDURE SetUndefValue(undefVal: REAL);
  (* has only an effect, if no array are currently delrared
     for reasons of consistency*)
PROCEDURE GetUndefValue(VAR undefVal: REAL);
  (* undefVal is assigned to any statistical value, which can not
     be calculated, because the number of observations is not sufficient,
     e.g. means if n=0, of stDevs is n=1.
     This value is also used to display values in the graph,
     that could not be calculated, e.g. mean if the number
     of observations is 0. You should use an undefVal,
     that does not occur in your data
     The default undefVal is -1.0E30; *)

PROCEDURE SetTolerance(tol: REAL);
PROCEDURE GetTolerance(VAR tol: REAL);
  (* tol is the maximal tolerance in which values of the
     independent variable are accepted. The value of the independent
     variable has to lie within the interval [x-tol,x+tol], where x is
     the first value given as independent.

```

```

The default tolerance is 10E-4 *)

PROCEDURE PutValue(statArray: StatArray; index: INTEGER; x, y: REAL);
  (* adds a value y to the stat array *)

PROCEDURE GetValue(statArray: StatArray; index: INTEGER;
  VAR count, x, sumY, sumYSquare: REAL);

(*****
(* Statistics *)
*****)

PROCEDURE GetSingleStatistics(statArray: StatArray; index: INTEGER;
  VAR count, x, sumY, sumYSquare, meansY, stdDevsY, confIntsY: REAL;
  confProb: Prob2Tail);
  (* gives statistical values describing a single observation point.
  count = number of observations at any observation point
  x = independent variable
  stdDevsY = standard deviation
  confIntsY = half confidence interval for confProb
  of any observation point in the array. The true mean lies within the
  interval [mean-confIntervalY, mean+confIntervalY] with
  a probability confProb.

  The statistics are given as follows for any observation point:
  if N = 0 ==> at any observation point, sumY, sumYSquare = 0,
               all other statistical values are = undefVal
  if N = 1 ==> the mean,sumY & sumYSquare are the single value resp. its
               square and all other values are = undefVal
  if N = 2 ==> all values are calculated
  *)

PROCEDURE GetStatistics(statArray: StatArray;
  VAR N, X, sumY, sumYSquare, meansY, stdDevsY, confIntsY: ARRAY OF REAL;
  confProb: Prob2Tail; VAR length: INTEGER);

  (* gives statistical values describing the data.
  N = number of observations at any observation point
  X = independent variable

  For further explanations see text of PROC GetSingleStatistics
  *)

(*****
(* Graphical display *)
*****)

PROCEDURE DeclDispMV(statArray: StatArray;
  mDepVar: Model; VAR mvDepVar: REAL;
  mIndepVar: Model; VAR mvIndepVar: REAL);
  (* Each data array to be displayed in the graph window must be associated
  with a dependent and an independent variable, which should both be declared
  as MVs in the client model. If time should be the independent variable,
  then SimGraphUtils.timeIsIndep can be given as parameter.
  See SimGraphUtils.DEF for description of the monitoring mechanism. *)

PROCEDURE DisplayArray(statArray: StatArray;
  withErrBars: BOOLEAN;
  confProb: Prob2Tail);
  (* The data are displayed in the graph if the following conditions are met:
  1. the associated MV must be set as isY
  2. the associated indepVar must be set as isX, respectively if the
  simulation time is chosen, none of the MVs must be set as isX.

  error bars with probability confProb are displayed, if withErrBars=TRUE
  and all observation points have an N = 2.
  If no values have been stored at any observation point, these values are

```

```

        displayed as undefVal. Make sure that undefVal lies outside your
        scaling range. *)

PROCEDURE DisplayAllArrays(withErrBars:  BOOLEAN;
                          confProb:      Prob2Tail);
  (* The data of all array are displayed. You can select the variables you
     want to display as isY. *)

  (*****)
  (* File output *)
  (*****)

  (* supports the file output of StatArray data together with labels,
     written on the top of the data and the independent variable values,
     written in the leftmost column. The data are written from the
     current position of the file f, which should be open. *)
TYPE
  RealFileFormat = RECORD
    rf:          RealFormat;
    n, dec:      CARDINAL;
  END;
  FileOutFormat = RECORD
    means, counts, sumsY, sumsYSquare,
    stdDevsY, confIntsY:
      BOOLEAN;
    indepsFormat, meansFormat, sumsYFormat, sumsYSquareFormat,
    stdDevsYFormat, confIntsYFormat:
      RealFileFormat;
    confProb:    Prob2Tail;
  END;
  (* The labels are written with the following suffixes:
     mean          -Ø
     count         -N'
     sum Y        - Y
     standard deviation  -stdev
     condifence intervals -CIL resp. -CIH for low and high limit *)

  VAR (* read only! *)
    meansOnly,
      (* writes only means *)
    meansSDCI,
      (* writes means, standard deviations and confidnce intervals *)
    allVals:      FileOutFormat;
      (* writes all stored and calculated values *)
      (* default RealFormat:
         rf = ScientificNotation;
         n  = 10
         dec = 5
         default confProb = prob950*)

PROCEDURE DumpStatArray (VAR f:          TextFile;
                        label:         Str31;
                        statArray:     StatArray;
                        fof:           FileOutFormat);
PROCEDURE DumpStatArrays(VAR f:          TextFile;
                        labels:        ARRAY OF Str31;
                        statArrays:    ARRAY OF StatArray;
                        fof:           FileOutFormat;
                        nArs:          INTEGER);
  (* The independent values of the first StatArray are written in the
     leftmost column. In case the arrays have not the same length, the
     length of the first array determines the number of values written.
     A character "N" is written in the positions where data are missing.
     Only the first nArs statArrays are dumped to the file. *)

END StochStat.

```



### D.3.10 *StructModAux*

*StructModAux* provides support for a model definition program, which consists of several modules, where each represents a submodel of a structured model. Mainly the dynamic activation and deactivation of submodels from within the simulation environment is made accessible via menu commands, even during simulations. For a typical usage of this optional auxiliary module see the sample model *GreenHouse*, which demonstrates the technique of modular modeling in the context of the green-house effect and the carbon fluxes between atmosphere and biosphere. The sample model *CarPollution* and the research sample model *LBM* also use *StructModAux*.

DEFINITION MODULE StructModAux;

(\*\*\*\*\*)

Module StructModAux (Version 1.0)

Copyright (c) 1993 by Andreas Fischlin and Swiss  
Federal Institute of Technology Zürich ETHZ

Purpose Utilities, which are of use when working with  
structured ModelWorks models

Remarks This module imports from the ModelWorks client  
interface

Implementation restriction: Assumes a single  
Master Model Definition Program (MDP), i.e. it can not  
support simultaneously more than one MDP!

Programming

o Design and Implementation  
A. Fischlin 4/1/94

Systems Ecology  
Institute of Terrestrial Ecology  
Department of Environmental Sciences  
Swiss Federal Institute of Technology Zurich ETHZ  
Grabenstr. 3  
CH-8952 Schlieren/Zurich  
Switzerland

Last revision of definition: 4/1/94 AF

(\*\*\*\*\*)

FROM DMMenu IMPORT Menu, Command;  
FROM SimBase IMPORT MWindowArrangement;

TYPE  
StructModelSet = BITSET;  
BooleanFct = PROCEDURE (): BOOLEAN;

VAR  
customM: Menu; (\* may be used to install more commands \*)  
chooseCmd: Command;

PROCEDURE InstallCustomMenu(title, chooseCmdTxt, chooseAlChr: ARRAY OF CHAR);  
(\*  
Installs a menu with the title 'title', and as the first

command (i.e. 'chooseCmd') a command with the text 'chooseCmdTxt'. The latter menu command is associated with procedure 'ChooseModel' and can also be activated with the alias char (keyboard equivalent) 'chooseAlChr'.

Typical usage: InstallCustomMenu("Models","Activation...","L");  
executed from within a InitSimEnv procedure.

See also example at the end of this definition!

\*)

```
PROCEDURE ChooseModel;
```

(\*

'ChooseModel' is the procedure associated with the menu command 'chooseMCmd' which allows the simulationist to activate previously installed sub models (see procedure 'AssignSubModel') dynamically. Note that this routine calls implicitly 'SetSimEnv'.

\*)

```
PROCEDURE AssignSubModel(VAR which: INTEGER; descr: ARRAY OF CHAR;
                        act,deact: PROC; isact: BooleanFct);
```

(\*

Installs a sub model with the descriptor 'descr' and uses the routines 'act', 'deact' respectively 'isact' to activate or deactivate respectively to investigate the current presence of the sub model. Upon successful assign the submodel gets the number 'which'; use it when calling procedure 'SetSimEnv', e.g. to denote those submodels you want to be active by default. NOTE: Implementation restriction, only up to a maximum of 16 sub models can be assigned. Submodels can't be deassigned, unless they have been assigned by a subprogram level which is to be terminated. In the latter case this module automatically deassignes any disappearing submodel.

\*)

```
PROCEDURE InstallMyGlobPreferences(myPrefs: PROC);
```

(\*

Installs routine 'myPrefs' which is used to define defaults of global parameters such as default window positions (e.g. by a call to routine 'PlaceGraphOnSuperScreen') or global simulation parameters (e.g. by a call to the routine 'SetDefltGlobSimPars' from module 'SimBase').

\*)

```
PROCEDURE SetSimEnv(sms: StructModelSet);
```

(\*

Sets the defaults (i.e. executes the previously installed routine 'myPrefs' whenever needed) and activates all the models specified in 'sms' according to the sequence in which sub models were installed by calls to the routine 'AssignSubModel'.

\*)

(\* =====

Typical example of a master module collecting several sub models by means of above routines:

...  
...

```
VAR
    atmos, bios, obs: INTEGER;
```

```
PROCEDURE InitSimEnv;
BEGIN
```

```

    InstallCustomMenu("Models","Activation...","L");
    SetSimEnv(atmos,obs); (* default activation *)
END InitSimEnv;

PROCEDURE SetMyGlobPreferences;
BEGIN
    SetDefltdGlobSimPars(1900.0, 2300.0, 0.5, 0.0001, 1.0, 10.0);
    PlaceGraphOnSuperScreen(tiled);
END SetMyGlobPreferences;

BEGIN (* body MyMaster *)
    InstallMyGlobPreferences(SetMyGlobPreferences);
    AssignSubModel(atmos, atmosModelDescr,
        ActivateAtmosModel, DeactivateAtmosModel, AtmosModelIsActive);
    AssignSubModel(bios, biosModelDescr,
        ActivateBiosModel, DeactivateBiosModel, BiosModelIsActive);
    AssignSubModel(obs, obsModelDescr,
        ActivateObsModel, DeactivateObsModel, ObsModelIsActive);
    RunSimEnvironment( InitSimEnv );
END MyMaster;

===== *)

END StructModAux.

```

### D.3.11 *TabFunc*

*TabFunc* allows to compute function values by linear interpolation and extrapolation from so-called table functions, i.e. functions only given by a series of x,y-value pairs instead of an analytically defined function such as  $y = \sin x$ . Besides algorithms for inter- and extrapolation *TabFunc* has also an user interface, which allows to inspect and edit table functions interactively via a table or a graphical display. For a typical usage of this optional auxiliary module see the sample model *UseTabFunc*<sup>1</sup> and *SwissPop*.

This section does not list the definition module of *TabFunc*, instead it describes the user as well as the client interface of this module in more details.

#### D.3.11.a User Interface

As soon as at least one table function has been declared successfully (see also below section *Declaration of table functions*) the module *TabFunc* activates a user interface. It consists mainly of the menu *TabFuncs* (Fig. A10) and some entry forms associated with its menu commands, plus a window *Table Function Editor*.

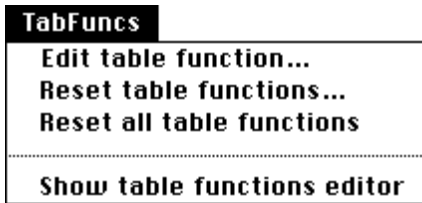


Fig. A10: Menu *TabFuncs*

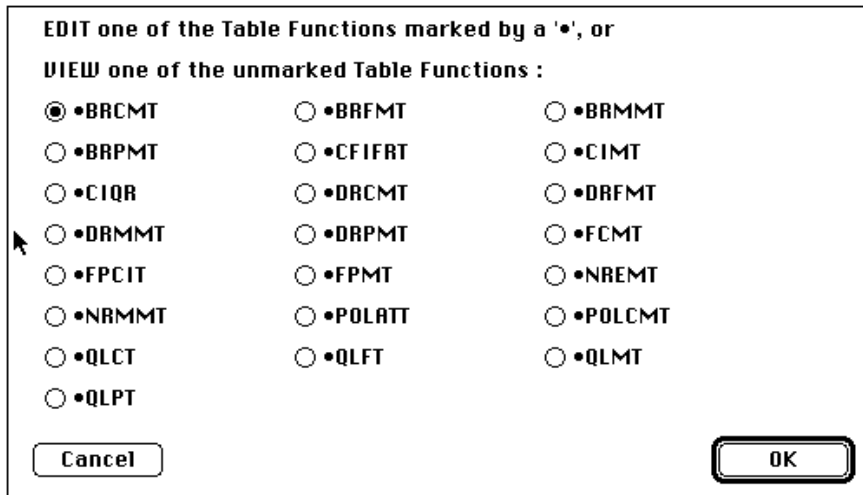


Fig. A11: Entry form to select a table function for the editing or viewing of the values of a particular table function. The functions are listed with their identifiers only. Functions marked with a '\*' are modifiable, i.e. they can actually be edited, in contrast to those which can only be viewed.

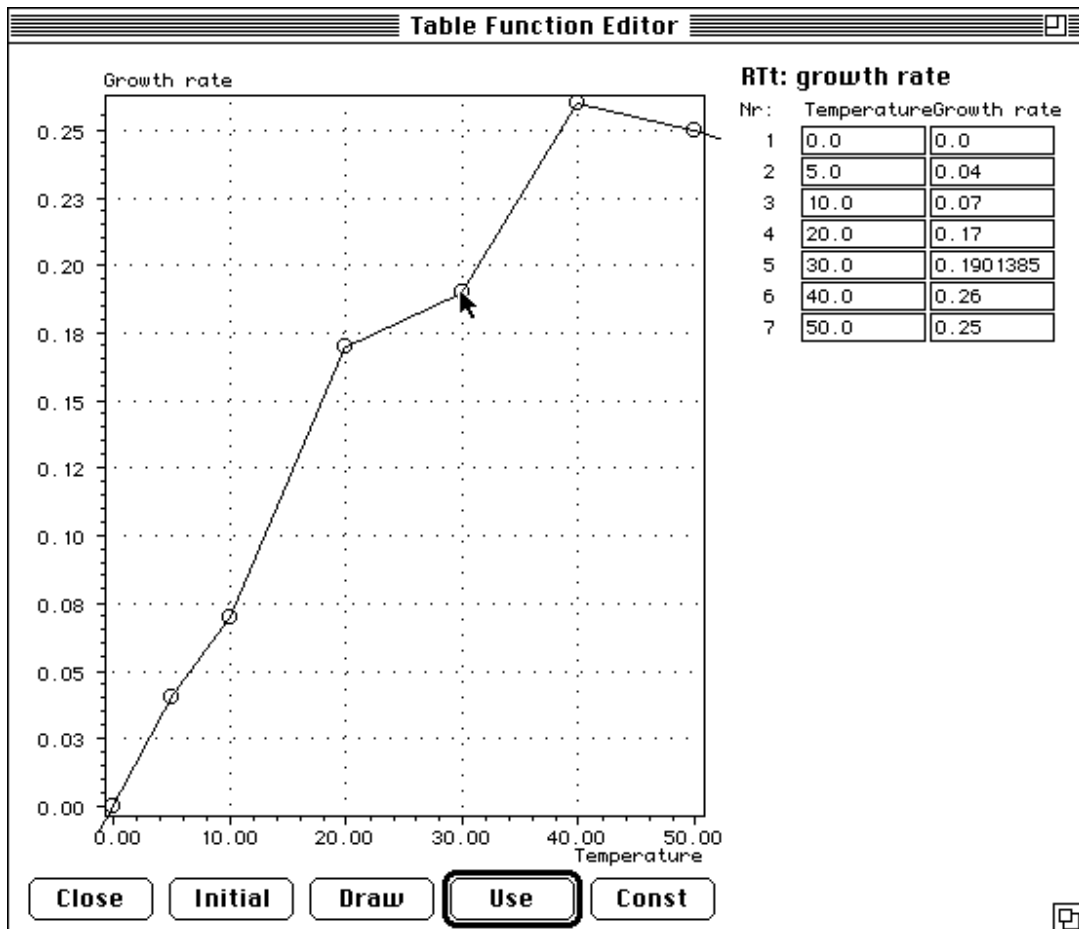
<sup>1</sup>Only distributed but not listed in this *Appendix*

The menu serves the editing and resetting of the declared table functions.

*Edit table function...:* Lets the user edit or inspect a table function.

First the table function has to be selected by clicking into the corresponding radio button in an entry form similar to the one shown in Fig. A11. Every table function is identified via its name (formal parameter *tabName* of procedure *DeclTabF*).

Secondly the window *Table Function Editor* is displayed, which looks similar to the example shown in Fig. A12. A graph of the current table function is drawn to the left; a table of the current x,y-pairs, i.e. supporting points, is shown to the right.



**Fig. A12:** *Table Function Editor* to display (*Draw*) and edit values of a table function. In contrast to analytical functions, such a function is defined by interpolation within a table of x,y-pairs, i.e. a series of supporting points. Modifiable table functions can be edited, either by dragging supporting points in the graph visible in the left part of the window, or by typing new values for supporting points in the table shown to the right. The depicted table function (the one from the sample model *UseTabFunc.MOD*) returns values computed from linear interpolation (within the range [0,50]) respectively extrapolation (outside [0,50]); since *ExtrapolMode = lastSlope*, extrapolation slopes are given by the closest two neighboring points, i.e. first two resp. last two points).

A modifiable table function can be edited in three ways: First by entering new values in the fields of the table, second in the graph by dragging a supporting point with the mouse, and third by transforming the function into a constant.

First editing method: The left column in the table contains the values of the independent variable (x-values) and the right column the values of the corresponding dependent function value (y-values). Every [x,y]-pair defines a supporting point of the function. Once editing is completed, the push button *Draw* allows to see the new table function in graphical form.

Except for the following restrictions, the tabulated values may be edited freely: It is required that the x-values are always in ascending order and that all x-values must be different, i.e. if there are  $n$  x-values with the index  $i$ , they must satisfy the condition  $x_1 < x_2 < x_3 < \dots < x_i < \dots < x_{n-1} < x_n$ . The user is asked to correct values which do not satisfy this condition.

In addition to typing a series of [x,y]-pairs, it is also possible to enter a single value, hereby transforming the whole table function into a single constant (see below third editing method).

Second editing method: In order to change a dependent function value (y-value), drag the circle which denotes the corresponding point vertically. You may drag a point outside the graph's panel, as long as the range limits defined at declaration time (see below section *Declaration of table functions*) are not exceeded. To change a value of the independent variable (x-value), press the option-key while dragging in a horizontal direction. However, in the latter case you are not allowed to drag a point beyond the adjacent x-values. The point's new numerical values, visible in the table to the right, are updated accordingly.

Third editing method: In order to quickly transform a whole table function into a single constant or model parameter, enter the constant in the top field of the x-values and click into the push button *Const*. The top x-value is then copied to all other x-values and the graph redrawn.

Even if a table function is not modifiable, it is possible to open the window *Table Function Editor*. In this case the corresponding graph is depicted and the x,y-values of the supporting points are tabulated. However, neither dragging of points nor editing of values is possible.

The push buttons at the lower left corner may be used for editing or to issue commands.

-- Push button *Close* (or the keyboard equivalent **W**) closes the window; in case the table function has actually been modified, the user is first asked whether the changes shall be really used (see above button *Use*) or whether the actual table function shall remain untouched, exactly as it was before the menu command *TabFuncs/Edit/View table function...* has been chosen.

Push button *Initial* discards all previous editing and reverts the values originally specified when the table function has been declared (see below section *Declaration of table functions*) as if the user would have retyped these values into the table in the upper right corner. Thus, do not confound this with a reset (see menu command *TabFuncs/Reset table functions*), since the equivalent to a reset would require to perform actually the following: First to push the button *Initial*, then without any editing inbetween to push the button *Use* immediately after.

- Push button *Draw* to (re)draw the graph with the current supporting values, as shown in the table at the right of the graph. You may push this button as many times you wish, of course also while editing the table.

- Push button *Use* accepts the edited coordinates of the supporting points as the new current values and redraws the graph. Note that from then on all interpolations or extrapolations done with this table function will use the new values. (Since it is the default button, it may also be pushed via a keyboard equivalent, i.e. by pressing either the key *Return* or *Enter*).
- Push button *Const* supports the third editing method (see above) and allows to turn a table function quickly into a single model parameter.

*Reset table function...:* Resets the values of an individual table function to the values specified when the table function has been declared. This command displays an entry form similar to the one shown in Fig. A13, which lists all modifiable table functions. Select the table function to be reset by clicking into the corresponding radio button. Again table functions are listed by their names (formal parameter *tabName* of procedure *DeclTabF*). This command is equivalent to first selecting a table function for editing via the menu command *TabFuncs/Edit/View table function...* plus pushing the buttons *Initial*, *Use*, and *Close* from within the window *Table Function Editor*.

RESET one of the following Table Functions:

<input checked="" type="radio"/> BRCMT	<input type="radio"/> BRFMT	<input type="radio"/> BRMMT
<input type="radio"/> BRPMT	<input type="radio"/> CFIFRT	<input type="radio"/> CIMT
<input type="radio"/> CIQR	<input type="radio"/> DRCMT	<input type="radio"/> DRFMT
<input type="radio"/> DRMMT	<input type="radio"/> DRPMT	<input type="radio"/> FCMT
<input type="radio"/> FPCIT	<input type="radio"/> FPMT	<input type="radio"/> NREMT
<input type="radio"/> NRMMT	<input type="radio"/> POLATT	<input type="radio"/> POLCMT
<input type="radio"/> QLCT	<input type="radio"/> QLFT	<input type="radio"/> QLMT
<input type="radio"/> QLPT		

CANCEL OK

Fig. A13: Entry form to select a particular table function for its resetting. Only modifiable functions, denoted by their identifiers, are listed.

*Reset all table functions:* Resets all modifiable table functions to their originally declared values without asking the user for a selection of a particular table function.

*Show table functions editor:* Shows the table function editor window by bringing it to the front.

The user interface, in particular the menu *TabFuncs*, vanishes as soon as the last table function has been removed (see also below section *Removing table functions*).

#### D.3.11.b Declaration of table functions

It is recommended to initialize the variables of type *TabFUNC* with the value *notExistingTabF* within the body of the corresponding scope before declaring the table functions.

```
TYPE TabFUNC;

VAR notExistingTabF: TabFUNC; (* read only! *)
```

Table functions may be declared with the procedure *DeclTabF* or *DeclTabFM*. The first alternative requires to specify the values with two arrays of reals (formal parameters *xx* and *yy* of procedure *DeclTabF*). The second allows to use the data type *Matrix* (formal parameter *xyVecs* of procedure *DeclTabF*) from the auxiliary module *Matrices*. Once declared there will be no difference between table functions, regardless of their declaration method.

