OpenDEVS: A Proposal for a Standardized DEVS Model Exchange Format

C. Thomas* H. Lückhoff* T. G. Kim**

*Research & Technology Dept.
Daimler-Benz AG
Alt-Moabit 96a
10559 Berlin, Germany
thomas@DBresearch-berlin.de

**Electrical Engineering Dept.
KAIST
373-1 Kusong-Dong, Yusong-Gu
Taejon 305-701, Korea
tkim@ee.kaist.ac.kr

Abstract

In this paper, we propose an open and extensible model exchange format for modeling, analysis, and simulation tools that are based on the DEVS (discrete event system specification) formalism. The exchange format — named OpenDEVS *— will provide a stable and standardized basis for the development of academical and commercial DEVS-based software systems and can facilitate cooperation and model exchange among business units in industry and among research groups. The proposal is intended to initiate a discussion process through which finally the standardized OpenDEVS exchange format will be defined. To provide a starting point, a preliminary version of OpenDEVS has been defined based on previous work done at Daimler-Benz and at KAIST [1, 2]. Strongly focused on the constructs of the set-theoretic DEVS formalism, it supports the specification of DEVS models utilizing object-oriented concepts like inheritance and information hiding. The format can be extended to support the various modifications and extensions that have been made to the basic formalism. It also takes into account the necessities of commercial model library vendors by supporting the separation of model declaration and implementation.*

1 Motivation

Despite its sound theoretical foundation and its proven suitability in many different areas of model management and simulation, the DEVS formalism so far did not gain the same level of attention from industry and simulation practitioners as other approaches did. One major reason for this situation may be the fragmentation of research efforts due to a lack of interoperability between the various systems and tools developed so far. As a consequence, practitioners as well as researchers are not able to take full advantage of the already impressing variety of existing tools. Providing an open standard for model exchange could be a substantial contribution to both broadening the acceptance of DEVS inside the simulation community and boosting the productivity of DEVS research teams.

Another obstacle on the way to a broader acceptance inside the simulation community is the missing availability of DEVS-based commercial products. As many examples in the recent past have shown, a widely accepted and supported standard has the potential to raise the interest of commercial vendors. High quality commercial products are undoubtedly a key factor for a further diffusion of the DEVS technology since they usually offer stable development environments, sophisticated user interfaces and the stable perspective for further product development that is demanded for by simulation practitioners. An open standard for model exchange can serve as a common basis for the integration of the growing spectrum of DEVS based systems and tools. In addition, it will enable the development of commercial model libraries whose vendors could benefit from substantially improved market opportunities.

Being aware of the necessities and opportunities outlined above a first attempt is made to formulate basic elements, requirements and approaches for an OpenDEVS model exchange format. Special emphasis is laid on the partly conflicting objectives like respecting the characteristics of different DEVS derivatives while preserving a formal uniformity which should allow future extensions of the format without violating its basic design principles.

2 Basic Design Principles

2.1 The DEVS Formalism

One of the basic concepts followed is the strong adherence to the mathematical constructs of the DEVS modeling formalism. The DEVS formalism is a set-theoretic formalism that supports the specification of discrete event models in a modular hierarchical form [3, 4]. DEVS models are constructed from two kinds of components: atomic models and coupled models.

Atomic models are components that cannot be further decomposed. The model specification consists of an interface description (namely the input event set X and the output event set Y, both typically further refined into port names and variable values), the sequential states set S, the internal and external event functions (δ_x, δ_i), the output function λ, and the time advance function τ:

$$AtomicModel = < X, Y, S, \delta_x, \delta_i, \lambda, \tau > .$$

Coupled models describe, how larger systems are composed from other components. A coupled model specification comprises again an interface description, a set D of component names and — for each of these names — a DEVS model M_i. By the use of coupled models as components it is possible to construct hierarchical models. For each component i, the coupling is specified by the set I_i of influencees and the sets $Z_{i,j}$ of output-input mappings between the component and their influencees. Additionally, there is a select function σ that resolves conflicts during simulation:

$$CoupledModel = < X, Y, D, M_i, \{I_i\}, \{Z_{i,j}, \sigma\} > .$$

Detailed introduction into the DEVS formalism is given, e.g., in [4].

As will be shown below, the mathematical constructs of the formalism have their direct counterpart in the OpenDEVS model exchange format. This provision eases the mapping from the different implementations of the DEVS formalism into the model exchange format and vice versa.

