Extended DEVSML as a Model Transformation Intermediary to Make UML Diagrams Executable

Jianpeng Hu1,2, Linpeng Huang1, Bei Cao1, Xuling Chang1
1Dept. of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
2College of Electrical and Electronic Engineering,
Shanghai University of Engineering Science, Shanghai, China
mr@sues.edu.cn, huang-lp@sjtu.edu.cn, caobei.sjtu@gmail.com, changxl@sjtu.edu.cn

Abstract—The Unified Modeling Language (UML) has been widely used for software and system design. To reduce the cost and risk of the system development, it is very important to validate and evaluate the system precisely in early design phase. Many efforts were made to make UML executable by transforming single diagram to executable model such as Colored Petri Nets (CPN), however, approach like this could not provide more systematic and intuitive simulation of the entire system. Therefore we choose the Discrete Event System Specification (DEVS) as the target formalism and transform UML Diagrams to systematically simulatable models in this paper. To achieve this goal, we extend the DEVS modeling language (DEVSML) by enhancing its capability of describing complex behavior of systems, and provide an automated code generation process using Extended DEVSML (E-DEVSML) as a model transformation intermediary to help modelers to acquire the benefits of DEVS framework without delving in the DEVS theory.

Keywords- UML; DEVS; Model Transformation; Simulation; Executable Model

I. INTRODUCTION

The Unified Modeling Language (UML) has become the de facto standard modeling language for software and system design. With extensions in different specific domains, such as Service oriented architecture Modeling Language (SoaML) in service engineering, the UML family has powerful ability to describe the systems from diverse viewpoints. However, systems described by those diagrams in UML are only static models, which are hard to validate and verify. Rigorous validation and verification of system specifications requires executable models, which typically come with various simulation capabilities to further support dynamic analysis and evaluation of the systems. A lot of researches are made to transform the UML models to executable discrete-event models. Colored Petri Nets (CPN) and Discrete Event System Specification (DEVS) [1] are two extensively referenced and used modeling and simulation formalisms. CPN are especially well suited to model workflows of systems where concurrent events can take place. Commonly a single UML diagram such as Sequence Diagram can be transformed into CPN to perform a validation [2]. As to DEVS which starts from general system theory, it may provide more systematic and intuitive simulation of the entire system by transforming a combination of several diagrams to executable models. In addition, DEVS has been proven to be a universal formal mechanism to express a variety of discrete-event system subclasses, including Petri Net, Cellular Automata and Generalized Markov Chain [3]. Thus we argue that DEVS is a better choice if we need a systematic validation and evaluation of the system. In this paper our research devotes to realize seamless connection between UML and DEVS simulation by model transformation. To achieve this goal, we propose an Extended DEVSML (E-DEVSML) by enhancing its capability of describing complex behavior of systems, and provide an automated model transformation approach between UML and DEVS.

II. PARALLEL DEVS FORMALISM

Parallel DEVS removes constraints in the classic DEVS that originated with the sequential operation and hindered the exploitation of parallelism [1]. DEVS models can fall into two categories: atomic and coupled. The atomic model is the irreducible model definition that specifies the behavior for any modeled entity. The coupled model represents the composition of several atomic and coupled models connected by explicit couplings. An atomic model M and a coupled model N are defined by the following equations:

\[ M = \langle IP, OP, X, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta > \quad (1) \]

\[ N = \langle IP, OP, X, Y, D, EIC, EOC, IC > \quad (2) \]

In an atomic model, S is the state space; IP, OP are the set of input and output ports; X, Y are the set of Inputs/Outputs, which are basically lists of port-value pairs.

\[ X = \{ (p,v) / p \in IP, v \in X_p \} \quad Y = \{ (p,v) / p \in OP, v \in Y_p \} \]

where \(X_p\) and \(Y_p\) are input/output values on port \(p\).

\(\delta_{\text{int}}: S \rightarrow S\) is the internal transition function; \(\delta_{\text{ext}}: Q \times X^\theta \rightarrow S\) is the external transition function, where \(Q = \{ (s,e) / s \in S, 0 \leq e \leq \tau(a(s)) \}\) is the total state set, \(e\) is the time elapsed since last transition, and \(X^\theta\) is a set of bags composed of elements in X; \(\delta_{\text{con}}: S \times X^\theta \rightarrow S\) is the confluent transition function, which decides the order between \(\delta_{\text{int}}\) and \(\delta_{\text{ext}}\) in cases of collision between simultaneous external and internal events.

\(\lambda: S \rightarrow Y^\theta\) is the output function, where \(Y^\theta\) is a set of bags composed of elements in Y; \(\tau(a(s)): S \rightarrow R_0^+ \cup \infty\) is the time advance function which decide how much time the system stays at the current state in the absence of external events.

