A modeling environment for decision support systems

van Hee, K.M.; Somers, L.J.A.M.; Voorhoeve, M.

Published: 01/01/1990

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 12. Dec. 2018
A MODELING ENVIRONMENT FOR
DECISION SUPPORT SYSTEMS

by

K.M. van Hee
L.J. Somers
M. Voorhoeve

EINDHOVEN UNIVERSITY OF TECHNOLOGY
F. du Buisson
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN

Juli 1990
K.M. van Hee
Eindhoven University of Technology
Faculty of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

M. Voorhoeve
Eindhoven University of Technology
Faculty of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

L.J.A.M. Somers
Eindhoven University of Technology
Faculty of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, the Netherlands
ABSTRACT

We describe a formal framework for modeling and prototyping complex systems. Our framework consists of a meta-model for discrete event systems, a language based upon this meta-model and a software environment for editing and validating system descriptions. The possibilities for using our framework for decision support systems are indicated and illustrated by a job shop planning example.

1. INTRODUCTION

We regard a decision support system (DSS) as a special class of automated information systems, i.e. systems that collect, maintain and produce information about some other system called target system. A DSS is focused at decisions that affect the target system. It can evaluate the effects of proposed decisions and generate decisions satisfying user-given criteria. Some authors (e.g. [Keen 87]) require a facility for selecting a decision in the case of conflicting criteria. A fourth facility is for sensitivity analysis of a given decision, which is connected to its evaluation.

The term decision is used here for the unit of information sent to the target system. Its impact depends upon the granularity of the control process. A decision might be a simple control action (that influences the target system for a short period of time) or a set of such actions (e.g. a production plan for a batch of jobs), but it can also be a function that assigns actions to observed behaviour of the target system. In the latter case, a single decision may control the target system for an arbitrarily long period of time.

Designing a classical information system means automating an existing well-defined manual system. When designing a DSS, though, it is unwise to automate the existing (heuristic) decision processes entirely: care has to be taken in deciding which decision tasks can be automated. Many DSS projects in the past have failed by assigning too many tasks.
to the machine. The decision situation proved more complicated than foreseen by the designers, rendering the automated procedures inadequate (cf. [Hee 89a]).

We describe a framework for the formal description and simulation of a wide variety of systems, called discrete event systems. These are characterized by a finite or countable state space and a behaviour that can be described as a succession of states. This class of systems contains information systems as well as physical systems (e.g. robots). Solutions of differential equations and the like are out of scope.

Our framework plays a role in the requirements engineering phase of DSS design. Here, designers and decision makers (future DSS users) together try to establish the functionality of the DSS. The use of prototypes for this purpose is recommended. Experiments with the prototype can be conducted, showing the strong and weak points of the prototype. With little turnaround time, the prototype can be adapted, until the users are satisfied with the functionality offered.

Prototypes of the DSS only are not always sufficient, though. The prototype experiments should be conducted under realistic but controllable circumstances. So the relevant parts of the target system should be simulated. Both tasks, prototyping of the DSS and simulation of the target system, can be performed by our framework, provided both the target system and the DSS can be modeled as a discrete event systems. The model of the target system should reflect its performance characteristics (namely the effects of decisions made by or through the DSS), while the model of the DSS should reflect its functionality.

Our framework thus integrates many activities (specification, prototyping and simulation) and aspects (data models, process models and process interaction) of DSS construction: moreover, it does so in a natural way close to intuition. It is possible to construct libraries of reusable components (model banks) to speed up the construction of DSS prototypes and target system simulation models. However, this paper stresses the way in which models are described and not how they are maintained and used (model management systems).

Our paper is organised as follows. In section 2, we introduce our framework. In section 3, we define its model, summarizing some properties. In section 4, we introduce its language. In section 5, we give an example of an interesting decision situation showing the compactness of specifications within the framework. In [Hee 89 b,c] the contents of sections 3 and 4 is also covered. Note that this paper does not pretend to give all details. We only give the flavour of using an integrated framework and hope that it catches.