```

PROCEDURE DeclTabF( VAR   t           : TabFUNC;
                   xx, yy  : ARRAY OF REAL;
                   NValPairs : INTEGER;
                   modifiable : BOOLEAN;
                   tabName,
                   xName, yName,
                   xUnit, yUnit : ARRAY OF CHAR;
                   xMin, xMax,
                   yMin, yMax   : REAL );

PROCEDURE DeclTabFM( VAR   t           : TabFUNC;
                    xyVecs      : Matrix;
                    modifiable   : BOOLEAN;
                    tabName,
                    xName, yName,
                    xUnit, yUnit : ARRAY OF CHAR;
                    xMin, xMax,
                    yMin, yMax   : REAL );

```

The meaning of the formal parameters is as follows:

- t*                    The variable *t* is of the opaque type *TabFUNC* . It is used to identify the table function. If a table function is already associated to *t* a warning will be displayed and *t* is left untouched.
- xx, yy*                The vector *xx* contains the independent and *yy* the dependent values of the table function being defined. The elements of the *xx* vector must be in ascending order otherwise the table function will not be declared.
- xyVecs*                This matrix contains the values of the independent variable in the first column and the values of the dependent variable of the table function in the second column. Again the independent values must be given in ascending order, otherwise the table function will not be declared.
- NValPairs*            Number of elements in the *xx* and *yy* vectors holding a valid value.
- modifiable*           If TRUE the table function may be modified by the table function editor, otherwise only viewed.
- tabName*              Is used for the identification of the table function in the user interface, such as its selection e.g. in the table function editor's entry form (Fig. A11 and A4).
- xName, yName, xUnit, yUnit*    The names of the table function's axis variables and their unit. These strings will be used in the table function editor window.
- xMin, xMax, yMin, yMax*    Define the upper and lower bounds for each axis. Attempts to drag points or to enter values outside of these ranges are not possible or will not be accepted by the table function editor. If during declaration any value of the independent or dependent variables is outside of these ranges, a warning message will be displayed and the table function will not be declared.

Note that if *DeclTabF* or *DeclTabFM* declares the first table function, the menu *TabFuncs* will be installed and becomes visible, given the "Dialog Machine" is currently running (see above section *User Interface*).



D.3.11.c Modification of table functions

```

PROCEDURE GetTabF( t: TabFUNC;
  VAR xx, yy      : ARRAY OF REAL;
  VAR NValPairs  : INTEGER;
  VAR modifiable : BOOLEAN;
  VAR tabName,
      xName, yName,
      xUnit, yUnit : ARRAY OF CHAR;
  VAR xMin, xMax,
      yMin, yMax   : REAL );

PROCEDURE SetTabF( t      : TabFUNC;
  xx, yy      : ARRAY OF REAL;
  NValPairs  : INTEGER;
  modifiable : BOOLEAN;
  tabName,
  xName, yName,
  xUnit, yUnit : ARRAY OF CHAR;
  xMin, xMax,
  yMin, yMax   : REAL );

PROCEDURE GetTabFM( t      : TabFUNC;
  VAR xyVecs  : Matrix;
  VAR modifiable : BOOLEAN;
  VAR tabName,
      xName, yName,
      xUnit, yUnit : ARRAY OF CHAR;
  VAR xMin, xMax,
      yMin, yMax   : REAL );

PROCEDURE SetTabFM( t      : TabFUNC;
  xyVecs  : Matrix;
  modifiable : BOOLEAN;
  tabName,
  xName, yName,
  xUnit, yUnit : ARRAY OF CHAR;
  xMin, xMax,
  yMin, yMax   : REAL );

```

The procedures *GetTabF* and *GetTabFM* retrieve the current values of the table function *t*. The procedures *SetTabF* and *SetTabFM* redefine the table function *t*; you may even change its dimensions by passing for parameter *NValPairs* another value than used during the declaration of *t*. If the table function *t* does not exist, a warning will be displayed. The meaning of the formal parameters is exactly the same as explained in section *Declaration of table functions*

```
PROCEDURE EditTabF ( t: TabFUNC );
```

This procedure opens the window *Table Function Editor* and allows to edit the table function *t*.

```
PROCEDURE ResetTabF( t: TabFUNC );
```

This procedure resets a table function to the original values specified when it has been declared and discards all interactive editing. Every successful call of *DeclTabF* respectively *DeclTabFM*, or *SetTabF* respectively *SetTabFM*, sets new default as well as current values, thus performs also an implicit reset. Any editing via the table function editor affects only the current values.

```

PROCEDURE FreezeEditorGraphBounds( VAR t      :TabFUNC;
                                   xMin, xMax,
                                   yMin, yMax : REAL );
PROCEDURE UnfreezeEditorGraphBounds( VAR t:TabFUNC );

```

The call of the procedure *FreezeEditorGraphBounds* freezes the range of the axes between the values *xMin* and *xMax*, resp. *yMin* and *yMax*. Before a call of this procedure or after the call of

*UnfreezeEditorGraphBounds* the scaling of the axes shown in the graph of the window *Table Function Editor* is adjusted such that the curve fills always the entire graph. To suppress this autoscaling of the axes call *FreezeEditorGraphBounds*, albeit, note that this may lead to a situation where the graph might show no part of the curve at all.

D.3.11.d Inter- and extrapolations with table functions

Table functions allow to compute function values within the defined domain [ *xx[1]*, *xx[n]* ] by linear interpolation or outside this range by extrapolation (Fig. A14) by using one of the following function procedures:

```
PROCEDURE Yie( t: TabFUNC; x: REAL ): REAL;
PROCEDURE Yi ( t: TabFUNC; x: REAL ): REAL;
```

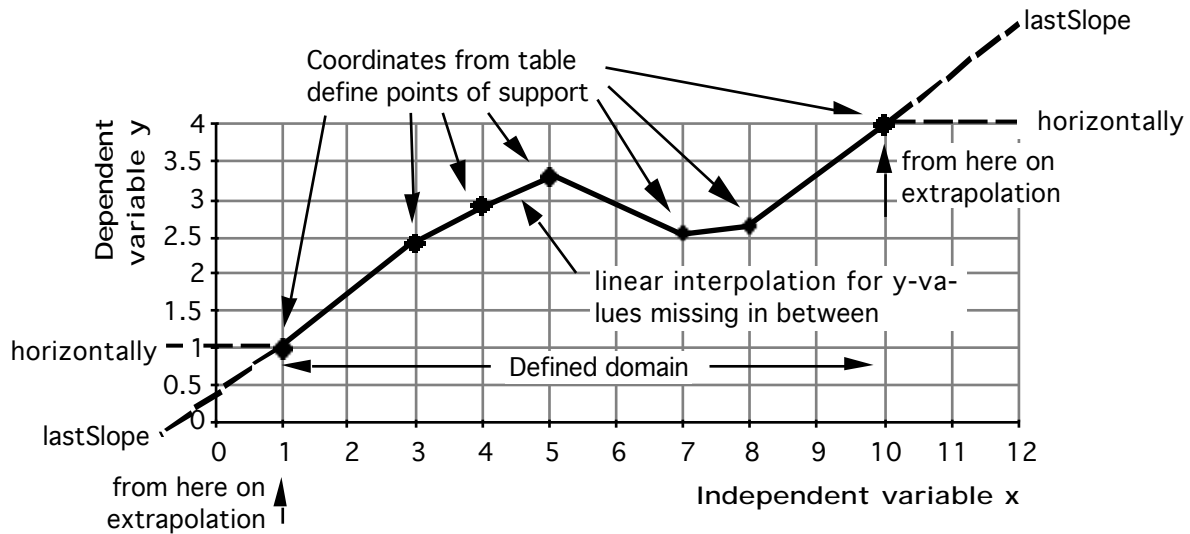


Fig. A14: Interpolation and extrapolations computed by the function procedures *Yie* (inter- and extrapolation) and *Yi* (only interpolation) for a function declared as a so-called table function. The table function is defined by supporting points given in form of coordinates within the domain of definition. Inside the domain *Yi* and *Yie* compute linear interpolations, outside *Yie* computes linear extrapolations, depending on the mode either horizontally or along the slope defined by the two adjacent supporting points.

Extrapolations are allowed if the function procedure *Yie* is used (read *Yie* as follows: returns dependent value Y by linear inter- or extrapolation). If you use *Yi* (returns dependent value Y by interpolation only) any attempt to compute a function value *y* for an independent value *x* outside the defined range [ *xx[1]*, *xx[n]* ] will result in a warning message, but the value returned is the same as if *Yie* would have been called.

```
TYPE ExtrapolMode = ( lastSlope, horizontally );
PROCEDURE DefineExtrapolationMode( VAR t:TabFUNC; extrapolation: ExtrapolMode );
PROCEDURE ExtrapolationMode ( t:TabFUNC ): ExtrapolMode;
```

The above procedures allow to define the extrapolation mode of the table function.

The extrapolation modes are defined as follows (*n* = number of value pairs):

*horizontally:*

$$\begin{array}{ll} y = yy[1] & \text{if } x < xx[1] \\ y = yy[n] & \text{if } x > xx[n] \end{array}$$

*lastSlope:*

$$y = yy[1] + \frac{(x - xx[1]) (yy[2] - yy[1])}{xx[2] - xx[1]} \quad \text{if } x < xx[1]$$

$$y = yy[n] + \frac{(x - xx[n]) (yy[n] - yy[n-1])}{xx[n] - xx[n-1]} \quad \text{if } x > xx[n]$$

### D.3.11.e Removing table functions

Table functions can be removed by calling the following procedure:

```
PROCEDURE RemoveTabF (VAR t: TabFUNC);
```

Upon a successful return from *RemoveTabF* *t* has the value *notExistingTabF*. Note that if *RemoveTabF* removes the last table function, the menu *TabFuncs* will also be removed (see above section *User Interface*).

### D.3.12 WriteDatTim

*WriteDatTim* may be used to record data and time at the begin and end of a long, e.g. several hours lasting structured simulation (see sample model *Markov* for such a use). This module is best used in conjunction with the module *DMClock*, which allows to access the internal, built in clock in a hardware independent way.

```
DEFINITION MODULE WriteDatTim;
```

```
(*****
```

```
Module WriteDatTim (Version 2.02)
```

```
Copyright ©1988 by Andreas Fischlin and CELTIA,  
Swiss Federal Institute of Technology Zürich ETHZ
```

```
Version for MacMETH V2.6.2 1-Pass Modula-2 implementation
```

```
Purpose Writing of date and time
```

```
Programming
```

- Design/Implementation  
A. Fischlin (16/Mai/88)

```
Swiss Federal Institute of Technology Zurich  
Project Centre IDA  
Pilot Project CELTIA  
[Computer-aided Explorative Learning and Teaching  
with Interactive Animated Simulation]  
ETH-Zentrum  
CH-8092 Zurich  
Switzerland
```

```
Last revision: 25 Nov 90 (A.F.)
```

```
*****)
```

```

CONST
  Jan = 1; Feb = 2; Mar = 3; Apr = 4; Mai = 5; Jun = 6;
  Jul = 7; Aug = 8; Sep = 9; Oct = 10; Nov = 11; Dec = 12;
  Sun = 1; Mon = 2; Tue = 3; Wed = 4; Thur = 5; Fri = 6; Sat = 7;

TYPE
  Months = INTEGER;
  WeekDays = INTEGER;
  DateAndTimeRec =
    RECORD
      year: INTEGER;          (* 1904,1905,...2040 *)
      month: Months;
      day,                    (* 1,...31 *)
      hour,                   (* 0,...,23 *)
      minute,                 (* 0,...,59 *)
      second: INTEGER;       (* 0,...,59 *)
      dayOfWeek: WeekDays;   (* Sun = 1, Sat = 7 *)
    END;

  WriteProc = PROCEDURE (CHAR);
  DateFormat = ( brief,      (* only numbers: e.g. 31/05/88 *)
                letMonth,   (* month in letters: e.g. 31/Mai/1988 *)
                full       (* full in letters: e.g. 31st Mai 1988 *)
                );
  TimeFormat = ( brief24h,   (* 24 hour format brief: e.g. 23:15 *)
                brief24hSecs, (* 24 hour brief & secs: e.g. 23:15:02 *)
                let24hSecs, (* hour in letters: e.g. 23h 15' 02" *)
                full24hSecs, (* full in letters: e.g. 23 hours
                               15 minutes 02 seconds*)
                brief12h    (* 24 hour format brief: e.g. 11:15 pm *)
                );

(* the following procedures write information in English only *)
PROCEDURE WriteDate(d: DateAndTimeRec; w: WriteProc; df: DateFormat);
PROCEDURE WriteTime(d: DateAndTimeRec; w: WriteProc; tf: TimeFormat);

END WriteDatTim.

```

## E Quick References

### E.1 AUXILIARY LIBRARY

The following quick reference lists the exports of the more important modules contained in the auxiliary library *AuxLib*. Its modules are all directly or indirectly based on the "Dialog Machine" only. Some modules import also from the client interface of ModelWorks, some also from other auxiliary modules. For details on how to work with the auxiliary library see part II *Theory*, chapter *ModelWorks Functions*, section *Module structure of ModelWorks*, and this appendix section *Sample Models*.

Auxiliary Library Modules based on the Dialog Machine Version 2.2 © 1994 Andreas Fischlin, Olivier Roth, Dimitrios Gyalistras, and Markus Ulrich  
Swiss Federal Institute of Technology Zurich ETHZ, Switzerland.

```
(===== SYSTEM ECOLOGY MODULES =====)

(***** Buttons *****)

TYPE Button; ButtonActionProc = PROCEDURE( Button );
ButtonDrawProc = PROCEDURE( Button, RectArea ); PaletteDrawProc = PROCEDURE( INTEGER );

VAR notInstalledButton: Button;

PROCEDURE InstallButton ( VAR btn : Button; btnFrame : RectArea;
                          buttonAction: ButtonActionProc; drawButton : ButtonDrawProc );
PROCEDURE ButtonExists( btn: Button ): BOOLEAN;
PROCEDURE RemoveButton( VAR btn: Button ); PROCEDURE RemoveAllButtonsOfWindow( w: Window );
PROCEDURE SetToDefaultButton ( w: Window; btn: Button; drawDeflBtn: ButtonDrawProc );
PROCEDURE NoDefaultButton ( w: Window );
PROCEDURE GetDefaultButton( w: Window; VAR btn: Button );
PROCEDURE SetButtonAliasChar ( btn: Button; modif: BITSET; aliasChar: CHAR );
PROCEDURE DisableButton( btn: Button ); PROCEDURE EnableButton ( btn: Button );
PROCEDURE IsEnabled ( btn: Button ): BOOLEAN;
PROCEDURE SetButtonNr(btn: Button; btnNr: INTEGER); PROCEDURE ButtonNr ( btn: Button ): INTEGER;
PROCEDURE OwnerWindow( btn: Button ): Window;
PROCEDURE SetButtonAttr(btn: Button; btnFrame: RectArea;
                        btnAction: ButtonActionProc; drawButton: ButtonDrawProc);
PROCEDURE GetButtonAttr(btn: Button; VAR btnFrame: RectArea;
                        VAR btnAction: ButtonActionProc; VAR drawButton: ButtonDrawProc);
PROCEDURE DrawTextButton ( btnFrame: RectArea; butText: ARRAY OF CHAR );
PROCEDURE DrawDeflButtonFrame ( frame: RectArea );
PROCEDURE AggregatePalette( palNr: INTEGER; fstBtn, lstBtn: Button; drawPalette: PaletteDrawProc );
PROCEDURE DummyButtonDrawing( dummyBtn: Button; dummyBtnFrame: RectArea );
PROCEDURE OwnerPalette(btn: Button ): INTEGER;
PROCEDURE GetPaletteDrawProc( palNr: INTEGER; VAR pdp: PaletteDrawProc; VAR done: BOOLEAN );
PROCEDURE DisaggregatePalette( palNr: INTEGER );
PROCEDURE RedrawAllButtons ( w: Window );
PROCEDURE DimmAllDisabledButtons ( w: Window );
PROCEDURE DoForAllButtonsOfWindow ( w: Window; proc: ButtonActionProc );

(***** CellAutoOut *****)

TYPE CellAutoID; Symbol = CHAR; UnderlayMode = (underlayWhite, dontUnderlay);
CellProcess = PROCEDURE (INTEGER, INTEGER);

VAR notExistingCA: CellAutoID;

PROCEDURE DeclCellOutput(VAR caID: CellAutoID; VAR outWindow: Window; plotFrame: RectArea;
                          numX, numY: INTEGER; withGridlines: BOOLEAN);
PROCEDURE RemoveCellOutput(VAR caID: CellAutoID); PROCEDURE RemoveAllCellOutputs;
PROCEDURE CellOutputExists(caID: CellAutoID): BOOLEAN;
PROCEDURE SelectCellOutput(caID: CellAutoID); PROCEDURE GetCurCellOutput(VAR caID: CellAutoID);
PROCEDURE ClearCell(x,y: INTEGER);
PROCEDURE FillCell(x,y: INTEGER; pattern: Pattern; col: Color);
PROCEDURE DrawInCell(x, y: INTEGER; sym: Symbol; underlay: UnderlayMode);
PROCEDURE DoForAllCells(doCellProc: CellProcess);
PROCEDURE CalcCellArea(x,y: INTEGER): RectArea;
PROCEDURE CalcCellMiddle(x, y: INTEGER; VAR xCoord,yCoord: INTEGER);
PROCEDURE GetPlotFrame(caID: CellAutoID): RectArea;

(***** Confidence *****)

PROCEDURE FInvNormalStand(alfa: REAL): REAL; (*  $\mu = 0$ ,  $\sigma = 1$  *)
PROCEDURE FInvNormal (mu,sigma,alfa: REAL): REAL;
PROCEDURE FInvStudent (mu: INTEGER; alfa: REAL): REAL;
PROCEDURE FInvChiSquare (mu: INTEGER; alfa: REAL): REAL;
PROCEDURE FInvF (mu1,mu2: INTEGER; alfa: REAL): REAL;
PROCEDURE FInvBinomial (k,N: INTEGER; alfa: REAL): REAL;
PROCEDURE FInvPoisson (lambda: INTEGER; alfa: REAL): REAL;
PROCEDURE FInvNegBinomial(mu,k: REAL; alfa: REAL): REAL;

(***** FileNameStrs *****)

VAR extSeparator, pathSeparator, volSeparator: CHAR;

PROCEDURE StripExt(fromName: ARRAY OF CHAR; VAR toName: ARRAY OF CHAR);
PROCEDURE SetNewExt(fromName, newExt: ARRAY OF CHAR; VAR toName: ARRAY OF CHAR);
PROCEDURE ExtractExt(pathAndFileName: ARRAY OF CHAR; VAR extension: ARRAY OF CHAR);
PROCEDURE ExtractFileName(pathAndFileName: ARRAY OF CHAR; VAR fName: ARRAY OF CHAR);
PROCEDURE ExtractPath(pathAndFileName: ARRAY OF CHAR; VAR path: ARRAY OF CHAR);
PROCEDURE ExtractRelPath(fullPath, basePath: ARRAY OF CHAR; VAR relPath: ARRAY OF CHAR );
PROCEDURE ExtractVolName(fullPath: ARRAY OF CHAR; VAR volName: ARRAY OF CHAR );
PROCEDURE SplitPathFileName(pathAndFileName: ARRAY OF CHAR; VAR path,fName: ARRAY OF CHAR);
PROCEDURE SplitVolPathFileName(fullPath: ARRAY OF CHAR; VAR volN, pathN, fileN: ARRAY OF CHAR );
PROCEDURE CompletePath(basePath, relPath: ARRAY OF CHAR; VAR fullPath: ARRAY OF CHAR );
```

## ModelWorks V2.2 - Appendix (Interfaces and Libraries)

```

PROCEDURE CompletePathFileName (volN, pathN, fName: ARRAY OF CHAR; VAR fullPathAndFileName: ARRAY OF CHAR );
(***** Help *****)

PROCEDURE ShowHelpWindow;
PROCEDURE SetHelpFileName (fn: ARRAY OF CHAR); PROCEDURE SetResourceFileName (fn: ARRAY OF CHAR );
PROCEDURE SetInstallationErrorHandler ( errMsg: PROC );
PROCEDURE SetDebugMode ( debugOn: BOOLEAN );
PROCEDURE ResetHelp;

(***** Histograms *****)

TYPE Histogram; HistoAct = PROCEDURE (Histogram);

PROCEDURE DefineHistogram(w: Window; VAR h: Histogram; r: RectArea; fromClass,toClass: INTEGER;
    xLabel: ARRAY OF CHAR; maxFrequency: CARDINAL; freqNumbs: BOOLEAN;
    barCol: Color; barPat: Pattern);
PROCEDURE SetYTickInterval( h: Histogram; interval: INTEGER );
PROCEDURE ClearHistogram(h: Histogram); PROCEDURE DrawHistogram(h: Histogram);
PROCEDURE MidTopPoint(h: Histogram; class: INTEGER; f: CARDINAL; VAR x,y: INTEGER);
PROCEDURE PlotBar(h: Histogram; class: INTEGER; f: CARDINAL);
PROCEDURE SetPlotBarMode( h: Histogram; wipeOut: BOOLEAN );
PROCEDURE GetPlotBarMode( h: Histogram; VAR wipeOut, done: BOOLEAN );
PROCEDURE RemoveHistogram(VAR h: Histogram); PROCEDURE DoForAllHistograms(p: HistoAct);

(***** IdentifyPars *****)

TYPE RealFct = PROCEDURE (): REAL; MinMethod = (halfDouble, amoeba, price, random, Brent, powell, simplex);
PROCEDURE MarkParForIdentification( VAR p: REAL ); PROCEDURE UnmarkParForIdentification( VAR p: REAL );
PROCEDURE UnmarkAllParsForIdentification;
PROCEDURE SetDeflTMinim( meth: MinMethod; maxIter: INTEGER; convC: REAL);
PROCEDURE GetDeflTMinim( VAR meth: MinMethod; VAR maxIter: INTEGER; VAR convC: REAL);
PROCEDURE MinimizeAfterDialog( func: RealFct );
PROCEDURE Minimize( method: MinMethod; maxIter: INTEGER; func: RealFct );

(***** Jacobi *****)

CONST VecSize=40;

TYPE Vector = ARRAY [1..VecSize] OF REAL; Matrix = ARRAY [1..VecSize] OF Vector;

PROCEDURE Jacobi(VAR mat: Matrix; dim: INTEGER; VAR eigVals: Vector; VAR eigVecs: Matrix; VAR numRot: INTEGER);
PROCEDURE EigSort(VAR eigVals: Vector; VAR eigVecs: Matrix; dim: INTEGER );

(***** JulianDays *****)

CONST Jan = 1; Feb = 2; Mar = 3; Apr = 4; Mai = 5; Jun = 6;
    Jul = 7; Aug = 8; Sep = 9; Oct = 10; Nov = 11; Dec = 12;
    Sun = 1; Mon = 2; Tue = 3; Wed = 4; Thur = 5; Fri = 6; Sat = 7;

TYPE Month = [Jan..Dec]; WeekDay = [Sun..Sat];
DateAndTime = RECORD
    year: INTEGER; (* e.g. 1582,...,1994,...,2040 etc.*)
    month: Month; day: INTEGER; (* [1..31] (depends on month) *)
    hour, (* [0..23] *)
    min: INTEGER; sec: INTEGER; (* [0..59] *)
    dayOfWeek: WeekDay; (* e.g. Sun *)
    secFrac: REAL; (* fraction of a second, e.g. 0.13 for 13 hundredth of a second *)
END;
PROCEDURE DateTimeToJulDay(dt: DateAndTime): LONGREAL;
PROCEDURE JulDayToDateTime(jd: LONGREAL; VAR dt: DateAndTime);
PROCEDURE DateToJulDay(day,month,year: INTEGER): LONGINT;
PROCEDURE JulDayToDate(jd: LONGINT; VAR day: INTEGER; VAR month: Month; VAR year: INTEGER; VAR dayOfWeek: WeekDay);
PROCEDURE IsLeapYear(yr: INTEGER): BOOLEAN;
PROCEDURE SetCalendarRange(firstYear,lastYear,firstSunday: INTEGER);

(***** Lists *****)

