International Journal of Computer Applications (0975 - 8887)
Volume 180 - No.8, December 2017

Safety Design for Simulation Models based on Formal Methods

Wassim Trojet
ESIGELEC-IRSEEM
Technopole du Madrillet,
Avenue Galilee - BP 10024,
76801 Saint-Etienne du Rouvray Cedex

ABSTRACT

Control theory researchers have been using DEVS models to formalize discrete event systems for a long time. Despite such systems are one of the main targets of Software Engineers, the DEVS formalism lacks tools offering representing and verifying safety properties. The general scope of the paper consists of extending the DEVS framework to support safety properties and verify them by using formal methods. Thus, we offer a possibility for DEVS user to describe safety properties and to verify formally if these properties are preserved during the evolution of the system. We called the extended formalism "ϕ*DEVS*". Safety verification is made once a "ϕ*DEVS*" model is translated to a formal specification using Z notation by performing proof obligation.

Keywords

Safety, DEVS, Discrete Event Simulation , Z, Formal Methods, Formal Verification

1. INTRODUCTION

The central assumption of a discrete event system is that the system changes instantaneously in response to certain discrete events. Discrete EVent System Specification (DEVS) formalism [1] permits modeling and simulation (M&S) of discrete event systems.
By definition [2], the M&S system must model and predict the behavior of some real world entity. This problem has been called "operational validation." A second problem for M&S systems is "conceptual model verification," which is concerned with ensuring that the model represents correctly the system description and does not contain errors. Our approach deals with the second problem. Hence, we offer for the DEVS users the possibility to describe the safety properties and to check these properties by using formal verification techniques developed by formal methods community.
Formal methods (FM) [3] have shown a potential for detecting major errors in system specification by applying a formal analysis. FM are mathematical techniques for the specification, design and analysis of complex systems. A formal specification is a system description using a mathematical notation. A formal verification is the use of a formal technique to check a formal specification: this technique can be either model checking or theorem proving.
Formal methods have been used to a great degree in computer hardware design. The two areas in which FMs have been used most are security and safety applications. Foundation'02 [4] noted significant progress, particularly with "lightweight" formal methods (LFMs), and pointed out that FMs "may be the only approach capable of demonstrating the absence of undesirable system behavior" [5]. However, FMs are not used as much as they could be in modeling and simulation, in part because FMs can not catch the whole aspects of complicated simulation models. In other part, it's difficult to use FMs and M&S in the same framework. Most of the research, publications, and work in and with formal methods continue to occur outside the modeling and simulation communities.
The subject of this paper is to improve the DEVS conceptual model by adding a component describing safety properties and conducting a formal verification in order to verify that are preserved in all system states before the simulation process. We will use the approach developed in [6] wich consists of: (1) transforming a DEVS model into an equivalent Z specification and (2) verifying the consistency of the DEVS model on the resulting specification using the tools developed by the Z community.

2. EXTENDING DEVS TO ϕDEVS

Lamport [7] defined a safety property called invariant property as the following: "Something bad never happens." Thus, a safety property is a property which has to be satisfied whatever is the state of the system. Hence, a conflict occurs when an invariant property is violated.
Example:
$property_i$: The system is always in motion.
$property_j$: WHEN the system stops THEN the door has to be open.
$property_i$ is a safety property, $property_j$ contradicts $property_i$, therefore there is a conflict.
The safety property guarantees the absence of an eventual conflict in the specification. According to the Z community, "the invariant" is the set of properties which must be satisfied whatever the state of the system. In Z it is possible to verify if all operations preserve the invariant by applying proof obligation.

2.1 Concept of invariant in Z

An invariant is a constraint that must always be satisfied: it defines a relationship which is true in every state of the system and is maintained by every operation [8]. That is, given a sequential system, defined with an initialization schema Init and a collection Op_1,..., Op_n of operations, a predicate I is an invariant if Init $\Rightarrow$ I, and for

every i ∈ 1..k, I is preserved by the ith operation (that is, we have I ∧ $Op_i \rightarrow$ I'). I should be added to the constraint part of the global state schema. Such invariants can be important for a variety of reasons:

—An analyst might wish to verify that the operations of a system preserve some important safety properties.