2. OVERVIEW OF THE FRAMEWORK

One can model the world as a collection of data types, functions and procedures, as encountered in third-generation languages. These concepts can be combined into models (like e.g. linear programming); [Dolk-Konsynski 84] give a language and a tool for describing and manipulating models. The model does not indicate how its procedures and functions are activated.
The activation issue is settled by embedding the model in an automaton. Automata are characterized by a state space, input-output alphabet and transition function. Upon receiving input, the state of an automaton is modified and output is produced by applying the transition function to the old state and the received input. An expedient of this approach is Z (cf. [Hayes 87]): a high-level mathematical language is used to describe transition functions.

Automata are however static entities; it is of course possible to describe e.g. parallelism or temporal behaviour, but this has to be done explicitly. Our framework is essentially dynamic: we describe a system as a network of automata that are communicating internally and with the outside world. Modeling systems as dynamic networks is very old; there exist a multitude of data flow diagramming techniques (e.g. [Yourdon 89]), although these techniques had no formal semantics.

We thus are led to distinguish the following three aspects of a discrete event system (reception of input and production of output are modeled as state transformations)

- the state spaces of components,
- the state transformations of components.
- the interaction structure.

Our model embodies these aspects; it is related to (coloured) Petri nets (CP nets, cf. [Jensen 87], for other Petri net variants cf. [Reisig 85]). Like in CP nets, triggers (tokens) have a colour and reside in typed channels (places). Processors (transitions) continuously consume and produce triggers. Unlike in CP nets, a colour has two components: a time stamp and a value. The consumption of triggers by a processor depends only upon their time stamps (lower time stamps have a higher priority) and is globally determined. The production of triggers depends only upon their values. This separation of concerns gives more clarity and better execution performance than in CP nets (where one can specify conditions upon the colours for token consumption), without sacrificing modeling power.

We call our model the DES model and a system specified by it a DES. We can use Petri net theory to verify structural properties of systems modeled in our framework (cf. [Genrich 81]). Our model has at least the same power as DEVS [Concepcion 88].

Our language ExSpect is executable and thus allows prototyping and experimentation. It encourages the reuse of specified components and possesses an interface for the incorporation of external software. It has a functional, a dynamic and a structural part. The functional part is used to define types and functions: a many-sorted algebra. The type system consists of primitive types and type constructors to define new types. A sugared lambda calculus is used to define new functions from a set of primitive ones. The dynamic part of ExSpect uses types and functions to specify processors and channels and it defines their interaction structure. Finally, the structural part of ExSpect supports its use for large projects by modularization.

The state of a DES is the configuration of triggers in the channels. The state space of a channel is characterized by its type: the set of all multisets (bags) over this type. We thus specify state spaces by types, state transformations by functions and interaction
structures by networks of processors and channels. A special kind of channels is called *stores*. These channels contain one and only one trigger. A processor that consumes a trigger from it must replace it immediately. This way of modeling corresponds to SADT [Marca 88].

We have developed a software tool to support the specification process. It consists of a graphical editor, a type checker that verifies the type correctness and an interpreter that simulates the specified system. The interpreter is connected to an asynchronous end-user interface. Decision makers, supported by the DSS prototype, can partake in the simulation of the target system and see the effects of their decisions.

3. MODEL OF DISCRETE EVENT SYSTEMS

A DES consists of two kinds of components: *processors* and *channels* (transitions and places). To each channel corresponds a type (a set). A processor is connected to one or more input channels and zero or more output channels. To each association of an input channel to a processor a weight is attached (most weights equal 1). Channels are shared by processors. Channels contain *triggers* (tokens). A trigger has a *value* that belongs to the type of its channel and a real-valued *time stamp*. There may be more triggers in the same channel with the same value and time stamp. So a channel actually contains a bag of triggers.