TYPE List; SelectionMode = (single, multipleAdjacent, multipleDisconnected);
DispListItemProc = PROCEDURE ( ADDRESS, INTEGER, INTEGER );
ItemSelection = ( selected, notSelected, all );
ListItemProc = PROCEDURE ( ADDRESS ); ListItemWhileProc = PROCEDURE ( ADDRESS, VAR BOOLEAN );
IsSuccessorProc = PROCEDURE ( ADDRESS, ADDRESS ): BOOLEAN;
ConditionProc = PROCEDURE ( ADDRESS ): BOOLEAN;

VAR noList: List; (* read only variable! *)

PROCEDURE DeclList ( VAR list: List; listName: ARRAY OF CHAR );
PROCEDURE RemoveList( VAR list: List );
PROCEDURE ListExists( list: List ): BOOLEAN;
PROCEDURE InsertInList ( list: List; aListItem, beforeItem: ADDRESS );
PROCEDURE DeleteFromList( list: List; VAR aListItem: ADDRESS );
PROCEDURE ListItemExists( list: List; listItem: ADDRESS ): BOOLEAN;
PROCEDURE DoWithListItems ( list: List; doWith: ItemSelection; doSomething: ListItemProc );
PROCEDURE DoWithListItemsWhile( list: List; doWith: ItemSelection; doSomething: ListItemWhileProc );
PROCEDURE SortList( list: List; isSuccessor: IsSuccessorProc );
PROCEDURE InstallLISBox(list: List; window: Window; scrBFrame: RectArea; title: ARRAY OF CHAR;
    dispListItem: DispListItemProc; cellW, cellH: INTEGER; selMode: SelectionMode);
PROCEDURE RemoveLISBox( list: List );
PROCEDURE RedrawLISBox( list: List );
PROCEDURE DeclLISBox ( list: List; window: Window; scrBFrame: RectArea; title: ARRAY OF CHAR;
    dispListItem: DispListItemProc; cellW, cellH: INTEGER; selMode: SelectionMode);
PROCEDURE SetLISBoxAttr( list: List; title: ARRAY OF CHAR; dispListItem: DispListItemProc;
    selMode: SelectionMode );
PROCEDURE GetLISBoxAttr( list: List; VAR title: ARRAY OF CHAR; VAR dispListItem: DispListItemProc;
    VAR selMode: SelectionMode);
PROCEDURE SetLISBoxFraming( li: List; boxFramed: BOOLEAN );
PROCEDURE GetLISBoxFraming( li: List; VAR boxFramed: BOOLEAN );
PROCEDURE SetScrollBarPlace( li: List; dx,dy,h: INTEGER );
PROCEDURE GetScrollBarPlace( li: List; VAR dx,dy,h: INTEGER );
PROCEDURE ScrollLISBox( li: List; by: INTEGER );
PROCEDURE FlipLISBox( li: List; direction: BOOLEAN );
PROCEDURE EnableLISBox ( list: List );
PROCEDURE DisableLISBox( list: List );
PROCEDURE ToggleLISBoxItem( li: List; item: ADDRESS; shifted: BOOLEAN );
PROCEDURE SetSelectionForAllIf( li: List; ifp: ConditionProc; isSelected: BOOLEAN);
PROCEDURE SetSelectionForAll( li: List; isSelected: BOOLEAN);
PROCEDURE JumpToLISBoxItem( list: List; item: ADDRESS );
PROCEDURE TopLISBoxItem( list: List ): ADDRESS;
PROCEDURE BotLISBoxItem( list: List ): ADDRESS;
PROCEDURE NextLISBoxItem( list: List; item: ADDRESS ): ADDRESS;
PROCEDURE PrevLISBoxItem( list: List; item: ADDRESS ): ADDRESS;
PROCEDURE IsSelected( list: List; item: ADDRESS ): BOOLEAN;
PROCEDURE GetListItemSelection( list: List; VAR lis: ARRAY OF ADDRESS; VAR nSelected: INTEGER );
PROCEDURE SetListItemSelection( list: List; VAR lis: ARRAY OF ADDRESS; nSelected: INTEGER );

```

```

(***** Matrices *****)
TYPE Matrix; Cell = RECORD row, col: INTEGER; END(*Cell*);
  Selection = RECORD tople : Cell; botri : Cell; active: Cell; END(*Selection*);
(***** MatAccess *****)
PROCEDURE SetMatrixEle( m: Matrix; row, col: INTEGER; val: REAL );
PROCEDURE GetMatrixEle( m: Matrix; row, col: INTEGER; VAR val: REAL );
PROCEDURE MEle( m: Matrix; row, col: INTEGER ): REAL;
PROCEDURE FillMatrix ( m: Matrix; v : REAL );
PROCEDURE SetMatrixRow( m: Matrix; nrRow : INTEGER; VAR rowArr : ARRAY OF REAL );
PROCEDURE SetMatrixCol( m: Matrix; nrCol : INTEGER; VAR colArr : ARRAY OF REAL );
PROCEDURE GetMatrixRow( m: Matrix; nrRow : INTEGER; VAR rowArr : ARRAY OF REAL );
PROCEDURE GetMatrixCol( m: Matrix; nrCol : INTEGER; VAR colArr : ARRAY OF REAL );
PROCEDURE SetIndexRangeChecking( doCheck: BOOLEAN );
PROCEDURE GetIndexRangeChecking( VAR doCheck: BOOLEAN );
PROCEDURE SetMatrixName( m: Matrix; VAR name: ARRAY OF CHAR );
PROCEDURE GetMatrixName( m: Matrix; VAR name: ARRAY OF CHAR );
(***** MatCopy *****)
VAR selOneOne : Selection; (* read only ! *)
PROCEDURE AssignMatrix(VAR myMatrix: ARRAY OF BYTE; m,n: INTEGER; VAR mat: Matrix);
PROCEDURE RetrieveMatrix(mat: Matrix; VAR myMatrix: ARRAY OF BYTE; m,n: INTEGER);
PROCEDURE CopyMatrix( a: Matrix; VAR b: Matrix );
PROCEDURE SelWholeMat( m: Matrix; VAR sel: Selection );
PROCEDURE CopySelection( sourceMat: Matrix; area: Selection; destMat : Matrix; topLeft: Cell );
PROCEDURE SwapSelections( mat1: Matrix; area: Selection; mat2: Matrix; topLeft: Cell );
PROCEDURE SwapRows( mat1: Matrix; row1: INTEGER; mat2: Matrix; row2: INTEGER );
PROCEDURE SwapCols( mat1: Matrix; col1: INTEGER; mat2: Matrix; col2: INTEGER );
PROCEDURE FillDown ( mat: Matrix; area: Selection );
PROCEDURE FillRight( mat: Matrix; area: Selection );
(***** MatDeclare *****)
VAR notExistingMatrix: Matrix; (* read only variable! *)
PROCEDURE DeclMatrix( VAR m: Matrix; nRows, nCols: INTEGER; name : ARRAY OF CHAR );
PROCEDURE MatrixExists( m: Matrix ): BOOLEAN;
PROCEDURE RemoveMatrix( VAR m: Matrix );
PROCEDURE SetMatrixDim( VAR m: Matrix; nRows, nCols: INTEGER );
PROCEDURE GetMatrixDim( m: Matrix; VAR nRows, nCols: INTEGER );
(***** MatFile *****)
TYPE MatFormOut = RECORD realF : RealFormat; len, dec : CARDINAL;
  separator: ARRAY[0..63] OF CHAR; crCol : INTEGER; eOM : ARRAY[0..63] OF CHAR; END(*RECORD*);
  MatFormIn = RECORD separator: ARRAY[0..63] OF CHAR; nCols: INTEGER;
  rowSep : ARRAY[0..63] OF CHAR; eOM : ARRAY[0..63] OF CHAR; END(*MatFormIn*);
VAR standardO : MatFormOut; standardI : MatFormIn; matFileOk : BOOLEAN;
PROCEDURE SetMatFormIn ( mf: MatFormIn ); PROCEDURE GetMatFormIn ( VAR mf: MatFormIn );
PROCEDURE SetMatFormOut( mf: MatFormOut); PROCEDURE GetMatFormOut( VAR mf: MatFormOut);
PROCEDURE WriteMatrix( f : TextFile; m: Matrix );
PROCEDURE WriteRow ( f : TextFile; m: Matrix; rowNr : INTEGER );
PROCEDURE WriteCol ( f : TextFile; m: Matrix; colNr : INTEGER );
PROCEDURE WriteEle ( f : TextFile; m: Matrix; row,col: INTEGER );
PROCEDURE ReadMatrix ( f : TextFile; m: Matrix );
PROCEDURE ReadRow ( f : TextFile; m: Matrix; rowNr : INTEGER );
PROCEDURE ReadCol ( f : TextFile; m: Matrix; colNr : INTEGER );
PROCEDURE ReadEle ( f : TextFile; m: Matrix; row,col: INTEGER );
(***** MathProcs *****)
PROCEDURE PowerI(x: REAL; iexp: INTEGER): REAL; PROCEDURE Power ( x, exp: REAL ): REAL;
PROCEDURE Lg (x: REAL): REAL; PROCEDURE Fac (k: CARDINAL): CARDINAL;
PROCEDURE Round (x: REAL): INTEGER; PROCEDURE Int (x: REAL): INTEGER;
PROCEDURE Imax (i1,i2: INTEGER): INTEGER; PROCEDURE Imin (i1,i2: INTEGER): INTEGER;
PROCEDURE Rmax (x1,x2: REAL): REAL; PROCEDURE Rmin (x1,x2: REAL): REAL;
PROCEDURE Pi (): REAL; PROCEDURE Tan (x: REAL): REAL;
PROCEDURE ArcSin(x: REAL): REAL; PROCEDURE ArcCos(x: REAL): REAL;
(***** MsgFiles *****)
CONST english = 0; german = 1; french = 2; italian = 3; myLanguage1 = 4; myLanguage2 = 5; undefMsgNr = -1;
PROCEDURE SetMessageLanguage(l: INTEGER);
PROCEDURE SetAsMessageFile(fn: ARRAY OF CHAR; VAR done: BOOLEAN);
PROCEDURE GetMessage(msgnr: INTEGER; VAR msg: ARRAY OF CHAR);
PROCEDURE GetNumberedMessage(msgnr: INTEGER; VAR msg: ARRAY OF CHAR);
(***** MultiNormal *****)
TYPE MultiNDistr;
VAR notDeclaredMultiNDistr: MultiNDistr; (* read only *)
PROCEDURE DeclareMultiNDistr( VAR mVec : Vector; VAR sigVec : Vector; VAR corMat : Matrix;
  dim : INTEGER; VAR mnd : MultiNDistr);
PROCEDURE MultiNDistrDeclared( mnd: MultiNDistr ): BOOLEAN;
PROCEDURE MultiN( mnd: MultiNDistr; VAR vals: Vector );
PROCEDURE UndeclareMultiNDistr( VAR mnd: MultiNDistr );
(***** Queues *****)
TYPE FIFOQueue; ItemAction = PROCEDURE (Transaction);
VAR notExistingFIFOQueue: FIFOQueue; (* read only *)
PROCEDURE CreateFIFOQueue (VAR q: FIFOQueue; maxLength: INTEGER);
PROCEDURE EmptyFIFOQueue (q: FIFOQueue);
PROCEDURE FileIntoFIFOQueue (q: FIFOQueue; ta: Transaction);
PROCEDURE FirstInFIFOQueue (q: FIFOQueue): Transaction;
PROCEDURE Take1stFromFIFOQueue(q: FIFOQueue): Transaction;
PROCEDURE FIFOQueueLength (q: FIFOQueue): INTEGER;
PROCEDURE IsFIFOQueueFull (fifoc: FIFOQueue): BOOLEAN;
PROCEDURE IsFIFOQueueEmpty(fifoc: FIFOQueue): BOOLEAN;
PROCEDURE DoForAllInFIFOQueue (q: FIFOQueue; ia: ItemAction);
PROCEDURE FIFOQueueExists (q: FIFOQueue): BOOLEAN;
PROCEDURE DiscardFIFOQueue(VAR q: FIFOQueue);

```

## ModelWorks V2.2 - Appendix (Interfaces and Libraries)

```

(***** RandGen *****)
PROCEDURE SetSeeds(z0,z1,z2: INTEGER); (*defaults: z0=1, z1=10000, z2=3000*)
PROCEDURE GetSeeds(VAR z0,z1,z2: INTEGER);
PROCEDURE Randomize; PROCEDURE ResetSeeds;
PROCEDURE U(): REAL; (*U~(0,1), cycle length > 2.78 E13 ~ 220 years for 1000 U/sec*)

(***** RandGen0 *****)
PROCEDURE J(): INTEGER; PROCEDURE Jp(min, max: INTEGER): INTEGER;
PROCEDURE SetJPar( min,max: INTEGER); PROCEDURE GetJPar(VAR min,max: INTEGER);
PROCEDURE R(): REAL; PROCEDURE Rp(min, max: REAL): REAL;
PROCEDURE SetRPar( min,max: REAL); PROCEDURE GetRPar(VAR min,max: REAL);
PROCEDURE NegExpP(): REAL; PROCEDURE NegExpP(lambda: REAL): REAL;
PROCEDURE SetNegExpPar( lambda: REAL); PROCEDURE GetNegExpPar(VAR lambda: REAL);

TYPE URandGen = PROCEDURE(): REAL;
PROCEDURE InstallU0(u0: URandGen); PROCEDURE InstallU1(u1: URandGen);

(***** RandGen1 *****)
PROCEDURE Weibull(): REAL; PROCEDURE WeibullP(alpha,beta: REAL): REAL;
PROCEDURE SetWeibullPars( alpha,beta: REAL);
PROCEDURE GetWeibullPars(VAR alpha,beta: REAL);
PROCEDURE Triang(): REAL; PROCEDURE TriangP(min,mode,max: REAL): REAL;
PROCEDURE SetTriangPars( min,mode,max: REAL);
PROCEDURE GetTriangPars(VAR min,mode,max: REAL);
PROCEDURE VM(): REAL; PROCEDURE VMP(mean,kappa: REAL): REAL;
PROCEDURE SetVMPars( mean,kappa: REAL);
PROCEDURE GetVMPars(VAR mean,kappa: REAL);

TYPE URandGen= PROCEDURE(): REAL;
PROCEDURE InstallU0(u0: URandGen); PROCEDURE InstallU1(u1: URandGen);

(***** RandNormal *****)
TYPE URandGen = PROCEDURE(): REAL;
PROCEDURE InstallU(U: URandGen); (* do always call *)

PROCEDURE N(): REAL; (* N~(μ,σDev) *) PROCEDURE Np(μ,σDev: REAL): REAL;
PROCEDURE SetPars( μ,σDev: REAL); (* defaults μ = 0, σDev = 1 *)
PROCEDURE GetPars(VAR μ,σDev: REAL);
PROCEDURE ResetN: (* call after SetSeeds for full reset of N *)

(***** ReadData *****)
VAR dataF: TextFile; readingAborted: BOOLEAN;

PROCEDURE OpenDataFile( VAR fn: ARRAY OF CHAR; VAR ok: BOOLEAN ); (* always with dialog *)
PROCEDURE OpenDataFile ( VAR fn: ARRAY OF CHAR; VAR ok: BOOLEAN ); (* normally no dialog *)
PROCEDURE ReReadDataFile: (* performs a reset *)
PROCEDURE CloseDataFile;

PROCEDURE SkipHeaderLine;
PROCEDURE ReadHeaderLine(VAR labels: ARRAY OF String; VAR nrVars: INTEGER );
(* assign NIL to labels before first use! *)

PROCEDURE ReadLn ( VAR txt: ARRAY OF CHAR );
PROCEDURE GetChars( VAR str: ARRAY OF CHAR );
PROCEDURE GetStr ( VAR str: String );
PROCEDURE SkipGapOrComment: (* skips <= " " and "( * . . . . * )" *)
PROCEDURE ReadCharsUnlessAComment( VAR string: ARRAY OF CHAR );
PROCEDURE GetInt ( desc : ARRAY OF CHAR; loc: INTEGER; VAR x: INTEGER; min, max: INTEGER );
PROCEDURE GetReal( desc : ARRAY OF CHAR; loc: INTEGER; VAR x: REAL; min, max: REAL );
PROCEDURE SetMissingValCode( missingValCode: CHAR); (* default "N"; used in dataF *)
PROCEDURE GetMissingValCode(VAR missingValCode: CHAR);
PROCEDURE SetMissingReal ( missingReal: REAL); (* default 0.0; value used for a real *)
PROCEDURE GetMissingReal (VAR missingReal: REAL);
PROCEDURE SetMissingInt ( missingInt: INTEGER); (* default 0; value used for an integer *)
PROCEDURE GetMissingInt (VAR missingInt: INTEGER);
PROCEDURE SetEOSCode( eosCode: CHAR ); (* default ASCII us (unit separator) 37C *)
PROCEDURE GetEOSCode(VAR eosCode: CHAR );
PROCEDURE FindSegment(segNr: CARDINAL; VAR found: BOOLEAN); (* first segNr = 1 *)
PROCEDURE SkipToNextSegment(VAR done: BOOLEAN);
PROCEDURE AtEOL(): BOOLEAN; PROCEDURE AtEOS(): BOOLEAN; PROCEDURE AtBOF(): BOOLEAN;
PROCEDURE TestBOF; (* use only where you don't expect EOF (shows alert) *)

TYPE Relation = ( smaller, equal, greater );
PROCEDURE Compare2Strings( a, b: ARRAY OF CHAR ): Relation;

CONST negLogDelta = 0.01; (*offset to plot log scale if values <= 0*)

TYPE ErrorType = (NoInt, NoReal, TooBig, TooSmall, NotEqual, EndOfFile, FileNotFound, DataFNotOpen);
NumType = (Real, Integer);
Error = RECORD
  errorType : ErrorType; strFound : ARRAY[0..63] OF CHAR;
CASE numType : NumType OF
  Integer : minI, maxI: INTEGER
  | Real : minR, maxR: REAL
ELSE END;
desc :ARRAY [0..255] OF CHAR; loc :INTEGER
END;
ErrMsgProc = PROCEDURE(Error);

PROCEDURE SetErrMsgP( errMsgP: ErrMsgProc );
PROCEDURE GetErrMsgP( VAR currErrMsgP: ErrMsgProc );
PROCEDURE UseDefaultErrMsg;

(***** StateEvents *****)
TYPE StateEvt;
VAR unexpectedStateEvt: StateEvt; (* read only! *)

PROCEDURE ExpectStateEvt(VAR evt: StateEvt; x: StateVar; theta1,theta2: REAL);
PROCEDURE StateEvtExpected(evt: StateEvt): BOOLEAN;
PROCEDURE IsStateEvt(evt: StateEvt; x: StateVar): BOOLEAN;
PROCEDURE SetStateEvt(evt: StateEvt; x: StateVar; theta1,theta2: REAL);
PROCEDURE GetStateEvt(evt: StateEvt; VAR theta1,theta2: REAL);
PROCEDURE IgnoreStateEvt(VAR evt: StateEvt);

(***** StatLib *****)
TYPE FunctionProc = PROCEDURE( REAL ): REAL; InRangeProc = PROCEDURE( REAL, REAL, REAL ): BOOLEAN;
PROCEDURE MinX ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;

```



```

PROCEDURE MaxX ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE WSumX ( VAR X: ARRAY OF REAL; N: CARDINAL; FX: FunctionXProc ): REAL;
PROCEDURE SumX ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE SumXY ( VAR X, Y: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE SumX2 ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE SumX3 ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE SumX4 ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE WMeanX ( VAR X: ARRAY OF REAL; N: CARDINAL; FX: FunctionXProc ): REAL;
PROCEDURE MeanX ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE VarX ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE SDevX ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE SkewX ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE KurtX ( VAR X: ARRAY OF REAL; N: CARDINAL ): REAL;
PROCEDURE LinearReg ( VAR X, Y: ARRAY OF REAL; N: CARDINAL; VAR a, b, r2: REAL );
PROCEDURE FuncX ( VAR X, Y: ARRAY OF REAL; N: CARDINAL; FX: FunctionXProc );
PROCEDURE CountX ( VAR X: ARRAY OF REAL; N: CARDINAL; XLow, XHigh: REAL; InRangeX: InRangeProc ): CARDINAL;
PROCEDURE InsertX ( VAR X: ARRAY OF REAL; N, j: CARDINAL; xValue: REAL );
PROCEDURE DeleteX ( VAR X: ARRAY OF REAL; N, j: CARDINAL );
PROCEDURE ClearX ( VAR X: ARRAY OF REAL; N: CARDINAL; xValue: REAL );
PROCEDURE SortX ( VAR X: ARRAY OF REAL; N: CARDINAL );
PROCEDURE NormDist ( z1, z2: REAL ): REAL;
PROCEDURE Factorial ( N: CARDINAL ): REAL;
PROCEDURE Combination ( N, R: CARDINAL ): REAL;
PROCEDURE Permutation ( N, R: CARDINAL ): REAL;

(***** StochStat *****)

TYPE StatArray; ProbZTail = (prob999, prob990, prob950, prob900, prob800); Str31 = ARRAY [0..31] OF CHAR;

VAR notExistingStatArray: StatArray; (* read only *)

PROCEDURE StatArrayExists(statArray: StatArray): BOOLEAN;
PROCEDURE DeclStatArray(VAR statArray: StatArray; length: INTEGER);
PROCEDURE RemoveStatArray(VAR statArray: StatArray); PROCEDURE RemoveAllStatArrays;
PROCEDURE ClearStatArray(statArray: StatArray); PROCEDURE ClearAllStatArrays;
PROCEDURE SetStatArray(statArray: StatArray; N, X, sumY, sumYSquare: ARRAY OF REAL);
PROCEDURE SetUndefValue( undefVal: REAL); PROCEDURE GetUndefValue(VAR undefVal: REAL);
PROCEDURE SetTolerance( tol: REAL); PROCEDURE GetTolerance(VAR tol: REAL);
PROCEDURE PutValue(statArray: StatArray; index: INTEGER; x, y: REAL);
PROCEDURE GetValue(statArray: StatArray; index: INTEGER; VAR count, x, sumY, sumYSquare: REAL);
PROCEDURE GetSingleStatistics(statArray: StatArray; index: INTEGER;
    VAR count, x, sumY, sumYSquare, meansY, stdDevsY, confIntsY: REAL; confProb: ProbZTail);
PROCEDURE GetStatistics(statArray: StatArray; VAR N, X, sumY, sumYSquare, meansY, stdDevsY, confIntsY: ARRAY OF REAL;
    confProb: ProbZTail; VAR length: INTEGER);
PROCEDURE DeclDispMv(statArray: StatArray; mDepVar: Model; VAR mDepVar: REAL; mIndepVar: Model; VAR mIndepVar: REAL);
PROCEDURE DisplayArray(statArray: StatArray; withErrBars: BOOLEAN; confProb: ProbZTail);
PROCEDURE DisplayAllArrays(withErrBars: BOOLEAN; confProb: ProbZTail);
TYPE RealFileFormat = RECORD rF: RealFormat; n, dec: CARDINAL; END;
FileOutFormat = RECORD
    means, counts, sumsY, sumsYSquare, stdDevsY, confIntsY: BOOLEAN;
    indepsFormat, meansFormat, sumsYFormat, sumsYSquareFormat, stdDevsYFormat, confIntsYFormat: RealFileFormat;
    confProb: ProbZTail; END;
VAR (* read only! *) meansOnly, meansSDCI, allVals: FileOutFormat;

PROCEDURE DumpStatArray (VAR f: TextFile; label: Str31; statArray: StatArray; fof: FileOutFormat);
PROCEDURE DumpStatArrays(VAR f: TextFile; labels: ARRAY OF Str31;
    statArrays: ARRAY OF StatArray; fof: FileOutFormat; nArs: INTEGER);

