Die Entwicklung interaktiver Modellierungs- und Simulationssoftware mit Modula-2

Klara Vancso[1], Andreas Fischlin[1,2] und Walter Schaufelberger[1]

[1] Projekt-Zentrum IDA [Informatik dient allen], Eidgenössische Technische Hochschule Zürich
[2] Institut für Phytomedizin, Eidgenössische Technische Hochschule Zürich

Zusammenfassung

Sinnvoller Einsatz interaktiver Modellierungs- und Simulationssoftware auf Kleinrechnern und Arbeitsplatzrechnern verlangt ein Überdenken der Arbeitsweisen und Architekturen herkömmlicher Simulationssoftware aus zwei Gründen: heutigen Anforderungen an die Mensch-Maschine Benutzerschnittstelle und den Fortschritten in der Modellierungstheorie. Es werden der Entwurf einer interaktiven Modellierungs- und Simulationssoftware für Arbeitsplatzrechner, deren Architekturen und Implementierungen beschrieben und die Tauglichkeit von Modula-2 zur Entwicklung, die sowohl modernen Ansprüchen in Bezug auf Mensch-Maschine Benutzerschnittstelle (hochauflösende Bitmap-Graphik, Menü- und Fenstertechnik, Zeigegerät z.B. Maus) und der Modellierungstheorie genügen kann, diskutiert.

Abstract

The Development of Interactive Modelling and Simulation Software With Modula-2: The sensible usage of interactive modelling and simulation software on personal computers and modern working stations requires modifications of traditional approaches and conventional software architectures mainly for two reasons: todays requirements for a modern man-machine interface and the progress made in the field of modelling theory. The paper describes the architecture for a new kind of interactive modelling and simulation software designed for modern working stations, which supports not only a graphical oriented user interface (bit-map graphics, menu and window technique, pointing device such as mouse) but is also based on modelling theory. It discusses the suitability of Modula-2 for such a task.

1. Einleitung

Die interaktive Modellierung und Simulation hat in den letzten Jahren, insbesondere durch die zunehmende Verbreitung von Kleinrechnern und Arbeitsplatzrechnern, an Bedeutung zugenommen. Ein sinnvoller Einsatz derartiger Informatikwerkzeuge verlangt jedoch ein Überdenken der Arbeitsweisen und Architekturen herkömmlicher Simulationssoftware. Neben den vielen Gründen, die anderweitig schon eingehend diskutiert worden sind (CELLIER 1979; CELLIER & FISCHLIN 1980), ist diese Arbeit vor allem aus den folgenden zwei Beweggründen in Angriff genommen worden: Erstens kommt der Mensch-Maschine Benutzerschnittstelle beim interaktiven Softwareeinsatz auf Arbeitsplatzrechnern eine ausschlaggebende Bedeutung zu; beispielsweise entscheidet sie im heutigen Markt oft schon allein über die Verwendbarkeit fast aller Software, weitgehendst unabhängig von der zugrundeliegenden Funktionalität der Programme. Zweitens sind in der Modellierungstheorie Fortschritte erzielt worden, die in den heute meistens zum Einsatz gelangenden Simulationsprogrammen noch kaum berücksichtigt werden. Was sich demzufolge aufdrängt, ist eine Neuentwicklung von Softwarearchitekturen, die interaktives Modellieren und Simulieren, gestützt auf die neueren Modellierungstheorien, ermöglichen.

Modula-2 ist eine moderne höhere Programmiersprache, die die Entwicklung komplexer Softwaresysteme gut unterstützt und für die viele effiziente und kostengünstige Implementierungen auf allen gängigen Arbeitsplatzrechnern zur Verfügung stehen. Die Tauglichkeit von Modula-2 zur Entwicklung benutzerfreundlicher Mensch-Maschine Schnittstellen auf modernen Arbeitsplatzrechnern mit hochauflösendem Graphikbildschirm, Fenstertechnik, und Zeigegerät (Maus) ist verschiedentlich gezeigt worden (WIRTH 1985; FISCHLIN 1986). Deren Tauglichkeit für eine interaktive Modellierungs- und Simulationssoftware für Arbeitsplatzrechner, die ebenfalls modernen Ansprüchen in Bezug auf die Modellierungstheorie genügen kann, ist hingegen unseres Wissens noch nie untersucht worden.

In der vorliegenden Arbeit werden über erste Ergebnisse aus unseren Forschungs- und Entwicklungsarbeiten berichtet. Sie beschäftigt sich insbesondere mit dem Teilaspekt modelltheoretisch ausgerichteter Simulationssoftware. Zuerst werden nur ganz knapp das allgemeine Anforderungsprofil und eine Übersicht über die Grobstruktur einer derartigen Software, dann werden die Realisierungen der Modelltheoretischen Definitionen von Systemen mit Modula-2 vorgestellt. Abschliessend diskutieren wir die Tauglichkeit von Modula-2 als Implementierungssprache für derartige Softwareprojekte, die sich mit wenigen, jedoch nicht unwichtigen Ausnahmen bestätigen liess.