This work was supported by the National Natural Science Foundation of China under Grant No. 61232007, 91118004
In a coupled model, IP, OP, X and Y have similar connotation as in atomic model, but mean external elements; D is a set of DEVS component models. EIC is the external input coupling relation, EOC is the external output coupling relation, and IC is the internal coupling relation. The coupled model can itself be a component of a larger coupled model giving rise to a hierarchical DEVS model construction.

III. EXTENDED DEVS MODELING LANGUAGE

In practice, when we depict a system with previous DEVS Modeling Language (DEVSMML) version [4] based on Parallel DEVS and Finite Deterministic DEVS, it is inconvenient to deal with nondeterministic state transitions, any state transition based on the message content is not realizable, and there may be a collision if two messages simultaneously arrive on more than two ports. Therefore we take appropriate measures to extend DEVSMML to deal with these complex situations.

A. Separation of Message Processing From State Transition

Sometimes, simultaneous arrivals of inputs on different ports make it harder to describe the behavior of the external transition function, because only one transition can be invoked at the same time. If we choose one of the possible transitions to trigger, other inputs may be ignored or lost. Therefore we try to separate the processing of inputs from state transition. For example, we could first store these inputs in some variables before state transitions. From a functional perspective, an abstract description of an atomic model will be: accepting inputs or incentives and then generating an output with state changes. This behavior can be expressed with the two following formulas:

\[ y = \lambda(s_n) / s_n \in S, y \in X^b \text{ and } s_n = \delta(x, s_{n-1}) / x \in X^b \cup \emptyset \quad (3) \]

where \( S = \{s_0, s_1, ..., s_{n-1}, s_n\} \) is a set of states. We could divide this input/output process into three parts: receiving a message, data processing during a series of transitions and sending a message. Three associate functions are defined as following:

\[ v_i = \text{Rev}(x), y = \text{Sed}(v_i), v_i = \text{DP}(v_i) \quad (4) \]

where \( x \in X^b, y \in Y^b, v_i \in V_x, v_i \in V_y \), we define a set of input-port-associate variables \( V_x \) and a set of output-port-associate variables \( V_y \). These variables may be simple data types (e.g. integer, floating number and string) or containers of simple data types (e.g. queue, stack and list). Deriving from equations 3 and 4, we can get equation 5:

\[ y = \lambda(s_n, v_i) \text{ and } (s_n, v_i) = \delta(v_x, s_{n-1}) \quad (5) \]

When an external transition is triggered, the receiving function Rev stores the inputs into corresponding \( V_i \) by unpacking a message bag. And DP function for data processing from \( V_i \) to \( V_j \) could be implemented within external transitions or internal transitions. Similarly, the sending function Sed for processing of outputs may be independently implemented in output function \( \lambda \). This separation saves modelers from digging into the DEVS semantics but only their own behavior description of the systems.

B. Support for Nondeterministic State Transitions

First, we’ll show the differences between Nondeterministic State Transitions (NST) and Deterministic State Transitions (DST) in formal definition. Based on basic transition functions, DST and NST can be defined as follow:

\[ DST = \delta : S \times X^b \rightarrow S \quad (6) \]
\[ NST = \delta : S \times X^b \rightarrow P(S) \quad (7) \]

where \( \delta(s, x) = \delta_{\text{ext}}(s, x) \cup \delta_{\text{con}}(s, x) \), and \( \delta(s, \emptyset) = \delta_{\text{int}}(s); P(S) \) denotes the power set of \( S \). In DST, the transition function \( \delta \) is a surjection. Thus the term deterministic means that there is at most one invoked transition to a target state from any source state with a specific input. On the contrary, NST has more alternatives for target state with the same inputs. In practice, it is very important to convert easier-to-implement NSTs into more efficiently executable DSTs. To achieve this goal, we defined a Conditional Transition (CT):

\[ CT = \delta : S \times X^b \times C \rightarrow S \quad (8) \]

It shows the transition is triggered only when some pre-conditions are satisfied. It also can be described as \( \delta(s, x, c) = s' \), where \( s, s' \in S, c \in C, x \in X^b \cup \emptyset \), \( C \) is a set of constraints. We could make an assumption that, for any target state we can find a disjoint pre-condition satisfied (may be specific inputs, variable correlation, time correlation, etc.) to invoke a deterministic transition from a source state, because we need ensure the system running as expected and there must be only one option of transitions in the same condition. So a NST described as \( \delta(s, x, c) = \{s_1, s_2, s_3, ..., s_i\ / s_i \in S\} \), may be divided into equations as follow:

\[ \delta(s, x, c_1) = s_1, \delta(s, x, c_2) = s_2, ..., \delta(s, x, c_i) = s_i \quad (9) \]

where \( c_i \in C, s, s_1 \in S \). We could consider equation 9 as an approximate realization of a NST using a DST format.