(***** StructModAux *****)

TYPE StructModelSet = BITSET; BooleanFct = PROCEDURE (): BOOLEAN;

VAR customM: Menu; chooseCmd: Command; (* may be used to install more commands *)

PROCEDURE InstallCustomMenu(title, chooseCmdTxt, chooseAlChr: ARRAY OF CHAR);
PROCEDURE AssignSubModel(VAR which: INTEGER; descr: ARRAY OF CHAR; act, deact: PROC; isact: BooleanFct);
PROCEDURE ChooseModel;
PROCEDURE InstallMyGlobPreferences(myPrefs: PROC);
PROCEDURE SetSimEnv(sms: StructModelSet);

(***** TabFunc *****)

TYPE TabFUNC; TabFProc = PROCEDURE (VAR TabFUNC );

VAR notExistingTabF: TabFUNC; (* read only! *)

PROCEDURE DeclTabF(VAR t: TabFUNC; xx, yy: ARRAY OF REAL; NvalPairs: INTEGER; modifiable: BOOLEAN;
    tabName, xName, yName, xUnit, yUnit: ARRAY OF CHAR;
    xMin, xMax, yMin, yMax: REAL );
PROCEDURE DeclTabFM(VAR t: TabFUNC; xyVecs: Matrix; modifiable: BOOLEAN;
    tabName, xName, yName, xUnit, yUnit: ARRAY OF CHAR;
    xMin, xMax, yMin, yMax: REAL );
PROCEDURE SetTabF( t: TabFUNC; xx, yy: ARRAY OF REAL; NvalPairs: INTEGER; modifiable: BOOLEAN;
    tabName, xName, yName, xUnit, yUnit: ARRAY OF CHAR;
    xMin, xMax, yMin, yMax: REAL );
PROCEDURE GetTabF(t: TabFUNC; VAR xx, yy: ARRAY OF REAL; VAR NvalPairs: INTEGER; VAR modifiable: BOOLEAN;
    VAR tabName, xName, yName, xUnit, yUnit: ARRAY OF CHAR;
    VAR xMin, xMax, yMin, yMax: REAL );
PROCEDURE SetTabFM( t: TabFUNC; xyVecs: Matrix; modifiable: BOOLEAN;
    tabName, xName, yName, xUnit, yUnit: ARRAY OF CHAR;
    xMin, xMax, yMin, yMax: REAL );
PROCEDURE GetTabFM( t: TabFUNC; VAR xyVecs: Matrix; VAR modifiable: BOOLEAN;
    VAR tabName, xName, yName, xUnit, yUnit: ARRAY OF CHAR;
    VAR xMin, xMax, yMin, yMax: REAL );
PROCEDURE RemoveTabF( VAR t: TabFUNC );
PROCEDURE EditTabF ( t: TabFUNC );
PROCEDURE ResetTabF ( t: TabFUNC );
PROCEDURE FreezeEditorGraphBounds (VAR t: TabFUNC; xMin, xMax, yMin, yMax: REAL );
PROCEDURE UnfreezeEditorGraphBounds(VAR t: TabFUNC );

TYPE ExtrapolMode = ( lastSlope, horizontally ); (* default lastSlope *)

PROCEDURE DefineExtrapolationMode( VAR t: TabFUNC; extrapolation: ExtrapolMode );
PROCEDURE ExtrapolationMode( t: TabFUNC ): ExtrapolMode;
PROCEDURE Yi ( t: TabFUNC; x: REAL ): REAL; (* interpolate only ELSE HALT *)
PROCEDURE Yie ( t: TabFUNC; x: REAL ): REAL; (* inter- and extrapolate *)
PROCEDURE DoForAllTabF( p: TabFProc );

(===== P U B L I C D O M A I N M O D U L E S =====)

(***** Curves3D *****)

CONST nRun = 5; nVal = 250;

```

## ModelWorks V2.2 - Appendix (Interfaces and Libraries)

```

TYPE ProjectionEnumerator = (xyPlane, xzPlane, yzPlane, spacial);
Projections = [xyPlane..spacial]; ProjectionSet = SET OF ProjectionEnumerator;

PROCEDURE SelectSymbol(theProjection: Projections; symbol: CHAR);
PROCEDURE ClearUpdateStore;
PROCEDURE StartNewCurve(projection:ProjectionSet; firstPoint: Point3D);
PROCEDURE PlotTo3D(P: Point3D); PROCEDURE ReplotAll;
PROCEDURE GetCurrentProjection():ProjectionSet;
PROCEDURE StorageOff; PROCEDURE StorageOn;

(***** SortLib *****)

PROCEDURE QuickSortX ( VAR a: ARRAY OF REAL; n: CARDINAL );
PROCEDURE QuickSortXY ( VAR a, b: ARRAY OF REAL; n: CARDINAL );
PROCEDURE BinarySortX ( VAR a: ARRAY OF REAL; n: CARDINAL );
PROCEDURE StrSelSortX ( VAR a: ARRAY OF REAL; n: CARDINAL );
PROCEDURE StrSelSortXY ( VAR a, b: ARRAY OF REAL; n: CARDINAL );

(***** WriteDatTim *****)

CONST
Jan = 1; Feb = 2; Mar = 3; Apr = 4; Mai = 5; Jun = 6;
Jul = 7; Aug = 8; Sep = 9; Oct = 10; Nov = 11; Dec = 12;
Sun = 1; Mon = 2; Tue = 3; Wed = 4; Thur = 5; Fri = 6; Sat = 7;

TYPE Months = INTEGER; WeekDays = INTEGER;
DateAndTimeRec = RECORD
year: INTEGER; (* 1904,1905,..2040 *) month: Months;
day, (* 1,..31 *) hour, (* 0,..,23 *) minute, second: INTEGER; (* 0,..,59 *)
dayOfWeek: WeekDays; END;
WriteProc = PROCEDURE (CHAR);
DateFormat = ( brief, (* only numbers: e.g. 31/05/88 *)
letMonth, (* month in letters: e.g. 31/Mai/1988 *)
full ); (* full in letters: e.g. 31st Mai 1988 *)
TimeFormat = ( brief24h, brief24hSecs, let24hSecs, full24hSecs, brief12h);
(* the following procedures write information in English only *)
PROCEDURE WriteDate(d: DateAndTimeRec; w: WriteProc; df: DateFormat);
PROCEDURE WriteTime(d: DateAndTimeRec; w: WriteProc; tf: TimeFormat);

(===== - E N D - =====)

The auxiliary library modules may be freely copied but not for profit!

```

## E.2 DIALOG MACHINE<sup>1</sup>

For details on how to work with the "Dialog Machine" see part II *Theory*, chapter *ModelWorks Functions*, section *User Interface Customization* and section *Module structure of ModelWorks*, and this appendix section *Research Sample Models*. On the "Dialog Machine" exist separate documentations (see *Literature*).

Dialog Machine Version 2.2 (19/Apr/96) (c) 1988-96 Andreas Fischlin, Systems Ecology, and Swiss Federal Institute of Technology Zurich ETHZ

```

===== K E R N E L =====
(***** DMConversions *****)

TYPE RealFormat = (FixedFormat, ScientificNotation);

PROCEDURE StringToCard(str: ARRAY OF CHAR; VAR card: CARDINAL; VAR done: BOOLEAN);
PROCEDURE CardToString(card: CARDINAL; VAR str: ARRAY OF CHAR; length: CARDINAL);
PROCEDURE StringToLongCard(str: ARRAY OF CHAR; VAR lcard: LONGCARD; VAR done: BOOLEAN);
PROCEDURE LongCardToString(lcard: LONGCARD; VAR str: ARRAY OF CHAR; length: CARDINAL);
PROCEDURE StringToInt(str: ARRAY OF CHAR; VAR int: INTEGER; VAR done: BOOLEAN);
PROCEDURE IntToString(int: INTEGER; VAR str: ARRAY OF CHAR; length: CARDINAL);
PROCEDURE StringToLongInt(str: ARRAY OF CHAR; VAR lint: LONGINT; VAR done: BOOLEAN);
PROCEDURE LongIntToString(lint: LONGINT; VAR str: ARRAY OF CHAR; length: CARDINAL);
PROCEDURE StringToReal(str: ARRAY OF CHAR; VAR real: REAL; VAR done: BOOLEAN);
PROCEDURE RealToString(real: REAL; VAR str: ARRAY OF CHAR; length, dec: CARDINAL; f: RealFormat);
PROCEDURE StringToLongReal(Str: ARRAY OF CHAR; VAR longReal: LONGREAL; VAR done: BOOLEAN);
PROCEDURE LongRealToString(longreal: LONGREAL; VAR str: ARRAY OF CHAR; length, dec: CARDINAL; f: RealFormat);
PROCEDURE HexStringToBytes(hstr: ARRAY OF CHAR; VAR x: ARRAY OF BYTE; VAR done: BOOLEAN);
PROCEDURE BytesToHexString(x: ARRAY OF BYTE; VAR hstr: ARRAY OF CHAR); PROCEDURE SetHexDigitsUpperCase(upperC: BOOLEAN);
PROCEDURE IllegalSyntaxDetected(): BOOLEAN;

PROCEDURE UndefREAL(): REAL; (* = NAN(017) *) PROCEDURE UndefLONGREAL(): LONGREAL; (* = NAN(017) *)
PROCEDURE IsUndefREAL(x: REAL): BOOLEAN; PROCEDURE IsUndefLONGREAL(x: LONGREAL): BOOLEAN;

(***** DMLanguage *****)

TYPE Language = (English, German, French, Italian, MyLanguage1, MyLanguage2);

PROCEDURE SetLanguage(l: Language); PROCEDURE CurrentLanguage(): Language;
PROCEDURE GetMsgString(msgNr: INTEGER; VAR str: ARRAY OF CHAR);

(***** DMMaster *****)

TYPE MouseHandlers = (WindowContent, BringToFront, RemoveFromFront, RedefWindow, CloseWindow);
MouseHandler = PROCEDURE (Window); KeyboardHandler = PROC; SubProgStatus = (normal, abnormal);

VAR MasterDone: BOOLEAN;

PROCEDURE AddSetupProc(sup: PROC; priority: INTEGER); PROCEDURE RemoveSetupProc(sup: PROC);
PROCEDURE AddMouseHandler(which: MouseHandlers; mhp: MouseHandler; priority: INTEGER);
PROCEDURE RemoveMouseHandler(which: MouseHandlers; mhp: MouseHandler);

PROCEDURE AddKeyboardHandler(khp: KeyboardHandler; priority: INTEGER); PROCEDURE RemoveKeyboardHandler(khp: KeyboardHandler);

PROCEDURE InspectKey(VAR ch: CHAR; VAR modifiers: BITSET); PROCEDURE KeyAccepted; PROCEDURE DoTillKeyReleased(p: PROC);
PROCEDURE SetKeyboardHandlerMode(readGetsThem: BOOLEAN; maxPriority: INTEGER); PROCEDURE Read(VAR ch: CHAR);
PROCEDURE GetKeyboardHandlerMode(VAR readGetsThem: BOOLEAN; VAR maxPriority: INTEGER); PROCEDURE BusyRead(VAR ch: CHAR);

PROCEDURE ShowWaitSymbol; PROCEDURE HideWaitSymbol; PROCEDURE Wait(nrTicks: LONGCARD); (* 1 tick = 1/60 second *)
PROCEDURE SoundBell; PROCEDURE PlayPredefinedMusic(fileName: ARRAY OF CHAR; musicID: INTEGER);

PROCEDURE InitDialogMachine; PROCEDURE RunDialogMachine; PROCEDURE DialogMachineIsRunning(): BOOLEAN;
PROCEDURE QuitDialogMachine; PROCEDURE AbortDialogMachine; PROCEDURE DialogMachineTask;
PROCEDURE CallSubProg(module: ARRAY OF CHAR; VAR status: SubProgStatus);

(***** DMMenus *****)

TYPE Menu; Command; AccessStatus = (enabled, disabled); Marking = (checked, unchecked); Separator = (line, blank);
QuitProc = PROCEDURE(VAR BOOLEAN); SeparatorPosition = (beforeCmd, afterCmd);

VAR MenusDone: BOOLEAN; notInstalledMenu: Menu; notInstalledCommand: Command;

PROCEDURE InstallAbout(s: ARRAY OF CHAR; w,h: CARDINAL; p: PROC);
PROCEDURE NoDeskAccessories;
PROCEDURE InstallMenu(VAR m: Menu; menuText: ARRAY OF CHAR; ast: AccessStatus);
PROCEDURE InstallSubMenu (inMenu: Menu; VAR subMenu: Menu; menuText: ARRAY OF CHAR; ast: AccessStatus);
PROCEDURE InstallCommand(m: Menu; VAR c: Command; cmdText: ARRAY OF CHAR; p: PROC; ast: AccessStatus; chm: Marking);
PROCEDURE InstallAliasChar(m: Menu; c: Command; ch: CHAR);
PROCEDURE InstallSeparator(m: Menu; s: Separator); PROCEDURE RemoveSeparator(m: Menu; s: CARDINAL);
PROCEDURE RemoveSeparatorAtCommand(m: Menu; cmd: Command; sp: SeparatorPosition);

PROCEDURE InstallQuitCommand(s: ARRAY OF CHAR; p: QuitProc; aliasChar: CHAR);
PROCEDURE HideSubQuit(onLevel: CARDINAL); PROCEDURE ShowSubQuit(onLevel: CARDINAL);
PROCEDURE UseMenu(m: Menu); PROCEDURE UseMenuBar;
PROCEDURE RemoveMenu(VAR m: Menu); PROCEDURE RemoveMenuBar;
PROCEDURE RemoveCommand(m: Menu; cmd: Command);
PROCEDURE EnableDeskAccessories; PROCEDURE DisableDeskAccessories;
PROCEDURE EnableMenu(m: Menu); PROCEDURE DisableMenu(m: Menu);
PROCEDURE EnableCommand(m: Menu; c: Command); PROCEDURE DisableCommand(m: Menu; c: Command);
PROCEDURE CheckCommand(m: Menu; c: Command); PROCEDURE UncheckCommand(m: Menu; c: Command);
PROCEDURE SetCheckSym(m: Menu; c: Command; ch: CHAR); PROCEDURE IsCommandChecked(m: Menu; c: Command): BOOLEAN;
PROCEDURE ChangeCommand(m: Menu; c: Command; p: PROC); PROCEDURE ChangeCommandText(m: Menu; c: Command;
newCmdText: ARRAY OF CHAR);
PROCEDURE ChangeAliasChar(m: Menu; c: Command; newCh: CHAR); PROCEDURE ChangeQuitAliasChar(onLevel: CARDINAL; newAliasCh: CHAR);
PROCEDURE ExecuteCommand(m: Menu; c: Command); PROCEDURE ExecuteAbout;

```

<sup>1</sup>For availability and installation see the separate booklet "Installation Guide and Technical Reference of the RAMSES software".

```

PROCEDURE MenuExists(m: Menu): BOOLEAN; PROCEDURE CommandExists(m: Menu; c: Command): BOOLEAN;
PROCEDURE MenuLevel(m: Menu): CARDINAL; PROCEDURE CommandLevel(m: Menu; c: Command): CARDINAL;
PROCEDURE GetMenuAttributes(m: Menu; VAR menuNr: CARDINAL; VAR menuText: ARRAY OF CHAR; VAR ast: AccessStatus;
VAR isSubMenu: BOOLEAN; VAR parentMenu: Menu);
PROCEDURE GetCommandAttributes(m: Menu; c: Command; VAR cmdNr: CARDINAL; VAR cmdText: ARRAY OF CHAR; VAR p: PROC;
VAR ast: AccessStatus; VAR chm: Marking; VAR chmCh, aliasCh: CHAR);

PROCEDURE InstallPredefinedMenu (fileName: ARRAY OF CHAR; menuID: INTEGER; VAR m: Menu);
PROCEDURE InstallPredefinedSubMenu (fileName: ARRAY OF CHAR; menuID: INTEGER; inMenu: Menu; VAR subMenu: Menu);
PROCEDURE InstallPredefinedCommand (fileName: ARRAY OF CHAR; menuID, itemNr: INTEGER; m: Menu; VAR c: Command; p: PROC);
PROCEDURE InstallPredefinedSeparator (fileName: ARRAY OF CHAR; menuID, itemNr: INTEGER; m: Menu);
PROCEDURE SaveAsPredefinedMenu (fileName: ARRAY OF CHAR; menuID: INTEGER; m: Menu);
PROCEDURE SaveAsPredefinedMenuSection (fileName: ARRAY OF CHAR; menuID: INTEGER; m: Menu; maxItemNr: INTEGER);

*****
***** DMessages *****
CONST INHREAK = 15C; undefMsgNr = -1; toScreen = 0; toJournalFile = 1;
TYPE MsgRetrieveProc = PROCEDURE ( INTEGER , VAR ARRAY OF CHAR );
MsgDevice = [toScreen..toJournalFile]; MsgWriteProc = PROCEDURE ( CHAR ); MsgWriteLnProc = PROC;

PROCEDURE Ask(question: ARRAY OF CHAR; butTexts: ARRAY OF CHAR; butWidth: CARDINAL; VAR answer: INTEGER);
PROCEDURE DisplayBusyMessage(msg: ARRAY OF CHAR); PROCEDURE DiscardBusyMessage;
PROCEDURE Inform (paragraph1, paragraph2, paragraph3: ARRAY OF CHAR);
PROCEDURE DoInform (msgnr: INTEGER; modIdent, locDescr, insertions: ARRAY OF CHAR);
PROCEDURE Warn (paragraph1, paragraph2, paragraph3: ARRAY OF CHAR);
PROCEDURE DoWarn (msgnr: INTEGER; modIdent, locDescr, insertions: ARRAY OF CHAR);
PROCEDURE Abort (paragraph1, paragraph2, paragraph3: ARRAY OF CHAR);
PROCEDURE DoAbort (msgnr: INTEGER; modIdent, locDescr, insertions: ARRAY OF CHAR);

PROCEDURE SetMsgRetrieveProc(rp: MsgRetrieveProc); PROCEDURE GetMsgRetrieveProc(VAR rp: MsgRetrieveProc);
PROCEDURE UseForMsgJournaling(wp: MsgWriteProc; wlnp: MsgWriteLnProc);
PROCEDURE SetMaxMsgs (max: INTEGER);
PROCEDURE SetMsgDevice (forAsk,forInform,forWarn,forAbort: MsgDevice);
PROCEDURE GetMsgDevice (VAR forAsk,forInform,forWarn,forAbort: MsgDevice);
PROCEDURE AskPredefinedQuestion(fileName: ARRAY OF CHAR; alertID: INTEGER;
str1,str2,str3,str4: ARRAY OF CHAR; VAR answer: INTEGER);

*****
***** DMStorage *****
PROCEDURE Allocate(VAR p: ADDRESS; size: LONGINT); PROCEDURE AllocateOnLevel(VAR adr: ADDRESS; size: LONGINT; onLevel: INTEGER);
PROCEDURE Deallocate(VAR p: ADDRESS); PROCEDURE DeallocateOnLevel(VAR p: ADDRESS; onLevel: INTEGER);

(* IBM PC compatibility: *)
PROCEDURE ALLOCATE(VAR p: ADDRESS; size: CARDINAL); PROCEDURE DEALLOCATE(VAR p: ADDRESS; size: CARDINAL);

*****
***** DMStrings *****
TYPE String; StringRelation = (smaller, equal, greater);
VAR notAllocatedStr: String; ResourceStringsDone: BOOLEAN;

PROCEDURE AllocateStr(VAR strRef: String; s: ARRAY OF CHAR); PROCEDURE DeallocateStr(VAR strRef: String);
PROCEDURE SetStr(VAR strRef: String; s: ARRAY OF CHAR); PROCEDURE GetStr(strRef: String; VAR s: ARRAY OF CHAR);
PROCEDURE StrLevel(strRef: String): CARDINAL; PROCEDURE StrLength(strRef: String): INTEGER;
PROCEDURE Length(VAR string: ARRAY OF CHAR): INTEGER;
PROCEDURE AssignString(source: ARRAY OF CHAR; VAR d: ARRAY OF CHAR);
PROCEDURE Append(VAR dest: ARRAY OF CHAR; source: ARRAY OF CHAR); PROCEDURE AppendCh(VAR dest: ARRAY OF CHAR; ch: CHAR);
PROCEDURE AppendStr(VAR strRef: String; s: ARRAY OF CHAR); PROCEDURE AppendChr(VAR strRef: String; ch: CHAR);
PROCEDURE Concatenate(first,second: ARRAY OF CHAR; VAR result: ARRAY OF CHAR);
PROCEDURE CopyString (VAR from: ARRAY OF CHAR; i1,lrChs: INTEGER; VAR to: ARRAY OF CHAR; VAR i2: INTEGER);
PROCEDURE Copy(from: ARRAY OF CHAR; startIndex, nrOfChars: INTEGER; VAR to: ARRAY OF CHAR);
PROCEDURE ExtractSubString(VAR curPosInSrcS: INTEGER; VAR srcS,destS: ARRAY OF CHAR; delimiter: CHAR);
PROCEDURE FindInString (VAR theString: ARRAY OF CHAR; searchStr: ARRAY OF CHAR; lastCh: INTEGER): BOOLEAN;
PROCEDURE CompareStrings(s1,s2: ARRAY OF CHAR): StringRelation;
PROCEDURE CompVarStrings( VAR a, b: ARRAY OF CHAR): StringRelation;
PROCEDURE CompStr( VAR a: ARRAY OF CHAR; bS: String): StringRelation;
PROCEDURE LoadString(fileName: ARRAY OF CHAR; stringID: INTEGER; VAR string: ARRAY OF CHAR);
PROCEDURE StoreString(fileName: ARRAY OF CHAR; VAR stringID: INTEGER; string: ARRAY OF CHAR);
PROCEDURE GetRString(stringID: INTEGER; VAR str: ARRAY OF CHAR);
PROCEDURE SetRStringName (fileName: ARRAY OF CHAR; stringID: INTEGER; name: ARRAY OF CHAR);
PROCEDURE GetRStringName (fileName: ARRAY OF CHAR; stringID: INTEGER; VAR name: ARRAY OF CHAR);
PROCEDURE NewString(VAR s: ARRAY OF CHAR): String; PROCEDURE PutString(VAR strRef: String; VAR s: ARRAY OF CHAR);

*****
***** DMSystem *****
CONST startUpLevel = 1; maxLevel = 5;

PROCEDURE CurrentDMLevel(): CARDINAL; PROCEDURE LevelisDMLevel(l: CARDINAL): BOOLEAN;
PROCEDURE TopDMLevel(): CARDINAL; PROCEDURE DoOnSubProgLevel(l: CARDINAL; p: PROC);
PROCEDURE ForceDMLevel(extraLevel: CARDINAL); PROCEDURE ResumeDMLevel(normalLevel: CARDINAL);

PROCEDURE InstallInitProc(ip: PROC; VAR done: BOOLEAN); PROCEDURE ExecuteInitProcs;
PROCEDURE InstallTermProc(tp: PROC; VAR done: BOOLEAN); PROCEDURE ExecuteTermProcs;

PROCEDURE GetDMVersion(VAR vers,lastModifDate: ARRAY OF CHAR); PROCEDURE SystemVersion(): REAL;
PROCEDURE GetComputerName(VAR name: ARRAY OF CHAR);
PROCEDURE GetCPUName(VAR name: ARRAY OF CHAR); PROCEDURE GetFPUPName(VAR name: ARRAY OF CHAR);
PROCEDURE FPUPresent(): BOOLEAN; PROCEDURE GetROMName(VAR name: ARRAY OF CHAR);

PROCEDURE ScreenWidth(): INTEGER; PROCEDURE ScreenHeight(): INTEGER;
PROCEDURE MainScreen(): INTEGER;
PROCEDURE SuperScreen(VAR whichScreen, x,y,w,h, nrOfColors: INTEGER; colorPriority: BOOLEAN);