2.2 Extensibility

In addition to the "pure" DEVS formalism, modified and extended DEVS flavors have been developed to be used for modeling and simulation of special kinds of systems such as variable structure systems and combined discrete/continuous systems [5, 6]. Also, modifications have been introduced to ease the process of modeling by providing means for type-safe modeling and model consistency checking [7]. Since it has been our goal to provide a unifying exchange platform for all these approaches and the respective tools, OpenDEVS must be open for such extensions. It also has to provide the means to differentiate between the different flavors of DEVS, such that tools operating on the specifications are able to identify the context and can handle the supplied information accordingly. In this paper, we focus on the "pure" DEVS formalism. Examples for specifications of combined discrete/continuous and variable structure models are discussed in [2], using a preliminary version of OpenDEVS.

2.3 Information Separation

A model specification can be seen as an entity consisting of three parts:

- the description of the model behavior, which manifests itself in the model *interface* to the outside world and is represented by the input and output ports and the description of the port properties,
- the actual *implementation* of the model behavior as an atomic or coupled component, which is completely independent of the interface description and preferably hidden to the model user,
- the *initial values* for parametric model variables which also influence the model behavior.

The separation of model interface and implementation is one of the fundamental notions of the object-oriented paradigm leading to reduced complexity, enhanced re-usability and easier maintenance of existing model libraries. Interface and implementation description form what we call a model prototype (fig. 1). A prototype specifies the overall behavior of a model and has to be distinguished from model instances. Instantiating a model means the process of allocating the basic resources of a model and connecting the model with its prototype. Since initialization can also change the model behavior, it is likewise a part of the prototype.

Existing DEVS based systems show a great variety in implementation styles (e.g., C++ libraries or DEVS Scheme) and potentially even further future diversification (e.g., Java applications downloaded from World Wide Web servers). Thus separation of interface and implementation is an adequate response to this existing heterogeneity that also offers an incremental approach to the development of the OpenDEVS standard itself. In a first step, only interface and coupled

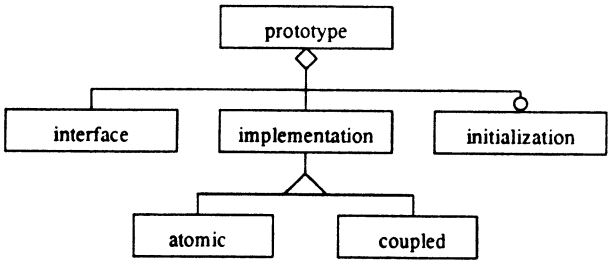

Figure 1: Contents of a model prototype

model specification may be standardized, while the specification of the implementation part may be system dependent. Later on, when a broad consensus concerning the functional specification has evolved, DEVS based systems can be upgraded to support this standard. In addition, the separation concept allows to hide the actual implementation of models held with system-specific model libraries. This feature is essential for commercial model library vendors that have a strong interest in protecting their intellectual property.

2.4 OpenDEVS Features

Object-orientation: The paradigm of object-orientation has a number of well elaborated and widely accepted benefits which made it the leading software and systems engineering approach of today. The DEVS formalism inherently supports certain object-oriented concepts such as encapsulation and data hiding, which makes it easy to integrate further object oriented features like inheritance and function overloading without conflicting with its underlying design principles. We propose three different and independent types of inheritance:

- *Interface* inheritance is strongly related to interface-oriented classification as a means of improved pre-runtime consistency check of DEVS models [7]. Using interface-oriented classification, a hierarchy of model classes is built up solely on interface specifications which can be seen as the set of all operations that can be performed on the specific models.

  A model interface can be regarded as derived from another if it comprises the parent interface. 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. In the case of output ports, it is required that the output ports of the derived interface do not handle a greater variety of message types than that of the parent.

- *Implementation* inheritance corresponds to the notion of inheritance normally used in the context of object-oriented programming systems. It covers the structural aspects of a model and is completely independent of the interface inheritance hierarchy. An important use of implementation inheritance is the specialization of atomic models by means of function overloading.

- *Initialization* information is used to overload the initial values of publicly accessible state variables. An inheritance scheme can be established in that sense that initialization specifications are cascaded. The initial values as defined in the atomic implementation are inherited by the first initialization specification where they can be overloaded. The result from this overloading is inherited by the next derived initialization specification. The initialization specifier can be regarded as a prototype name since it is connected with a complete model specification. Thus, it can be used directly for model instantiation while ensuring a correct initialization without the need of any explicit initialization actions to be taken by the modeler.