C. Abstract Syntax of E-DEVSML

E-DEVSMML is specified using Extended Backus-Naur Form (EBNF) notation like the previous version. Considering of both convenience of use and conformity with the DEVS formalism, we specify it with modular and object-oriented features. We realize this language by Eclipse Xtext which is a powerful framework for the development of a DSL. Models in E-DEVSMML are divided into three primary elements: the Entity, the Atomic and the Coupled.

1) Entity: Message objects are exchanged according to the port-value pairs, and the datatyp of a input value can be defined as an entity and reused by some ports. Figure 1 gives the definition of Entity in EBNF. We define a variable type named container which is a common data structure (e.g. queue) to store a series of entities. The keyword default assign values to variables when model is started or restarted.

2) Atomic: The Atomic is the most important and complicated part of DEVS. The Atomic model is specified in EBNF grammar as figure 2 and figure 3 showing. The keyword vars defines a set of variables. The interfaceIO specification gives the definition of ports with specific data
type which is referenced as an Entity type. And state-time-advance defines a set of states and the associated time-advances. The time-advance TimeAdv can have values of either DOUBLE, infinity or a Variable already declared. The state-machine contains the initial state InitState and four behavioral functions. The expression (code=Code) implies that there may be a code block associated with setting up of the initial state.

```plaintext
Entity = {extend name=ID 'extends' superType = [Entity | QualifiedName])
  ('{ (attributes = Variable)*') ;
 Variable:
 type = VarType name=ID (default def=STRING);
 VarType:
 simple = ( [int | double | string | boolean] )
 container = ( [queue | stack | list] )
 complex = [Entity | QualifiedName];
```

**Figure 1.** Definition of Entity in EBNF

Atomic : 'atomic' name=ID

```plaintext
{extend superType=[Atomic | QualifiedName])
  ('{ (variables = Variable)*')
  'interface-Advance' ( (msgs = Port)* )
  'state-time-Advance' ( (state = STA)* )
  'state-machine' ( start in 'init-InitState' ( asm=Deltext? ) ( asm=Outfn? ) ( asm=Delint? (asm=Confluent? ) ( func = UserFunction* ) ))
 Port : type = ( 'input' | 'output' )
 ref=Entity [qualifiedName] name=ID;
 STA = name=ID time-Advance;
 TimeAdv: tav=DOUBLE inf=Infinity tvar=Variable;
 InitState: state=STA code=Code;
 CodeTemplate = 'code-template' name=ID
 ( (codePara = Parameter* )
 ( 'templateBody' = codeBody CodePara* )
 UseTemplate: 'use[Template]'
 UserFunction: 'code=Code' ;
 Parameter: st=STRING;
 Code: '{(str=STRING)}';
```

**Figure 2.** Definition of Atomic in EBNF

Deltext: 'Deltext' '{(rm = ReceiveMessage) (st = StateTransition? ) (Outfn: Outfn' ( (so = StateOutput? ) )
Confluent: 'Confluent function=Ignore-Input (input-only) | input-first | input-later)
SendMessage: 'send [oute=Port] senders=Variable;
ReceiveMessage: 'receive' '{in=Port}(, subMsg = QualifiedName)'?
StateOutput: 'at=STA' ( asm=StateOutput? )
StateTransition: 'at=STA' ( asm=StateTransition? )
     '{(ct = ConditionalTransition? )}'
ConditionalTransition: ( 'when' '{(condition=STRING)' act = Action? )
Action: ( 'const=continue' sig=SetSigma) sig=SetSigma;
SetSigma: 'sigs=DOUBLE inf=Infinity var=Variable? )
Transition: 'goto' target=STA sig=SetSigma;
CodeProcess = 'code=Code' ;
UseTemplate: 'ut=UseTemplate';
```

**Figure 3.** Definition of behavior of an Atomic in EBNF

DEVS has four functions to specify the behavior of an atomic model which are the main featured parts of E-DEVSML: Deltext, Delint, Outfn and Confluent. And the conditional transition, receiving function, sending function and DP function mentioned above are also implemented as shown in figure 3. The keyword continue allows the model to stay in the same source state and the SetSigma rule allows the resetting of time-advance. To enable interoperability between the coupled model and its components, user-defined method defined by UserFunction in an atomic model is allowed to be called by itself or the coupled model comprising it. In addition, UseTemplate rule can reuse a code block with parameters which is already defined in CodeTemplate rule.

3) Coupled: The specification of a coupled model is shown in figure 4. To put it simply, the coupled model has the same interface specification as the atomic model. It is composed of a set of models which can be either atomic component or coupled component.

Coupled: 'coupled' name=ID