2. Anforderungsprofil und Softwarearchitektur

Interaktives Modellieren und Simulieren ermöglicht eine Arbeitsteilung zwischen Mensch und Rechner, die für das Arbeiten mit schlechtdefinierten Systemen besonders attraktiv ist. Allerdings entstehen dadurch besonders hohe Anforderungen an die Benutzeroberfläche der verwendeten Software. Eine gut, d.h. effizient und sachgerecht gestaltete Benutzeroberfläche kann die Funktionalität eines interaktiven Programmes stark beeinflussen. Es können völlig neuartige Einsatzmöglichkeiten erschlossen werden, die ansonsten unrealisierbar blieben: Z.B. ist die explorative Analyse schlecht-definierter Modelle schon derart komplex, dass die interaktive Simulationssoftware bloss dann verwendbar ist, wenn es der Benutzeroberfläche gelingt, die Verwaltung des komplexen Geschehens auf ein erträgliches Mass zurückzuschrauben. Ansonsten droht die Gefahr, dass schon allein die Bedienung der Software alle intellektuellen Kräfte aufbraucht. Erfüllt die zur Verfügung stehende Software die Anforderung einer benutzerfreundlichen Oberfläche nicht, so ist Stapelbetrieb vorteilhafter, obwohl gerade in diesem Anwendungsbereich durch die interaktive Arbeitsweise das Abwechseln zwischen typischen Computeraufgaben (numerische Berechnungen und deren graphischer Darstellung) und qualitativ orientierten Überlegungen (Mensch) besonders fruchtbar wäre.

Da sich die folgende Arbeit vor allem mit der Entwicklung einer modelltheoretisch abgestützten Simulationssoftware beschäftigt, beschränkt sich die folgende Beschreibung des Anforderungsprofils auf den Teilaspekt Simulation. Es erweist sich als günstig, die interaktive Simulation in sogenannte Spezifikations-, eigentlichen Simulations- und Resultatanalysesessionen aufzuteilen. Im Verlaufe dieser Tätigkeiten fallen die verschiedensten Aufgaben an: in der Spezifikationssession einer Simulationssession die Festlegung verschiedenster Parameter, z.B. Modellwahl und Festlegung der Integrationsverfahren, Simulationsdauer, Modellparameter-, Anfangs- und Randwerte, sowie die Auswahl der Grössen, die zur Überwachung der Simulationssession beigezogen werden sollen. Während dem eigentlichen Simulationsgeschehen, muss neben dem einfachen Unterbruch noch die interaktive Abänderung der Simulationsparameter, natürlich vorbehältlich Nichtverletzung der während der Modellierungssession festgelegten Modellkonsistenz, unterstützt werden. Da im allgemeinen während der Simulationssession nicht alle berechneten Ergebnisse dargestellt

werden können, ist eine anschliessende Resultatanalysesession vorgesehen, während der interaktiv die Ausgabesteuerung für die Postauswertung in Form von zusätzlichen Graphiken, Tabellen, oder Speicherungen auf Massenspeichermedien, der Vergleich von simulierten Zeitreihen mit Messungen usw. vorgenommen werden kann.

Die Verwendung modelltheoretischer Konzepte erweist sich für eine strukturierte Modellierung als besonders förderlich. Ein hierarchischer, modularer Aufbau der Modelle unterstützt die Modellierung komplexer Systeme durch Aufteilung in besser beherrschbare Teilschritte und durch die Auswechselbarkeit einzelner Komponenten zwecks Untersuchung ihrer Relevanz. Eine Abbildung dieser Konzepte auf die Mächtigkeit der Formalismen wie sie durch moderne höhere Programmiersprachen wie Modula-2 angeboten werden, ermöglicht die Modellierung von nichtklassisch mathematischen Formalismen in Form rein algorithmischer Beschreibungen, z.B. Rekursion (FISCHLIN 1982). Schliesslich unterstützt die Modularisierung das Mischen von Modellexperimenten mit realen Experimenten, das Mischen von Systemen verschiedener Klassen, z.B. zeitkontinuierlichen mit zeitdiskreten Systemen, und die Erweiterbarkeit wesentlich.