—An implementor can take advantage of an invariant to simplify an implementation. If, for example, x = y + z is an invariant, then one of x , y, or z may be computed as needed rather than stored explicitly in the state. If some sequence is always ordered, efficient searching algorithms are possible.

2.2 The invariant in the DEVS formalism

The concept of invariant has not been adopted by the DEVS community. Thus if system requirements state that there is a condition which has to be satisfied whatever the system state, this is traduced by holding this condition on all atomic DEVS transitions (see Fig. 1). Therefore, the model will be crowded because of condition redundancy. In addition, the simulation time will increase due to repetitive condition tests.

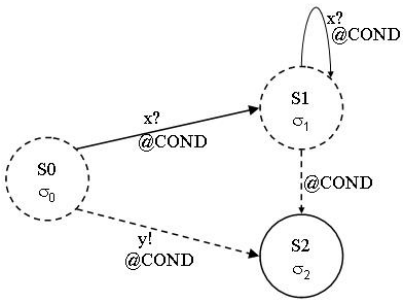

Fig. 1. Example of preserving a condition in all transitions

2.3 Extending DEVS to ϕDEVS

We propose a DEVS modelling framework which adopts the concept of "the invariant," i.e., conditions which have to be preserved throughout the model evolution: we call this framework *ϕDEVS*. This extended DEVS preserves the basic DEVS and provides a new structure, which we call "ϕ," to contain the conditions. That is, the structure of ϕDEVS is:

$$\phi\text{DEVS}=¡ \text{ X, S, Y, } \delta_{int}, \delta_{ext}, \lambda, \text{ D, } \phi¿$$

where $\phi=\{P(s) \mid P$ *is a Z formula involving a state* $s\}$.

2.4 The verification of safety properties in a ϕDEVS model

Given a ϕDEVS (¡DEVS, ϕ¿) model, we apply the transformation cited in Section **??** to extract the equivalent Z specification from the basic DEVS model. Then we encode conditions of the ϕ structure into the form of Z notation; the resulting conditions represent the Z invariant. The latter is added into the constraint part of the state schema of the Z package describing the DEVS model.

M
$s : S$
ϕ

We can check the coherency of the *ϕDEVS* model using proof obligation on the resulting Z specification. The proof obligation consists of:

—Proving the initial state, indeed if the initial state preserves the state invariant, it is considered "true." In our case, the Z formula which permits to check such a state is:
Therorem *canInit*
$\exists M' \bullet InitM$
This theorem permits us to check if the initial state of the *ϕDEVS* model is valid.

—Preconditions calculation: this step permits of checking whether all the operations of a Z package can be performed, i.e., whether the operations preserve the invariant.

DEFINITION 1. *Z provides a reference to the precondition of an operation schema [8]; for a schema Op b* $\triangleq$*[ΔS; in? : IN; out! : OUT] the schema reference pre Op is equivalent to* $\exists$ *S'; out! : OUT • Op, and describes the initial state for which an output and a final state are possible. If the operation is total (does not depend on another operation to be performed), it can be executed in any starting state and with any inputs; thus* $\forall S$*; in?: IN • pre Op should be a theorem. Trying to prove this conjecture can uncover any missing hypotheses. A correct precondition theorem, of the form* $\forall$ *S; in? : IN | P • pre Op, where P gives the precondition, can be a useful form of documentation of a specification.*

In our case, the Z formula which permits us to check the precondition of an operation $''op''$ resulting from δ_{ext}:
Theorem *CheckingPrecondition*
$\forall M;\ in? : domain_{in?} \mid in? = x \wedge s = source \wedge cond_{src-trg} \bullet$ **pre** *op*

This theorem permits us to check if the corresponding external transition is performed without violation of the invariant.
The Z formula which permits of checking the precondition of an operation $''op''$ resulting from δ_{int} and λ is:
Theorem *CheckingPrecondition*
$\forall M \mid s = source \wedge cond_{src-trg} \bullet$ **pre** *op*
This theorem permits us to check if the corresponding internal transition is performed without violation of the invariant.