Type checking: Type checking and the resulting type safety is a powerful tool for reducing the risk of modeling errors. Especially for inexperienced users there are subtle traps and pitfalls while working with untyped systems.

Candidates for type checking are variables, messages and ports. Though we do not want to propagate a rigid framework for type safety and implicit type conversion rules at the very moment, there should be at least clear semantics for type declaration. The actual degree of type safety to be applied could be left to the simulator design. For simulation systems utilizing only the basic DEVS formalism, a type **`universal`** can be introduced signaling that any kind of type can be handled. For systems that intensively make use of types — like the VSIM modeling and simulation system [8] — types may be declared for different kinds of ports and for variables and message contents.

Namespaces: With the introduction of third-party model libraries, there is a potential for name conflicts that can be especially annoying if the model specifications of such libraries are not available. Therefore it is necessary to have a clear concept concerning the

scope of names and symbols in the model specification. In OpenDEVS, we propose three scope levels which are related to prototypes, libraries (i.e., sets of prototypes), and the global scope, respectively. This provides a sufficient degree of granularity while keeping the necessary tool implementation efforts modest.

3 OpenDEVS Syntax Proposal

3.1 Basic Syntactical Structure

The basic syntactical unit of the OpenDEVS exchange format is a statement. A statement is composed of one or more of the following components: specifiers, identifier keywords and expressions. Specifiers refer to type information like variable types or prototypes whereas identifiers denominate names for model instances, variables or ports. Expressions can be any kind of model implementation code or arithmetic or constant expressions.

The second basic syntactical construct is the statement list. A list can contain any number of statements, but one statement may contain not more than one list. A single statement list can also be considered a statement. Lists can be nested, i.e., lists may contain other lists as well as statements which contain a list. Statement lists are always enclosed in braces. Every statement must be terminated by a semicolon, whereas the elements of identifier lists are separated by commas.

3.2 Selected Statements

To give an overview about the most important OpenDEVS statements we use the DEVS model of a generator/buffer/processor system taken from [1] (fig. 2). We present fragments of the OpenDEVS specification of this example to clarify the use of some basic OpenDEVS statements. A more detailed discussion of OpenDEVS statements can be found in [2]; the complete OpenDEVS syntax proposal is available through the authors.

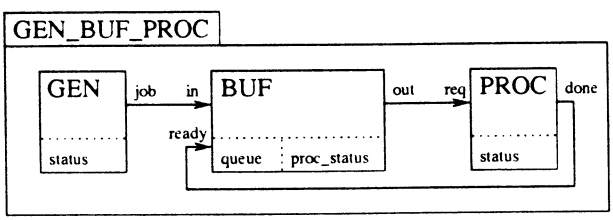

Figure 2: Structure of the GEN_BUF_PROC model

Context description: Through the description statement, the parser processing an OpenDEVS file is informed about which OpenDEVS constructs are used in the current context. By this means, the parser is able to determine whether it meets the requirements that are necessary to handle the DEVS flavor used in the file. Unique names (here: **stdDEVS**) are used to identify descriptions. Description statements can be cascaded to ease the declaration of an extended context while preserving the properties of the basic context.

```
description {
  system { // supported abstract simulators
    "atomic"; "coupled"
  };
  port { // no special ports supported here
    "universal"
  };
  var {"integer"; "boolean"; "double";
    "complex"; "time"; "string"; "list";
    // more variable types if necessary  ...
  };
} stdDEVS;
```

In the **system** section, the description declares which system types are supported in the described context. By this, it also states which abstract simulators are necessary to interpret the specifications correctly. Additionally, all the generic data and message types that are required by the model are declared in the **port** and **var** sections. Since the "pure" DEVS formalism applied in the example does not utilize special kinds of ports, only the **universal** port type is listed.

