International Journal of Computer Applications (0975 - 8887)
Volume 132 - No.17, December 2015

System Reliability using Simulation Models and Formal Methods

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

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

ABSTRACT

The general scope of the paper consists of improving the verification of simulation models through the integration of formal methods. We offer a formal verification approach of DEVS models based on Z notation. DEVS is a formalism that allows the description and analysis of the behavior of discrete event systems, i.e., systems whose state change depends on the occurrence of an event. A DEVS model is essentially validated by the simulation which permits of verifying whether it correctly describes the behavior of the system. However, a simulation does not cover all possible cases that means the model is validated only for the expected behaviors. For this reason, we have integrated the Z formal specification language in the DEVS formalism to detect errors before simulation which is still an important step for the validation. This integration 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. Such consistency is fulfilled by determinism and completeness. Thus, a DEVS model is subjected to an automatic formal verification before proceeding to its simulation.

Keywords

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

1. INTRODUCTION

Modelling and simulation (M&S) play increasingly an important role in the understanding of how things function, and they are essential to effective and efficient design, evaluation, and operation of new products and systems [1]. Modelling is the process of producing a model; a model is a representation of the construction and working of some system of interest. One purpose of a model is to enable the analyst to predict the effect of changes to the system. A simulation is an execution of a model. The outputs of the simulation can be studied, and hence, properties concerning the behavior of the actual system or a subsystem can be inferred. In its broadest sense, simulation is a tool to evaluate the performance of a system, existing or proposed, under different configurations of interest and over long periods of real time. 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.
The conceptual model describes what is to be executed by the simulation [2], thus it must include assumptions about the system and its environment, equations and algorithms, data, and relationships between model entities. Although algorithms and equations are necessarily formal statements, the assumptions and relationships are most often described using natural language, which introduces the potential for ambiguities and misunderstandings between developers, users, and subject matter experts. 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.
The subject of this paper is discrete event simulation, in which the central assumption is that the system changes instantaneously in response to certain discrete events. Therefore, we will focus on the Discrete EVent System Specification (DEVS) formalism [4]. Our aim is to improve the DEVS conceptual model by conducting a formal verification in order to detect eventual errors before the simulation process (see Fig. 1). An error can be either an ambiguity or an incompleteness. There is no DEVS tool which detects automatically these errors. However we suggest to do this by transforming the DEVS model into a Z formal specification and checking for such errors in the resulting specification by reusing Z tools already available.
The main reasons to choose the Z specification language are the similarity and the complementarity of DEVS and Z. In fact :

—Z and DEVS are state/transition approaches: DEVS model transitions can be interpreted by operations in Z, and the set of states of the DEVS model can be interpreted by the abstract state in Z.

—Z and DEVS are modular: the modularity in DEVS can be interpreted by the promotion process in Z, which consists of forming the global specification by combining the partial specifications.

—Z permits a functional description of a system while DEVS permits a behavioral one.

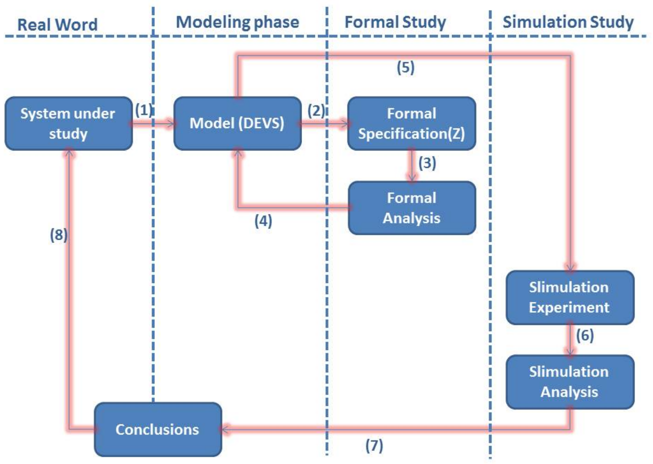


Fig. 1. Integration of Z within DEVS

—Z has a logical semantics while DEVS has an operational semantics.