At any moment a *transition* may occur, i.e. a change in the configuration of triggers in the channels (the *state*). (We attach a different meaning to the term transition than Petri net theory). Transitions occur instantaneously and are *executed* by processors. A processor *p* may execute if it is able to consume the right number of triggers from each of its input channels. This number must be equal to the weight of the input channel for *p*. The execution of a processor implies the consumption of the triggers from its input channels and the production of triggers in its output channels. The number and value of the produced triggers is a function of the values of the consumed triggers. An *event* is an assignment of triggers to a processor *p* such that *p* can execute. The *event time* of an event is the maximum of the time stamps of the triggers to be consumed. The *transition time* of a system in a certain state is the minimum of the possible event times. Being in a certain state, a system will select an event of which the event time equals the transition time and execute it, causing a state transition. The time stamps of produced triggers will be at least equal to the event time. It is thus clear that the transition times of successive events will be non-descending.

Formally, a DES is represented by the following components.

- A set *P* of processors.
- A set *C* of channels *c*, each with a set *V*(c) of possible trigger values.
- A function *I* assigning to each processor a bag (weighted set) of input channels.
- A function *O* assigning to each processor a set of output channels.
- A set *T* of possible time stamps.
- For each processor *p* in *P*, a function *f*ₚ, satisfying the requirements below.
The domain of a function \( f_p \) is a set \( S \) of bags \( B \) of channel-trigger pairs. For any \( B \) in \( S \), the number of pairs in \( B \) with channel \( c \) is equal to the weight of \( c \) in \( I(p) \). If all weights in \( I(p) \) are 1, \( I(p) \) becomes an ordinary set and \( S \) a set of functions \( B \) from channels in \( I(p) \) to triggers (since each channel \( c \) occurs only once in a \( B \)).

The range of \( f_p \) is a bag of channel-trigger pairs such that the channels are in \( O(p) \). Note that there is no weight restriction here; the number of triggers produced on each output channel is not fixed. This is the reason why \( O(p) \) is a set and \( I(p) \) a bag.

An invariant (pre- and post) condition for the functions \( f_p \) is that any trigger coupled to channel \( c \) has a value in the set \( V(c) \). A postcondition is that the time stamp of any output trigger must be at least equal to the maximal time stamp of the input triggers.

We can derive from the above components a transition system, i.e. a state space \( S \) and a relation \( R \) giving for each state in \( S \) the set of possible next states (the transition relation). This is done as follows. \( S \) is the set of all bags of channel-trigger pairs \((c,t)\), with \( t \) in \( V(c) \). For any \( s \) in \( S \), we compute the possible events, i.e. subbags \( B \) of \( S \) and a processor \( p \) such that \( f_p(B) \) is defined. We may say that \( p \) can execute or fire, caused by \( B \). The event time of such an event is the maximal time stamp of the triggers in \( B \). We construct the set \( E \) of possible events with minimal event time (the transition time). A next state is then computed by selecting a \( B, p \) pair from \( E \), subtracting \( B \) from \( S \) and adding \( f_p(B) \) to the outcome.

The above formalization of the DES-framework has been elaborated in [Hee 89 b,c]. It is even slightly complicated further, since an event can consist of the simultaneous firing of more processors, thus allowing parallelism. However, a state transition caused by the such an event can be reached also by a sequence of single-processor transitions (serializability).

To represent a DES we use a diagram technique like for Petri nets. A diagram is isomorphic to an ordered graph containing the network of processors and channels (the first four components above). To such a diagram we must add a type for each channel and a function to each processor. Networks can be defined separately and used as a subnetwork in a larger network. It is thus possible to use a decomposition hierarchy (cf. [Huber 89]). A high-level DES diagram is very similar to a data flow diagram (cf. [Yorndon 89]).