(* low level routines *)
PROCEDURE MenuBarHeight(): INTEGER; PROCEDURE TitleBarHeight(): INTEGER; PROCEDURE ScrollBarWidth(): INTEGER;
PROCEDURE GrowIconSize(): INTEGER;
PROCEDURE NumberOfColors(): INTEGER; (* supported by DM regardless of currently used screen *)
PROCEDURE HowManyScreens(): INTEGER; PROCEDURE GetScreenSize(screen: INTEGER; VAR x,y,w,h: INTEGER);
PROCEDURE NumberOfColorsOnScreen(screen: INTEGER): INTEGER;

CONST unknown = 0;
Mac512KE = 3; MacSE30 = 9; MacLC = 19; MacPowerBook140 = 25; SUN = 101; IBMPC = 201;
MacPlus = 4; MacPortable = 10; MacQuadra900 = 20; MacLCII = 19; SUN3 = 102; IBMAT = 202;
MacSE = 5; MacIICI = 11; MacPowerBook170 = 21; MacQuadra950 = 26; SUNSparc = 103; IBMPS2 = 203;
MacII = 6; MacIIFx = 13; MacQuadra700 = 22;
MacIIX = 7; MacClassic = 17; MacClassicII = 23;
MacIICx = 8; MacIISI = 18; MacPowerBook100 = 24;
PROCEDURE ComputerSystem(): INTEGER;

CONST CPU68000 = 1; CPU8088 = 201; CPU80186 = 203; FPU68881 = 1;
CPU68010 = 2; CPU8086 = 202; CPU80286 = 204; FPU68882 = 2;
CPU68020 = 3; CPU80386 = 205; FPU68040 = 3;
CPU68030 = 4; CPU80486 = 206;
CPU68040 = 5;

```

```

PROCEDURE CPType(): INTEGER;          PROCEDURE FPType(): INTEGER;

CONST MacKeyboard = 1;    AExtendKbd = 4;    PortableISOKbd = 7;    ADEKbdII = 10;    PwrBkISOKbd = 13;
      MacKbdAndPad = 2;    ADEKeyboard = 5;    EastwoodISOKbd = 8;    ADBISOKbdII = 11;
      MacPlusKbd = 3;    PortableKbd = 6;    SaratogaISOKbd = 9;    PwrBkADEKbd = 12;
PROCEDURE Keyboard(): INTEGER;

CONST ROM64k = 1;    ROM128k = 2;    ROM256k = 3;    ROM512k = 4;    ROM1024k = 5; (* ROM types *)
PROCEDURE ROMType(): INTEGER;          PROCEDURE ROMVersionNr(): INTEGER;          PROCEDURE QuickDrawVersion(): REAL;

(*****                               DMWindIO                               *****)

TYPE MouseModifiers = (ordinary, cmded, opted, shifted, capsLock, controlled);          ClickKind = SET OF MouseModifiers;
DragProc = PROCEDURE (INTEGER, INTEGER);

VAR WindowIODone: BOOLEAN;

PROCEDURE PointClicked(x,y: INTEGER; maxDist: INTEGER): BOOLEAN;
PROCEDURE RectClicked(rect: RectArea): BOOLEAN;
PROCEDURE PointDoubleClicked(x,y: INTEGER; maxDist: INTEGER): BOOLEAN;
PROCEDURE RectDoubleClicked(rect: RectArea): BOOLEAN;
PROCEDURE GetLastClick(VAR x,y: INTEGER; VAR click: ClickKind): BOOLEAN;
PROCEDURE GetLastDoubleClick(VAR x,y: INTEGER; VAR click: ClickKind): BOOLEAN;
PROCEDURE GetCurMousePos(VAR x,y: INTEGER);
PROCEDURE GetLastMouseClick(VAR x,y: INTEGER; VAR click: ClickKind);
PROCEDURE DoTillMbutReleased(p: PROC);
PROCEDURE Drag(duringDragP,afterDragP: DragProc);
PROCEDURE SetContSize(u: Window; contentRect: RectArea);          PROCEDURE GetContSize(u: Window; VAR contentRect: RectArea);
PROCEDURE SetScrollStep(u: Window; xStep,yStep: INTEGER);          PROCEDURE GetScrollStep(u: Window; VAR xStep, yStep: INTEGER);
PROCEDURE GetScrollBoxPos(u: Window; VAR posX,posY: INTEGER);
PROCEDURE SetScrollBoxPos(u: Window; posX,posY: INTEGER);
PROCEDURE GetScrollBoxChange(u: Window; VAR changeX,changeY: INTEGER);
PROCEDURE AutoScrollProc(u: Window);
PROCEDURE SetScrollProc(u: Window; scrollP: RestoreProc);          PROCEDURE GetScrollProc(u: Window; VAR scrollP: RestoreProc);
PROCEDURE ScrollContent(u: Window; dx,dy: INTEGER);          PROCEDURE MoveOriginTo(u: Window; x0,y0: INTEGER);
PROCEDURE SelectForOutput(u: Window);          PROCEDURE CurrentOutputWindow(): Window;

TYPE PaintMode = (replace, paint, invert, erase);
Hue = [0..359];          GreyContent = (light, lightGrey, grey, darkGrey, dark);          Saturation = [0..100];
Color = RECORD hue: Hue; greyContent: GreyContent; saturation: Saturation; END;
PatLine = BYTE;          Pattern = ARRAY [0..7] OF PatLine;

VAR pat: ARRAY [light..dark] OF Pattern;          black, white, red, green, blue, cyan, magenta, yellow: Color;

PROCEDURE SetMode(mode: PaintMode);          PROCEDURE GetMode(VAR mode: PaintMode);
PROCEDURE SetBackground(c: Color; pat: Pattern);          PROCEDURE GetBackground(VAR c: Color; VAR pat: Pattern);
PROCEDURE SetColor(c: Color);          PROCEDURE GetColor(VAR c: Color);
PROCEDURE SetPattern(p: Pattern);          PROCEDURE GetPattern(VAR p: Pattern);
PROCEDURE IdentifyPos(x,y: INTEGER; VAR line,col: CARDINAL);
PROCEDURE IdentifyPoint(line,col: CARDINAL; VAR x,y: INTEGER);
PROCEDURE MaxCol(): CARDINAL;          PROCEDURE MaxLn(): CARDINAL;
PROCEDURE CellWidth(): INTEGER;          PROCEDURE CellHeight(): INTEGER;
PROCEDURE StringArea (s: ARRAY OF CHAR; VAR a: RectArea; VAR baseLine,sepSpace: INTEGER);
PROCEDURE StringWidth (VAR s: ARRAY OF CHAR): INTEGER;
PROCEDURE BackgroundWidth(): INTEGER;          PROCEDURE BackgroundHeight(): INTEGER;
PROCEDURE SetBOWAction(u: Window; action: PROC);          PROCEDURE GetBOWAction(u: Window; VAR action: PROC);
PROCEDURE EraseContent;          PROCEDURE RedrawContent;
PROCEDURE SetClipping(cr: RectArea);          PROCEDURE GetClipping(VAR cr: RectArea);
PROCEDURE RemoveClipping;

TYPE WindowFont = (Chicago, Monaco, Geneva, NewYork);          FontStyles = (bold, italic, underline);
LaserFont = (Times, Helvetica, Courier, Symbol);          FontStyle = SET OF FontStyles;

PROCEDURE SetWindowFont(wf: WindowFont; size: CARDINAL; style: FontStyle);
PROCEDURE GetWindowFont(VAR wf: WindowFont; VAR size: CARDINAL; VAR style: FontStyle);
PROCEDURE SetLaserFont(lf: LaserFont; size: CARDINAL; style: FontStyle);
PROCEDURE GetLaserFont(VAR lf: LaserFont; VAR size: CARDINAL; VAR style: FontStyle);
PROCEDURE SetPos(line,col: CARDINAL);          PROCEDURE GetPos(VAR line,col: CARDINAL);
PROCEDURE ShowCaret(on: BOOLEAN);          PROCEDURE Invert(on: BOOLEAN);
PROCEDURE Write(ch: CHAR);          PROCEDURE WriteString(s: ARRAY OF CHAR);
PROCEDURE WriteLn;          PROCEDURE WriteVarString(VAR s: ARRAY OF CHAR);
PROCEDURE WriteCard(c,n: CARDINAL);          PROCEDURE WriteLongCard(lc: LONGCARD; n: CARDINAL);
PROCEDURE WriteInt(c: INTEGER; n: CARDINAL);          PROCEDURE WriteLongInt(li: LONGINT; n: CARDINAL);
PROCEDURE WriteReal(r: REAL; n,dec: CARDINAL);          PROCEDURE WriteRealSci(r: REAL; n,dec: CARDINAL);
PROCEDURE WriteLongReal(lr: LONGREAL; n,dec: CARDINAL);          PROCEDURE WriteLongRealSci(lr: LONGREAL; n,dec: CARDINAL);
PROCEDURE SetPen(x,y: INTEGER);          PROCEDURE GetPen(VAR x,y: INTEGER);
PROCEDURE SetBrushSize(width,height: INTEGER);          PROCEDURE GetBrushSize(VAR width,height: INTEGER);
PROCEDURE Dot(x,y: INTEGER);          PROCEDURE LineTo(x,y: INTEGER);
PROCEDURE Circle(x,y: INTEGER; radius: CARDINAL; filled: BOOLEAN; fillpat: Pattern);          PROCEDURE CopyArea(sourceArea: RectArea; dx,dy: INTEGER);
PROCEDURE MapArea(sourceArea,destArea: RectArea);
PROCEDURE DisplayPredefinedPicture (fileName: ARRAY OF CHAR; pictureID: INTEGER;          f: RectArea);
PROCEDURE GetPredefinedPicture(fileName: ARRAY OF CHAR; pictureID: INTEGER; VAR f: RectArea);
PROCEDURE StartPolygon;          PROCEDURE CloseAndFillPolygon(pat: Pattern);
PROCEDURE DrawAndFillPoly(nPoints: CARDINAL; VAR x, y: ARRAY OF INTEGER; VAR withEdge: ARRAY OF BOOLEAN;
      VAR edgeColors: ARRAY OF Color; isFilled: BOOLEAN; fillColor: Color; fillPattern: Pattern);

TYPE QDVHSelect = (v,h);          QDVHSelectR = [v..h];
QDPoint = RECORD CASE: INTEGER OF 0: v,h: INTEGER; | 1: vh: ARRAY QDVHSelectR OF INTEGER; END; END;
QDRect = RECORD CASE: INTEGER OF 0: top,left,bottom,right: INTEGER; | 1: topLeft,botRight: QDPoint; END; END;

PROCEDURE XYToQDPoint(x,y: INTEGER; VAR p: QDPoint);          PROCEDURE RectAreaToQDRect(r: RectArea; VAR qdr: QDRect);
PROCEDURE SelectRestoreCopy(u: Window);          PROCEDURE SetRestoreCopy(u: Window; rcp: ADDRESS);
PROCEDURE Turn(angle: INTEGER);          PROCEDURE TurnTo(angle: INTEGER);
PROCEDURE ScaleUC(r: RectArea; xmin,xmax,ymin,ymax: REAL);          PROCEDURE GetUC(VAR r: RectArea; VAR xmin,xmax,ymin,ymax: REAL);
PROCEDURE ConvertPointToUC(x,y: INTEGER; VAR xUC,yUC: REAL);          PROCEDURE ConvertUCToPoint(xUC,yUC: REAL; VAR x,y: INTEGER);
PROCEDURE UCFrame;          PROCEDURE EraseUCFrame;          PROCEDURE EraseUCFrameContent;
PROCEDURE SetUCPen(xUC,yUC: REAL);          PROCEDURE GetUCPen(VAR xUC,yUC: REAL);
PROCEDURE UCdot(xUC,yUC: REAL);          PROCEDURE UCLineTo(xUC,yUC: REAL);
PROCEDURE DrawSym(ch: CHAR);

(*****                               DMWindows                               *****)

TYPE Window;
WindowKind = (GrowOrShrinkOrDrag, FixedSize, FixedLocation, FixedLocTitleBar);
ModalWindowKind = (DoubleFrame, SingleFrameShadowed);
ScrollBars = (WithVerticalScrollBar, WithHorizontalScrollBar, WithBothScrollBars, WithoutScrollBars);
CloseAttr = (WithCloseBox, WithoutCloseBox);          ZoomAttr = (WithZoomBox, WithoutZoomBox);
RectArea = RECORD x,y,w,h: INTEGER END;          WindowFrame = RectArea;
WFFixPoint = (bottomLeft, topLeft);          RestoreProc = PROCEDURE (Window);
WindowProc = PROCEDURE (Window);
WindowHandlers = (clickedInContent, broughtToFront, removedFromFront,
      redefined, onlyMoved, disappeared, reappeared, closing);

VAR background: Window;          WindowsDone: BOOLEAN;          notExistingWindow: Window;

```

```

PROCEDURE NoBackground; PROCEDURE ReshowBackground;
PROCEDURE OuterWindowFrame(innerf: WindowFrame; wk: WindowKind; s: ScrollBars; VAR outerf: RectArea);
PROCEDURE InnerWindowFrame(outerf: WindowFrame; wk: WindowKind; s: ScrollBars; VAR innerf: RectArea);
PROCEDURE CreateWindow(VAR u: Window; wk: WindowKind; s: ScrollBars; c: CloseAttr; z: ZoomAttr;
    fixPoint: WFFixPoint; f: WindowFrame; title: ARRAY OF CHAR; Repaint: RestoreProc);
PROCEDURE CreateModalWindow(VAR u: Window; wk: ModalWindowKind; s: ScrollBars; f: WindowFrame; Repaint: RestoreProc);
PROCEDURE UsePredefinedWindow(VAR u: Window; fileName: ARRAY OF CHAR; windowID: INTEGER;
    fixPoint: WFFixPoint; Repaint: RestoreProc);
PROCEDURE CreateTitledModalWindow(VAR u: Window; title: ARRAY OF CHAR; f: WindowFrame); CONST DoubleFrameTitled = 3;
PROCEDURE RedefineWindow(u: Window; f: WindowFrame); PROCEDURE RedrawTitle(u: Window; title: ARRAY OF CHAR);
PROCEDURE MakeWindowInvisible(u: Window); PROCEDURE MakeWindowVisible(u: Window);
PROCEDURE IsNowVisible(u: Window): BOOLEAN;
PROCEDURE WindowLevel(u: Window): CARDINAL;
PROCEDURE GetWindowCharacteristics(u: Window; VAR wk: INTEGER; VAR modalKind: BOOLEAN; VAR s: ScrollBars; VAR c: CloseAttr;
    VAR z: ZoomAttr; VAR fixPoint: WFFixPoint; VAR f: WindowFrame; VAR title: ARRAY OF CHAR);
PROCEDURE DummyRestoreProc(u: Window); PROCEDURE AutoRestoreProc(u: Window);
PROCEDURE SetRestoreProc(u: Window; r: RestoreProc); PROCEDURE GetRestoreProc(u: Window; VAR r: RestoreProc);
PROCEDURE StartAutoRestoring(u: Window; r: RectArea); PROCEDURE StopAutoRestoring(u: Window);
PROCEDURE AutoRestoring(u: Window): BOOLEAN; PROCEDURE GetHiddenBitmapSize(u: Window; VAR r: RectArea);
PROCEDURE UpdateWindow(u: Window); PROCEDURE InvalidateContent(u: Window);
PROCEDURE UpdateAllWindows;
PROCEDURE AddWindowHandler(u: Window; wh: WindowHandlers; wpp: WindowProc; prio: INTEGER);
PROCEDURE RemoveWindowHandler(u: Window; wh: WindowHandlers; wpp: WindowProc);
PROCEDURE GetWindowFrame(u: Window; VAR f: WindowFrame); PROCEDURE GetWFFixPoint(u: Window; VAR loc: WFFixPoint);
PROCEDURE DoForAllWindows(action: WindowProc);
PROCEDURE UseWindowModally(u: Window; VAR terminateModalDialog, cancelModalDialog: BOOLEAN);
PROCEDURE PutOnTop(u: Window); PROCEDURE FrontWindow(): Window;
PROCEDURE RemoveWindow(VAR u: Window); PROCEDURE RemoveAllWindows;
PROCEDURE WindowExists(u: Window): BOOLEAN; PROCEDURE RedrawBackground;
PROCEDURE AttachWindowObject(u: Window; obj: ADDRESS); PROCEDURE WindowObject(u: Window): ADDRESS;

(===== O P T I O N A L M O D U L E S =====)

(***** DM2DGraphs *****

TYPE Graph; Curve;
LabelString = ARRAY[0..255] OF CHAR; GridFlag = (withGrid, withoutGrid);
ScalingType = (lin, log, negLog); PlottingStyle = (solid, slash, slashDot, dots, hidden, wipeout);
Range = RECORD min,max: REAL END; GraphProc = PROCEDURE(Graph);
AxisType = RECORD range: Range; scale: ScalingType; dec: CARDINAL; tickD: REAL; label: LabelString; END;

VAR DM2DGraphsDone: BOOLEAN; notExistingGraph: Graph; notExistingCurve: Curve;

PROCEDURE DefGraph(VAR g: Graph; u: Window; r: RectArea; xAxis, yAxis: AxisType; grid: GridFlag);
PROCEDURE DefCurve(g: Graph; VAR c: Curve; col: Color; style: PlottingStyle; sym: CHAR);
PROCEDURE RedefGraph(g: Graph; r: RectArea; xAxis,yAxis:AxisType; grid: GridFlag);
PROCEDURE RedefCurve(c: Curve; col: Color; style: PlottingStyle; sym: CHAR);
PROCEDURE ClearGraph(g: Graph); PROCEDURE DrawGraph(g: Graph);
PROCEDURE DrawLegend(c: Curve; x,y: INTEGER; comment: ARRAY OF CHAR);
PROCEDURE RemoveGraph(VAR g: Graph); PROCEDURE RemoveAllGraphs(u: Window);
PROCEDURE RemoveCurve(VAR c: Curve);
PROCEDURE GraphExists(g: Graph ): BOOLEAN; PROCEDURE CurveExists(g: Graph; c: Curve): BOOLEAN;
PROCEDURE DoForAllGraphs(u: Window; gp: GraphProc);
PROCEDURE SetNegLogMin(nlm: REAL); PROCEDURE SetGapSym(ch: CHAR); PROCEDURE GetGapSym(VAR ch: CHAR);

PROCEDURE Move(c: Curve; x,y: REAL); PROCEDURE Plot(curve: Curve; newX,newY: REAL);
PROCEDURE PlotSym(g: Graph; x,y: REAL; sym: CHAR); PROCEDURE PlotCurve(c: Curve; nrOfPoints: CARDINAL; x,y: ARRAY OF REAL);
PROCEDURE GraphToWindowPoint(g: Graph; xReal,yReal: REAL; VAR xInt,yInt: INTEGER);
PROCEDURE WindowToGraphPoint(g: Graph; xInt,yInt: INTEGER; VAR xReal,yReal: REAL);

(***** DMAlerts *****

PROCEDURE WriteMessage(line,col: CARDINAL; msg: ARRAY OF CHAR);
PROCEDURE ShowAlert(height,width: CARDINAL; WriteMessages: PROC);
PROCEDURE ShowPredefinedAlert(fileName: ARRAY OF CHAR; alertID: INTEGER; str1,str2,str3,str4: ARRAY OF CHAR);

(***** DMClipboard *****

TYPE EditCommands = (undo, cut, copy, paste, clear);

VAR ClipboardDone: BOOLEAN;

PROCEDURE InstallEditMenu(UndoProc, CutProc, CopyProc, PasteProc, ClearProc: PROC);
PROCEDURE RemoveEditMenu; PROCEDURE UseEditMenu;
PROCEDURE EnableEditMenu; PROCEDURE DisableEditMenu;
PROCEDURE EnableEditCommand(whichone: EditCommands); PROCEDURE DisableEditCommand(whichone: EditCommands);

PROCEDURE PutPictureIntoClipboard;
PROCEDURE GetPictureFromClipboard(simultaneousDisplay: BOOLEAN; destRect: RectArea);
PROCEDURE PutTextIntoClipboard;
PROCEDURE GetTextFromClipboard(simultaneousDisplay: BOOLEAN; destRect: RectArea; fromLine: LONGINT);

(***** DMClock *****

CONST Jan = 1; Feb = 2; Mar = 3; Apr = 4; Mai = 5; Jun = 6; Jul = 7; Aug = 8; Sep = 9; Oct =10; Nov =11; Dec = 12;
Sun = 1; Mon = 2; Tue = 3; Wed = 4; Thu = 5; Fri = 6; Sat = 7;

PROCEDURE Today(VAR year, month, day, dayOfWeek: INTEGER); PROCEDURE Now(VAR hour, minute, second: INTEGER);
PROCEDURE NowInSeconds(): LONGINT;
PROCEDURE InterpretSeconds(secs: LONGINT; VAR year, month, day, hour, minute, second, dayOfWeek: INTEGER);
PROCEDURE ConvertDateToSeconds(year, month, day, hour, minute, second: INTEGER; VAR secs: LONGINT);

(***** DMEditFields *****

TYPE EditItem; RadioBut; EditHandler = PROCEDURE(EditItem);
ItemfType = (charField, stringField, textField, cardField, intField, realField,
    pushButton, radioButtonSet, checkBox, scrollbar); Direction = (horizontal, vertical);

VAR EditFieldsDone: BOOLEAN; notInstalledEditItem: EditItem; notInstalledRadioBut: RadioBut;

PROCEDURE MakeCharField(u: Window; VAR ei: EditItem; x,y: INTEGER; ch: CHAR; charset: ARRAY OF CHAR);
PROCEDURE MakeStringField(u: Window; VAR ei: EditItem; x,y: INTEGER; fw: CARDINAL; string: ARRAY OF CHAR);
PROCEDURE MakeTextField(u: Window; VAR ei: EditItem; x,y: INTEGER; fw,lines: CARDINAL; string: ARRAY OF CHAR);
PROCEDURE MakeCardField(u: Window; VAR ei: EditItem; x,y: INTEGER; fw: CARDINAL; card: CARDINAL; minCard,maxCard: CARDINAL);
PROCEDURE MakeLongCardField(u: Window; VAR ei: EditItem; x,y: INTEGER; fw: CARDINAL;
    card: LONGCARD; minCard,maxCard: LONGCARD);
PROCEDURE MakeIntField(u: Window; VAR ei: EditItem; x,y: INTEGER; fw: CARDINAL; int: INTEGER; minInt,maxInt: INTEGER);
PROCEDURE MakeLongIntField(u: Window; VAR ei: EditItem; x,y: INTEGER; fw: CARDINAL;
    int: LONGINT; minInt,maxInt: LONGINT);
PROCEDURE MakeRealField(u: Window; VAR ei: EditItem; x,y: INTEGER; fw: CARDINAL; real: REAL; minReal,maxReal: REAL);
PROCEDURE MakeLongRealField(u: Window; VAR ei: EditItem; x,y: INTEGER; fw: CARDINAL;
    real: LONGREAL; minReal,maxReal: LONGREAL);

```

```

PROCEDURE MakePushButton(u: Window; VAR ei: EditItem; x,y: INTEGER;
    buttonWidth: CARDINAL; buttonText: ARRAY OF CHAR; pushButtonAction: PROC);
PROCEDURE UseAsDefaultButton(pushButton: EditItem);
PROCEDURE BeginRadioButtonSet(u: Window; VAR ei: EditItem);
PROCEDURE AddRadioButton(VAR radButt: RadioBut; x,y: INTEGER; text: ARRAY OF CHAR);
PROCEDURE EndRadioButtonSet(checkedRadioButton: RadioBut);
PROCEDURE MakeCheckBox(u: Window; VAR ei: EditItem; x,y: INTEGER; text: ARRAY OF CHAR; boxChecked: BOOLEAN);
PROCEDURE MakeScrollBar(u: Window; VAR ei: EditItem; x, y, length: INTEGER; sbd: Direction; minVal,maxVal: REAL;
    smallStep, bigStep: REAL; curVal: REAL; actionProc: PROC);