We apply the theorems of the proof obligation already elaborated by the Z community [8] on the resulting Z specification to verify the invariant.
If $card(\phi) \neq 0$, the DEVS simulator has to check the invariant condition(s) in all DEVS model transitions. However, proof obligation permits to avoid this redundancy and verify formally the consistency of a *ϕDEVS* model by checking the consistency of the resulting Z specification.

3. SOFTWARE DESIGN FOR THE Z TRANSFORMATION AND SAFETY VERIFICATION PROCESS

We developed a tool which implements the algorithm of the transformation and the verification of a ϕDEVS model: we call this tool "ϕDEVS-Compiler" (see Fig. 2). Once ϕDEVS-Compiler reads a ϕDEVS model saved in an XML file it generates another XML file

containing the equivalent Z specification and loads this file on Z theorem prover and enables the checking process. Afterward, it recuperates the analysis results and return them to the DEVS user. If there are some conflicts in the model, the DEVS user can fix them and verify the model again via ϕDEVS-Compiler. Once the model is consistent (no conflicts in the model), the user proceeds to the simulation process via the DEVS simulator.

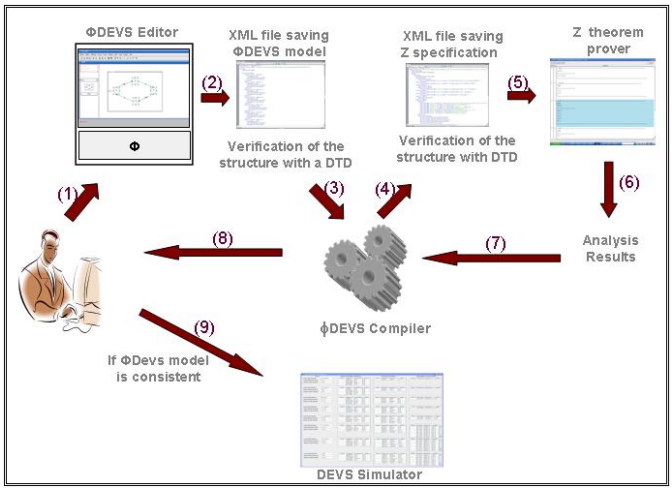

Fig. 2. Formal verification of a ϕDEVS model with Z notation

3.1 ϕDEVS-Compiler design

ϕDEVS-Compiler is based on *XSLT(eXtensible Stylesheet Language Transformations)* which is an XML-based language used for the transformation of XML documents into other XML documents. In our case, the source XML file saves a ϕDEVS model (Fig. 3) and the resulting XML file saves the equivalent Z specification (Fig. 4). We did not develop the whole trees for space reason. The transformation and the verification algorithms are given below.

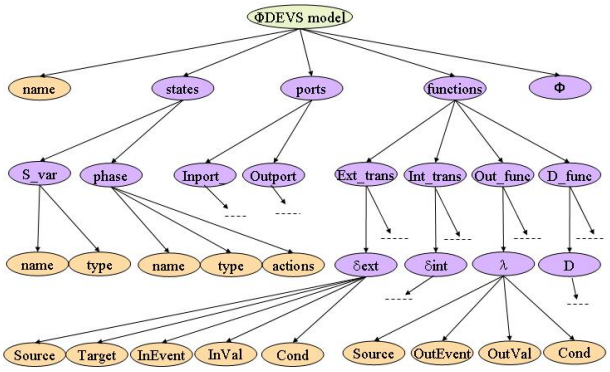

Fig. 3. The tree of a DEVS XML file

3.1.1 Transformation algorithm. This algorithm concerns the transformation of a *ϕDEVS* XML file into a Z file. It is described by the algorithm 1.