In ExSpect, the computation of values of triggers is separated from the computation of their time stamps. The time stamps are computed by a delay depending on the values of the consumed triggers. This delay is added to the event time to obtain the time stamps of the produced triggers. When no delay is specified, the value 0 is taken. By not specifying delays, we have a timeless model; events are then ordered only by causality: triggers can be consumed only after production.

4. LANGUAGE OVERVIEW

As we remarked in the previous section, a DES diagram must be complemented by attaching types to channels and functions to processors. The definition of types and functions can be done in the functional part of our language ExSpect. Processors, channels and (sub)systems can be defined in the dynamic part of our language, graphically or by means
Types are defined by means of type expressions, consisting of basic types and type operators. Functions are defined by means of expressions, built from basic constants and functions. We shall treat in this order basic types, type expressions, type definitions, expressions and function definitions. Then we move to the dynamic part and treat processor and system definitions. Channels are defined within a system. We do not treat modules; they are similar to modules in modern programming languages and affect the scope of definitions.

The basic types are void, bool, num, real and str. These correspond to the empty set, the booleans, the rationals, the floating point numbers and the strings, respectively. The basic type operators are set (denoted by a $ prefix) and cart (denoted by a $\times$ infix). If $A$ and $B$ are type expressions, then $\$A$ denotes the set of finite subsets of $A$ and $A \times B$ denotes the set of pairs of objects, the first in $A$, the second in $B$. The type operator map (denoted by a $\rightarrow$ infix) is derived from set and cart. The type expression $A \rightarrow B$ denotes the set of mappings from $A$ to $B$; a mapping is a finite set of pairs with different first components. Precedence is indicated by brackets.

The type definitions are represented as follows:

```
type id from te,
```

where $id$ is an identifier and $te$ a type expression. The following example illustrates some type definitions.

```
type coord from real><real;
type loc from coord->str;
```

A store of type loc can be regarded as a file containing the names of cities and their coordinates (longitude and latitude). The ExSpect type system is hierarchical; a type "inherits" all functions that can be applied to its supertypes.

Our set of basic functions includes all well-known set-theoretical, logical and numerical constants and functions. Actually only a few of them are truly basic; the others can be derived from them. Many of these functions are "sugared" to their customary symbolic infix or "circumfix" notations. For instance, the application of a mapping (to an element as well as to a set) is indicated by a dot infix. All binary function applications can be denoted in infix form. Expressions are built by constants, variables, function application and mapping construction. A mapping construction is an expression of the form

```
[ id : setexpr | expr ]
```

where $id$ is an identifier, setexpr an expression denoting a set and $expr$ an expression (probably containing $id$ as parameter). Its meaning is the mapping $\lambda id \in setexpr : expr$.

Constant and function definitions are represented respectively as follows:

```
id := expr : te,
id [ id_1 : te_1 , \ldots , id_n : te_n ] := expr : te,
```

where the $ids$ are identifiers, the $tes$ type expressions and $expr$ an expression. We give some examples.
min [x: real, y: real] := if \( x > y \) then \( x \) else \( y \) fi: real;
min [x: $\{\text{real}\}] := \text{if } \text{rest}(x) = {} \text{ then pick}(x)
\text{ else min(rest(x)) min pick(x) fi: real;}
min [x: T->real] := min(rng(x)) : real;
upd [x: T->S, y: T, z: S] :=
\\[ t: y \text{ ins dom}(x) \text{ if } t=y \text{ then } z \text{ else } x.t \text{ fi } : T->S;\]
upd [x: T->S, y: T->S] :=
\\[ t: \text{dom}(x) \text{ union dom}(y) \text{ if } t \in \text{dom}(y) \text{ then } y.t \text{ else } x.t \text{ fi } : T->S;\]
pinf [x: T->real] := pick($\{t: \text{dom}(x) \text{ if } x.t = \text{min}(x)\}): T;

