A Neutral Model-Solver Protocol for Discrete Systems

H. Prähofer and G. Reisinger and C. Thomas and A. Uhrmacher

November 28, 1996

Contents

1 Introduction and Motivation

Due to its sound theoretical foundation, the DEVS (discrete event system specification) formalism that has been developed by Zeigler [27] and the techniques associated to DEVS have proven to be powerful means for model management and simulation. There are a variety of DEVS-based modeling and simulation systems (e.g., DEVS-Scheme [29], DEVS-CLOS [7], STIMS [15], DEVSim++ [10], and VSIM [22, 2]), including parallel simulator versions [16, 6]. The DEVS formalism has been adapted to combined discrete/continuous modeling [17], to variable structure modeling [4, 12, 5], and to the modeling of symmetric kernel structures (e.g., cellular and hypercube structures [29]). DEVS also has been used to formalize the semantics of conventional discrete event simulation world views, viz. event oriented, activity scanning, and process interaction [27]. The DEVS formalism has also been used in knowledge-based configuration systems [20, 19], for event-based control [30, 18], and for the synthesis of discrete event control algorithms [21].

An open standard for DEVS can be a basis for the development of model libraries. DEVS provides a level of modularity which allows to exchange and integrate models from very different sources. If there is a way to hide the knowledge inside such model libraries, development and distribution of application-oriented model libraries will also be a commercially attractive market. It is easily imaginable that companies even accompany their products with DEVS models like they do it now in the electric circuit market with SPICE and VHDL models.

With these ideas in mind, even the realization of framework software systems seems feasible. Such frameworks could provide the interface, communication and data management services shared by a variety of compatible modeling and simulation tools. Every software tool in the framework would serve a specific task; there would be a variety of modeling tools, sequential and parallel simulators, output analysis and visualization tools, 2D- and 3D-animators, etc.

Based on our experience with DEVS-based modeling and simulation systems and other model-based techniques, we propose a first draft of a DEVS model interchange format in this paper. To emphasize the open and extensible nature of the approach we have taken, we will refer to the interchange format as *OpenDEVS* throughout this report.

With the approach taken here we try to satisfy two sometimes contradictory goals:

- to narrow down the differences between the diverse approaches to discrete event modeling and simulation, and
- to force people to our way of thinking and doing things in as few situations as possible.

2 Review of System Modeling Concepts

2.1 Modular, hierarchical modeling

Modular, hierarchical system modeling [28, 29, 13] is an approach to complex dynamic system modeling where modular building blocks, i.e., system components with a well defined interface in the form of *input* and *output ports*, are coupled in a hierarchical manner to form complex systems. System modeling means modeling complex systems by interface based object composition.

A modular building block is an input/output model or black box model

$$IO =_{df} \langle X,\ Y, \rangle$$

with X a set of inputs and Y a set of outputs. Usually X and Y are structured set owning several input and output ports.

In system modeling, one distinguishes between atomic and coupled models. While an atomic model specifies its internal structure in terms of its set of states and state transition functions (see next sections), a coupled model's internal structure is specified by its components and coupling scheme, i.e., how ports are connected.

A coupled model is a structure

$$CM =_{df} \langle X, Y,\ D,\ C,\ Select \rangle$$

with X and Y being the input/output interface specification, D the set of component systems, C the set of couplings, and $Select : 2^M \leftarrow M$ the tie-breaking function to arbitrate between components having equal event times [1].

Modularity allows for setting up bases of reusable building blocks which can be plugged into a system through their well defined input and output interface.

2.2 DEVS-based atomic model specification

The DEVS formalism, being an application and computer implementation independent system formalism for discrete event modeling, provides a basis for system based, multi-formalism simulation modeling. In the following we review the DEVS-based concepts for multi-formalism modeling.

In DEVS modeling the state behavior is event like. The input and output trajectories are *DEVS-segments*, i.e., events are defined for a finite number of points in time and between the event times there is the none event (Fig. 1 (a)). In distinction to DEVS-segments, state trajectories in DEVS are piecewise constant (Fig. 1 (b)) and trajectories of continuous systems are continuous or piecewise continuous (Fig. 1 (c)).

[1] only used in discrete event systems

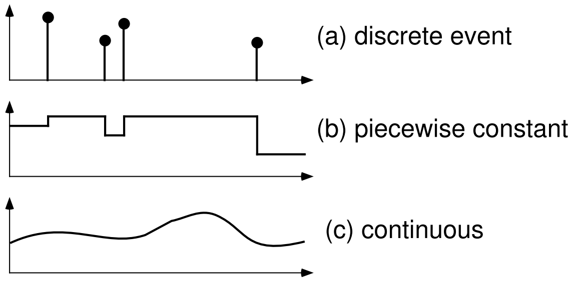

Figure 1: (a) DEVS-segment, (b) piecewise constant, and (c) continuous trajectories.

In the following we review the DEVS formalism and various exentsion thereof which will serve as a basis for a neutral model specification.

2.2.1 DEVS formalism

DEVS have eventistic behavior. It reacts to *external input events* by the *external state transition function* and to *internal, time scheduled events* by an *output event function* and the *internal transition function.* Time scheduling of internal events is accomplished by the *time advance function* which defines the time span from one event to the next internal event.

Formally an atomic DEVS is a structure [28, 29]

$$DEVS =_{df} \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$$

where $X = \{(x_1, x_2, ..., x_m)|x_1 \in X_1, x_2 \in X_2, ..., x_m \in X_m\}$ is the structured set of inputs with m input ports x_i, $Y = \{(y_1, y_2, ..., y_p)|y_1 \in Y_1, y_2 \in Y_2, ..., y_p \in Y_p\}$ is the structured set of outputs with p output ports y_i, $S = \{(s_1, s_2, ...s_n)|s_1 \in S_1, s_2 \in S_2, ..., s_m \in S_m\}$ is the structured set of sequential states, $\delta_{ext} : Q \times X \to S$ is the external state transition function, with $Q = \{(s, e) : s \in S, 0 \leq e \leq ta(s)\}$ the set of total states, $\delta_{int} : S \to S$ is the internal state transition function, $\lambda : S \to Y$ is the output event function, $ta : S \to \mathcal{R}_0^+ \cup \{\infty\}$ is the time advance function.

2.2.2 Value coupling and conditional events

Input and output trajectories in DEVS are event like only. Events are the only means in DEVS through which components can communicate. To ease simulation modeling, we extend DEVS to allow value couplings. Input and output *variables* are introduced and can be connected to transmit variable values from one component to another. These input and output variables can be coupled in the same way as input and output ports. The trajectories however are piecewise constant or continuous as shown in Fig. 1(b) and (c).

Additionally, we allow the internal events to be conditional, i.e., the occurrence of an internal event depends on an *internal event condition.* Only when the time given by the time advance function has elapsed *and* the internal event condition evaluates to true, the event is executed, otherwise the event is *due.* This scheme corresponds to the *activity scanning* model [27] for event processing known from other simulation approaches.

We define a structure *extended DEVS* in the following way:

$$extDEVS =_{df} \langle X, X_c, Y, Y_c, S, \delta_{ext}, C_{int}, \delta_{int}, \lambda, ta, \lambda_c \rangle$$

where X, Y, S are the same as in conventional DEVS, $X_c = \{(x_{c1}, x_{c2}, ...) | x_{c1} \in X_{c1}, x_{c2} \in X_{c2}, ...\}$ is the structured set of value inputs with input variables x_{ci}, $Y_c = \{(y_{c1}, y_{c2}, ...) | y_{c1} \in Y_{c1}, y_{c2} \in Y_{c2}, ...\}$ is the structured set of value outputs with output variables y_{ci}, $\delta_{ext} : Q \times X_c \times X \rightarrow S$, $\delta_{int} : S \times X_c \rightarrow S$, $\lambda : S \times X_c \rightarrow Y$, $ta : S \times X_c \rightarrow \mathcal{R}_0^+ \cup \{\infty\}$, $\lambda_c : S \times X_c \rightarrow Y_c$ is the *value output function* to define the output variable values, and $C_{int} : S \times X_c \rightarrow \mathcal{B}$ is the *internal condition function* for conditioning the execution of internal events.