PROCEDURE SetChar(ei: EditItem; newCh:CHAR); PROCEDURE SetString(ei: EditItem; newStr: ARRAY OF CHAR);
PROCEDURE SetText(ei: EditItem; VAR text: ARRAY OF CHAR);
PROCEDURE SetCardinal(ei: EditItem; newValue: CARDINAL); PROCEDURE SetLongCardinal(ei: EditItem; newValue: LONGCARD);
PROCEDURE SetInteger(ei: EditItem; newValue: INTEGER); PROCEDURE SetLongInteger(ei: EditItem; newValue: LONGINT);
PROCEDURE SetReal(ei: EditItem; newValue: REAL); PROCEDURE SetLongReal(ei: EditItem; newValue: LONGREAL);
PROCEDURE SetRadioButtonSet(ei: EditItem; checkedRadioButton: RadioBut);
PROCEDURE SetCheckBox(ei: EditItem; boxChecked: BOOLEAN);
PROCEDURE SetScrollBar(ei: EditItem; newValue: REAL);

PROCEDURE IsChar(ei: EditItem; VAR ch:CHAR): BOOLEAN; PROCEDURE GetString(ei: EditItem; VAR str: ARRAY OF CHAR);
PROCEDURE GetText(ei: EditItem; VAR text: ARRAY OF CHAR);
PROCEDURE IsCardinal(ei: EditItem; VAR c: CARDINAL): BOOLEAN; PROCEDURE IsLongCardinal(ei: EditItem; VAR c: LONGCARD): BOOLEAN;
PROCEDURE IsInteger(ei: EditItem; VAR i: INTEGER): BOOLEAN; PROCEDURE IsLongInteger(ei: EditItem; VAR i: LONGINT): BOOLEAN;
PROCEDURE IsReal(ei: EditItem; VAR r: REAL): BOOLEAN; PROCEDURE IsLongReal(ei: EditItem; VAR r: LONGREAL): BOOLEAN;
PROCEDURE GetRadioButtonSet(ei: EditItem; VAR checkedRadioButton: RadioBut);
PROCEDURE GetCheckBox(ei: EditItem; VAR boxChecked: BOOLEAN);
PROCEDURE GetScrollBar(ei: EditItem; VAR r: REAL);

PROCEDURE InstallEditHandler(u: Window; eh: EditHandler); PROCEDURE GetEditHandler(u: Window; VAR eh: EditHandler);
PROCEDURE SelectField(ei: EditItem); PROCEDURE ClearFieldSelection (u: Window);

PROCEDURE EnableItem(ei: EditItem); PROCEDURE DisableItem(ei: EditItem); PROCEDURE IsEnabled(ei: EditItem): BOOLEAN;

PROCEDURE EditItemExists(ei: EditItem): BOOLEAN; PROCEDURE GetEditItemType(ei: EditItem; VAR it: ItemType);
PROCEDURE RadioButtonExists(rb: RadioBut): BOOLEAN;
PROCEDURE EditItemLevel(ei: EditItem): CARDINAL; PROCEDURE RadioButtonLevel(rb: RadioBut): CARDINAL;
PROCEDURE RemoveEditItem(VAR ei: EditItem); PROCEDURE RemoveAllEditItems(u: Window);
PROCEDURE AttachEditFieldObject(ei: EditItem; obj: ADDRESS); PROCEDURE EditFieldObject(ei: EditItem): ADDRESS;

(***** DMEntryForms *****
TYPE FormFrame = RECORD x,y: INTEGER; lines,columns: CARDINAL END; DefltUse = (useAsDeflt, noDeflt); RadioButtonID;
VAR FieldInstalled: BOOLEAN; notInstalledRadioButton: RadioButtonID;

PROCEDURE WriteLabel(line,col: CARDINAL; text: ARRAY OF CHAR);
PROCEDURE CharField(line,col: CARDINAL; VAR ch: CHAR; du: DefltUse; charset: ARRAY OF CHAR);
PROCEDURE StringField(line,col: CARDINAL; fw: CARDINAL; VAR string: ARRAY OF CHAR; du: DefltUse);
PROCEDURE CardField(line,col: CARDINAL; fw: CARDINAL; VAR card: CARDINAL; du: DefltUse; minCard,maxCard: CARDINAL);
PROCEDURE LongCardField (line,col: CARDINAL; fw: CARDINAL; VAR longCard: LONGCARD; du: DefltUse; minLCard,maxLCard: LONGCARD);
PROCEDURE IntField(line,col: CARDINAL; fw: CARDINAL; VAR int: INTEGER; du: DefltUse; minInt,maxInt: INTEGER);
PROCEDURE LongIntField (line,col: CARDINAL; fw: CARDINAL; VAR longInt: LONGINT; du: DefltUse; minLInt,maxLInt: LONGINT);
PROCEDURE RealField(line,col: CARDINAL; fw: CARDINAL; VAR real: REAL; du: DefltUse; minReal,maxReal: REAL);
PROCEDURE LongRealField (line,col: CARDINAL; fw,dig: CARDINAL; fmt: RealFormat; VAR longReal: LONGREAL; du: DefltUse;
    minLReal,maxLReal: LONGREAL);
PROCEDURE PushButton(line,col: CARDINAL; buttonText: ARRAY OF CHAR; buttonWidth: CARDINAL; pushButtonAction: PROC);
PROCEDURE DefineRadioButtonSet(VAR radioButtonVar: RadioButtonID);
PROCEDURE RadioButton(VAR radButt: RadioButtonID; line,col: CARDINAL; text: ARRAY OF CHAR);
PROCEDURE CheckBox(line,col: CARDINAL; text: ARRAY OF CHAR; VAR checkBoxVar: BOOLEAN);
PROCEDURE UseEntryForm(bf: FormFrame; VAR ok: BOOLEAN);

(***** DMFiles *****
CONST EOL = 36C;

TYPE Response = (done, filenotfound, volnotfound, cancelled, unknownfile, toomanyfiles, diskfull, memfull,
    alreadyopened, isbusy, locked, notdone);
HiddenFileInfo: IOMode = (reading, writing);
TextFile = RECORD
    res: Response;
    filename: ARRAY [0..255] OF CHAR;
    path: ARRAY [0..63] OF CHAR;
    curIOMode: IOMode;
    curChar: CHAR;
    fhint: HiddenFileInfo;
END;

VAR
    legalNum: BOOLEAN; (* read only *) PROCEDURE LastResultCode(): INTEGER;
    neverOpenedFile: TextFile; (* read only *)

PROCEDURE GetExistingFile(VAR f: TextFile; prompt: ARRAY OF CHAR);
PROCEDURE CreateNewFile(VAR f: TextFile; prompt, defaultName: ARRAY OF CHAR);
PROCEDURE Lookup(VAR f: TextFile; pathAndFileName: ARRAY OF CHAR; new: BOOLEAN);
PROCEDURE ReadOnlyLookup(VAR f: TextFile; pathAndFileName: ARRAY OF CHAR);
PROCEDURE Close(VAR f: TextFile); PROCEDURE IsOpen(VAR f: TextFile): BOOLEAN;
PROCEDURE FileExists(VAR f: TextFile): BOOLEAN; PROCEDURE FileLevel(VAR f: TextFile): CARDINAL;

PROCEDURE Delete(VAR f: TextFile); PROCEDURE Rename(VAR f: TextFile; filename: ARRAY OF CHAR);
PROCEDURE Reset(VAR f: TextFile); PROCEDURE Rewrite(VAR f: TextFile);
PROCEDURE AppendAtEOF(VAR f: TextFile); PROCEDURE FileSize(VAR f: TextFile): LONGINT;

PROCEDURE EOF(VAR f: TextFile): BOOLEAN;
PROCEDURE ReadByte(VAR f: TextFile; VAR b: BYTE); PROCEDURE WriteByte(VAR f: TextFile; b: BYTE);
PROCEDURE ReadChar(VAR f: TextFile; VAR ch: CHAR); PROCEDURE WriteChar(VAR f: TextFile; ch: CHAR);
PROCEDURE ReadChars(VAR f: TextFile; VAR string: ARRAY OF CHAR); PROCEDURE WriteChars(VAR f: TextFile; string: ARRAY OF CHAR);
PROCEDURE WriteBOL(VAR f: TextFile); PROCEDURE WriteVarChars(VAR f: TextFile; VAR string: ARRAY OF CHAR);
PROCEDURE SkipGap(VAR f: TextFile); PROCEDURE Again(VAR f: TextFile);
PROCEDURE GetCardinal(VAR f: TextFile; VAR c: CARDINAL); PROCEDURE GetLongCard(VAR f: TextFile; VAR c: LONGCARD);
PROCEDURE PutCardinal(VAR f: TextFile; c: CARDINAL; n: CARDINAL); PROCEDURE PutLongCard(VAR f: TextFile; lc: LONGCARD;
    n: CARDINAL);
PROCEDURE GetInteger(VAR f: TextFile; VAR i: INTEGER); PROCEDURE GetLongInt(VAR f: TextFile; VAR i: LONGINT);
PROCEDURE PutInteger(VAR f: TextFile; i: INTEGER; n: CARDINAL); PROCEDURE PutLongInt(VAR f: TextFile; li: LONGINT;
    n: CARDINAL);
PROCEDURE GetReal(VAR f: TextFile; VAR x: REAL); PROCEDURE GetLongReal(VAR f: TextFile; VAR x: LONGREAL);
PROCEDURE PutReal(VAR f: TextFile; x: REAL; n, dec: CARDINAL); PROCEDURE PutRealSci(VAR f: TextFile; x: REAL; n: CARDINAL);
PROCEDURE PutLongReal(VAR f: TextFile; lr: LONGREAL; n,dec: CARDINAL);
PROCEDURE PutLongRealSci(VAR f: TextFile; lr: LONGREAL; n,dec: CARDINAL);

PROCEDURE AlterIOMode (VAR f: TextFile; newMode: IOMode);
PROCEDURE SetFilePos (VAR f: TextFile; pos: LONGINT ); PROCEDURE GetFilePos (VAR f: TextFile; VAR pos: LONGINT );
PROCEDURE ReadByteBlock ( VAR f: TextFile; VAR buf: ARRAY OF BYTE; VAR count: LONGINT );
PROCEDURE WriteByteBlock (VAR f: TextFile; VAR buf: ARRAY OF BYTE; VAR count: LONGINT );

PROCEDURE SetFileFilter(f1,f2,f3,f4: ARRAY OF CHAR); PROCEDURE GetFileFilter(VAR f1,f2,f3,f4: ARRAY OF CHAR);

```

## ModelWorks V2.2 - Appendix (Interfaces and Libraries)

```

PROCEDURE UseAsTypeAndCreator(fileType,creator: ARRAY OF CHAR); PROCEDURE UsedTypeAndCreator(VAR fileType,creator: ARRAY OF CHAR);
PROCEDURE HasTypeAndCreator(VAR f: TextFile; VAR fileType,creator: ARRAY OF CHAR);

(***** DMFloatEnv *****)

CONST invalid = 0; underflow = 1; overflow = 2; divideByZero = 3; inexact = 4;
  haltIfInvalid = 0; haltIfUnderflow = 1; haltIfOverflow = 2; haltIfDivideByZero = 3; haltIfInexact = 4;
  flagIfInvalid = 8; flagIfUnderflow = 9; flagIfOverflow = 10; flagIfDivideByZero = 11; flagIfInexact = 12;
IEEEFloatDefaultEnv = ; DMFloatDefaultEnv = haltIfInvalid, haltIfOverflow, haltIfDivideByZero;

TYPE Exception = [invalid..inexact]; FloatEnvironment = BITSET;
  RoundDir = (toNearest, upward, downward, towardZero); RoundPre = (extPrecision, dblPrecision, sglPrecision);

PROCEDURE HaltEnabled(which: Exception): BOOLEAN; PROCEDURE DisableHalt(which: Exception);
PROCEDURE EnableHalt(which: Exception); PROCEDURE ExceptionPending(which: Exception): BOOLEAN;
PROCEDURE RaiseException(which: Exception); PROCEDURE ClearException(which: Exception);
PROCEDURE SetPrecision(p: RoundPre); PROCEDURE GetPrecision(VAR p: RoundPre);
PROCEDURE SetRound(r: RoundDir); PROCEDURE GetRound(VAR r: RoundDir);
PROCEDURE GetEnvironment(VAR e: FloatEnvironment); PROCEDURE SetEnvironment(e: FloatEnvironment);
PROCEDURE ProcEntry(VAR savedEnv: FloatEnvironment); PROCEDURE ProcExit(savedEnv: FloatEnvironment);

(***** DMKeyChars *****)

CONST mouse=0; command=1; alt=1; option=2; shift=3; capslock=4; control=5;
VAR cursorUp, cursorDown, cursorLeft, cursorRight, homeKey, endKey, pageUp, pageDown, helpKey, enter, return, delete,
  backspace, tab, esc, hardBlank: CHAR; (* READ ONLY! *)

VAR BestCH: PROCEDURE(CHAR): CHAR; (* READ ONLY! *)
TYPE ComputerPlatform=( Mac, IBMPCCompatible, UNIXMachine); PROCEDURE ProgrammedOn( c: ComputerPlatform);

PROCEDURE PCCHAR( macCh: CHAR): CHAR; PROCEDURE MacCHAR( pcCh: CHAR): CHAR;
PROCEDURE PCASCII( pcCh: CHAR): CHAR; PROCEDURE MacASCII( macCh: CHAR): CHAR;

(***** DMMathLib/DMMathLF *****)

VAR undefREAL: REAL; undefLONGREAL: LONGREAL; (* read only *)

PROCEDURE Sqrt( x: REAL): REAL;
PROCEDURE Exp( x: REAL): REAL; PROCEDURE Ln( x: REAL): REAL;
PROCEDURE Sin( x: REAL): REAL; PROCEDURE Cos( x: REAL): REAL; PROCEDURE ArcTan(x: REAL): REAL;
PROCEDURE Real( x: INTEGER): REAL; PROCEDURE Entier(x: REAL): INTEGER;
PROCEDURE Randomize; PROCEDURE RandomInt(upperBound: INTEGER): INTEGER; PROCEDURE RandomReal(): REAL;

(***** DMOpSys *****)

CONST noError = 0; notDone = -2; inexistent = -1; notOpen = 0; readOnly = 1; alreadyWrite = 2; (* codes returned by CurrentFileUse*)

TYPE ProgStatus = (regular, moduleNotFound, fileNotFound, illegalKey, readError, badSyntax, noMemory, alreadyLoaded,
  killed, tooManyPrograms, continue, noApplication);
DirectoryProc = PROCEDURE (INTEGER, ARRAY OF CHAR, BOOLEAN, VAR BOOLEAN);
MessageResponder = PROCEDURE (ARRAY OF CHAR, ARRAY OF CHAR, INTEGER);
InitDocuHandlingProc = PROCEDURE (INTEGER); DocuHandler = PROCEDURE (INTEGER, ARRAY OF CHAR, ARRAY OF CHAR, VAR BOOLEAN);

VAR profileName: ARRAY [0..127] OF CHAR;

PROCEDURE CurWorkDirectory(VAR path: ARRAY OF CHAR); PROCEDURE GetLastResultCode(): INTEGER;
PROCEDURE Createdir( path, dirN: ARRAY OF CHAR; VAR done: BOOLEAN );
PROCEDURE Deletedir( path, dirN: ARRAY OF CHAR; VAR done: BOOLEAN );
PROCEDURE Renamedir( path, oldDirN, newDirN: ARRAY OF CHAR; VAR done: BOOLEAN );
PROCEDURE DirInfo( path, dirN: ARRAY OF CHAR; VAR dirExists, containsFiles: BOOLEAN );
PROCEDURE DoForAllFilesInDirectory(path: ARRAY OF CHAR; dp: DirectoryProc);
PROCEDURE CurrentFileUse( path, fileName: ARRAY OF CHAR): INTEGER;
PROCEDURE GetFileDialog(prompt, fileTypes: ARRAY OF CHAR; VAR path, fileName: ARRAY OF CHAR): BOOLEAN;
PROCEDURE GetApplication(VAR path, appName: ARRAY OF CHAR): BOOLEAN;
PROCEDURE GetFileTypeAndCreator(path, fn: ARRAY OF CHAR; VAR type, creator: ARRAY OF CHAR);
PROCEDURE SetFileTypeAndCreator(path, fn: ARRAY OF CHAR; type, creator: ARRAY OF CHAR);
PROCEDURE GetFileDates(path, fn: ARRAY OF CHAR; VAR creationDate, modificationDate: LONGINT);
PROCEDURE SetFileDates(path, fn: ARRAY OF CHAR; creationDate, modificationDate: LONGINT);
PROCEDURE NowSeconds(): LONGINT; PROCEDURE TouchFileDate(path, fn: ARRAY OF CHAR);
PROCEDURE CopyResourceFork(sourcePath, sourceFn, destPath, destFn: ARRAY OF CHAR; VAR done: BOOLEAN);
PROCEDURE CopyDataFork( sourcePath, sourceFn, destPath, destFn: ARRAY OF CHAR; VAR done: BOOLEAN);
PROCEDURE InstallInitDocuOpening( idhp: InitDocuHandlingProc); PROCEDURE InstallOpenDocuHandler( dh: DocuHandler);
PROCEDURE InstallInitDocuPrinting( idhp: InitDocuHandlingProc); PROCEDURE InstallPrintDocuHandler( dh: DocuHandler);
PROCEDURE SubLaunch(path, prog: ARRAY OF CHAR); PROCEDURE Transfer(path, prog: ARRAY OF CHAR);
PROCEDURE IsForegroundProgram(): BOOLEAN;
PROCEDURE SetMessageResponder( mr: MessageResponder); PROCEDURE GetMessageResponder(VAR mr: MessageResponder);
PROCEDURE SignalMessageToApplication(creatorOfAppl, eventClass, eventID: ARRAY OF CHAR;
  msgVal: INTEGER; VAR resultCode: INTEGER);

PROCEDURE EmulateKeyPress(ch: CHAR; modifier: BITSET); PROCEDURE EmulateMenuSelection(aliasChar: CHAR);
PROCEDURE EmulateMouseDown(x,y: INTEGER; modifier: BITSET);
PROCEDURE TurnMachineOff; PROCEDURE RestartMachine;

PROCEDURE SetNewPaths; PROCEDURE EmulateMacMETHCopyProtection;
PROCEDURE CallDMSubProg(prog: ARRAY OF CHAR; leaveLoaded: BOOLEAN; VAR st: ProgStatus);
PROCEDURE CallM2SubProg(prog: ARRAY OF CHAR; leaveLoaded: BOOLEAN; VAR st: ProgStatus);
PROCEDURE IncludeLibModules(prog: ARRAY OF CHAR; VAR st: ProgStatus);
PROCEDURE UnLoadM2Progs; PROCEDURE AbortM2Prog(st: ProgStatus);
PROCEDURE SetCompilerFileTypes(creator, sbmType, obmType, rfnType: ARRAY OF CHAR);
PROCEDURE GetCompilerFileTypes(VAR creator, sbmType, obmType, rfnType: ARRAY OF CHAR);

(***** DMPortab *****)

PROCEDURE SameProc( p1, p2: ARRAY OF BYTE): BOOLEAN;
PROCEDURE LongTRUNC( x: LONGREAL): LONGINT; PROCEDURE LongFLOAT( x: LONGINT): LONGREAL;
PROCEDURE LONGINTConst( str: ARRAY OF CHAR): LONGINT; PROCEDURE LONGREALConst( str: ARRAY OF CHAR): LONGREAL;

(***** DMPrinting *****)

TYPE PrinterFont = (chicago, newYork, geneva, monaco, times, helvetica, courier, symbol);

VAR PrintingDone: BOOLEAN;

PROCEDURE PageSetup; PROCEDURE SetHeaderText(h: ARRAY OF CHAR);
PROCEDURE SetSubHeaderText(sh: ARRAY OF CHAR); PROCEDURE SetFooterText(f: ARRAY OF CHAR);
PROCEDURE PrintPicture;
PROCEDURE PrintText(font: PrinterFont; fontSize: INTEGER; tabwidth: INTEGER);

(***** DMPTFiles *****)

VAR PTFileDone: BOOLEAN;

PROCEDURE DumpPicture(VAR f: TextFile);
PROCEDURE LoadPicture( VAR f: TextFile; simulDisplay: BOOLEAN; destRect: RectArea);

```



```

PROCEDURE DumpText(VAR f: TextFile);
PROCEDURE LoadText (VAR f: TextFile; simulDisplay: BOOLEAN; destRect: RectArea; fromLine: LONGINT);

(***** DMResources *****)

CONST nulCh = 21C;

TYPE ResourcePointer = POINTER TO Resource; Resource = ARRAY [0..32000] OF CHAR; Padding = (noPadding, padToEven, padToOdd);

VAR theResource: ResourcePointer; ResourcesDone: BOOLEAN;

PROCEDURE StartResourceComposition; PROCEDURE CurPosition(): INTEGER;
PROCEDURE AddBoolean(b: BOOLEAN);
PROCEDURE AddInt(int: INTEGER); PROCEDURE AddLongInt(lint: LONGINT);
PROCEDURE AddHexInt(int: INTEGER); PROCEDURE AddHexLongInt (lint: LONGINT);
PROCEDURE AddBinInt(int: INTEGER); PROCEDURE AddBinLongInt(lint: LONGINT);
PROCEDURE AddReal(r: REAL); PROCEDURE AddLongReal(lr: LONGREAL);
PROCEDURE AddHexReal(r: REAL); PROCEDURE AddHexLongReal(lr: LONGREAL);
PROCEDURE AddBinReal(r: REAL); PROCEDURE AddBinLongReal(lr: LONGREAL);
PROCEDURE AddChar(ch: CHAR); PROCEDURE AddString(s: ARRAY OF CHAR);
PROCEDURE AddString255(s: ARRAY OF CHAR; pad: Padding);
PROCEDURE OverWriteAtPos (VAR x: ARRAY OF BYTE; VAR theResource: ARRAY OF CHAR; VAR curPos: INTEGER);
PROCEDURE StoreResource(filename: ARRAY OF CHAR; resID: INTEGER);

PROCEDURE RetrieveResource(filename: ARRAY OF CHAR; resID: INTEGER);
PROCEDURE FetchBoolean(VAR b: BOOLEAN);
PROCEDURE FetchInt(VAR int: INTEGER); PROCEDURE FetchLongInt(VAR lint: LONGINT);
PROCEDURE FetchHexInt(VAR int: INTEGER); PROCEDURE FetchHexLongInt(VAR lint: LONGINT);
PROCEDURE FetchBinInt(VAR int: INTEGER); PROCEDURE FetchBinLongInt(VAR lint: LONGINT);
PROCEDURE FetchReal(VAR r: REAL); PROCEDURE FetchLongReal(VAR lr: LONGREAL);
PROCEDURE FetchHexReal(VAR r: REAL); PROCEDURE FetchHexLongReal(VAR lr: LONGREAL);
PROCEDURE FetchBinReal(VAR r: REAL); PROCEDURE FetchBinLongReal(VAR lr: LONGREAL);
PROCEDURE FetchChar(VAR ch: CHAR); PROCEDURE FetchString(VAR s: ARRAY OF CHAR);
PROCEDURE FetchString255(VAR s: ARRAY OF CHAR; pad: Padding);