Der entwickelte Prototyp der Simulationssoftware SAM (Simulation And Modelling) lässt es zu, dass die Modellstruktur direkt auf Strukturen in Modula-2 abgebildet werden. Das bedeutet beispielsweise, dass ein System oder Untersystem auf ein Modula-2 Modul abgebildet wird. Objekte die das System spezifizieren werden von diesen Modulen exportiert und der allgemeinen Simulationssoftware zwecks Simulationsspezifikation, Simulationsdurchführung oder Resultatanalysesession zur Verfügung gestellt. Für jede der drei wichtigen Sytemklassen sequentielle Maschine, Differentialgleichungsformalismus (DESS) und ereignisorientierter Systemformalismus (DEVS) (s.a. Anhang) sind Prototyen für den Defintionsteil entsprechender Module und der darin benötigten Basistypen durch SAM angeboten. Die sich daraus ergebende Gesamtstruktur der Software zeigt, dass der Modellierer durch das Einfügen von exportierenden Modulen in die SAM-Software nach erfolgter Compilation eine Simulation durchführen kann (Fig. 1).

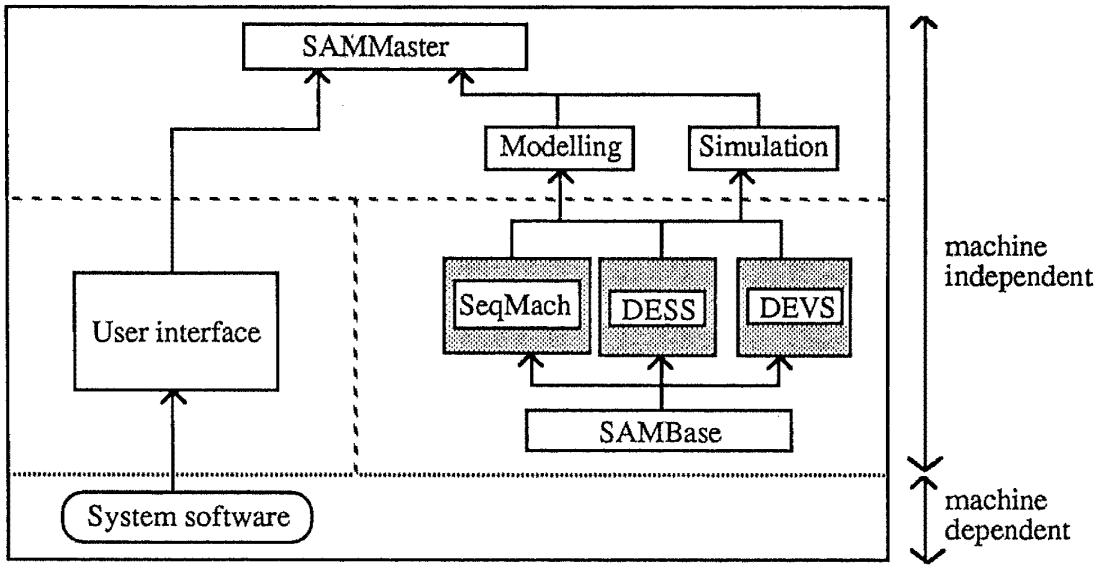


Fig. 1: Architektur der Simulationssoftware SAM. Die Komponenten entsprechen einzelnen Modula-2 Modulen oder Gruppen von Modulen. (Pfeile bedeuten Importe, Klientteile von SAM durch Musterung hervorgehoben). Die innere Struktur von SAM ist nicht detailliert dargestellt.

Dadurch zerfällt die Gesamtsoftware in einen durch den Modellierer zu erstellenden sog. Klientteil und den durch SAM vorgebenen Dienstteil. Der Dienstteil enthält alle Software um Spezifikationssessionen, Simulationssessionen und Resultatanalysesessionen durchführen zu können, insbesondere auch die Laufzeitbibliothek und Graphikbibliothek, wie sie auch in herkömmlicher Simulationssoftware anzutreffen ist.

Was hingegen fehlt, ist eine eigentliche Simulationssprache, dank derer z.B. eine automatische Sortierung von Modellgleichungen vorgenommen werden könnte. Dies wird jedoch durch die Mächtigkeit der verwendeten Programmiersprache mehr als wettgemacht. Im folgenden werden nun die Schnittstellen zwischen der Klient- und der Dienstsoftware beschrieben, in denen sich die die gefundene Lösung zur Implementierung der modelltheoretischen Konzepte besonders klar wiederspiegelt.

3. Realisierung der system- und modellierungstheoretischen Konzepte in Modula-2

Eine kurze Zusammenfassung der modellierungs- und systemtheoretischen Konzepte, die die Basis für die Simulationssoftware abgeben, ist im Anhang dargestellt. Die Übertragung modellierungstheoretischer Konzepte in Modula-2 Datenstrukturen und Programmstrukturen, stützt sich stark auf die folgenden Eigenschaften von Modula-2: den opaken Datentyp, den Typ Prozedur (Prozedurvariablen) und offene formale Feldparameter (open array parameters). Im Folgenden zeigen wir, wie die drei grundsätzlichen Formalismen, sequentielle Maschine, DESS, DEVS (s. Anhang), in Modula-2 definiert werden können.