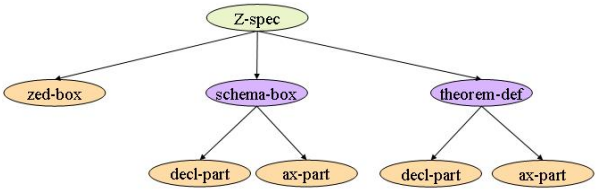

Fig. 4. The tree of the Z XML file

3.1.2 Verification algorithm. Once the transformation is performed, the *$\phi DEVS-Compiler$* proceeds to the verification process which is described by the algorithm 2.

3.2 ϕDEVS-Compiler interface

The ϕDEVS interface is shown in Fig. 5. We integrated our tool with LSIS_DME. Therefore, once the DEVS user establishes the model, invariant properties can be added by clicking on the "Add Constraints" button: thus a frame will open. This frame contains a text field to capture only one constraint. If the user wants to add a new constraint, the "new constraint" button should be clicked on, and then a new text field will appear. The user can also remove a constraint by clicking on the "delete" button. Finally, when the user has finished inserting all the constraints, the user should then click on the "validate" button. The user can also compile the model by clicking on the "ϕDEVS-Compiler" button. The compiling process is described in the algorithms mentioned in the previous chapter. It is divided into two steps, the first one is the transformation process: the XML file saving the ϕDEVS model is transformed into another XML file saving the equivalent Z specification, the second one is the verification process: the resulting XML file is loaded in the Z/EVES theorem prover [8] and verified using proving techniques offered by this tool. The analysis results generated by the compiling process are visualized for the user. When there are no conflicts in the ϕDEVS model, i.e., this is traduced by writing "the model is consistent" within the analysis results box, the user can go to the simulation process to analyze the behavior of the model. We offer the possibility of seeing the resulting Z specification on the Z/EVES editor by clicking on the "Show Z spec" button (see Fig. 6). This button is disabled until enabling the "ϕDEVS-Compiler" button. The results of proofs are shown in the second column on the left. The status of each paragraph is shown in two columns to the left of the paragraph. The leftmost column shows one of three symbols:

—"?" indicates that the paragraph has not been checked.

—"Y" indicates that the paragraph has been checked and has no syntax or type errors.

—"N" indicates that the paragraph has been checked and has errors.

This column permits checking that the transformation algorithm is valid. In fact, it shows that the resulting Z specification and theorems are correctly written and consistent with the Z notation.
The next column shows the proof status for the paragraph, using one of three symbols

—"?" indicates that the paragraph has not been successfully checked (so the proof status cannot be determined).

—"Y" indicates that the paragraph has no unproved goals.

—"N" indicates that the paragraph has an associated goal that is unproven.