The functions above are examples of basic functions defined in terms of still more primitive ones, c.q. \( \text{pi}_1, \text{pi}_2 \) (projections), \{\}, \text{ins, pick, rest, union} (resp. the empty set, insertion, picking an element from a set, the set without the picked element, set union), \text{dom and rng} (domain and range of mappings). \( \{t: A | P\} \) is our notation for the subset of \( A \) where \( P \) holds. Some definitions are recursive and some \textit{polymorphic}. A polymorphic definition uses \textit{type variables} \((T, S)\). Note the multiple definitions (overloading); they are allowed if no clashes occur. The \text{min} functions compute the minimum of two numbers, a set of numbers or a numerical mapping. The \text{upd} functions overwrite a mapping either by a specific value at a specific location or by a second mapping. Finally, \text{pinf}(x)\) selects a point from the domain of \( x \) where its minimum value is reached.

Processor definitions have a header and contents. The header contains the processor name and its parameters, (channels and stores, sometimes values). The contents consists of concurrent (conditional) assignments of expressions to output channels and stores. A system is an aggregate of processors, channels and stores. Its definition header is similar to a processor header and the contents name the internal processors, subsystems, (initialized) channels and stores and describe the graph interconnecting them. Examples of processor and system definitions are given in the next section.

Definitions can be followed by a \textit{where-part}, a list of definitions the scope of which is limited to the definition it follows. Where-parts can be nested; they belong to the structural component of ExSpect.

5. EXAMPLE

In this section we illustrate the concepts of the previous chapters by specifying a simple DSS for scheduling in a job shop. From an external source (the sales department), jobs are brought into the DSS. A job is divided into a set of tasks. A task has a \textit{machine type}, a \textit{standard duration} and a set of \textit{predecessors}. The shop floor may contain several machines of the same type. Machines may be unavailable due to malfunction. The standard duration of a task is its processing time on a (hypothetical) standard machine. A machine's speed is measured in relation the standard machine. The estimated duration of a task on a specific machine equals the task's standard duration divided by the machine's speed. The predecessors of a task \( t \) are the tasks (belonging to the same job) that need
to be finished before \( t \) can be started. The predecessor graph may contain no cycles.

The DSS produces schedules for the shop floor: the assignment of machines to tasks on a specific time (here a decision is a schedule). From another external source (the shop floor), feedback is given to the DSS: reports on e.g. completed tasks and machine breakdowns. The DSS generates reports for and receives decisions from the decision maker.

To model the DSS, we start by modeling the fundamental concepts playing a role in the scheduling process. Next we define the functions using these types that are needed throughout the scheduling system and end by formulating the constraints that schedules must satisfy. The constraints we consider are the following. They can be extended at will.

- Only known tasks can be scheduled.
- Tasks are scheduled for available machines of the prescribed type.
- Any predecessor of a scheduled task is scheduled too; the starting time of the task exceeds or equals the finishing time of any predecessor.
- The same machine cannot be occupied by two tasks at the same moment.

The above definitions in ExSpect are as follows.

--- type definitions ------------------------

\[
\begin{align*}
\text{type tid} & \quad \text{from str;} & \quad \text{-- identification for tasks} \\
\text{type mtyp} & \quad \text{from str;} & \quad \text{-- machine type} \\
\text{type tdur} & \quad \text{from real;} & \quad \text{-- time or duration} \\
\text{type tasks} & \quad \text{from tid -> $tid >< mtyp >< tdur;} \\
\text{type mid} & \quad \text{from str;} & \quad \text{-- machine identification} \\
\text{type machs} & \quad \text{from mid -> mtyp >< real >< bool;} \\
\text{-- type, rel. speed, available} \\
\text{type sched} & \quad \text{from tid -> mid >< tdur;} \\
\end{align*}
\]