3.1 Definition der sequentiellen Maschine

Eine sequentielle Maschine wird in der I/O-Systemebene und in der Strukturierten-I/O-Systemebene in einem sog. Definitionsmodul definiert. Der Unterschied zwischen der Systemspezifikation in der dritten und der vierten Ebene (s. Anhang) liegt darin, dass in der vierten Ebene die Mengen und Funktionen strukturiert sind, d.h. sie sind repräsentiert als ein kartesisches Produkt mehrerer elementarer Mengen und Funktionen.

```
TYPE
     Input; State; Output;                                          (* X, Q, Y *)
     SingleStepFunction = PROCEDURE (Input, State):State;           (* δM *)
     OutputFunction = PROCEDURE (State):Output;                     (* λ *)
     SeqMach = RECORD
                    Delta: SingleStepFunction;
                    Lambda: OutputFunction;
               END;
```

Die `Input,State,Output` sind lediglich als opake Datentypen vereinbart, d.h., in dem Definitionsmodul sind nur ihre Bezeichner vorgegeben. Das hat den Vorteil dass der Modellierer diese Typen in dem dazugehörigen Implementierungsteil des Moduls als beliebige Datenstrukturen definieren kann. Auf diese Weise können grundsätzlich unterschiedliche Systeme gleichartig modelliert werden. Der Typ `SeqMach` ist durch zwei Funktionsprozeduren gegeben, bei welchen die Typen der formalen Parameter durch das oben angeführte Definitionsmodul vorgeschrieben sind. Die Tatsache, dass in Modula-2 auch strukturierte Datentypen vereinbart werden können, ermöglicht es, dass die gleichen Definitionen für Systemdefinitionen sowohl für die dritte wie auch die vierte Ebene gleichzeitig verwendet werden können. Als Beispiele zeigen wir die Definitionen zweier Systeme: Das erste ist ein lineares, zeitdiskretes System und das zweite ist eine einfache logische Schaltung. Die Programmtexte enthalten die Systembeschreibungen in der Form, wie sie von dem Modellierer in einem Implementierungsmodul programmiert werden sollen.

Beispiel 1: Ein lineares, diskretes System als Sequentielle Maschine:

$$x_{k+1} = A \cdot x_k + B_k \cdot u_k \qquad y_k = C \cdot x_k$$

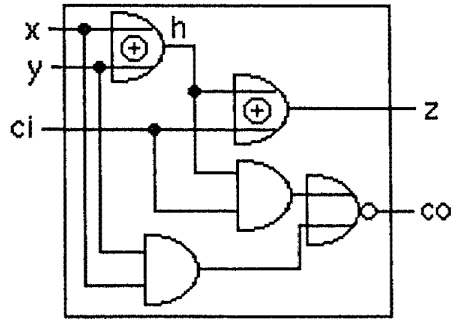
$$x = \begin{matrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -2 & -2 & -4 \end{matrix} \qquad B = \begin{matrix} 0 \\ 0 \\ 1 \end{matrix} \qquad C = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

```
IMPLEMENTATION MODULE SeqMach;

  FROM LinAlgebra IMPORT TypeOfElement, Matrix, FillMatrixI, MulMatrix, AddMatrix;
  TYPE
       Input = POINTER TO InputItem;
       State = POINTER TO StateItem;
       Output = POINTER TO OutputItem;
       InputItem = Matrix;
       StateItem = Matrix;
       OutputItem = Matrix;
```