Every set of OpenDEVS statements is valid only with respect to a description. To identify the context of statements, the unique description name has to be used. All specifications must be placed inside the braces of a context statement.

```
stdDEVS {
  // any number of OpenDEVS statements
};
```

Interface specification: The **interface** statement basically consists of the input and output specifications. It makes the model known and accessible to the modeler.

```
interface BUF {
  inports {
    universal in, ready;
  };
```

```
  outports {
    universal out;
  };
};
```

As already mentioned, standard DEVS does not utilize special kinds of ports or typed messages. Therefore, all ports are declared as **universal**, and no message types are specified. In other approaches, such as the Hierarchical Object Nets used in the VSIM simulation system, different kinds of ports and typed messages may be used to ensure message type safety and port compatibility, and to ease the process of modeling. For such systems, extended declarations can be formulated utilizing the extensibility of OpenDEVS.

Atomic model implementation: The atomic model specification starts with the declaration of the local state variables in the **variables** section. The variables have types that must be declared in the **var** section of the description statement corresponding to the current context or using a type declaration statement. Note, that for types declared in this way only a limited set of operators is applicable.

```
type status_t { free, busy };

atomic BUF {
  variables {
    status_t proc_status;
    integer queue;
  };
  ...
};
```

In the **functions** section, the transition, output, and time advance functions are specified. Although a final syntax hasn't been defined yet, we present an example on how the function definition may look like. The specification style is straight forward: If the precondition on the left side of the "=>"-operator is fulfilled, then the postcondition is evaluated to **true** by binding the post-values (indicated by an apostroph) accordingly. The precondition also may be omitted.

```
atomic BUF {
  ...
  functions {
    internal()
    { ((proc_status = free) && (queue > 0))
      => ((proc_status' = busy) &&
          (queue' = queue - 1)); };

    external(p,_)
    { (p = in) => (queue' = queue + 1);
      (p = ready) => (proc_status' = free);
    };

    output(out, y')
    { ((proc_status = free) && (queue > 0))
      => (y' = 1); };
  };
};
```

Note, that the external event function carries parameters that specify the port and the message content type. The parameters can be used in two different ways: If they are variables (as **p** is above), they are bound to the identifier of the port where the event occurred and can be used to modify the model behavior accordingly. If the parameters are constants or sets of constants, they define the input space segment, which the function can be applied to ("_" is the "don't care" symbol). Especially for models that react very differently on input events of different kind or at different ports, this feature eases the modeling process. Thus, the external event function could also be split into to functions, yielding the same behavior as the one function above.

```
external(in,_)
{ queue' = queue + 1; };

external(ready,_)
{ proc_status' = free; };
```

The same mechanism is used to partially or completely overload the specified function in derived atomic model implementations. Here, a buffer of limited capacity is derived from the more general **BUF** model.

```
atomic LIM_BUF {
  variables {
    integer max_cap;
  }
  functions {
    external(in,_)
    { (queue < max_cap)
      => queue' = queue + 1; };
  };
};
```

The overloading works for both specification styles: It either partially overloads the global **external(p,_)** function of the first example or completely overloads the **external(in,_)** function of the second example.

Coupled model implementation: The **coupled** statement describes the implementation of coupled models. In the **components** section, all model components are listed. The **couplings** section denominates the links between ports to be connected.

```
coupled GEN_BUF_PROC {
  components {
    GEN gen; BUF buf; PROC proc; };
  couplings {
    link {gen.job; buf.in};
    link {buf.out; proc.req};
    // named link
    link {proc.done; buf.ready};
  };
} gen_buf_proc; // instance of GEN_BUF_PROC
```

For this example it is assumed, that the prototypes **GEN**, **BUF**, and **PROC** have been made publicly accessible by **interface** statements. Alternatively, the specification of components may be placed directly into the components section of the coupled model. In this case, the component prototype is "local" to the coupled model and may not be used elsewhere. We introduced these different options to specify a model for two reasons. On the one hand, we want to enable the modeler to limit the inflationary use of globally known prototype names which can be problematic especially in complex models. On the other hand, nested model definition can be a good method to structure a complex model specification because the models are defined where they are actually used.

Model initialization: The initial values of model state variables may be specified using the **initialize** statement.

```
initialize LIM_BUF::BUF_10 {
  proc_status = free;
  queue = 0;
  max_cap = 10;
};
```

The identifier **BUF** on the left side of the "::"-operator indicates that the initializer refers to the atomic model **BUF**. **BUF_10** is the name of the initializer. To demonstrate the inheritance mechanism suppose the following example.

```
initialize LIM_BUF::BUF_20 : BUF_10 {
  max_cap = 20;
};
```

This statement introduces a new initializer **BUF_20** which is derived from the previously defined initializer **BUF_10**. This means that all the initial values which were defined for **BUF_10** are inherited by **BUF_20**. Only those variables which are explicitly re-initialized by **BUF_20** will be overwritten with the new values. Therefore, the value for the variable **queue** in the context of **BUF_20** is still **0** as the result of the inheritance mechanism, while the value of **max_cap** is set to **20**.