2.2.3 Combined discrete/continuous modeling

Differential equation specified systems (DESS) are a dual concept to DEVS for continuous modeling. They define a modular interface in the form of a set of real valued input and output variables. The dynamic behavior of the multidimensional state space is defined by a derivative function. DESS can be coupled modularilly and hierarchically in the same way as DEVS.

Many real world modeling problems require the integration of continuous and discrete event modeling elements, i.e., a combined discrete/continuous modeling and simulation methodology. The DEVS formalism therefore has been extended to allow multi-formalism combined modeling and simulation. The DEV&DESS formalism [14, 13] has been introduced as a universal formalism for combined discrete/continuous modeling. It is defined in the following way [2]:

$$DEV\&DESS =_{df} \langle X, X_c, Y, Y_c, S, \delta_{ext}, C_{int}, \delta_{int}, \lambda, f, \lambda_c \rangle$$

where X, X_c, Y, Y_c, δ_{ext}, C_{int}, δ_{int}, λ, and λ_c are the same as in extended DEVS, $S = S_d \times S_c$ is the Cartesian product of discrete states and continuous states with $S_c = \{(s_{c1}, s_{c2}, ...) | s_{c1} \in S_{c1}, s_{c2} \in S_{c2}, ...\}$, $f : S \times X_c \rightarrow S_c$ is the derivative function.

The derivative function together with the value output function specify the continuous behavior. The external state transition function and the conditional internal state transition are used to model the event part. In

[2]Here we use a slightly simplified version of the original formalism in [14] which does not allow for time scheduled events.

particular, the internal condition and state transition function is used to model *state events*, i.e., events which occur upon conditions specified on the continuous states and inputs becoming true, typically, when continuous state variables reach certain thresholds.

2.3 Variable structure modeling

Variable structure models are models that entail in their description the possibility to change their own structure. Referring to the epistemological hierarchy of systems introduced by Klir [11], variable structure models can be perceived as meta systems applied to structure systems. The question rises how those meta reasoning about structures can be realized in an object-oriented modeling approach.

DEVS defines models either as atomic models or as coupled models, as do other simulation systems [3]. Thereby, the behavior of a coupled model is typically completely determined by the behavior of the atomic models. This reductionistic modeling approach in combination with the ignorance of models referring to the environment they are embedded in inhibits some problems for the realization of variable structures [26].

One straightforward approach to overcome these difficulties is to extend coupled models with atomic behavior and give them access to their set of components and coupling structure. Coupled models that possess state and transition functions of their own are able to access and control their constitutive components and the couplings that exist among those. Thereby, a change of structure is directed from top to bottom of the compositional hierarchy.

A variable structure coupled model therefore is defined in the following way:

$$CM =_{df} \langle X,\ Y,\ D,\ C, Select, S, \delta_{int}, \delta_{ext}, \lambda, ta\rangle$$

with the respective interpretation of the atomic DEVS and coupled model definitions (see sections 2.1 and 2.2).

Thus, transition functions that describe structural changes constitute an integral part of a coupled model. It is responsible for changing the set of constitutive components as well as the coupling that exist among them [24], [1], [9].

A different approach to variable structure modeling is described in [25]. This approach focuses on the idea of agents manipulating their environment based on their knowledge about the environment. This knowledge is represented by means of an abstract structural model (*endomorphic agents*[29], [25]), which constitutes a reflection of composition and coupling realized by coupled models.

2.4 DEVS-representation of discrete event simulation strategies

In this section we make the connection from the system theoretical DEVS approach to the three conventional world views known in discrete event simulation. We do this by introducing structured DEVS systems to express event scheduling, activity scanning and process interaction models and then show how these models map to DEVS systems.

2.4.1 Event scheduling models

Event oriented models work with a prescheduling of all events and there is no provision for activating events by tests on the global state.

For event scheduling model definition we introduce so-called *event scheduling DEVS* models [27]. Those are characterized by owning several components which show individual event behavior. In distinction to coupled DEVS, coupling in structured DEVS is not modular but coupling between the individual components is done by providing for each component a set of other components influencing it.

Formally a event scheduling DEVS is a structure [27]

$$esDEVS =_{df} \langle X, Y, D, \delta_{ext}, Select \rangle$$

where X, Y, are the same as in conventional DEVS. Each $d \in D$ is a component

$$esD =_{df} \langle S_d, I_d, \delta_d, \lambda_d, ta_d \rangle$$

with S_d is the set of sequential states of d, $I_d \subset D$ is the set of influencing components, $\delta_d : \times_{i \in I_d} Q_i \to \times_{i \in I_d} S_i$ is the internal state transition function, with $Q_i = \{(s_i, e_i) : s_i \in S_i, 0 \leq e_i \leq ta(s)\}$ being the total state of a component i with e_i the time elapsed since the last event in i, $\lambda_d : \times_{i \in I_d} Q_i \to Y$ is the output event function, $ta_d : \times_{i \in I_d} Q_i \to \mathcal{R}_0^+ \cup \{\infty\}$ is the time advance function. $\delta_{ext} : \times_{i \in D} Q_i \to \times_{i \in D} S_i$ is the external state transition function of the structured DEVS and $Select : 2^D \to D$ is the tie-breaking function.

A event scheduling DEVS works in the following way: Internal events are scheduled individually by each component $d \in D$ by the individual time advance functions ta_d. Upon occurrence of an event in one of the components, this event is executed resulting in state changes through the internal transition function δ_d and eventually output events for the event scheduling DEVS defined in the output function λ_d of d. Note, that the state transition function may depend on and change any state of the influencing components. When events in different components are scheduled at the same time, the *Select*-function is used to arbitrate among them. External events at the structured system's input interface are handled by the external state transition function δ_{ext} and may result in state changes in any of the components.

Formally we associate the following atomic

$$DEVS = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$$

with an

$$esDEVS = \langle X_s, Y_s, D, \delta_{s,ext}, Select \rangle$$

with $X = X_s$, $Y = Y_s$, $\delta_{ext} = \delta_{s,ext}$, $S = \times_{d \in D} Q_d$. we define $ta(s) = min\{\sigma_d = ta_d(..., q_i, ...) - e_d : d \in D\}$ and we define the next scheduled component $i* = Select(\{d : ta_d(..., q_i, ...) - e_d = ta(s)\}$; then $\delta_{int} = \delta_{i*}$ and $\lambda = \lambda_{i*}$, i.e. the internal state transition of the the DEVS is defined by the state transition of the next scheduled component $i*$.

2.4.2 Activity scanning models

In contrast to the event scheduling approach, in the activity scanning approach events are conditioned on a contingency test. Whenever the event condition becomes true, the event is ready to be activated [3].

For activity scanning model definition we introduce so-called *activity scanning DEVS* which is similar to an event scheduling DEVS but has components working with the activity scanning approach.

Formally a activity scanning DEVS is a structure

$$asDEVS =_{df} \langle X, X_c, Y, Y_c, D, \delta_{ext}, Select \rangle$$

where X, X_c, Y, Y_c are the same as in extended DEVS. Each $d \in D$ is a component

$$asD =_{df} \langle S_d, I_d, \delta_d, \lambda_d, C_d \rangle$$

with S_d, I_d, δ_d, λ_d, δ_{ext}, and *Select* the same as in event scheduling DEVS. $C_d : \times_{i \in I_d} \times Q_i \rightarrow \mathcal{B}$ is the event condition function.

An activity DEVS works in the following way: Each component has its own state transitions whose occurrence is conditioned by the event condition function. Whenever the event condition becomes true, the component is ready to be activated. The *Select*-function is used to arbitrate between components having true event conditions simultaneously.