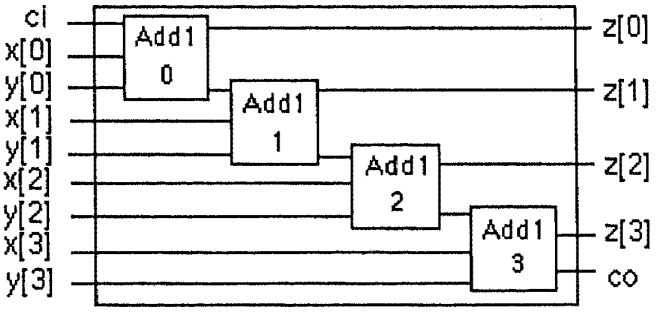
```
VAR
    LinearDiscreteSystem: SeqMach;

    A, B, C, u, x, Ax, Bu, Cx: Matrix;
    dataA, dataC: ARRAY [1..9] OF INTEGER;
    dataB, dataD: ARRAY [1..3] OF INTEGER;

PROCEDURE LinearSingleStepFunction(i:Input; s:State):State;     (* x:=Ax+Bu *)
  VAR newState: State;
BEGIN
  u:=i^; x:=s^;
  MulMatrix(A,x,Ax);  MulMatrix(B,u,Bu); AddMatrix(Ax, Bu, x);
  newState^:=x;
  RETURN newState;
END LinearSingleStepFunction;

PROCEDURE LinearOutputFunction(s:State):Output;      (* y:=Cx *)
  VAR y: Output;
BEGIN
  x:=s^; MulMatrix(C,x,Cx);
  y^:=Cx; RETURN y;
END LinearOutputFunction;

PROCEDURE CurrentSeqMach():SeqMach;
BEGIN
  LinearDiscreteSystem.Delta:=LinearSingleStepFunction;
  LinearDiscreteSystem.Lambda:=LinearOutputFunction;
  RETURN LinearDiscreteSystem;
END CurrentSeqMach;

BEGIN
    dataA[1]:=0;      dataA[2]:=1;    dataA[3]:=0;
    dataA[4]:=0;      dataA[5]:=0;    dataA[6]:=1;
    dataA[7]:=-2;     dataA[8]:=-3;   dataA[9]:=-4;
    dataB[1]:=0;      dataB[2]:=0;    dataB[3]:=1;
    FillMatrixI(A, 3, 3, integer, dataA);
    FillMatrixI(B, 3, 1, integer, dataB);
    dataC[1]:=1;      dataC[2]:=0;    dataC[3]:=0;
    dataC[4]:=0;      dataC[5]:=1;    dataC[6]:=0;
    dataC[7]:=0;      dataC[8]:=0;    dataC[9]:=1;
    FillMatrixI(C, 3, 3, integer, dataC);
END SeqMach.
```

Beispiel 2: Ein Schaltungskreis als sequentielle Maschine:

Add1:



Add4:



```
IMPLEMENTATION MODULE SeqMach;

  FROM BooleAlgebra IMPORT XOR;

  TYPE
    Input = POINTER TO InputItem;
    State = POINTER TO StateItem;
    Output = POINTER TO OutputItem;
    InputItem = RECORD
                  x: ARRAY [0..3] OF BOOLEAN;
                  y: ARRAY [0..3] OF BOOLEAN;
                  ci: BOOLEAN;
                END;
    StateItem = RECORD
                  z: ARRAY [0..3] OF BOOLEAN;
                  co: BOOLEAN;
                END;
    OutputItem = RECORD
                  z: ARRAY [0..3] OF BOOLEAN;
                  co: BOOLEAN;
                END;
  VAR
    CircuitAdd4: SeqMach;

  PROCEDURE Add4SingleStepFunction(i:Input; s:State):State;
    VAR
      newState: State; k: CARDINAL;
      c: ARRAY [0..4] OF BOOLEAN;

    PROCEDURE Add1(x, y, ci: BOOLEAN; VAR z,co: BOOLEAN);
      VAR h: BOOLEAN;
    BEGIN
      h:=XOR(x,y); z:=XOR(h,ci);
      co:=NOT ((x AND y) OR (h AND ci));
    END Add1;

  BEGIN (* Add4 *)
    c[0]:=i^.ci;
    FOR k:=0 TO 3 DO
      Add1(i^.x[k], i^.y[k], c[k], newState^.z[k], c[k+1]);
    END;
    newState^.co:=c[4];
    RETURN newState;
  END Add4SingleStepFunction;

  PROCEDURE Add4OutputFunction(s:State):Output;
    VAR output: Output;
  BEGIN
    output:=VAL(Output, s);
  END Add4OutputFunction;

  PROCEDURE CurrentSeqMach():SeqMach;
  BEGIN
    CircuitAdd4.Delta:=Add4SingleStepFunction;
    CircuitAdd4.Lambda:=Add4OutputFunction;
    RETURN CircuitAdd4;
  END CurrentSeqMach;

END SeqMach.
```

Um eine sequentielle Maschine in der IORO und IOFO Ebene definieren zu können müssen die Konzepte der Eingangs- und Ausgangssegmente und die sog. Eingangs- und Ausgangssegmentpaare eingeführt werden. Definiert man die Eingangs- und Ausgangs-segmente als lineare Liste, so wird das Eingangs- Ausgangssegmentpaar als strukturierter Datentyp folgendermassen realisiert:

```
TYPE
    InputSegment = POINTER TO Inputs;
    Inputs = RECORD
               i: Input;
              next: InputSegment;
             END;
    OutputSegment = POINTER TO Outputs;
    Outputs = RECORD
                o: Output;
               next: OutputSegment;
              END;
    IOSegmentPair = RECORD
                      is: InputSegment;
                      os: OutputSegment;
                    END;
```