PROCEDURE DeleteResource(filename: ARRAY OF CHAR; resID: INTEGER);
PROCEDURE SetResourceName(fileName: ARRAY OF CHAR; resID: INTEGER; name: ARRAY OF CHAR);
PROCEDURE GetResourceName(fileName: ARRAY OF CHAR; resID: INTEGER; VAR name: ARRAY OF CHAR);
PROCEDURE SetResourceType(type: ARRAY OF CHAR);
PROCEDURE GetResourceType(VAR type: ARRAY OF CHAR);

(***** DMTextFields *****)

TYPE TextPointer = POINTER TO TextSegment; TextSegment = ARRAY [0..32000] OF CHAR;

PROCEDURE RedefineTextField(textField: EditItem; wF: WindowFrame; withFrame: BOOLEAN);
PROCEDURE WrapText(textField: EditItem; wrap: BOOLEAN);
PROCEDURE CopyWTextIntoTextField(textField: EditItem; VAR done: BOOLEAN);
PROCEDURE CopyTextFromFieldToWText(textField: EditItem);

PROCEDURE SetSelection(textField: EditItem; beforeCh,afterCh: INTEGER);
PROCEDURE GetSelection(textField: EditItem; VAR beforeCh,afterCh: INTEGER);
PROCEDURE GetSelectedChars(textField: EditItem; VAR text: ARRAY OF CHAR);
PROCEDURE DeleteSelection(textField: EditItem);
PROCEDURE InsertBeforeCh(textField: EditItem; VAR text: ARRAY OF CHAR; beforeCh: INTEGER);

PROCEDURE GetTextSizes(textField: EditItem; VAR curTextLength, nrLns, charHeight, firstLnVis,lastLnVis: INTEGER);
PROCEDURE GrabText(textField: EditItem; VAR txtbeg: TextPointer; VAR curTextLength: INTEGER);
PROCEDURE ReleaseText(textField: EditItem);
PROCEDURE FindImText(textField: EditItem; stringToFind: ARRAY OF CHAR; VAR firstCh,lastCh: INTEGER): BOOLEAN;
PROCEDURE ScrollText(textField: EditItem; dcols,dlines: INTEGER);
PROCEDURE ScrollTextWithWindowScrollBars(textField: EditItem);
PROCEDURE AddScrollBarsToText(textField: EditItem; withVerticalScrollBar, withHorizontalScrollBar: BOOLEAN);

(***** DMWPictIO *****)

VAR PictIODone: BOOLEAN;

PROCEDURE StartPictureSave; PROCEDURE StopPictureSave;
PROCEDURE PausePictureSave; PROCEDURE ResumePictureSave;
PROCEDURE DisplayPicture(ownerWindow: Window; destRect: RectArea); PROCEDURE DiscardPicture;

PROCEDURE SetPictureArea(r: RectArea); PROCEDURE GetPictureArea(VAR r: RectArea);
PROCEDURE SetHairLineWidth(f: REAL); PROCEDURE GetHairLineWidth(VAR f: REAL);

(***** DMWTextIO *****)

VAR TextIODone: BOOLEAN;

PROCEDURE StartTextSave; PROCEDURE StopTextSave;
PROCEDURE PauseTextSave; PROCEDURE ResumeTextSave;
PROCEDURE DisplayText(ownerWindow: Window; destRect: RectArea; fromLine: LONGINT); PROCEDURE DiscardText;

PROCEDURE GrabWText(VAR txtbeg: ADDRESS; VAR curTextLength: LONGINT); PROCEDURE ReleaseWText;
PROCEDURE AppendWText(txtbeg: ADDRESS; length: LONGINT); PROCEDURE SetWTextSize(newTextLength: LONGINT);

(===== - E N D - =====)

```

The Dialog Machine may be freely copied but not for profit!

| Different from Version 1.

## E.3 MODEL WORKS CLIENT INTERFACE

The following listing of the client interface is identical for all ModelWorks versions (V2.2, V2.0/Reflex, V1.1/PC, V2.2/PC, and V2.2/II). For a detailed description see part III *Reference* the chapter *Client Interface*.

```

ModelWorks Version 2.2 (April 1996)
© 1989 - 1996 Andreas Fischlin, Dimitrios Gyalistras, Olivier Roth, Markus Ulrich,
Juerg Thoery, Thomas Nemecek, Harald Bugmann & Frank Thommen

Swiss Federal Institute of Technology Zurich ETHZ, Switzerland.

(===== CLIENT INTERFACE MODULES =====)

(***** SimBase *****)

(* Declaration of models and model objects: *)
(* ----- *)

TYPE
Model;
StateVar = REAL;      Derivative = REAL;      NewState = REAL;
AuxVar   = REAL;      Parameter   = REAL;
InVar    = REAL;      OutVar     = REAL;

IntegrationMethod = (Euler, Heun, RungeKutta4, RungeKutta45Var, stiff, discreteTime, discreteEvent);
RTCType = (rtc, noRtc);
StashFiling = (writeOnFile, notOnFile);
Tabulation = (writeInTable, notInTable);
Graphing = (isX, isY, isZ, notInGraph);

| VAR notDeclaredModel: Model; (* read only variable *)

PROCEDURE DeclM(VAR m: Model; defaultMethod: IntegrationMethod; initialize, input, output, dynamic, terminate: PROC;
declModelObjects: PROC; descriptor, identifier: ARRAY OF CHAR; about: PROC);
PROCEDURE DeclSV(VAR s: StateVar; VAR ds: Derivative (*or NewState*); defaultInitial, minCurInit, maxCurInit: REAL;
descriptor, identifier, unit: ARRAY OF CHAR);
PROCEDURE DeclP(VAR p: Parameter; defaultVal, minCurVal, maxCurVal: REAL; runTimeChange: RTCType;
descriptor, identifier, unit: ARRAY OF CHAR);
PROCEDURE DeclMV(VAR mv: REAL; defaultScaleMin, defaultScaleMax: REAL; descriptor, identifier, unit: ARRAY OF CHAR;
defaultSF: StashFiling; defaultT: Tabulation; defaultG: Graphing);

| PROCEDURE CurCalcM(): Model;
PROCEDURE CurAboutM(): Model;
PROCEDURE SelectM (m: Model; VAR done: BOOLEAN);

PROCEDURE NoInitialize; PROCEDURE NoInput; PROCEDURE NoOutput; PROCEDURE NoDynamic; PROCEDURE NoTerminate;
PROCEDURE NoModelObjects; PROCEDURE NoAbout; PROCEDURE DoNothing;

(* Modifying of models and model objects: *)
(* ----- *)

PROCEDURE GetDefltM (VAR m: Model; VAR defaultMethod: IntegrationMethod;
VAR initialize, input, output, dynamic, terminate: PROC;
VAR descriptor, identifier: ARRAY OF CHAR; VAR about: PROC);
PROCEDURE SetDefltM (VAR m: Model; defaultMethod: IntegrationMethod;
initialize, input, output, dynamic, terminate: PROC;
descriptor, identifier: ARRAY OF CHAR; about: PROC);
PROCEDURE GetDefltSV (m: Model; VAR s: StateVar; VAR defaultInit, minCurInit, maxCurInit: REAL;
VAR descriptor, identifier, unit: ARRAY OF CHAR);
PROCEDURE SetDefltSV (m: Model; VAR s: StateVar; defaultInit, minCurInit, maxCurInit: REAL;
descriptor, identifier, unit: ARRAY OF CHAR);
PROCEDURE GetDefltP (m: Model; VAR p: Parameter; VAR defaultVal, minVal, maxVal: REAL;
VAR runTimeChange: RTCType;
VAR descriptor, identifier, unit: ARRAY OF CHAR);
PROCEDURE SetDefltP (m: Model; VAR p: Parameter; defaultVal, minVal, maxVal: REAL;
runTimeChange: RTCType;
descriptor, identifier, unit: ARRAY OF CHAR);
PROCEDURE GetDefltMV (m: Model; VAR mv: REAL; VAR defaultScaleMin, defaultScaleMax: REAL;
VAR descriptor, identifier, unit: ARRAY OF CHAR;
VAR defaultSF: StashFiling; VAR defaultT: Tabulation;
VAR defaultG: Graphing);
PROCEDURE SetDefltMV (m: Model; VAR mv: REAL; defaultScaleMin, defaultScaleMax: REAL;
descriptor, identifier, unit: ARRAY OF CHAR;
defaultSF: StashFiling; defaultT: Tabulation;
defaultG: Graphing);

PROCEDURE GetM (VAR m: Model; VAR curMethod: IntegrationMethod);
PROCEDURE SetM (VAR m: Model; curMethod: IntegrationMethod);
PROCEDURE GetSV (m: Model; VAR s: StateVar; VAR curInit: REAL);
PROCEDURE SetSV (m: Model; VAR s: StateVar; curInit: REAL);
PROCEDURE GetP (m: Model; VAR p: Parameter; VAR curVal: REAL);
PROCEDURE SetP (m: Model; VAR p: Parameter; curVal: REAL);
PROCEDURE GetMV (m: Model; VAR mv: REAL; VAR curScaleMin, curScaleMax: REAL;
VAR curSF: StashFiling; VAR curT: Tabulation; VAR curG: Graphing);
PROCEDURE SetMV (m: Model; VAR mv: REAL; curScaleMin, curScaleMax: REAL;
curSF: StashFiling; curT: Tabulation; curG: Graphing);

| PROCEDURE ResetAllIntegrationMethods;
PROCEDURE ResetAllInitialValues;
PROCEDURE ResetAllParameters;
PROCEDURE ResetAllStashFiling;
PROCEDURE ResetAllTabulation;
PROCEDURE ResetAllGraphing;
PROCEDURE ResetAllScaling;

(* Model attributes: *)
(* ----- *)

| TYPE Attribute = INTEGER; CONST noAttr = MIN(Attribute);
PROCEDURE SetModelAttr(m: Model; val: Attribute);

```

```

PROCEDURE GetModelAttr(m: Model): Attribute;
PROCEDURE SetObjAttr(m: Model; VAR o: REAL; val: Attribute);
PROCEDURE GetObjAttr(m: Model; VAR o: REAL): Attribute;

(* Access helps for all models and all model objects: *)
(* ----- *)

PROCEDURE MDeclared(m: Model): BOOLEAN;
PROCEDURE SVDeclared(m: Model; VAR sv: StateVar): BOOLEAN;
PROCEDURE PDeclared(m: Model; VAR p: Parameter): BOOLEAN;
PROCEDURE MVDeclared(m: Model; VAR mv: REAL): BOOLEAN;

| TYPE ModelProc = PROCEDURE( VAR Model, VAR Attribute );      ModelObjectProc = PROCEDURE( Model, VAR REAL, VAR Attribute );

PROCEDURE DoForAllModels( p: ModelProc );
PROCEDURE DoForAllSVs ( m: Model; p: ModelObjectProc );
PROCEDURE DoForAllPs ( m: Model; p: ModelObjectProc );
PROCEDURE DoForAllMVs ( m: Model; p: ModelObjectProc );

(* Removing of models and model objects: *)
(* ----- *)

PROCEDURE RemoveM (VAR m: Model);
PROCEDURE RemoveSV (m: Model; VAR s : StateVar);
PROCEDURE RemoveMV (m: Model; VAR mv: REAL);
PROCEDURE RemoveP (m: Model; VAR p : Parameter);
PROCEDURE RemoveAllModels;

(* Global simulation parameters and project description: *)
(* ----- *)

PROCEDURE SetDefltGlobSimPars( t0, tend, h, er, c, hm: REAL);
PROCEDURE GetDefltGlobSimPars(VAR t0, tend, h, er, c, hm: REAL);

PROCEDURE SetDefltProjDescrs( title,remark,footer: ARRAY OF CHAR;
                             wtitle,wremark,autofooter,
                             recM, recSV, recP, recMV, recG: BOOLEAN);
PROCEDURE GetDefltProjDescrs(VAR title,remark,footer: ARRAY OF CHAR;
                             VAR wtitle,wremark,autofooter,
                             recM, recSV, recP, recMV, recG: BOOLEAN);
| PROCEDURE SetDefltTabFuncRecording( recTF: BOOLEAN);
| PROCEDURE GetDefltTabFuncRecording( VAR recTF: BOOLEAN);

| PROCEDURE SetDefltIndepVarIdent( descr,ident,unit: ARRAY OF CHAR);
| PROCEDURE GetDefltIndepVarIdent(VAR descr,ident,unit: ARRAY OF CHAR);

PROCEDURE SetGlobSimPars( t0, tend, h, er, c, hm: REAL);
PROCEDURE GetGlobSimPars(VAR t0, tend, h, er, c, hm: REAL);
PROCEDURE SetProjDescrs( title,remark,footer: ARRAY OF CHAR;
                        wtitle,wremark,autofooter,
                        recM, recSV, recP, recMV, recG: BOOLEAN);
PROCEDURE GetProjDescrs(VAR title,remark,footer: ARRAY OF CHAR;
                        VAR wtitle,wremark,autofooter,
                        recM, recSV, recP, recMV, recG: BOOLEAN);
| PROCEDURE SetTabFuncRecording( recTF: BOOLEAN);
| PROCEDURE GetTabFuncRecording(VAR recTF: BOOLEAN);

| PROCEDURE SetIndepVarIdent( descr,ident,unit: ARRAY OF CHAR);
| PROCEDURE GetIndepVarIdent(VAR descr,ident,unit: ARRAY OF CHAR);

| PROCEDURE ResetGlobSimPars;
| PROCEDURE ResetProjDescrs;

PROCEDURE SetMonInterval(hm: REAL); (* only for upward compatibility *)
PROCEDURE SetIntegrationStep(h: REAL); (* only for upward compatibility *)
PROCEDURE SetSimTime(t0,tend: REAL); (* only for upward compatibility *)

(* Control of Display and Monitoring: *)
(* ----- *)

PROCEDURE TileWindows;
PROCEDURE StackWindows;

| PROCEDURE InstallTileWindowsHandler(doAtTile:PROC);
| PROCEDURE InstallStackWindowsHandler(doAtStack:PROC);

TYPE MWindow = (MIOW, SVIOW, PIOW, MVIOW, TableW, GraphW, AboutMW, TimeW);

PROCEDURE SetWindowPlace(mmw: MWindow; x,y,w,h: INTEGER);
PROCEDURE GetWindowPlace(mmw: MWindow; VAR x,y,w,h: INTEGER; VAR isOpen : BOOLEAN);
PROCEDURE SetDefltWindowPlace(mmw: MWindow; x,y,w,h: INTEGER);
PROCEDURE GetDefltWindowPlace(mmw: MWindow; VAR x,y,w,h: INTEGER; VAR enabled: BOOLEAN);
PROCEDURE CloseWindow(w: MWindow);

TYPE
IOWColsDisplay = RECORD
  descrCol, identCol : BOOLEAN;
CASE iow: MWindow OF
  MIOW : m : RECORD
    integMethCol: BOOLEAN;
    END(*RECORD*);
  | SVIOW : sv: RECORD
    unitCol, sVinitCol: BOOLEAN;
    fw,dec: INTEGER;
    END(*RECORD*);
  | PIOW : p : RECORD
    unitCol, pValCol, pRtcCol: BOOLEAN;
    fw,dec: INTEGER;
    END(*RECORD*);
  | MVIOW : mv: RECORD
    unitCol, scaleMinCol, scaleMaxCol, mVmonSetCol: BOOLEAN;
    fw,dec: INTEGER;
    END(*RECORD*);
END(*CASE*)
END(*RECORD*);

PROCEDURE SetIOWColDisplay (mmw: MWindow; wd: IOWColsDisplay );
PROCEDURE GetIOWColDisplay (mmw: MWindow; VAR wd: IOWColsDisplay );
PROCEDURE SetDefltIOWColDisplay(mmw: MWindow; wd: IOWColsDisplay );
PROCEDURE GetDefltIOWColDisplay(mmw: MWindow; VAR wd: IOWColsDisplay );

```

## ModelWorks V2.2 - Appendix (Interfaces and Libraries)

```

PROCEDURE DisableWindow(w: MWindow);
PROCEDURE EnableWindow (w: MWindow);

TYPE MWindowArrangement = (current, stacked, tiled);
PROCEDURE SetDeflItWindowArrangement(a: MWindowArrangement);
PROCEDURE ResetWindows;

PROCEDURE SuppressMonitoring;
PROCEDURE ResumeMonitoring;
PROCEDURE InstallClientMonitoring( initClientMon, doClientMon, termClientMon: PROC );

PROCEDURE SetStashFileName      ( sfn: ARRAY OF CHAR);
PROCEDURE GetStashFileName      (VAR sfn: ARRAY OF CHAR);
PROCEDURE SetDeflItStashFileName( dsfn: ARRAY OF CHAR);
PROCEDURE GetDeflItStashFileName(VAR dsfn: ARRAY OF CHAR);
PROCEDURE SetStashFileType      ( filetype, creator: ARRAY OF CHAR);
PROCEDURE GetStashFileType      (VAR filetype, creator: ARRAY OF CHAR);
PROCEDURE SetDeflItStashFileType( dFileType, dCreator: ARRAY OF CHAR);
PROCEDURE GetDeflItStashFileType(VAR dFileType, dCreator: ARRAY OF CHAR);
PROCEDURE SwitchStashFile      (newsfn: ARRAY OF CHAR);
PROCEDURE ResetStashFile;

PROCEDURE Message(m: ARRAY OF CHAR);

TYPE
Stain = (coal, snow, ruby, emerald, sapphire, turquoise, pink, gold, autoDefCol);
LineStyle = (unbroken, broken, dashSpotted, spotted, invisible, purge, autoDefStyle);

CONST
autoDefSym = 200C;

PROCEDURE SetCurveAttrForMV(m: Model; VAR mv: REAL;
st: Stain; ls: LineStyle; sym: CHAR);
PROCEDURE GetCurveAttrForMV(m: Model; VAR mv: REAL;
VAR st: Stain; VAR ls: LineStyle; VAR sym: CHAR);
PROCEDURE SetDeflItCurveAttrForMV(m: Model; VAR mv: REAL;
st: Stain; ls: LineStyle; sym: CHAR);
PROCEDURE GetDeflItCurveAttrForMV(m: Model; VAR mv: REAL;
VAR st: Stain; VAR ls: LineStyle; VAR sym: CHAR);

PROCEDURE ResetAllCurveAttributes;

PROCEDURE ClearTable;
PROCEDURE ClearGraph;
PROCEDURE DumpGraph;

(* Assignment of predefined values to global default *)
(* values and resetting of all current values *)
(* ----- *)

PROCEDURE SetPredefinitions;
PROCEDURE ResetAll;

(* Preferences and simulation environment modes: *)
(* ----- *)

PROCEDURE SetDocumentRunAlwaysMode( dra: BOOLEAN );
PROCEDURE GetDocumentRunAlwaysMode( VAR dra: BOOLEAN );
PROCEDURE SetAskStashFileTypeMode(asft: BOOLEAN);
PROCEDURE GetAskStashFileTypeMode(VAR asft: BOOLEAN);

PROCEDURE SetRedrawTableAlwaysMode( rta: BOOLEAN );
PROCEDURE GetRedrawTableAlwaysMode( VAR rta: BOOLEAN );
PROCEDURE SetCommonPageUpRows( rows: CARDINAL );
PROCEDURE GetCommonPageUpRows( VAR rows: CARDINAL );

PROCEDURE SetRedrawGraphAlwaysMode( rga: BOOLEAN );
PROCEDURE GetRedrawGraphAlwaysMode( VAR rga: BOOLEAN );
PROCEDURE SetColorVectorGraphSaveMode(cvgs: BOOLEAN);
PROCEDURE GetColorVectorGraphSaveMode(VAR cvgs: BOOLEAN);

(* Customization of keyboard shortcuts for menu commands *)
(* ----- *)

TYPE
MMenuCommand = ( pageSetUpCmd, printGraphCmd, preferencesCmd, customizeCmd,
(*core commands*) setGlobSimParsCmd, setProjDescrCmd, selectStashFileCmd,
resetGlobSimParsCmd, resetProjDescrCmd, resetStashFileCmd,
resetWindowsCmd, resetAllIntegrMethodsCmd, resetAllInitialValuesCmd,
resetAllParametersCmd, resetAllStashFilingCmd, resetAllTabulationCmd,
resetAllGraphingCmd, resetAllScalingCmd, resetAllCurveAttrsCmd,
resetAllCmd, defineSimEnvCmd,
(*core commands*) tileWindowsCmd, stackWindowsCmd, modelsCmd, stateVarsCmd,
(*core commands*) modelParamsCmd, monitorableVarsCmd, tableCmd, clearTableCmd,
(*core commands*) graphCmd, clearGraphCmd,
(*core commands*) startRunCmd, haltOrResumeRunCmd, stopCmd, startExperimentCmd);

PROCEDURE SetMenuCmdAliasChar(cmd: MMenuCommand; alias: CHAR);
PROCEDURE GetMenuCmdAliasChar(cmd: MMenuCommand; VAR alias: CHAR);

PROCEDURE ResetCoreMenuCmdsAliasChars;
PROCEDURE ResetAllMenuCmdsAliasChars;

***** SimMaster *****

(* Running of the standard interactive simulation environment *)
(* ----- *)

PROCEDURE RunSimEnvironment( initSimEnv: PROC );
PROCEDURE SimEnvRunning( progLevel: CARDINAL ):BOOLEAN;
PROCEDURE InstallDefSimEnv( defineSimEnv: PROC );
PROCEDURE ExecuteDefSimEnv;

(* States of the simulation environment *)
(* ----- *)

```

```

TYPE
  MWState = (noSimulation, simulating, pause, noModel);
  MWSubState = (noRun, running, noSubState, stopped);

PROCEDURE GetMWState (VAR s: MWState);
PROCEDURE GetMWSubState(VAR ss: MWSubState);
PROCEDURE InstallStateChangeSignaling( doAtStateChange: PROC );

(* Simulation run conditions *)
(* ----- *)

TYPE TerminateConditionProcedure = PROCEDURE(): BOOLEAN;
   StartConsistencyProcedure = PROCEDURE(): BOOLEAN;

PROCEDURE InstallStartConsistency ( startAllowed: StartConsistencyProcedure );
PROCEDURE InstallTerminateCondition( isAtEnd: TerminateConditionProcedure);

(* Control of elementary and structured simulation runs *)
(* ----- *)

PROCEDURE SimRun;
PROCEDURE PauseRun;
PROCEDURE ResumeRun;
PROCEDURE StopRun;

PROCEDURE InstallExperiment( doExperiment: PROC );
PROCEDURE SimExperiment;
PROCEDURE StopExperiment;

PROCEDURE ExperimentRunning(): BOOLEAN;
PROCEDURE ExperimentAborted(): BOOLEAN;
PROCEDURE CurrentSimNr(): INTEGER;

PROCEDURE CurrentTime(): REAL;
PROCEDURE CurrentStep(): INTEGER;
PROCEDURE LastCoincidenceTime(): REAL;

(===== OPTIONAL CLIENT INTERFACE MODULES =====)
(***** SimEvents *****)

CONST minEventClass=0; maxEventClass=3000; unknownEventClass= maxEventClass; always = MIN(REAL); never = MAX(REAL);

TYPE EventClass=[ minEventClass.. maxEventClass]; Transaction= ADDRESS;
   StateTransitionFunction= PROCEDURE( Transaction); StateTransition= RECORD ec: EventClass; fct: StateTransitionFunction; END;
VAR
  nilTransaction: Transaction; (* read only! *)
  noStateTransition: ARRAY[0..0] OF StateTransition; (* read only! *)
  dummyDEVChg: REAL; (* read only! *) schedulingDone: BOOLEAN;

PROCEDURE EventClassExists( ec: EventClass): BOOLEAN;
PROCEDURE AsTransaction(VAR d: ARRAY OF BYTE): Transaction;