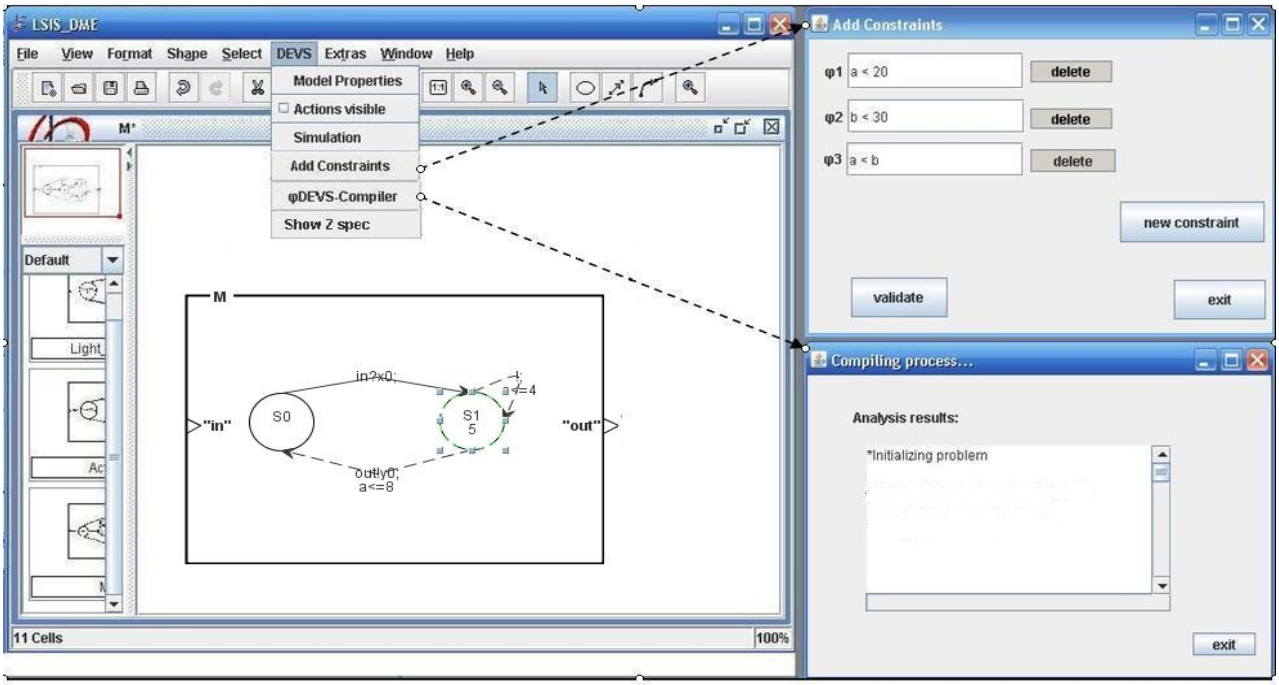

Fig. 5. ϕDEVS-compiler interface

This column permits checking the properties of the *ϕDEVS* model described with Z theorems.
Z/EVES uses here the regular definition expansion technique to rewrite the proofs, using the concerned state and operation schemas. Then it simplifies the calculation by putting all conditions at the same level (i.e., removing the local existential quantifiers) and replacing the dotted variables by their values.

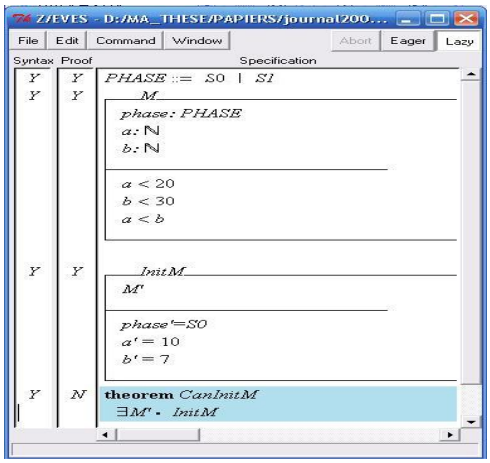

Fig. 6. The resulting Z specification appeared on the Z/EVES editor

4. CONCLUSION AND FUTURE WORK

This paper contributes to works dealing with the improvement of the *V&V* of simulation models via the integration of FM. Our goal was to extend DEVS framework to *ϕDEVS* in order to support safety properties modeling and verfication. *ϕDEVS* model could be translated to Z specification, then Z tools could be therefore used to verify the model consistency by checking wether the safety properties are preserved. This kind of verification is called proof obligation. We developed a tool automating the verification of safety properties, we call it *ϕDEVS − Compiler*. As known in software engineering, assertions are used to avoid programs to use insignificant tests that increase running time [9] [10]. Logically, it seems that our apporach which checks assertions using *ϕDEVS − Compiler*, reduces significantly the simulation time. We plan to extend our approach in order to verify other properties, such as liveness property which means ”something good must happen” [7].

5. REFERENCES

[1] B. Zeigler. *Theory of Modelling and Simulation*. Robert F. Krieger Publishing, 1976.

[2] D. R. Kuhn, D. Craigen, and M. Saaltink. Practical application of formal methods in modeling and simulation. In *SCSC'03*, 2003.

[3] M. Clarke and M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, December 1996.