Die Sequentielle Maschine in der Ebene 1 und 2 (IORO und IOFO) können als je eine Prozedur - Prozedur "IORelation" und "IOFunction" - realisiert werden. Die Prozedur "IORelation" gibt die Menge der Eingangs- und Ausgangssegmentpaare zurück. Weil die Kardinalität dieser Menge vorher nich bekannt ist, kann der Ausgangsparameter als ein offener Feldparameter vereinbart werden:

```
PROCEDURE IORelation(VAR R: ARRAY OF IOSegmentPair);
```

Die Prozedur IOFunction ist eine Funktionsprozedur, die ein Eingangssegment in ein Ausgangssegment abbildet.

```
PROCEDURE IOFunction(is: InputSegment):OutputSegment;
```

3.2 Definition des Differentialgleichungsformalismus

Die Typen für die Eingänge, Ausgänge und Zustände bei dem Differentialgleichungsformalismus (DESS) sind einheitlicher als die für die sequentielle Maschine. Die Zeit ist kontinuierlich, die Eingänge sind Vektoren von Zeitfunktionen. Der Zustand und die Werte der Ausgänge sind reelle Vektoren. Die Ableitungsfunktion und die Ausgangsfunktion lassen sich als Prozedurtypen realisieren:

```
TYPE
  Time = REAL;
  TimeInterval =  RECORD
                    initialTime: Time;
                    endTime: Time;
                  END;
  TimeFunction = PROCEDURE(Time):REAL;
  State = ARRAY [1..maxStateDim] OF REAL;
  Input = ARRAY [1..maxInputDim] OF TimeFunction;
  OutputValue = ARRAY [1..maxOutputDim] OF REAL;

  RateOfChangeFunction=PROCEDURE (Time,
                                  ARRAY OF REAL,          (*state*)
                                  ARRAY OF TimeFunction,  (*input*)
                                  VAR ARRAY OF REAL);     (*state*)
  OutputFunction = PROCEDURE (Time,
                              ARRAY OF REAL,             (*state*)
                              ARRAY OF TimeFunction,     (*input*)
                              VAR ARRAY OF REAL);        (*output*)
  DESS = RECORD
           f: RateOfChangeFunction;
           Lambda: OutputFunction;
         END;
```

Das Eingangssegment ist durch ein Zeitintervall und durch den Vektor der Eingangsfunktionen definiert. Im Ausgangssegment kann die Ausgangsfunktion nicht in analytischer Form auftreten, weil der Differentialgleichungsformalismus in der Regel numerisch gelöst wird. Ein derartiges Ausgangssegment ist eine Serie von Zeitpunkten mit den Werten der Ausgangsfunktion. Die Definition von Differentialgleichungssy-

stemen in der Ebene 1 und 2 sind ähnlich, wie die bei der sequentiellen Maschine ("IORelation" und "IO-Funktion").

```
TYPE
  TimeInterval =  RECORD
                    initialTime: Time;
                    endTime: Time;
                  END;
  InputSegment =    RECORD
                    tint:TimeInterval;
                    input: Input;
                  END;
  Output =  RECORD
              t: Time;
              oValue: OutputValue;
            END;
  OutputSegment = POINTER TO Outputs;
  Outputs = RECORD
              ou: Output;
              next: OutputSegment;
            END;
  IOSegmentPair =  RECORD
                     is: InputSegment;
                     os: OutputSegment;
                   END;

PROCEDURE IORelation(VAR R: ARRAY OF IOSegmentPair);
PROCEDURE IOFunction(is: InputSegment): OutputSegment;
```

3.3 Definition des ereignisorientierten Systemformalismus

Die Typen `SequentialState` und `ExternalEvent` sind, ähnlich, wie die Typen `Input`, State, Output bei der sequentiellen Maschine, als opake Datentypen vereinbart. Der Typ DEVS ist durch drei Funktionsprozeduren gegeben:

```
TYPE
  SequentialState;  ExternalEvent;

  InternalTransition = PROCEDURE(SequentialState): SequentialState;
  ExternalTransition = PROCEDURE(SequentialState, ExternalEvent): SequentialState;

  Transition =  RECORD
                  transInt: InternalTransition;
                  transExt: ExternalTransition;
                END;
  TimeAdvanceFunction = PROCEDURE(SequentialState):REAL;

  DEVS =  RECORD
            Delta: Transition;
            ta:TimeAdvanceFunction;
          END;
```