--- standard functions -----------------------

\[
\begin{align*}
\text{prs} \quad [t: \text{tasks}, x: \text{tid}] & := \text{pi1}(t.x) : \text{tid;} \\
\text{mtt} \quad [t: \text{tasks}, x: \text{tid}] & := \text{pi2}(t.x) : \text{mtyp;} \\
\text{sdur} \quad [t: \text{tasks}, x: \text{tid}] & := \text{pi3}(t.x) : \text{tdur;} \\
\text{mtm} \quad [m: \text{machs}, x: \text{mid}] & := \text{pi1}(m.x) : \text{mtyp;} \\
\text{sp} \quad [m: \text{machs}, x: \text{mid}] & := \text{pi2}(m.x) : \text{real;} \\
\text{av} \quad [m: \text{machs}, x: \text{mid}] & := \text{pi3}(m.x) : \text{bool;} \\
\text{ma} \quad [s: \text{sched}, x: \text{tid}] & := \text{pi1}(s.x) : \text{mid;} \\
\text{stm} \quad [s: \text{sched}, x: \text{tid}] & := \text{pi2}(s.x) : \text{tdur;} \\
\text{etm} \quad [s: \text{sched}, t: \text{tasks}, m: \text{machs}, x: \text{tid}] := \text{stm}(s,x) + \text{sdur}(t,x) \backslash \text{sp}(m,\text{ma}(s,x)) : \text{tdur;} \\
\end{align*}
\]

--- constraints -----------------------------

\[
\begin{align*}
\text{c1} & \quad [s: \text{sched}, t: \text{tasks}] := \text{dom}(s) \subset \text{dom}(t) : \text{bool;} \\
\text{c2} & \quad [s: \text{sched}, t: \text{tasks}, m: \text{machs}] :=
\end{align*}
\]
We shall now define a schedule updater. It checks whether a produced schedule meets the constraints; it is used for evaluating user-defined decisions.

\[
\text{proc supd [in d: sched, out ok: bool, store t: tasks, m: machs, s: sched]} :=
\]
\[
\begin{cases}
\text{if c(s upd d, t, m)} & \text{then ok <- true, s <- s upd d} \\
\text{else ok <- false fi}
\end{cases}
\]

A second processor is a schedule truncator: it deletes the tasks scheduled after a given time. It can be shown that this deletion strategy leaves the constraints invariant.

\[
\text{proc strunc [in h:tdur, store s: sched]} := s <- s \text{ restrict } \{x: \text{dom}(s) | \text{stm}(s, x) < h\};
\]

A third processor is an automatic scheduler; it generates decisions using a “greedy” algorithm. Given are stores \(m\), \(s\), and \(t\) as before and a scheduling range \(srg\) (the cartesian product of a start time and an end time). The processor adds unscheduled tasks to the existing schedule, giving priority to tasks whose predecessors have finished first. The ExSpect header of this processor is as follows (the definition is too lengthy to include here and can be found in an appendix).

\[
\text{proc salg [in srg:tdur><tdur, store s: sched, m: machs, t: tasks]}
\]

These processors and a user interface system \(ui\) form the DSS. Under control of the decision maker, the user interface switches back and forth between updater, truncator and algorithm, while reporting about the schedule being produced. We can add other subsystems to simulate the target system: a task generator \(tgen\) (simulating the sales department) and a panic generator \(pgen\) (simulating machine breakdowns and repairs at the shop floor). Breakdowns and repairs are reported to the decision maker; breakdowns also start an immediate automatic truncation and rescheduling operation to account for the modified situation. The truncation and rescheduling described here is complete from the moment of breakdown onward, but we could easily modify it so that only tasks that directly or indirectly suffer from the breakdown are rescheduled. The system described
where \( m_1 := \text{'m1'} : \text{mid end}; \)