Formally we associate the following extended DEVS

$$extDEVS = \langle X, X_c, Y, Y_c, S, \delta_{ext}, C_{int}, \delta_{int}, \lambda, ta, \lambda_c \rangle$$

with a

$$asDEVS = \langle X_s, X_{s,c}, Y_s, Y_{s,c}, D, \delta_{s,ext}, Select \rangle$$

[3] Activity scanning approaches have to be combined with an event scheduling approach to be operational. However, we present the pure form here, to elucidate the differences.

with $X = X_s$, $Y = Y_s$, $X_c = X_{s,c}$, $Y_s = Y_{s,c}$, $\delta_{ext} = \delta_{s,ext}$, $S = \times_{d \in D} Q_d$. we define $i* = Select(\{d : C_d(..., q_i, ...) = true\}$; then $C_{int} = C_{i*}$, $\delta_{int} = \delta_{i*}$ and $\lambda = \lambda_{i*}$, i.e. the internal state transition of the DEVS is defined by the state transition of the next scheduled component $i*$.

2.4.3 Process interaction models

Process interaction world view results from a combination of event scheduling and activity scanning approaches. They allow to preschedule events into the future as well as conditioning event occurrence upon contiguency tests.

To formally express process interaction models, we introduce the *process interaction DEVS* formalism as follows:

$$piDEVS =_{df} \langle X, X_c, Y, Y_c, D, \delta_{ext}, Select \rangle$$

with each $d \in D$ is a component

$$piD =_{df} \langle S_d, I_d, \delta_d, \lambda_d, C_d, ta_d \rangle$$

with interpretation analogous to components of event scheduling DEVS esD and activity scanning DEVS asD.

In process interaction DEVS, events in components can be prescheduled by the time advance function. When this time elapses, the condition function is employed to see if this event can be executed. Otherwise the event is due (Note that events which should not be scheduled in the future but are ready to be executed as soon as the event condition becomes true should be scheduled in 0 time units.)

Formally we associate the following extended DEVS

$$extDEVS = \langle X, X_c, Y, Y_c, S, \delta_{ext}, C_{int}, \delta_{int}, \lambda, ta, \lambda_c \rangle$$

with a

$$piDEVS = \langle X_s, X_{s,c}, Y_s, Y_{s,c}, D, \delta_{s,ext}, Select \rangle$$

with $X = X_s$, $Y = Y_s$, $X_c = X_{s,c}$, $Y_s = Y_{s,c}$, $\delta_{ext} = \delta_{s,ext}$, $S = \times_{d \in D} Q_d$; We define (1) $ta(s) = 0$ if $\exists d \in D : ta_d(..., q_i, ...) - e_d \leq 0 \wedge C_d(..., q_i, ...)$ or (2) $ta(s) = min\{\sigma_d = ta_d(..., q_i, ...) - e_d : d \in D \wedge \sigma_d > 0\}$; we define the next scheduled component $i* = Select(\{d : ta_d(..., q_i, ...) - e_d \leq ta(s) \wedge C_d(..., q_i, ...)\})$; then $C_{int} = C_{i*}$, $\delta_{int} = \delta_{i*}$ and $\lambda = \lambda_{i*}$, i.e. the internal state transition of the DEVS is defined by the state transition of the next scheduled component $i*$.

3 OpenDEVS Model Exchange Format

3.1 Object oriented concepts

OpenDEVS is based on object oriented concepts which augments system modeling concepts with object orientations. Object orientation means:

- Object identity: Objects are identified purely by their existence.
- Encapsulation: Objects have a well defined interface, the interior is hidden.
- Inheritance: Object definitions, viz. classes, can be organized in classification hierarchies where the inferior classes inherit the properties of the superior.
- Polymorphism: Polymorphic object operations have different realizations depending on the class of the object.

Zeigler [29] compares modular hierarchical system modeling and the object oriented paradigm. While both share various concepts they also show major differences. System models as well as objects use the concept of internal states. However, objects are not dynamic systems in the sense of system theory which show dynamic, concurrent behavior specified over a time base. Nor are objects modular units in the above sense. In particular, objects lack an explicit output interface. The output interface is only given implicitly by the method calls to other objects. Therefore, an object has to have knowledge of the other objects with whom it communicates. In contrast to that, a system model only defines an output port, the communication with other components is defined independently on the higher coupled system level. Object oriented systems also are not hierarchical in the above sense.

While polymorphic operations in system models are given by equal interface for different internal model structure and behavior, inheritance is not defined for system models. However, as depicted below, inheritance are employed advantageously for system modeling.

3.2 Interface specification

Model interfaces should be seen as controlled gateways in walls which are hiding the interior of a model [23]. Any attempt to get information about the hidden model contents or to influence the internal representation should be prohibited.

Interface declarations describe the "surface" of a model (i.e., everything what is visible from the outside). An interface declaration consists of a description of the ports which may be used to couple the model and its environment, and a description of the variables, in particular parameters used to parameterize the model (parameters are set prior to the simulation and are constant during the simulation) which are accessible for the user of the model. In accordance to the principle of *information hiding*, nothing is revealed about the models implementation. Therefore, no internal (private) state variables nor functions or components are to be mentioned in the interface description. Similarly, the keyword `interface` introducing an interface declaration gives no hint about the models type and implementation.

A model interface specifications represents an polymorphic interface and hides the interior realization of the model from outside access. A natural consequence of that is to separate the interface from the model realization, i.e., one interface can serve various, diverse model realizations.

An interface declaration consists of the declaration of available input and output ports including the message types allowed on these ports, parameters and state variables of the model which can be set by the user.

```
interface OneInputGate {
  inports {
    input1 { type{ boolean } };
  }
  outports {
    output { type{ boolean } };
  }
}
```

An interface declaration may be inherited to form another interface declaration. In the example below, the declaration of `OneInputGate` is extended by another input port in order to form the declaration of a gate with two input ports. The inherited ports are visible and usable in the derived model.

```
interface TwoInputGate:OneInputGate {
  inports {
    input2 { type { boolean } };
  }
}
```

If models differ only in their implementation from their parent model, their interface can be declared by just inheriting the interface of the parent model. Here, the interfaces of the abstract models [4] `OneInputGate` and `TwoInputGate` are inherited by concrete models of binary gates.

```
interface NotGate:OneInputGate;
interface AndGate:TwoInputGate;
interface OrGate:TwoInputGate;
interface XorGate:TwoInputGate;
```

From the interfaces introduced so far, an inheritance scheme shown in figure 2 can be extracted.

Models also may provide output and input variable interfaces (see section 2.2.2. They allow for value couplings.

A model of a server component might have the following interface:

[4] Abstract models are models without a completely implemented functionality which cannot be used for simulation.

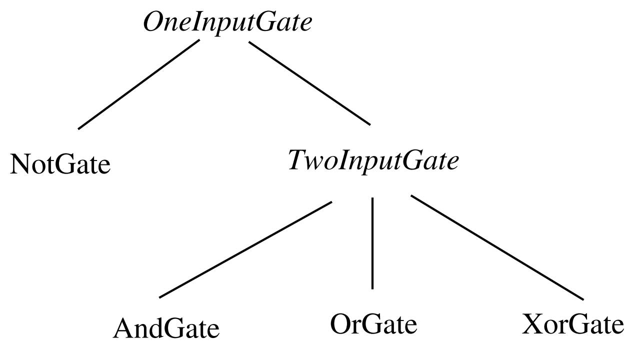

Figure 2: Interface-oriented inheritance scheme (abstract model identifiers shown in italics)

```
interface Server {
inports {
    in { type { Job } };
outports {
    out {type { Job } };
invars {
    nextfree {type { boolean } };
outvars {
    free {type { boolean } };
  }
}
```

In this example, the interface declaration of `Server` makes the variable `status`, which gives the free or busy status of the Server visible and accessible. The input variable `nextfree` is intended to provide the information of the status of the next component.