4. Diskussion

Ein Prototyp von SAM ist mit einem der neuen Modula-2 Compiler aus der Einpassfamilie implementiert worden. Mehrere Modelle aus allen der drei wichtigen Klassen (sequentielle Maschine, DESS, DEVS) sind durch ihre Programmierung gemäss SAM modelliert und anschliessend simuliert worden. Die bislang erzielten Resultate zeigen, dass es gelungen ist, die entworfene Simulationssoftware dank einiger spezifischer Eigenschaften der Programmiersprache Modula-2 (opake Datentypen, Prozedurtyp (Prozedurvariablen), offene formale Feldparameter) derart zu realisieren, dass der Grad der Abstraktion nicht kleiner wird, als er in den Modellierungstheorien von Wymore und Zeigler erreicht worden ist. Allerdings haben sich bei dieser Arbeit auch einige Schwierigkeiten ergeben, die sich in Modula-2 nur mit beschränkter Eleganz realisieren lassen. Die bei Modula-2 übliche, implizite Versionenüberprüfung der Module bereitet bei dem gewählten Ansatz Probleme, da der Dienstteil dann keine Importe mehr aus dem Klientteil vornehmen darf, um die Recompilierung des Dienstteils vermeiden zu können. Den eleganten Formulierungs- und Modularisierungsmöglichkeiten von Modula-2 wurden dadurch entscheidende Grenzen gesetzt, die man häufig gerne überschritten hätte. Alles in allem hat sich jedoch die Verwendung von Modula-2 für unser Vorhaben gut bewährt.

Als besonders vorteilhaft haben sich die Möglichkeiten von Modula-2 zur Defintion strukturierter Datentypen erwiesen. Dank dieser Eigenschaft war es möglich auf besonders elegante Art eine Systemspezifikation auf der dritten und vierten Ebene mit ein und demselben Modul vornehmen zu können. Durch die strenge Typenüberprüfung von Modula-2 liess sich auch die Robustheit der Simulationssoftware steigern (CELLIER 1984).

Die erzielten Resultate stellen bloss einen ersten Schritt im Hinblick auf die Entwicklung der entworfenen Simulationssoftware dar. Die dadurch entwickelte Grundlage für weitere Entwicklungsschritte scheint aber genügend tragfähig zu sein um Weiterentwicklungen darauf aufbauen zu können. Insbesondere ist in einem nächsten Schritt die Realisierung der computergestützten, graphischen Modellierung CAGM [Computer-Aided Graphic Modelling] vorgesehen.

5. Literaturverzeichnis

CELLIER, F., E. 1979. Combined continuous/discrete system simulation by use of digital computers: techniques and tools. Diss ETH Zürich No 6483, 266pp.

CELLIER, F., E. 1984: *How to enhance the Robustness of Simulation Software,* in: ÖREN, T. I., ZEIGLER, B. P., ELZAS, M. S.(EDS): *Simulation and Model-Based Methodologies: An Integrative View,* Springer-Verlag, 1984

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.

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. 294pp.

FISCHLIN, A. 1986. Simplifying the usage and programming of modern working stations with Modula-2: The Dialog Machine. In prep.

WIRTH, N. 1985: *Programming in Modula-2, Third, Corrected Edition,* Springer-Verlag, 1985

WYMORE, A. W. 1976: *Systems Engineering Methodology for Interdisciplinary Teams,* John Wiley & Sons, 1976

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., ELZAS, M. S., KLIR, G. J., ÖREN, T. I. (EDS) 1976: *Methodology in Systems Modelling and Simulation,* North-Holland Publishing Company, 1979

ZEIGLER, B. P. 1979:*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, 1984

ZEIGLER, B. P. 1984:*Theory of Modelling and Simulation,* John Wiley & Sons, 1976

6. Anhang (Modellierungstheorie von Wymore und Zeigler)

Die theoretischen Grundlagen für die Modelllierung und Simulation haben A. W. WYMORE (1976, 1984) und B. P. ZEIGLER (1976, 1979, 1984) geschaffen. Hier werden die formalen Definitionen der Systemtheorie kurz zusammengefasst.

Die Ebenen der Systemspezifikationen:

Ebene	Spezifikation	Formales Objekt
5	Kopplung von Systeme	$(D, \{S_d\}, \{I_d\}, \{Z_d\})$
4	Strukturiertes I/O System	$(A,D,\{A_d \mid d \in D\},i)$
3	I/O System	$(T, X, \Omega, Q, Y, \delta, \lambda)$
2	I/O Beobachtungsfunktion	(T, X, Ω, Y, F)
1	I/O Beobachtungsrelation	(T, X, Ω, Y, R)
0	Beobachtungsbereiche	(T,X,Y)

Ebene 0: O=(T,X,Y)

T: Zeitbasis (Menge): O ist *kontinuierlich*, wenn $T=\Re$ (Menge der reellen Zahlen) , und *diskrete* wenn $T=\Im$ (Menge der ganzen Zahlen).

X: Menge der Eingangswerte

Y: Menge der Ausgangswerte

Ebene 1: IORO=(T,X,Ω,Y,R)

(T,X,Y): Beobachtungsbereiche (wie in Ebene 0)

Ω: Menge der Eingangssegmente $\Omega \subseteq (X,T)$

R: I/O Relation, $R \subseteq \Omega \times (Y,T)$, so, dass wenn $(\omega,ß) \in R$, dann $dom(\omega)=dom(ß)$; $(\omega,ß)$ ist ein sog. Input/Output Segment Paar.