\[
\begin{align*}
\text{sys pgen [out rep: str, int: tdur><tdur,} \\
&\quad \text{store m: machs, sch: sched] :=} \\
&\quad \text{channel i: mav init true,} \\
&\quad \text{store s:real init 0.4321,} \quad \text{-- seed for random generator} \\
&\quad \text{p(in i, out i, rep, int, store m, sch, s);} \\
\text{sys main [in uicom: str, out uirep: str, panrep: str] :=} \\
&\quad \text{channel updsched: sched,} \\
&\quad \text{channel okupd: bool,} \\
&\quad \text{channel trnctim: tdur><tdur,} \\
&\quad \text{store cursched: sched init \{\},} \\
&\quad \text{store curtasks: tasks init \{\},} \\
&\quad \text{store machines: machs init \{ 'm1', \ldots \},} \quad \text{-- fill in machines} \\
&\quad \text{ui (in uicom, okupd, out uirep, updsched, algint, trnctim,} \\
&\quad \quad \text{store cursched),} \\
&\quad \text{supd (in updsched, out okupd,} \\
&\quad \quad \text{store cursched, curtasks, machines),} \\
&\quad \text{strunc (in trnctim, store cursched),} \\
&\quad \text{salg (in algint, store cursched, curtasks, machines),} \\
&\quad \text{tgen (store curtasks),} \\
&\quad \text{pgen (out panrep, algint, store machines, cursched);} \\
\end{align*}
\]

In the above system, decision makers can use and test the DSS under more or less realistic circumstances.
6. CONCLUSIONS

The power of our framework lies not so much in the novelty of its concepts (although there are several improvements of existing formalisms) as in the fact that it integrates known concepts, namely Petri nets, data modeling, functional programming, data flow diagrams and abstract data types. Descriptions, specifications, simulation models and prototypes for the target system and the DSS can be made by a single tool.

The type system of ExSpect, with polymorphy and type hierarchy as well as the various parametrization possibilities of its constructs make it feasible to construct libraries of reusable components. Backed up by such libraries (model banks) for DSS disciplines, ExSpect will be a very powerful modeling, simulation and prototyping tool.

APPENDIX

In this appendix we describe the scheduling algorithm. Given are stores \( m, s, \) and \( t \) as before and a scheduling range \( srg \) (a start time \( a = pi_1(srg) \) and an end time \( b = pi_2(srg) \)).

From the existing schedule, we select a task \( y \) with the following properties.

- \( y \) has not been scheduled yet,
- all predecessors of \( y \) have been scheduled
- \( mpt(y) \), the maximal completion time of the predecessors of \( y \), is minimal.

This task is then planned into the schedule with time limit \( \max(a, mpt(y)) \). We repeat this until all tasks \( y \) have been treated for which \( mpt(y) \leq b \). We thus arrive at a schedule for which no tasks can be scheduled at a time before \( b \).

The planning of a task \( y \) into a schedule \( x \) with time limit \( z \) is done as follows. Given a machine \( w, i \) is the set of tasks scheduled on \( w \), the end time of which exceeds \( z \). The set \( cst \) of candidate start times for \( y \) on \( w \) consists of the end times of jobs planned on \( w \), augmented, if \( w \) is idle at time \( z \), with \( z \). The set \( pst \) of possible start times is the subset of \( cst \) after which \( w \) is idle long enough to execute \( y \). We now treat \( w \) as a variable and take the minimum of \( pst(w) \) over all machines \( w \) that are available and of the prescribed type. This minimum time and (one of) the machine(s) where this minimum is reached is added to the schedule \( x \) for task \( y \), if the minimum time does not exceed \( b \).

The ExSpect code of the algorithm is as follows.

```prolog
proc salg [in srg:tdur><tdur, store s:sched, m:machs, t:tasks] :=
  s <- fill (s, pi1(srg), dom(t)\dom(s))
  where
  fill [x:sched, y:tdur, v:$tid] :=
    if v={} or y>=pi2(srg) then x
    else fill (nxtsch, nxttim, v del pinf(cands)) fi : sched
  where
    nxtsch := plan(x, pinf(cands), nxttim) : sched;
    nxttim := min(cands) max pi1(srg): tdur;
    cands := [z: ct| mpt(z)] : tid->tdur;
```
ct := $[z : v | \text{prs}(t,z) \subset \text{dom}(x)] : $tid;

mpt[z,tid] := \max[w : \text{prs}(t,z) \mid \text{etm}(x,t,m,w)] : \text{tdur};