Interface-based model classification Interface declaration and interface inheritance are strongly interconnected. Using interface-oriented classification, a hierarchy of model classes is built up solely on implementation independent information about models: their interfaces, i.e., their ports and variables the data types they do handle.

A model can be derived from another model, if its interface comprises the interface of the parent model. This condition is met if the model has at least the same input ports and these input ports can handle at least the same messages, and that it has at least the same output ports, and these output ports do not handle a greater variety of message types than that of the parent model.

3.3 Coupled Model Specification

In OpenDEVS a coupled system is specified by defining as variables a set of components and a set of couplings. The example below shows couplings of different boolean gates (Note: The interface of this model is specified above).

```
coupled XorGate {
  variables {
    set<component> components =
    { AndGate a1, AndGate a2,
      NotGate n1, NotGate n2,
      OrGate o1
    };
    set<coupling> couplings =
    { (input1, n2.input1), (input1, a1.input1),
      (input2, n1.input1), (input2, a2.input2),
      (n1.output, a1.input2),
      (n2.output, a2.input1),
      (a1.output, o1.input1),
      (a2.output, o1.input2),
      (o1.output, output)
    };
  }
}
```

In order to be able to specify also structure-variant systems, neither the set of components nor the coupling specification should be seen as constant over time. The set of components and couplings can be manipulated by set theoretic operations.

3.4 Atomic Model Behavior Specification

As stated in the introductory section, OpenDEVS shall be a *standardized* and *open* means for model exchange. Therefore we would like to have all systems using OpenDEVS being able to handle the most basic implementation specifications, but we also want to leave room for extensions and modifications as outlined in section 2.2. In practice, the names of a model type can be used as the keyword opening the implementation specification. DEVS-based software systems which support additional model types will recognize additional model type keywords, while all others reject implementation specifications containing such keywords. However, for the sake of interchangeability of models, the variety of supported model types should be kept to a minimum[5].

The state space and parameters of atomic models are specified by a set of `variables` and `parameters` of different types. Behavior specification is dependent on the particular formalism in hand and will be discussed in detail below.

An example a specification of an atomic devs model with variables looks as follows:

```
atomic-devs OrGate {
   parameters {
      real delay-time;
   }
   variables {
      boolean m;
   }
   ...
}
```

3.4.1 Event behavior specification

Refering to the discrete model types discussed in section 2.2, we have to provide specification constructs to specify:

- external events stemming from external inputs,
- internal time events scheduled by the time advance function, and
- conditional events depending on conditions.

In the following we discuss two forms for event specifications.

For the specification of event behavior, two styles seem to be appropriate:

- Use *situation/activity* pairs to describe, which action has to be taken when a certain situation is given (i.e., a certain state is reached).
- Use *precondition/postcondition* pairs to describe, which state has to be reached after the specified function was applied in a certain situation.

These different specification styles are discussed in the following.

[5]This goal may be met by introducing a standardizing committee, or just by early and intensive communication among the scientists interested in DEVS.

Situation/activity pairs: The specification style using situation/activity pairs is very much implementation oriented in that it suggests a special way of implementing the specified functionality. Each event is specified by defining when the event occurs and the actions to take at the event. Therefore, for the three different event types we use the following notations:

```
external(port, x) [when (cond)]
    { action;  ... }
```

External events are identified using the keyword `external` followed by the port and input value in brackets. Optionally there is a condition signified by the keyword `when`. These two parts together form the event condition which have to be fulfilled for the event to occur. The body is formed by an arbitrary number of actions (which are specified in C++ notation.)

```
time(time-span) [when (cond)]
    { action;  ... }
```

Internal events are identified using the keyword `time` which is followed by the time span to the time event. Again optionally there might be an additional condition. The body is formed by a sequence of actions.

```
when (cond)
    { action;  ... }
```

Also pure conditional events can be specified. They only have a event condition identified by the keyword `when`.

With this notation, a specification of a simple server component could look as shown below. The `continuous` clause thereby specifies the variable output value for `free` and will be discussed in detail in the section about combined modeling.

```
interface SimpleServer: Server {
  parameter {
    real service-time;
  }
};

atomic-devs SimpleServer {
  variables {
    Job m;
    boolean busy;
}
events {
   external(in, x) when (busy = false)
```

```
        { m := x; busy := true; }
   external(in, x) when (busy = true)
        { error; }
   time(service-time) when (busy = true & nextfree = false)
        { out := m; m := null; }
   }
continuous {
   if (true) { free := ! phase; }
}
```

This style of model specification eases the transformation of the specification into the simulation system target code (e.g., C++ or LISP). However, this style has serious drawbacks when it comes to symbolic manipulation or analysis of the specification. In order to analyze the behavior of the model specified above, the software would need to know about the implementation of the invoked subroutines.

Precondition/postcondition pairs: The drawbacks described above lead us to suggest the use of the precondition/postcondition style of specification. Using this style, an action (state change or alike) is specified by describing the preconditions for the action to be invoked (i.e., the state the model has to have reached) and the result of that action in terms of the state reached after the invocation of the action.

The notation for the precondition/postcondition pairs is borrowed from the Z-Notation: One thinks of having two sets of state variables of one atomic object: the state variables bound to the values they had before the action was invoked, and the state variables bound to the values they will have after action invocation. While the first set is referred to with the usual variable names (e.g., `var1`), apostrophes appended to these identifiers indicate that variables belong to the second set (e.g., `var1'`).

In preconditions, only variables of the first set can be used. Postconditions usually specify a balance condition, which has to be fulfilled. It does not specify, which functions have to be invoked to meet this balance condition. The following example illustrates this:

```
atomic-devs SimpleServer {
variables {
    Job m;
    boolean busy;
}
events {
   external(in, x) & (busy = false) ->
        { m' = x; busy' = true; }
   external(in, x) & (busy = true) ->
```

```
        { }
    time(service-time) & (busy = true & nextfree = false) ->
        { out' = m; m' = null; busy' = false; }
  }
}
continuous {
  if (true) { free := ! phase; }
}
```

Inheritance. Implementation specifications build up an inheritance tree on their own, which is independent from the inheritance tree built up using the rules of interface-oriented inheritance. The derivation of a model from a base model is expressed in the same way as for interface declarations. Both, state variable definitions and defined functions are inherited.

In the following example, a interruptable server is realized employing independent interface and implementation inheritance.

```
interface InteruptableServer : SimpleServer
inports {
    interupt { type { boolean } };
  }
}

atomic-devs InteruptableServer: SimpleServer {
variables {
boolean interrupted;
}
events {
   external(interrupt, _) when  (busy = true)
       { out := m; m := null; busy = false;}
   }
}
```

3.4.2 Combined discrete/continuous model specification

The DEV&DESS extension of DEVS, as discussed in section 2, allows for combined modeling. OpenDEVS allows for specification of DEV&DESS through the following concepts.

Input and output variables of type `double` are the means for continuous interface specification. Accordingly, variables of type `double` can represent continuous state variables. The state event behavior of DEV&DESS is specified by conditional events.

For the specification of continuous variables we introduce a `continuous` section in atomic model specification. Here the specification of the derivatives of the continuous state variables by first order differential equations is accomplished. Secondly, additional continuous variables and continuous output variables can be defined by algebraic expressions.