Ebene 2: IOFO=(T,X,Ω,Y,F), wo

T,X,Ω,Y: wie in der Ebene 1

F: die Menge der I/O Funktionen. Wenn $f \in F$ dann $f \subseteq \Omega \times (Y,T)$. f ist eine Funktion so, dass $dom(f(\omega))=dom(\omega)$.

Ebene 3: S=(T,X,Ω,Q,Y,δ,λ), wo

T,X,Ω,Y: wie in der Ebene 1

Q: Menge der internen Zustände

δ: Übergangsfunktion, $\delta: Q \times \Omega \rightarrow Q$

λ: Ausgangsfunktion, $\lambda: Q \rightarrow Y$

Ebene 4: Strukturiertes System:

Der Unterschied zwischen der Systemspezifikation auf dieser und der dritten Ebene liegt darin, dass die Mengen und Funktionen strukturiert sind, d.h. sie sind repräsentiert als ein kartesisches Produkt mehrerer elementarer Mengen und Funktionen. Eine strukturierte Menge ist eine Struktur $A=(A,D,\{A_d \mid d \in D\},i)$ wobei gilt

A: zu strukturierende Menge

D: geordnete Menge, $d_1,d_2,...,$ (die Koordinaten)

A_d: Wertebereich, $d \in D$

i: die Zuweisungsfunktion derart, dass $i: A \rightarrow \times_{d \in D} A_d$ (A eineindeutig)

Ebene 5: Kopplung von Systemen: Die Kopplung von Systemen ergibt die folgende Struktur, $N=(D, \{S_d\}, \{I_d\}, \{Z_d\})$, wobei

D: Menge der Namen der Komponenten, $\forall\, d \in D$:

S_d: System (Komponente d)

I_d: Menge der Systeme, die d beeinflussen

Z_d: Funktion, die Kopplungsfunktion von d so, dass wenn $S_d = (T,X_d,\Omega_d,Q_d,Y_d,\delta_d,\lambda_d)$ einSystem in der Ebene 4, dann $I_d \subseteq D$ und $Z_d: \times_{\beta \in I_d} Y_\beta \rightarrow X_d$.

Innerhalb einer Gesamtmenge mathematisch beschreibbarer Modelle gibt es Klassen, sog. Formalismen.

Zeitdiskrete Systeme: der Formalismus der sequentiellen Maschine

Der Formalismus der sequentiellen Maschinen oder Automatenformalismus kann als eine Spezialisierung diskreter Systeme interpretiert werden. Es kann gezeigt werden, dass jedes diskrete System als eine sequentielle Maschine spezifiziert werden kann. Eine sequentielle Maschine ist eine Struktur $\mathbf{M=(X, Q, Y, \delta_M, \lambda)}$ wobei

X, Q, Y:	Mengen der Eingänge, Zustände und Ausgänge.
δ_M:	Übergangsfunktion, $\delta_M: Q \times X \rightarrow Q$
λ:	Ausgangsfunktion

Mit einer sequentiellen Maschine kann ein strukturiertes System assoziiert werden.

Der Differentialgleichungsformalismus:

Der Differentialgleichungsformalismus schreibt nicht den nächsten Zustand direkt vor, sondern er definiert implizite Beschränkungen dafür, wie sich der Übergang ereignen soll. Um Differentialgleichungssysteme numerisch lösen zu können, werden sie in zeitdiskrete Systeme umgewandelt. Eine Differentialgleichungssystemspezifikation (DESS) ist eineStruktur: $\mathbf{D=(X,Q,Y, f, \lambda)}$

X :	Menge der Eingangswerte (reeller, endlich-dimensionaler Vektorraum, $\Re^n$)
Q:	Menge der Zustände ($\Re^n$)
Y:	Menge der Ausgange ($\Re^n$)
f :	Ableitungsfunktion, $f: Q \times X \rightarrow Q$
λ:	Ausgangsfunktion, $\lambda: Q \rightarrow Y$.

Eine DESS kann in ein strukturiertes I/O System überführt werden.

Der Ereignisorientierte Systemformalismus:

Eine diskrete Ereignisorientierte Systemspezifikation (DEVS) ist eine Struktur $\mathbf{M=(X,S,\delta,ta)}$

X:	Menge: die Namen äusserer Ereignistypen
S:	Menge der sequentiellen Zustände
δ:	Funktion: die Übergangsspezifikation
ta:	Funktion: die Zeitfunktion

Eine DEVS kann in ein strukturiertes I/O System übersetzt werden.