```plaintext
{extend superType=[Coupled | QualifiedName])
  ('{ (components = Component)* '}
  'interface-Adv' ( (ports = Port)* )
  'couplings' '{ (couplings = Coupling)* }
  'user-code' 'code=Code' );
 Component: 'AtomicComp | CoupledComp;
 AtomicComp: 'atomic' at=[Atomic | QualifiedName] name=ID;
 CoupledComp: 'coupled' cp=[Coupled | QualifiedName] name=ID;
 Coupling : ic=IC eco=EC eic=EIC
 EIC : 'eic' 'this' 'srcport = [Port] ->'
 dest=Component? destport = [Port] ;
 IC : ( 'ic' src=Component [QualifiedName] ?
  srctype = [Port];
 dest=Component? destport = [Port] ;
 EOC : 'eoc'
 src = ( 'Component' 'this' srcport = [Port] ->'
 dest = [Port];
```

**Figure 4.** Definition of Coupled in EBNF

IV. MODEL TRANSFORMATION USING E-DEVSML

This automated transformation process from UML to DEVS is a two-step approach. Due to the textual style of E-DEVSML, first we use Xpand, a language specialized on code generation, to implement transformation from UML to E-DEVSML. Then we can get executable codes through Xtext framework because Xtext is seamlessly integrated with the Eclipse Java framework by code generator with Xtext. At last, we use one of the open source DEVS simulator named DEVSSuite [5] to validate the executable models.

First, all the data types inherit from UML::Class can be translated into Entities in E-DEVSML. Second, from structural perspective, Class Diagram gives definition of attributes in an atomic model, while Component Diagram provides structural information about a coupled model including components in coupled model and couplings between them. At last, from behavioral perspective, State Machine Diagram (SMD) is quite suitable for providing behavioral information of an atomic model. In this diagram, the guard conditions of transitions certainly are mapped onto the constraints of Conditional Transitions. The receiving function and DP function are derived from transition effects, and the sending function is derived from state exit effects. Note that, there is no concept can be mapped onto \( \delta_{con} \). The default definition of confluent transition function simply applies \( \delta_{ext} \) before \( \delta_{int} \) in cases of collision between simultaneous external and internal events.

To illustrate the capability of the E-DEVSML, we are going to show an example of a multi-server architecture model “m”
as shown in figure 6-a. The coupled model “EFM” contains two models: the coupled model “ExpFrame” is an experimental frame for evaluating the performance of the other coupled model “m”. The generator “g” sends entity “job” at a periodic rate, and the transducer “t” keeps counting generated jobs as well as processed jobs. In the coupled model “m”, the coordinator “multiScop:serverCord” keeps track of the status of the servers in a queue called “freeServers”. Figure 5 shows a SMD of the coordinator “serverCord”: when a job arrives at the input port “in”, it is routed to the first “passive” server. If no server is free, the job will be stored in a queue “jobsToDo”. When a completed job returns on corresponding port “x”, the “serverCord” reroutes it to the output port “out.” Simultaneously arrived completed jobs will be stored in a queue “jobsDone”. Note that, “jobsToDo” and “freeServers” are input-port-associate variables, and “jobToDo” is an output-port-associate variable which is assigned in the DP function employing a code template “make_output”, while “jobsDone” is not only a $V_x$ but also a $V_y$. If we try to transform this diagram to the previous version of DEVSML, it’s difficult to deal with the NST to different target states which are triggered by a message on the same port or simultaneous messages on different ports, while the E-DEVSML solves this problem and makes the automated transformation process more smoothly.

After an automated transformation process from UML to DEVS, figure 6-c shows a comparison of the model “serverCord” in E-DEVSML and java. When the final generated codes are executed in DEVS-Suite, we can get runtime animations shown in “SimView” window (figure 6-a). After several simulations with different processing time for a job of single server, the statistics of system performance is shown in figure 6-b. All these features guarantee validation of the functional requirements and evaluation of nonfunctional requirements precisely in earlier design phase.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a DEVS based modeling language E-DEVSML which can helps modelers to realize systematic simulation of the systems. We also presented an approach to make UML diagrams executable through an automated model transformation process using E-DEVSMIL. It also saves modelers from digging into the DEVS simulator codes but only their own behavior description of systems. In the future, practical application of E-DEVSML in an industry case will be discussed. As the current status of DEVSML is only support for static-architecture coupled models, our future work also includes description of DEVS dynamic-architecture, which permits the coupled model to evolve over time.

REFERENCES