The specification of the derivatives is done in the following form:

```
d/dt (variable) := expression;
```

The continuous model specification part consists of a set of such clauses which are contained in an `if` clause representing the condition when the definitions apply. In the following an example of a continuous model specification is shown. It is a specialization of a Server model where the service of the jobs is accomplished by a continuous filling process.

```
interface FillingServer: Server {
    invars {
        double inflow;
}

atomic-devs FillingServer {
variables {
    Job m;
    boolean busy;
    double content;
    double full-level;
}
events {
   external(in, x) when (busy = false)
        { m := x; busy := true; }
   external(in, x) when (busy = true)
        { error; }
   when (content > full-level & nextfree = false)
        { out := m; m := null; content := 0; }
   }
continuous {
   if (true) { free := ! busy; }
   if (busy) {d/dt (content) := inflow;}
   if (! busy) {d/dt (content) := 0;}
  }
}
```

3.5 Variable Structure Model Specification

Variable structure systems combine the specification style of coupled models presented in section 3.3 and of atomic models presented in section 3.4. Additionally we introduce special constructs to manipulate the set of components and couplings. Similar to atomic models specification we distinguish two specification styles, viz. situation/activity and precondition/postcondition pairs.

Situation/activity pairs: In the situation/activity style we employ constructs to add and delete components and couplings. Those are as follows:

```
addcoupling(coupling);
deletecoupling (coupling);
```

adds and deletes the coupling specified.

```
addcomponent (componentname);
addcomponent (componenttype componentname);
deletecomponent (componentname);
```

adds and deletes the component specified, whereby with `addcomponent (componenttype componentname)` a new component of type `componenttype` and with name `componentname` is created. With a statement `deletecomponent (componentname)` also all the coupling to and from the component are deleted.

Additionally to these statements, predicates

```
existscomponent (componentname);
existscoupling (coupling);
```

are used to test for existence of components and couplings.

The following example is used to show the specification of variable structure models. The variable structure model specifies a gate that is able to switch from "AND" behavior to "NAND" behavior by adding an inverter gate to its inner structure (figure 3). The structure changes are event-based, i.e., upon a receipt of an external event on input port `do_negate` the structure switches.

In the specification, both, set of components and couplings are treated as variables, and can therefore be altered by state transition functions. In the following, the specification of this variable structure model is shown employing the situation/activity style of specification.

```
interface And/NandGate:TwoInputGate {
  inports {
    do_negate { type { boolean } };
  }
}

var_coupled And/NandGate {
```

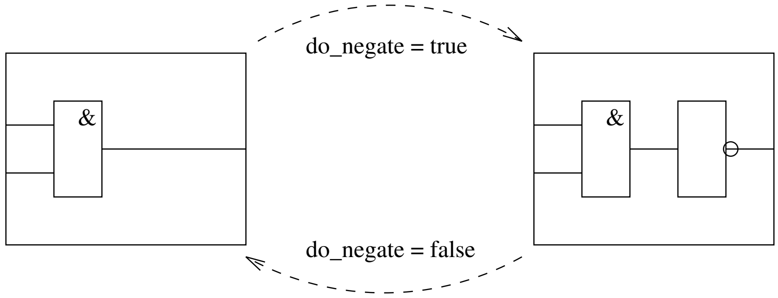

Figure 3: Switching structure gate changing from AND to NAND behavior and back

```
variables {
  set<components> components = { AndGate a };
  set<coupling> couplings =
  { (input1, a.input1),
    (input2, a.input2),
    (a.output, output)
  };
}
events {
  external(do_negate, x) when ((x = true) & !(existscomponent n))
     { addcomponent (NotGate n);
       deletecoupling(a.output, output);
       addcoupling(a.output, n.input);
       addcoupling(n.output, output);
     }

  external(do_negate, x) when ((x = false) & (existscomponent n)))
     { deltecomponent (n);
       addcoupling(a.output, output);
     }
  }
}
```

Precondition/postcondition pairs: In the precondition/postcondition style of specification we use basic set theoretic operations to specify changes in the set of components and set of couplings. Those are the following:

```
set1 union set2
set1 intersection set2
set1 minus set2
x member_of set
```

with obvious semantics. For new component creation the construct

```
(componenttype componentname)
```

is used.

In the following, the specification of the above variable structure model example is shown employing the precondition/postcondition style.

```
var_coupled And/NandGate {
  variables {
    set<components> components = { AndGate a };
    set<coupling> couplings =
    { (input1, a.input1),
      (input2, a.input2),
      (a.output, output)
    };
  }
  events {
    external(do_negate, x) & ((x = true) & !(n member_of components)) ->
        { components' = components union {(NotGate n)};
          couplings' = couplings minus {(a.output, output)}
                       union { (a.output, n.input), (n.output, output) };
        }
    external(do_negate, x) & ((x = false) & (n member_of components)) ->
        { components' = components minus {n};
          couplings' = couplings union {(a.output, output)} };
        }
    }
  }
```

3.6 Event Scheduling, Activity Scanning, Process Interaction Model Specification

To represent conventional event scheduling, activity scanning and process interation models, we introduce a special type of model specification, called `structured`. These models are similar to coupled models in that they have a set of components, which, however, are not modular, and differ to coupled models in that they specify the couplings, viz. influences, between the components by setting variables in components to influencing components or output ports. Additionally structured models can have state variables and event behavior, in particular, external events. The component of structured models are identified by keyword `atomic-pi`.

```
interface TwoServer {
inports {
```

```
    in { type { Job } };
outports {
    out {type { Job } };
  }
}

structured TwoServer {
  variables {
    set<components> components =
                        { BasicServer s1, BasicServer s2, Queue q };
    set<influencers> influencers =
    { (s1.queue, q),
      (s2.queue, q),
      (s1.out, out),
      (s2.out, out)
    };
  }
  events {
    external(input, x)
        { q.insert (x); }
    }
}
```

Components of structured systems are built similar to atomic devs with the difference that they do not have an interface specification. However, the same type of event behavior specification can be used, especially the `when` clause is used to specify the conditions typical for the activity scanning approach. In the following a definition of the `BasicServer` component is given.

```
atomic-pi BasicServer {
   variables {
     Queue queue;
     Output out;
     Job m;
     boolean busy;
     real service-time;
   }
   events {
    when ((busy = false) & ! queue.empty())
         { m := queue.pop(); busy := true; }
    time (service-time)
         { out := m; m := null; busy := false; }
   }
}
```

3.7 Models, Prototypes, and Model Instances

3.7.1 Prototypes and model initialization

A model is only completely specified, if all those state variables are assigned a value, which may influence its behavior. Therefore, the definition of initial state variable values is an important part of the model specification.

Often, models vary only in the values of their parameters and state variables, while the rest of the model specification is identical (e.g., a conveyor belt parameterized for a length of 10m and another one parameterized for 30m length). However, these models may have completely different behavior and should therefore be treated as completely different. This can be taken into account by separating the functional model specification from the definition of their initial state variable values, but treating the combination of functional specification and initialization values as an entity. For such entities, Thomas [24] introduced the term *model prototype*.

Prototypes may also be referred to as "abstract model types". A complete prototype comprises (see figure 4):

- the (sub-) formalism to be used during simulation,
- the functional specification of the model (i.e., the specification of its interface and implementation, including, e.g., variables and functions or the models inner structure),
- the initial values of all variant model elements (usually variables, but also, e.g., the structure of a structure-variant coupled model).

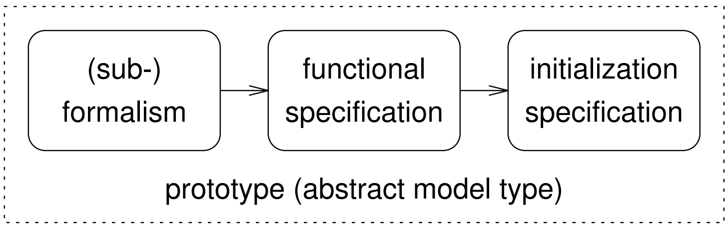

Figure 4: Hierarchy of specifications forming a prototype

3.7.2 Initialization value definition

Initialization information is used to overload the values of parameters and initial values of state variables that is defined in the implementation description of atomic models. Here, too, an inheritance scheme can be established in that sense that initialization informations are cascaded. Then, the initial values as defined in the atomic model implementation are inherited by the first initialization information, the result from this overloading act is inherited by the next initialization information in line and so forth.