\text{plan}[x: \text{sched}, y: \text{tid}, z: \text{tdur}] :=
\begin{cases}
\text{if } \min(fri) \geq \pi 2(srg) \text{ then } x & \text{ -- horizon exceeded} \\
\text{else } \text{upd}(x, y, <\text{pinf}(fri), \min(fri)>) & \text{ fi : sched}
\end{cases}

\text{where}
\begin{align*}
\text{fri} & := \{w : \text{pm} \\mid \text{min}(\text{pst}(w))\} : \text{mid} \rightarrow \text{tdur}; \\
\text{pm} & := \{w : \text{dom}(m) \mid \text{av}(m,w) \text{ and } \text{mtm}(m,w) = \text{mtt}(t,y)\} : \text{mid}; \\
\text{pst}[w: \text{mid}] & := \{v : \text{cst} \mid \forall[u : \text{stm}(x,u) \leq v \text{ or } \\
\text{stm}(x,u) & \geq v + \text{sdur}(t,y)/\text{sp}(m,w)]\} : \text{tdur}; \\
\text{where}
\end{align*}
\begin{align*}
\text{cst} & := \text{rng}[v : i | e(v)] \cup \\
\text{if } \forall[v : i | \text{stm}(x,v) > z] \text{ then } \{z\} \text{ else } \emptyset : \text{tdur}; \\
i & := \{v : \text{dom}(x) \mid \text{ma}(x,v) = w \text{ and } e(v) \geq z\} : \text{tid}; \\
e[v : \text{tid}] & := \text{etm}(x,t,m,v) : \text{tdur};
\end{align*}

The effect of the above algorithm is that tasks whose predecessors have finished first have priority over other tasks. Machines that are a bottleneck will process tasks on a first-come-first-served base. This is a fair and stable strategy. Of course we do not claim any originality for this crude algorithm, nor do we advocate its unmodified use in practical situations; we want merely to show that algorithms such as the above can be defined in a nice top-down manner.

Also note how the above algorithm resembles the scheduling done by the ExSpect interpreter as prescribed in section 3, with machines instead of processors and tasks instead of triggers. The main difference is that processors are immediately available, though triggers can be delayed. To simulate the algorithm in this way, one has to add a delayed self-triggering channel to each processor (machine).

\textbf{LITERATURE}

\textbf{Concepcion 88:} A.I. Concepcion, B.P. Zeigler,
DEVS formalism: a framework for hierarchical model development,

\textbf{Dolk-Konsynski 84:} D.R. Dolk, B.R. Konsynski.
Knowledge Representation for Model Management Systems.

\textbf{Genrich 81:} H.J. Genrich, K. Lautenbach.
System Modelling with high-level Petri Nets.


A formal framework for simulation of discrete event systems,
in: D. Murray-Smith, J. Stephenson, R.N. Zobel (eds.).
Proceedings 3rd European Simulation Congress.
Simulation Councils Inc. 1989.

Hee 89c: K.M. van Hee, L.J.A.M. Somers, M. Voorhoeve,
Executable specifications for distributed information systems,

Huber 89: P. Huber, K. Jensen, R.M. Shapiro, Hierarchies in Coloured Petri Nets,

Jensen 87: K. Jensen, Coloured Petri Nets,
in: G. Rosenberg(ed.), Petri Nets: central models and their properties,

Keen 78: P.G.W. Keen, M.S. Scott Morton,
Decision support systems: an organized perspective, Addison-Wesley 1978.

Marca 88: D.A. Marca, C.L. McGowan,

Reisig 85: W. Reisig, Petri Nets, EATCS Monographs on Theoretical Computer Science,


Address of the authors:
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513
5600MB Eindhoven, The Netherlands
Email: wsinmarc@lso.win.tue.nl