As mentioned in the previous section, the identifier of an initializer can be used directly as a prototype name.

```
coupled GEN_PROC_BUF_20 {
  components {
    BUF_20 buf;
    PROC proc, GEN gen;
  };
  couplings {
  ...
  };
};
```

It is noteworthy that no single explicit statement has to be made by the modeler to insure a proper initialization of the model.

4 Application potential

The OpenDEVS model exchange format provides an open and extensible platform for model exchange between various DEVS-based tools. It can be the basis for the integration of the various research efforts that have led to powerful tools for modeling, simulation, and analysis. OpenDEVS enables users to combine the strengths of these tools and to build powerful model-based engineering environments that would not be possible if one is restricted to just one tool supplier.

Every software tool in such an environment serves a specific task. New or dedicated tools can replace old ones; the integrated environment may be easily adapted to new application areas. We envision the use of OpenDEVS in, e.g.,

- user-friendly graphical modeling tools that support model library handling and knowledge-based system synthesis,
- model checking tools proving the syntactical consistency and the completeness of the specifications,

- model analysis tools, which help to investigate properties of the systems described like liveliness and reachability,
- simulation tools supporting sequential simulation on desktop workstations as well as high-performance simulation using massively parallel computers,
- animation tools for two-dimensional and three-dimensional system visualization,
- documentation tools that generate graphical and cross reference documentation from model specifications describing systems under development.

Within such environments, the tools complement each other in a way that improves the usability of each of them and betters the chance for commercial success for every single tool. In this way, a standardized OpenDEVS format also broadens the market for DEVS-based software tools; thus making DEVS and DEVS tool development interesting for commercial tool developers.

In addition to the application in the tool interoperability field, the OpenDEVS standard also can be a basis for the development of model libraries. It provides a level of modularity which allows to exchange and integrate models from very different sources. Interaction between research groups and between academia and industry will profit from this possibility. Also — since the information separation approach allows to hide the knowledge inside of non-readable model libraries — development and distribution of application-oriented model libraries will be a commercially attractive market.

5 Summary

In this paper, we proposed an open and extensible DEVS model exchange format named OpenDEVS. Rationale for a standardized model exchange format comes from the expectation, that the existence of such a format supports tool inter-operability and enhances the possibilities for co-operation in research and industry. Moreover, a standardized model exchange format makes the development of tools and model libraries commercially attractive and enhances the acceptance for the DEVS formalism in industrial environments.

We hope, that the ideas outlined in this paper initiate a discussion that finally leads to a mutually agreed upon standardized OpenDEVS. To keep this effort going, commitment from research and industrial partners is necessary to support an upcoming DEVS model exchange format with their tools. Also, active contributions from the DEVS community are needed in order to continue the standardization work.

References

[1] Gyung Pyo Hong and Tag Gon Kim, "A framework for verifying discrete event models using a dual specification approach", *SCS Transactions*, Accepted for publication.

[2] Carsten Thomas, "The DEVS model interchange format OpenDEVS. A proposal", Tech. Rep. F3S-95-015, Daimler-Benz AG, Berlin, Germany, 1995.

[3] Bernard P. Zeigler, *Theory of Modeling and Simulation*, John Wiley and Sons, New York, NY, USA, 1976.

[4] Bernard P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, London, UK, 1984.

[5] Fernando J. Barros, M. T. Mendes, and Bernard P. Zeigler, "Variable DEVS – variable structure modeling formalism: An adaptive computer architechture application", In Fishwick [9], pp. 185–191.

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

[7] Carsten Thomas, "Interface-oriented classification of DEVS models", In Fishwick [9], pp. 208–213.

[8] Carsten Thomas, "Hierarchical object nets — a methodology for graphical modeling of discrete event systems", in *1993 Winter Simulation Conference Proceedings*, G. W. Evans, Ed., Los Angeles, CA, USA, Dec. 1993, pp. 650–656, IEEE.

[9] Paul A. Fishwick, Ed., *Proceedings of the 5. Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, Gainsville, FL, USA, Dec. 1994. IEEE.