Following a simplistic approach, one can require that all state variables need to be assigned a default value. This can be proven easily by comparing the model implementation specifications with the initialization value definitions. However, a more elaborate approach may use the notion of sorts of variables as explained in the beginning of this paper. Then, only those state variables need to be assigned a default value which are used as input to state transition and output functions. These variables can be distinguished from supervisory state variables by scanning the functional part of the model implementation specification. For both approaches it can be assured that no model misbehavior stems from uninitialized state variables.

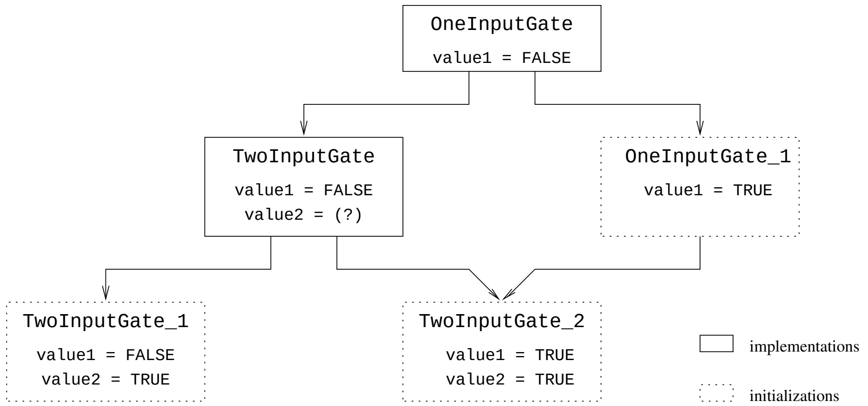

Figure 5: Inheritance among implementations and initializations

Initialization is a part of the specification that forms an individual inheritance tree (to some extend independent from the implementation specification inheritance tree). Therefore, initialization information is kept in an independent specification structure, that is indicated by the keyword `initialization`:

```
initialization OneInputGate_1( OneInputGate )
{ value1 = TRUE; }
```

This initialization of `OneInputGate_1` refers to `OneInputGate` and overrides the initial value for `value1` that has been given there (see figure 5). For the specification of initial values for a derived model implementation, one can imagine two distinct versions. The first is straightforward and works as the one above:

```
initialization TwoInputGate_1( TwoInputGate )
{ value2 = TRUE; }
```

TwoInputGate_1 re-initializes the initial state variable values that are inherited from TwoInputGate (value1 = FALSE, value2 is undefined). Therefore, the initial state of TwoInputGate_1 is (FALSE, TRUE).

However, inheritance features can also be used for initialization specifications as well:

```
initialization TwoInputGate_2( TwoInputGate ) : OneInputGate_1
{ value1 = TRUE; }
```

Here, the initial state inherited from TwoInputGate is partially overwritten by the initialization specification OneInputGate_1. Naturally, that results in an initial state (TRUE, TRUE) for TwoInputGate_2.

4 Discrete Solver Realization

4.1 Polymorphic Simulator Protocol

In the following we propose a solver realization for a mulitformalism simulation environment based on DEVS. This solver, relies on a polymorphic simulation protocol similar to the abstract simulator concept developed by Zeigler [28].

In the discrete solver realization, each component in the model owns a simulator object which is responsible for its correct execution. For each individual model type one individual simulator type is defined which realizes the special simulation algorithm underlying the model type. The different simulators, however, obey to the same abstract polymorphic protocol. The simulator objects are coordinated and activated by a superior coordinator (or several coordinators in the distributed case) which rely on the abstract protocol.

Figure 6 depicts the protocol interface of a simulator object. It consists of the following elements:

- An message interface `init (t)` to initialize the simulator with time `t`.
- An value output `time-of-next-event/due` which provides the time of next event, in the case of events scheduled in the future, and an due event indicator, meaning that that there is a due event in scope of the respective model.
- An message interface `do-internal-event (t)` activating the simulator to execute the next event at time `t` within its scope.
- An message interface `do-input-event (t, port, event)` activating the simulator to handle an input event `event` on an input port `port` at time `t`.

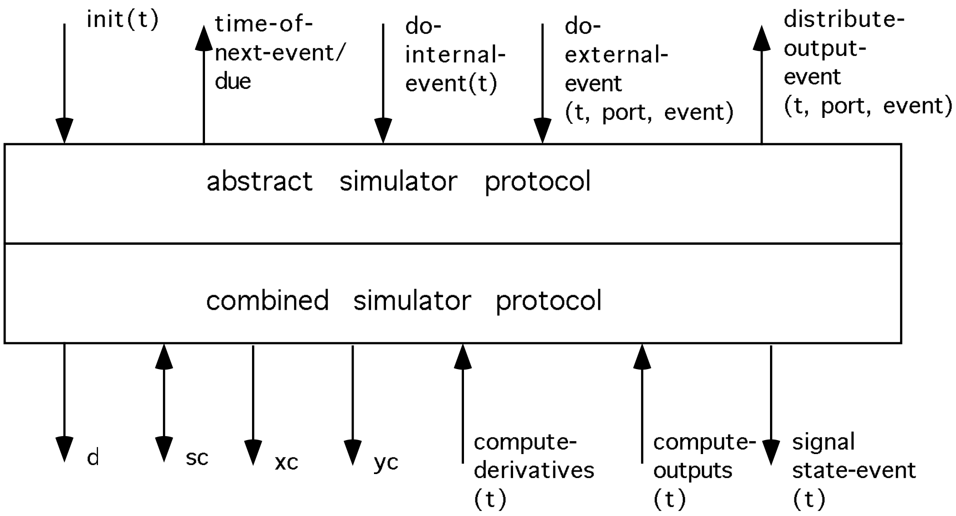

Figure 6: Abstract simulator protocol

- An message output interface `distribute-output-event (t, port, event)` signaling the superior coordinator to distribute the output event `event` at port `port`.

Additionally, simulators for combined discrete/continuous models have to provide the following interface:

- An message interface `compute-derivatives(t)` to compute the derivatives of the continuous state variables.
- An message interface `compute-outputs(t)` to compute the values of the continuous output variables.
- A data output `d`, `sc`, `xc`, and `yc` interface which provides the derivative, continuous states and continuous input and output information, respectively.
- A data input interface `sc` to set the continuous state variables.
- An message output interface `signal-state-event(t)` to report the occurrence of a state event at time `t`.

A superior *coordinator* (Fig. 7) is responsible for the coordinated execution of the simulators within its scope. For the pure discrete case it typically manages an event queue and time of next event and realizes a classical event handling algorithm. It has to provide a protocol dual to the simulators.

In the combined discrete/continuous case, the solver additionally has a numerical integrator which uses the protocol of the combined model simulators to realize various numerical integration algorithms.

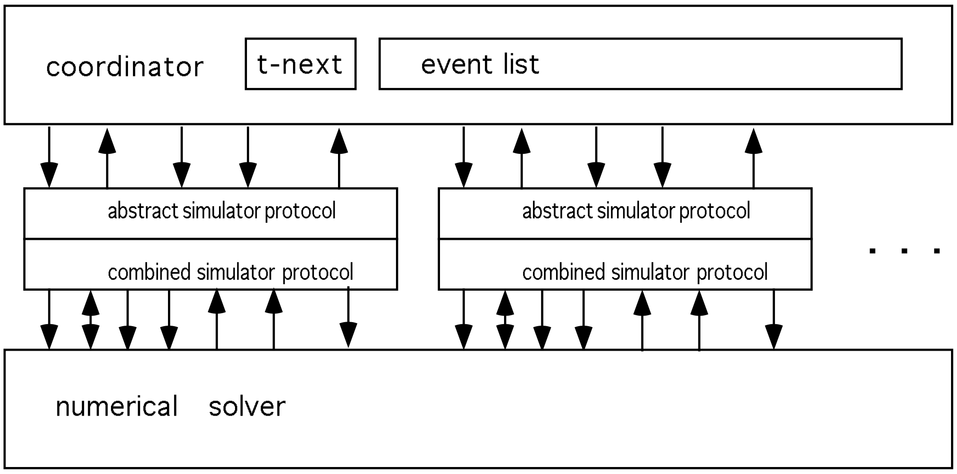

Figure 7: Hierarchical simulator

Hierarchical Simulators Simulator objects for complex models, e.g. process interaction DEVS, have to handle a set of components by their own. Such a simulator object has to realize a coordinator for the inferior simulator objects. It is therefore a combination of a simulator object, as it realizes the protocol of a simulator for its superior coordinator as well as a coordinator protocol for its inferior simulators (Fig. 8).