PROCEDURE DeclDEVM (VAR m: Model; initialize, input, output: PROC;
  statetransfct: ARRAY OF StateTransition; terminate, declModelObjects: PROC;
  descriptor, identifier: ARRAY OF CHAR; about: PROC);
PROCEDURE GetDefltdEVm(VAR m: Model; VAR initialize, input, output: PROC;
  VAR statetransfct: ARRAY OF StateTransition; terminate: PROC;
  VAR descriptor, identifier: ARRAY OF CHAR; about: PROC);
PROCEDURE SetDefltdEVm(VAR m: Model; initialize, input, output: PROC;
  statetransfct: ARRAY OF StateTransition; terminate: PROC;
  descriptor, identifier: ARRAY OF CHAR; about: PROC);

PROCEDURE InitEventScheduler;
PROCEDURE ScheduleEvent(ec: EventClass; tau: REAL; alfa: Transaction);
PROCEDURE NextEventAt(): REAL;
PROCEDURE ProbeNextPendingEvent(VAR ec: EventClass; VAR when: REAL; VAR alfa: Transaction);
PROCEDURE GetNextPendingEvent (VAR ec: EventClass; VAR when: REAL; VAR alfa: Transaction);
PROCEDURE PendingEvents(): INTEGER;
PROCEDURE SchedulingOnlyAfter(tmin: REAL);
PROCEDURE DiscardEventsAfter(ec: EventClass; aftert: REAL; alfa: Transaction);
PROCEDURE DiscardEventsBefore(beforet: REAL);

(***** SimObjects *****)

FROM SYSTEM IMPORT ADDRESS; FROM DMStrings IMPORT String; FROM SimBase IMPORT Model;

TYPE RefAttr;

VAR aDetachedRefAttr: RefAttr;

PROCEDURE AttachRefAttrToModel (m: Model; VAR a: RefAttr; val: ADDRESS);
PROCEDURE DetachRefAttrFromModel(m: Model; VAR a: RefAttr );
PROCEDURE AttachRefAttrToObject (m: Model; VAR o: REAL; VAR a: RefAttr; val: ADDRESS);
PROCEDURE DetachRefAttrFromObject(m: Model; VAR o: REAL; VAR a: RefAttr );
PROCEDURE FindModelRefAttr (m: Model; VAR a: RefAttr);
PROCEDURE FindObjectRefAttr(m: Model; VAR o: REAL; VAR a: RefAttr);
PROCEDURE SetRefAttr(a: RefAttr; val: ADDRESS);
PROCEDURE GetRefAttr(a: RefAttr): ADDRESS;

PROCEDURE CurCalcMRefAttr(): ADDRESS;
PROCEDURE CurAboutMRefAttr(): ADDRESS;

PROCEDURE ModelLevel (m: Model):CARDINAL;
PROCEDURE ObjectLevel(m: Model; VAR o: REAL):CARDINAL;

TYPE
  MWObj = (Mo, SV, Pa, MV, AV );
  ExportObjectType = (stateVar, modParam, outAuxVar);
  RealPtr = POINTER TO REAL;
  PtrToClientObject = ADDRESS;
  ObjPtr = POINTER TO ObjectHeader;
  ObjectHeader = RECORD
    ident : String;
    descr : String;
    unit : String;
    varAdr : RealPtr; min, max : REAL;
    nrAttr : INTEGER;
    refAttr : PtrToClientObject;
    chAttr : CHAR;
    kind : MWObj;
    parentM : Model;

```

```

next      : ObjPtr;
prev      : ObjPtr;
END;

PROCEDURE FirstModel(): ObjPtr;
PROCEDURE FirstSV( m: Model ): ObjPtr;
PROCEDURE FirstP ( m: Model ): ObjPtr;
PROCEDURE FirstMW( m: Model ): ObjPtr;
PROCEDURE LastModel(): ObjPtr;
PROCEDURE LastSV( m: Model ): ObjPtr;
PROCEDURE LastP ( m: Model ): ObjPtr;
PROCEDURE LastMW( m: Model ): ObjPtr;

PROCEDURE AllowForRAMESEExport (owner: Model; VAR obj: REAL; ident: ARRAY OF CHAR; eot: ExportObjectType);

(***** SimDeltaCalc *****)

TYPE DeltaVar;      DeltaProc = PROCEDURE ( (*ySim~*)REAL, (*yData*)REAL ): REAL;

VAR defaultDelta: DeltaProc;

PROCEDURE InstallDeltaProc( VAR mvDepVar: REAL; compDelta: DeltaProc );
PROCEDURE InitDeltaStat( VAR mvDepVar: REAL; xSim, ySim: REAL; VAR dv: DeltaVar );
PROCEDURE AccuDelta( dv: DeltaVar; xSim, ySim: REAL );
PROCEDURE GetDeltaStat( VAR mvDepVar: REAL; VAR sumY, sumY2, sumAbsY: REAL; VAR count: INTEGER );
PROCEDURE SetDeltaStat( VAR mvDepVar: REAL; sumY, sumY2, sumAbsY: REAL; count: INTEGER );
PROCEDURE WriteDeltaStatMsg( VAR mvDepVar: REAL );

(***** SimGraphUtils *****)

FROM SimBase  IMPORT MWindowArrangement, Model, Stain, LineStyle, Graphing;
FROM DMWindow IMPORT Color;
FROM Matrices IMPORT Matrix;

TYPE Curve;      VAR nonexistent : Curve; (* read only! *)

PROCEDURE PlaceGraphOnSuperScreen(deflTwa: MWindowArrangement);
PROCEDURE SelectForOutputGraph;
PROCEDURE DefineCurve( VAR c: Curve; st: Stain; style: LineStyle; sym: CHAR );
PROCEDURE RemoveCurve( VAR c: Curve );
PROCEDURE DrawLegend( c: Curve; x, y: INTEGER; comment: ARRAY OF CHAR );
PROCEDURE Plot( c: Curve; newX, newY: REAL );
PROCEDURE Move( c: Curve; newX, newY: REAL );
PROCEDURE PlotSym( x, y: REAL; sym: CHAR );
PROCEDURE PlotCurve( c: Curve; nOfPoints: CARDINAL; x, y: ARRAY OF REAL );
PROCEDURE GraphToWindowPoint( xReal, yReal: REAL; VAR xInt, yInt: INTEGER );
PROCEDURE WindowToGraphPoint( xInt, yInt: INTEGER; VAR xReal, yReal: REAL );

TYPE Abscissa = RECORD isMV: POINTER TO REAL; xMin, xMax: REAL END;

VAR timeIsIndep: REAL;

PROCEDURE InstallGraphClickHandler(gch: PROC);
PROCEDURE MValToPoint(val: REAL; m: Model; VAR mv: REAL; VAR curG: Graphing): INTEGER;
PROCEDURE PointToMVal(xInt, yInt: INTEGER; m: Model; VAR mv: REAL; VAR curG: Graphing): REAL;
PROCEDURE CurrentAbscissa(VAR a: Abscissa);
PROCEDURE TimeIsX(): BOOLEAN;

PROCEDURE StainToColor( stain: Stain; VAR color: Color );
PROCEDURE ColorToStain( color: Color; VAR stain: Stain );

TYPE DisplayTime = ( showAtInit, showAtTerm, noAutoShow );
DispDataProc = PROCEDURE( Model, VAR REAL );
PROCEDURE DeclDispData( mDepVar : Model; VAR mvDepVar : REAL;
mIndepVar : Model; VAR mvIndepVar: REAL;
x, v,
vLo, vUp : ARRAY OF REAL;
n : INTEGER;
withErrBars: BOOLEAN;
dispTime : DisplayTime );
PROCEDURE DisplayDataNow( mDepVar : Model; VAR mvDepVar : REAL );
PROCEDURE DisplayAllDataNow;
PROCEDURE DoForAllDispData( p: DispDataProc );
PROCEDURE RemovedispData( mDepVar : Model; VAR mvDepVar : REAL );

PROCEDURE DeclDispDataM( mDepVar : Model; VAR mvDepVar : REAL;
mIndepVar : Model; VAR mvIndepVar: REAL;
data : Matrix;
withErrBars: BOOLEAN;
dispTime : DisplayTime );
PROCEDURE SetDispDataM( mDepVar: Model; VAR mvDepVar: REAL; data: Matrix );
PROCEDURE GetDispDataM( mDepVar: Model; VAR mvDepVar: REAL; VAR data: Matrix );

(***** SimIntegrate *****)

PROCEDURE Integrate ( m: Model; from, till: REAL);

(===== - E N D - =====)

```

ModelWorks may be freely copied but not for profit!

## Index

### Symbols

«Mini RAMSES Shell» viii, 20, 32, 35

«RAMSES Shell» **20**

### A

application *see stand-alone application*

Ask for stash file type **97**, 104

auxiliary library 89, **90**

auxiliary variable **14**, 69, 71, **131**

AuxVar 131

availability of menu commands **82**

### B

button 84

button palette **84**

### C

calculation order of procedures 49, 50, **66**, 126

callee 71, 72

client *see modeler*

client interface vi, **12**, 53, 89, 90, 122

auxiliary library 90, **123**

mandatory 273

mandatory part 89, 90, **122**, 342

optional client interface **122**

optional part 89, 90

client monitoring **78**

calling sequence 79

termination 79

client procedure 66, 68, **71**, 72

coincidence interval **46**, 47, **50**, 101

coincidence point 46, 47

consistency check *see start consistency check*

continuous time 13, **40**, 41, 43, 44, 45, 46, 47, 135, 224

core menu commands **98**

coupling of models **44**, 47, 48, 209, 224, 243

current value **14**, 22, **58**, **59**, 107, 133

curve attribute **24**, 25, 120, 148, 149

automatic definition 24, 62, 120, 121, 149

in legend 150

customization **98**

### D

declaration of

experiment 143

model 30, 58, 125

model during a simulation 73

model object 29, 34, 71, 128

model parameter 28, 30, 130

monitorable variable 28, 30, 131

state variable 28, 29, 128

table function 323

DeclM 126

DeclMV 131

DeclP 130

DeclSV 129

default value vi, 13, **14**, 59, 105, 133

Derivative 128

derivative of state variable 13, 14, 29, **44**, 68, 69, 70, 127, 129

derivative vector **41**

DESS **40**

DEVS 41, 42, 162, 273

"Dialog Machine" vi, **17**, 54, **90**, 91, 124

difference equation 12, 13, **40**, **41**

difference equation system v

differential equation 12, 13, 26, **40**, 127

differential equation system v

discrete event 12, 13, **41**, **42**, 126, 162

external **42**

internal **42**

discrete event formalism v

discrete event model viii

discrete time 13, **41**, 43, 45, 46, 47, 126, 224

discrete time step 41, **42**, **46**

## E

event class 41, **42**

event input 41

event output 40, 41, 46  
external 45

event scheduling viii, 42  
by DESS **40**  
by DEVS **41**  
by SQM **41**

existence of model objects 71

experiment 195, *see structured simulation*

external event 45, 46

## F

first order difference equation 14

first order differential equation 14

full reset **62**

function vector **41**

## G

global parameters **53**

global simulation parameter 62, 101, 133, 134

graph 36, 76, **78**, 79, 96, 108, 118, 119, 123, 132, 148, 150

graphing 119

## H

hardware requirements vi

hierarchical system 43, 49

## I

independent variable **46**, 68, 78, 117, 118, 134, 136

initial events **41**

initial value **13**, 14, 15, 40, 41, 59, 70, 105, 113, 127, 129

input 40, 41, 42, 49, 68, 69, 127

input variable 131

installing procedure 71

instantaneous state transition function  
13, **40**, **41**, 42, 162

integration method 26, 51, **59**, 68, 112, 126

integration step 26, **50**, **59**, **69**

interactive simulation environment 106

internal event 46

InVar 131

IO-window **21**, **82**, 107, 109, 144, 145

IO-window default action **110**, 112, 114, 116, 119

## K

keyboard shortcut **94**, 98, 109, 110, 111, 112, 114, 116, 117, 118, 119, 121

## L

## M

MacMETH vi

mixed model 14, 43, **46**, **47**, 224

model 13, **40-43**, 59, 110

model attributes 139

model base **53**, 73

model definition program 12, **15**, 16, 28, 32, 87-88, **90**, 94, 124

model development **12**, 32-87

model object **13**, 59, 70, 71, 82

model parameter **13**, 14, 41, 59, 70, 114

model validation 278

modeller **12**, 71

ModelWorks window 146

modifying current value

automatic curve attribute definition  
121

curve attribute 24, 120

discrete time step 101

global simulation parameter 101, 135, 136

graphing 118, 119

initial value 23, 113, 114, 139



- integration method 26, 112, 139
- integration step 26, 101
- model parameter 23, 114, 116, 139
  - run time change 26, 30, 81, 108, 115, 130
- monitorable variable 23, 24-25, 116, 139
- monitoring interval 25, 101, 136
- project description 136
- recording flags 103, 136
- scaling 23, 119, 139
- simulation start time 101, 136
- simulation stop time 101, 136
- stash filing 117
- table function 139
- tabulation 118
- user defined curve attributes 121
- modifying default value
  - global simulation parameters 135
  - initial value 138
  - integration method 34, 138
  - integration step 136
  - model 138
  - model parameter 138
    - run time change 138
  - monitorable variables 138
  - monitoring interval 136
  - project description 135
  - recording flags 135
  - scaling 138
  - simulation start time 136
  - simulation stop time 136
  - state variable 138
- Modula-2 vi, 16
- modular modeling 88, 209
- module structure of ModelWorks 90
- monitorable variable **13**, 14, 23, 24, 59, 70, 76, 116
- monitoring 23, 68, **76-79**, 116-121, 146-151
- monitoring interval 59, **70, 76**, 101
- monitoring time **76**
- Monte-Carlo simulation 205
- MS DOS vi, 272
- N**
- new state 68
- new value of state variable 13, 14, 29, **45**, 70, 127, 129
- NewState 128
- No run *see* *substate of simulation environment*
- No simulation *see* *state of simulation environment*
- O**
- object *see* *model object*
- object selection *see* *selection of object*
- operand 84
- operator 84
- output 40, 41, 42, 49, 68, 69, 127
- output variable 131
- output-input coupling **44, 45**
- OutVar 131
- P**
- page up **78**, 97, 150
- parallel model v, 48, 243
- parameter *see* *model parameter*
- parameter identification 89, 278
- Pause *see* *state of simulation environment*
- performance index 278
- personal computer vi
- predefined defaults 62
- predefinitions **62**
- preferences *see* *simulation environment mode*
- program control **73**
- program stack **80**, 94
  - conditional reset 82
  - global simulation parameters 82
  - quitting subprogram level 81
- pseudo random number generator 195
- Q**
- quitting of a program level **80**
- R**
- RAMSES v

availability 272  
 recommended preferences 97  
 removing of  
     model 58, 125, 141  
     model object 58, 141  
 reset **14**, 59, **60**, 62, 101, 105  
 run *see simulation run*  
 run time change *see modifying  
 current value: model parameter*  
 run-time system **54**  
 Running *see substate of simulation  
 environment*  
 RunSimEnvironment 94

## S

sample model **154**  
 scaling **13**, 23, 78  
 Scope All 84  
 selection of model **83**  
 selection of object 22, 83  
 selection scope 83  
 selectionscope **83**  
 sensitivity analysis 89-183, 278  
 sequential machine **41**, 42  
 SimDeltaCalc 278  
 SimEvents 273  
 SimGraphUtils 281  
 SimIntegrate 288  
 SimObjects 290  
 Simulating *see state of simulation  
 environment*  
 simulation environment **15**, 16, 31, 53  
 simulation environment mode 96, 97,  
 104, **148**, 150  
 simulation run 22, 53, 65, **66**, **67**, 108,  
 141  
 simulation session 53, **63**  
 simulation time 22, 31, **40**, 69, 101, 134,  
 135  
 simulationist **12**  
 solving models simultaneously **80**

source *see model definition  
 program*  
 SQM *see sequential machine*  
 standard interactive simulation  
     environment 124  
 standard user interface **54**, **79**  
     states 82  
 start consistency check **66**, 142  
 start time 46  
 stash file 26, **76**, 77, 96, 103, 104, 105,  
 134, 148, 150  
     recording flags 103, 134, 136  
     renaming 75  
     switching 75  
 state discontinuity **42**  
 state of simulation environment 55, 56,  
     **81**, 82, 143  
     EmptySimEnvironment 54  
     No model 55, 81, 82  
     No simulation 54, 55, 56, 81, 82  
     Pause 26, 54, 55, 56, 81, 82  
     Simulating 54, 55, 56, 81, 82, 108  
 state variable **13**, 14, 68, 69, 70, 113  
 state vector **41**  
 StateVar 128  
 stochastic simulation 195  
 structured model 40, **43**, 48, 51, 88, 224  
 structured simulation 56, 65, 67, **88**, 89,  
 109, 141, 143, *see experiment*  
 structured simulation run 195  
 submodel 40, **43**, 49, 50, 51  
 subprogram 94  
 subprogram level **80**  
 subrun **75**  
 subrun break **75**  
 subsequent monitoring 146  
 substate of simulation environment **55**,  
     **56**, 143  
     No run 56  
     Running 56  
 system  
     continuous time 40  
     discrete event 40  
     discrete time 40

system specification  
  continuous time **40**  
  discrete event **41**, 162  
  discrete time **41**

## T

table 25, **76**, **78**, 107, 150  
table function 176, 320, 323, **326**  
table function editor 320, 321  
terminate condition **66**, 142  
time *see continuous or discrete time*  
transaction **41**  
TYPE ExtrapolMode = ( lastSlope,  
  horizontally ) **326**

## U

user *see simulationist*  
user interface **12**, 53, 94  
user interface customization **84**  
  additional menus 85  
  disable functions 84  
  initialization (InstallDefSimEnv) 85  
  nonstandard user  
    interface(MySimEnv) 85  
  override predefined settings 84

## V

versions of ModelWorks of  
  ModelWorks 272

## W

work object 20

## X

## Y

## Z



**BERICHTE DER FACHGRUPPE SYSTEMÖKOLOGIE**  
**SYSTEMS ECOLOGY REPORTS**  
**ETH ZÜRICH**

---

Nr./No.

- 1 FISCHLIN, A., BLANKE, T., GYALISTRAS, D., BALTENSWEILER, M., NEMECEK, T., ROTH, O. & ULRICH, M. (1991, erw. und korr. Aufl. 1993): Unterrichtsprogramm "Weltmodell2"
- 2 FISCHLIN, A. & ULRICH, M. (1990): Unterrichtsprogramm "Stabilität"
- 3 FISCHLIN, A. & ULRICH, M. (1990): Unterrichtsprogramm "Drosophila"
- 4 ROTH, O. (1990): Maisreife - das Konzept der physiologischen Zeit
- 5 FISCHLIN, A., ROTH, O., BLANKE, T., BUGMANN, H., GYALISTRAS, D. & THOMMEN, F. (1990): Fallstudie interdisziplinäre Modellierung eines terrestrischen Ökosystems unter Einfluss des Treibhauseffektes
- 6 FISCHLIN, A. (1990): On Daisyworlds: The Reconstruction of a Model on the Gaia Hypothesis
- 7 \* GYALISTRAS, D. (1990): Implementing a One-Dimensional Energy Balance Climatic Model on a Microcomputer (*out of print*)
- 8 \* FISCHLIN, A., & ROTH, O., GYALISTRAS, D., ULRICH, M. UND NEMECEK, T. (1990): ModelWorks - An Interactive Simulation Environment for Personal Computers and Workstations (*out of print*] for new edition see title 14)
- 9 FISCHLIN, A. (1990): Interactive Modeling and Simulation of Environmental Systems on Workstations
- 10 ROTH, O., DERRON, J., FISCHLIN, A., NEMECEK, T. & ULRICH, M. (1992): Implementation and Parameter Adaptation of a Potato Crop Simulation Model Combined with a Soil Water Subsystem
- 11 \* NEMECEK, T., FISCHLIN, A., ROTH, O. & DERRON, J. (1993): Quantifying Behaviour Sequences of Winged Aphids on Potato Plants for Virus Epidemic Models
- 12 FISCHLIN, A. (1991): Modellierung und Computersimulationen in den Umweltnaturwissenschaften
- 13 FISCHLIN, A. & BUGMANN, H. (1992): Think Globally, Act Locally! A Small Country Case Study in Reducing Net CO<sub>2</sub> Emissions by Carbon Fixation Policies
- 14 FISCHLIN, A., GYALISTRAS, D., ROTH, O., ULRICH, M., THÖNY, J., NEMECEK, T., BUGMANN, H. & THOMMEN, F. (1994): ModelWorks 2.2 – An Interactive Simulation Environment for Personal Computers and Workstations
- 15 FISCHLIN, A., BUGMANN, H. & GYALISTRAS, D. (1992): Sensitivity of a Forest Ecosystem Model to Climate Parametrization Schemes
- 16 FISCHLIN, A. & BUGMANN, H. (1993): Comparing the Behaviour of Mountainous Forest Succession Models in a Changing Climate
- 17 GYALISTRAS, D., STORCH, H. v., FISCHLIN, A., BENISTON, M. (1994): Linking GCM-Simulated Climatic Changes to Ecosystem Models: Case Studies of Statistical Down-scaling in the Alps
- 18 NEMECEK, T., FISCHLIN, A., DERRON, J. & ROTH, O. (1993): Distance and Direction of Trivial Flights of Aphids in a Potato Field
- 19 PERRUCHOUD, D. & FISCHLIN, A. (1994): The Response of the Carbon Cycle in Undisturbed Forest Ecosystems to Climate Change: A Review of Plant–Soil Models
- 20 THÖNY, J. (1994): Practical considerations on portable Modula 2 code
- 21 THÖNY, J., FISCHLIN, A. & GYALISTRAS, D. (1994): Introducing RASS - The RAMSES Simulation Server

---

\* Out of print

- 22 GYALISTRAS, D. & FISCHLIN, A. (1996): Derivation of climate change scenarios for mountainous ecosystems: A GCM-based method and the case study of Valais, Switzerland
- 23 LÖFFLER, T.J. (1996): How To Write Fast Programs
- 24 LÖFFLER, T.J., FISCHLIN, A., LISCHKE, H. & ULRICH, M. (1996): Benchmark Experiments on Workstations
- 25 FISCHLIN, A., LISCHKE, H. & BUGMANN, H. (1995): The Fate of Forests In a Changing Climate: Model Validation and Simulation Results From the Alps
- 26 LISCHKE, H., LÖFFLER, T.J., FISCHLIN, A. (1996): Calculating temperature dependence over long time periods: Derivation of methods
- 27 LISCHKE, H., LÖFFLER, T.J., FISCHLIN, A. (1996): Calculating temperature dependence over long time periods: A comparison of methods
- 28 LISCHKE, H., LÖFFLER, T.J., FISCHLIN, A. (1996): Aggregation of Individual Trees and Patches in Forest Succession Models: Capturing Variability with Height Structured Random Dispersions
- 29 FISCHLIN, A., BUCHTER, B., MATILE, L., AMMON, K., HEPPELLE, E., LEIFELD, J. & FUHRER, J. (2003): Bestandesaufnahme zum Thema Senken in der Schweiz. Verfasst im Auftrag des BUWAL
- 30 KELLER, D., 2003. *Introduction to the Dialog Machine*, 2<sup>nd</sup> ed. Price,B (editor of 2<sup>nd</sup> ed)

Erhältlich bei / Download from

<http://www.ito.umnw.ethz.ch/SysEcol/Reports.html>

Diese Berichte können in gedruckter Form auch bei folgender Adresse zum Selbstkostenpreis bezogen werden /  
Order any of the listed reports against printing costs and minimal handling charge from the following address:

SYSTEMS ECOLOGY ETHZ, INSTITUTE OF TERRESTRIAL ECOLOGY GRABENSTRASSE 3, CH-8952 SCHLIEREN/ZURICH, SWITZERLAND
--