[4] In *a Workshop on Model and Simulation Verification and Validation for the 21st Century, Laurel, MD, CD-ROM*. The Society for Modeling and Simulation, 2002.

[5] D. R. Kuhn, M. Chandramouli, and R. W. Butler. Cost effective use of formal methods in verification and validation. In *Proc. Workshop on Foundations for M&S and V&V in the 21st Century, SCS, San Diego, CA*, 2002.

Algorithm 1: transformation algorithm

STEP 1. *Creation of the Z file*

(*1*) *CREATE a Z file for writing the Z specification corresponding the ϕDEVS model.*

STEP 2. *Free types definition*

(*2*) *CREATE zed-box node:*
PHASE ::= "FOR EACH phase node WRITE phase.name |"

(*3*) *GO TO S_var (states variables) node:*
IF S_var.type = finite set
THEN Creating zed-box node (Free type definition of the finite set)

(*4*) *GO TO Input_var (Input variables) node:*
IF Input_var.type = finite set
THEN Creating zed-box node (Free type definition of the finite set)

(*5*) *GO TO Output_var (Output variables) node:*
IF Output_var.type = finite set
THEN Creating zed-box node (Free type definition of the finite set)

STEP 3. *Creation of the abstract state*

(*6*) *CREATE schema-box node:*
name = "ϕDEVS.name"
decl-part = " FOR EACH S_var node WRITE
S_var.name : S_var.type"
ax-part = "ϕDEVS.ϕ"

STEP 4. *Creation of the initializing operation schema*

(*7*) *GO TO phase node WHERE phase.type= "Initial"*

(*8*) *CREATE schema-box node:*
name = "Initializing_ϕDEVS.name"
decl-part = " ϕDEVS.name'"
ax-part = "phase.actions"

STEP 5. *Creation of operations schemas from external transitions*

(*9*) *FOR EACH δ_{ext} node CREATE schema-box node:*
name = source.phase.name-target.phase.name
decl-part = " Δ ϕDEVS.name
InEvent? : Inport_var.InEvent.type"
ax-part = " InEvent? = InVal
FOR EACH S_var node WRITE
S_var.name = Source.S_var.val
Cond
FOR EACH S_var node WRITE
S_var.name' = Target.S_var.val"

STEP 6. *Creation the operations schemas from internal transitions and output functions*

(*10*) *FOR EACH δ_{int} node AND*
IF $\exists$ λ node WHERE λ .source=δ_{int}.source THEN
CREATE schema-box node:
name = source.phase.name-target.phase.name
decl-part = " Δ ϕDEVS.name
OutEvent! : Outport_var.OutEvent.type"
ax-part = " FOR EACH S_var node WRITE
S_var.name = Source.S_var.val
Cond
FOR EACH S_var node WRITE
S_var.name' : Target.S_var.val"
OutEvent!=OutVal"
ELSE lines containing outEvent! are deleted.

Algorithm 2: verification algorithm

STEP 1. *Proof obligation*

(*1*) *CREATE theorem-def node :*
decl-part = "canInit"
ax-part = " $\exists$ ϕDEVS.name' $\bullet$ InitϕDEVS.name "

(*2*) *FOR EACH δ_{ext} and δ_{int} node CREATE theorem-def node:*
decl-part = " theorem name"
ax-part = " pre-condition theorem"

[6] W. Trojet and T. Berradia. System reliability using simulation models and formal methods. *International Journal of Computer Applications*, 132(17):1–8, December 2015. Published by Foundation of Computer Science (FCS), NY, USA.

[7] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on software engineering*, March 1977.

[8] M. Saaltink. *The Z/EVES 2.0 User's Guide*. TR-99-5493-06a. ORA Canada, 1999.

[9] K. Mughwal and R. Rasmussen. *A programmer's guide to Java scjp certification (chapter 6 section 10)*. Addison wesley, third edition, december 2008.

[10] D. Rosenblum. A practical approach to programming with assertions. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 21, january 1995.