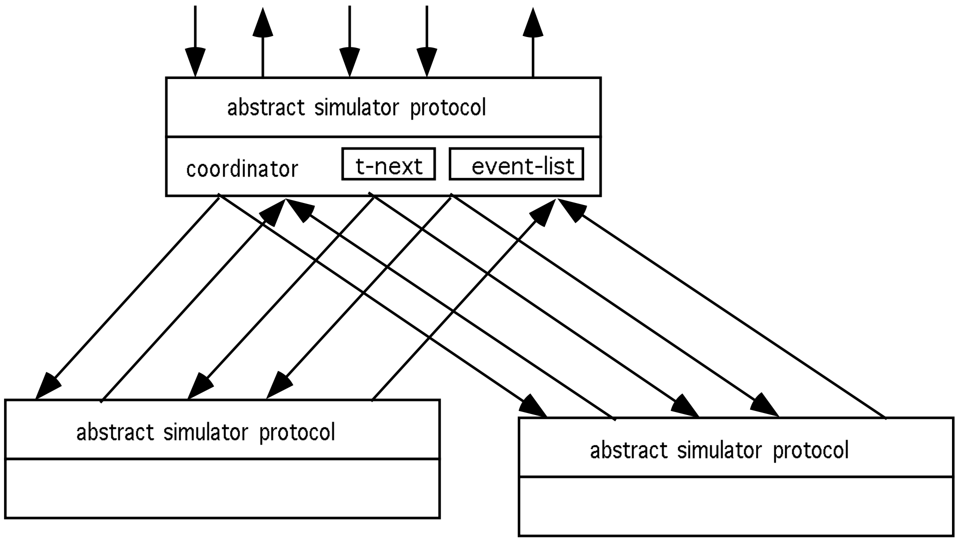

Figure 8: Hierarchical simulator

Distributed Simulators The above outlined scheme can be extended for distributed simulation [31, 32]. In distributed simulation, the model clusters executed on one parallel process, are handled by one coordinator. The coordination of models in different clusters can be done in a distributed manner by the local coordinators or a global synchronization mechanism [31].

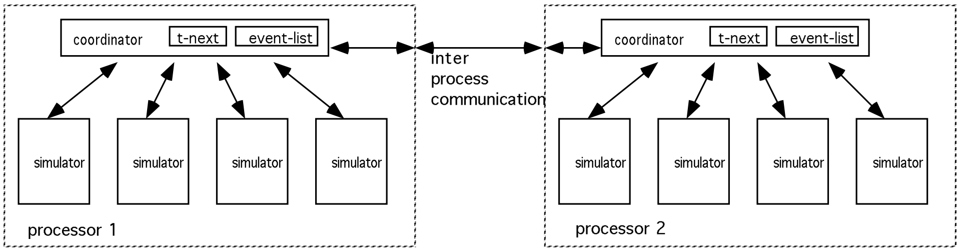

Figure 9: Distributed simulator

Simulator Integration Model encapsulation by interface specification and polymorphic simulator protocols as well as the formal foundation provided by the DEVS formalism can be a basis for simulator integration. Simulation models in foreign formalisms and simulation systems can be wrapped by an event interface providing controlled access to the model. Additionally, a simulator targeted to the coordinated execution of the model with its own simulation system has to be realized.

4.2 Compiling OpenDEVS Models

Figure 11 depicts an simulation system based on an OpenDEVS model specification. Using an OpenDEVS translator, models could be imported from other systems by automatically compiling the specifications into the programming language constructs (figure 11). Moreover, models can also be implemented directly in OpenDEVS, that provides a model implementation level that is much nearer to the DEVS formalism that programming language or subsets thereof.

5 Conclusions

Neutral model solver protocol needs a universal basis for discrete event and combined discrete/continuous modeling. As has been shown in this paper, the DEVS formalism as an application and computer language independent formalism can provide such a basis. This theoretical basis is augmented by the OpenDEVS approach which provides an common language for model exchange between different simulation systems. The polymorphic abstract simulator protocol can be a basis for the realization of a solver for a DEVS-based multi-formalism simulation system.

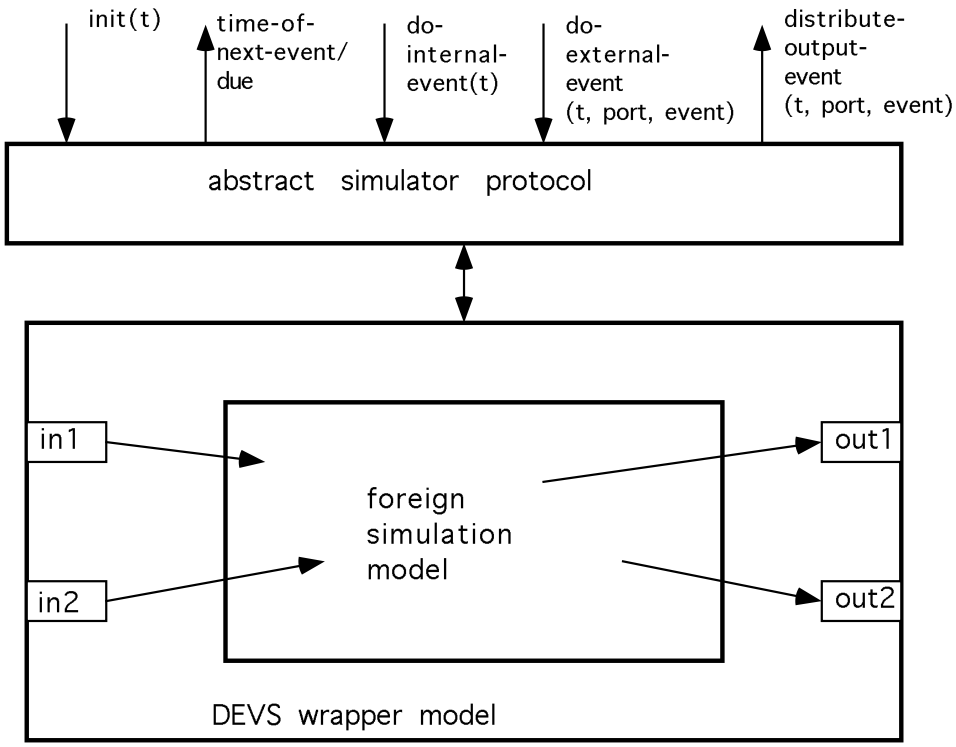

Figure 10: Simulator integration by modular model encapsulation and abstract simulator

The idea of a commonly usable DEVS model interchange format has been first raised during a DEVS Working Group meeting at the AIS Conference in 1994. Since then, the idea has substantially gained shape. The first proposal on OpenDEVS documented in this report provides a good starting point for further discussion, and makes path for the involvement of a broader audience.

To keep this development effort going, commitment from research and industrial partners is necessary to support an upcoming DEVS model interchange format with their tools. Also, active contributions from the DEVS community are needed in order to continue the standardization work.

Acknoledgement

The idea of an generic OpenDEVS format for model exchange stems from a work carried out at Daimler-Benz Corporate Research that aims at making a prototypical DEVS-based modeling and simulation system commercially available. Several people contributed to this work. Especially we'd like to thank Dr. Kunke from GSC mbH, Dr. Aposporidis, Mr. Lückhoff, and Mr.

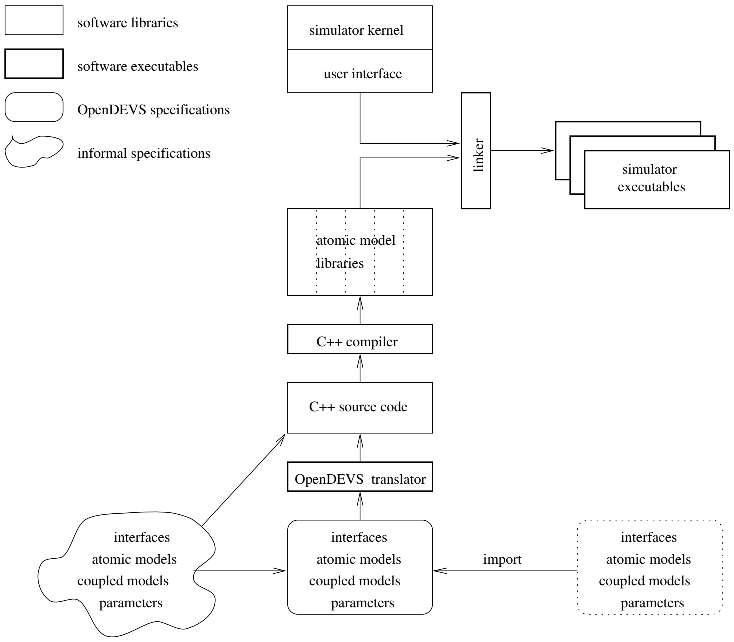

Figure 11: OpenDEVS usage for a compiling (in this case, C++ based) system

Hoffmann.

References

[1] Uhrmacher A.M. Reasoning about changing structure: A modeling concept for ecological systems. *International Journal on Applied Artificial Intelligence*, 9(2):178–184.

[2] Eustathios Aposporidis, Harbahjan Singh, and Carsten Thomas. Entwurf eines Modellierungs- und Simulationssystems für diskrete Prozesse. Technical Report F2S-94-004, Daimler-Benz AG, Berlin, Germany, August 1994.

[3] Schmidt B. Simulationssysteme der 5.generation. In A. Sydow, editor, *Fortschritte in der Simulationstechnik - 8. ASIM Symposium Berlin*. ASIM, 1993.

[4] Fernando J. Barros and M. T. Mendes. Modelling and simulation of an intergrated circuit assembly flow-shop in DEVS-V. In A. Verbraek and E. J. Kerckhoffs, editors, *Proceedings of the 1993 European Simulation Symposium*, pages 103–108. SCS, SCS Publications, 1993.

[5] Fernando J. Barros, M. T. Mendes, and Bernard P. Zeigler. Variable DEVS – variable structure modeling formalism: An adaptive computer architechture application. In Fishwick [8], pages 185–191.

[6] Alex C. Chow, Bernard P. Zeigler, and D. H. Kim. Abstract simulator for the parallel DEVS formalism. In Fishwick [8], pages 157–163.

[7] J. M. Farrow and S. Sevinc. Modelling tools for a Common LISP object-oriented system environment. In *Proceedings of the 2. Annual Conference on AI, Simulation and Planning in High Autonomy Systems*, pages 219–224. IEEE CS Press, 1991.

[8] Paul A. Fishwick, editor. Gainsville, FL, USA, December 1994. IEEE.

[9] Barros F.J. and Mendes M.T.and Zeigler B.P. Variable devs - variable structure modeling formalism: An adaptive computer architecture application. In Fishwick [8], pages 185–191.

[10] Tag Gon Kim and Sung B. Park. The DEVS formalism: Hierarchical modular systems specification in C++. In Philippe Geril, editor, *Proceedings of the 1992 European Simulation Multiconference*, pages 152–156. SCS International, SCS Publications, 1992.

[11] Georg J. Klir. *Architecture of Systems Problem Solving*. Plenum Press, New York, 1985.

[12] T. Pawletta, S. Pawletta, and E. Dimitrov. Modellierung und Simulation strukturvariabler Systeme. In *ASIM 94: 9. Symposium Simulationstechnik*, number 8 in Fortschritte in der Simulationstechnik, pages 59–64, Braunschweig, Germany, 1994. ASIM, Vieweg-Verlag.

[13] F. Pichler and H. Schwärtzel, editors. *CAST Methods in Modelling*. Springer-Verlag, 1992.

[14] H. Praehofer. *System Theoretic Foundations for Combined Discrete-Continuous System Simulation*. PhD thesis, Johannes Kepler University of Linz, Linz, Austria, 1991.

[15] H. Prähofer, F. Auernig, and G. Reisinger. An environment for DEVS-based multiformalism simulation in Common LISP/CLOS. *Discrete Event Dynamic Systems: Theory and Applications*, 3(2):119–149, 1993.

[16] Herber Prähofer and Gernot Reisinger. Distributed simulation of DEVS-based multiformalism models. In Fishwick [8], pages 150–156.

[17] Herbert Prähofer. *System Theoretic Foundations for Combined Discrete-Continuous System Simulation*. PhD thesis, Johannes-Keppler-Universität, Linz, Austria, 1991.

[18] Herbert Prähofer. Automatic abstraction of event-based control models from continuous base models. Technical report, Johannes-Kepler-Universität, Linz, Austria, 1994.

[19] J. W. Rozenblit and Y. M. Huang. Rule-based generation of model structures in multifacetted modeling and system design. *ORSA Journal on Computing*, 3(4):330–344, 1991.

[20] Jerzy W. Rozenblit and Jhyfang Hu. Integrated knowledge representation and management in simulation-based design generation. *Mathematics and Computers in Simulation*, 32(3-4):261–282, September 1992.

[21] H. S. Song and Tag Gon Kim. The DEVS framework for discrete event systems control. In Fishwick [8], pages 228–234.

[22] Carsten Thomas. Hierarchical object nets — a methodology for graphical modeling of discrete event systems. In G. W. Evans, editor, *1993 Winter Simulation Conference Proceedings*, pages 650–656, Los Angeles, CA, USA, December 1993. IEEE.

[23] Carsten Thomas. Interface-oriented classification of DEVS models. In Fishwick [8], pages 208–213.

[24] Carsten Thomas. *Ein objektorientierter Ansatz zur Modellierung und Simulation komplexer Systeme*. PhD thesis, Technische Universität, Braunschweig, Germany, 1995. To be published.

[25] Arnold R. Uhrmacher A.M. Maintaining knowledge in variable structure environments. In A. Sydow, editor, *Proc. of the 5th Annual Conference on Artificial Intelligence*, pages 178–184, 1994.

[26] Zeigler B.P. Uhrmacher A.M. Variable structure models in object-oriented simulation. *International Journal on General Systems*, 24:359–375, 1996.

[27] B. P. Zeigler. *Theory of Modelling and Simulation.* John Wiley, New York, 1976.

[28] B. P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, London, 1984.

[29] B. P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models.* Academic Press, London, 1990.

[30] Bernard P. Zeigler. DEVS representation of dynamical systems: Event-based intelligent control. *Proceedings of the IEEE*, 77(1):72–80, 1989.

[31] B.P. Zeigler and D. Kim. Distributed simulation of devs-based multiformalism models. In *in Workshop on Artificial Intelligence Simulation and Planing in High-Autonomy Systems*, Gainesville, FL, 1994. IEEE/CS Press.

[32] B.P. Zeigler and D. Kim. Design of high level modelling / high performance simulation environments. In *in 10th Workshop on Parallel and Distributed Simulation*, Philadelphia, 1996.