—Verification and Validation (V&V) in Z is performed via formal techniques while V&V in DEVS is performed via simulation techniques.

2. INTEGRATION OF FORMAL METHODS INTO DISCRETE EVENT M&S

2.1 Related works

In the literature of DEVS, there are some works which deal with the integration of formal methods into the DEVS framework.
Traoré [5–7] defined a formalism called "ZDEVS" which represents each DEVS entity (model or simulator) in the Z-Object language. The static structure of ZDEVS identifies data and operations of the model/simulator and represents them with Z class schemas. The dynamic structure of ZDEVS is described within a pattern which has the style of a finite state machine generating the dynamics (temporal evolution) of the static structure of ZDEVS. According to Traoré, ZDEVS is used to analyze the DEVS model, check for inconsistencies and incompleteness, prove properties, and generate traces. These properties are common to any system, but they are specifically relevant to simulation due to the importance of these systems in decision making processes. Thus, the ultimate goal is to build models of better quality and to increase the understanding of key concepts such as Verification, Validation and Accreditation (VV&A), reuse and composability.
Traoré has shown how the resulting specification allows the reuse of tools supporting the FM to perform formal reasoning on the source model. However the major drawback of his approach is that the transformation is not subjected to formal rules. Indeed, it is based on case studies that were not generalized. Hence, it can not be automatized.
Crisitia [8, 9] specified the equivalent TLA+ (Temporal Logic of Action +) specification [10] of a DEVS model [atomic and coupled] under certain rules. In addition, he explained how to interpret the notion of elapsed time in DEVS with TLA+. The motivation of his approach consists of bridging the gap between DEVS and other formal notation to make DEVS known by the FM community of Computer Science. This translation is beneficial for DEVS since it lays the basis for the formal semantics of this powerful modelling language. Having a TLA+ specification of a DEVS model enables the formal verification of the model or the model-checking by means of the tools already available for TLA+. This approach permits DEVS models to be written in a widely known formalism, used by FM researchers and practitioners to specify hardware- or software-based reactive or concurrent systems.
Cristia has shown how a DEVS model can be transformed into a Z specification. This transformation is subjected to detailed formal rules. However the major drawback of his approach is that the verification process was not elaborated. In fact, Cristia did not explain which kinds of properties can be proved and how can the transformation be used to improve the V&V of DEVS models.
Hong [11] integrated TL (Temporal Logic) into the DEVS framework. In fact, temporal constraints that present temporal properties of the system are required for logical analysis (proving that there are no logical conflicts in the procedure rules). These constraints are expressed by using TL. The TL formula is translated into a finite state automaton. The state information is obtained from the automaton. This information is applied to the DEVS model to obtain a projected state space. Logical analysis is performed by using the projected state space and the finite automaton of TL formulas to check whether a TL assertion accepts the state transition sequences of the DEVS model. This approach offers a way to reduce the state spaces of DEVS models by using the projection mechanism.
The DEVS-TL approach is distinguished by the duality of specification languages. The main advantage of the dual language approach is its flexibility: the use of a specification language provides an uniform notation for expressing a wide variety of correctness properties and it separates models from reachability assertions [12]. Unfortunately the dual language approach also has the state explosion problem for complex systems. Therefore a partial proof against given specifications is a reasonable solution to these problems [13]. The dual language approach with a projection mechanism can be an efficient method for the validation of large systems. The use of the projection mechanism (projection of DEVS model states in the states of the finite states automaton of TL assertions) is a part of this specification. However, this mechanism is done manually (there are no tools that support it), this probably encourages projection errors especially in complex systems. In addition, the state variables of DEVS models are considered to take fixed values and can not describe functions: this reduces the set of models on which the approach can be applied.
Our approach permits of avoiding the drawbacks of the dual approach. Besides, we offer an automation of the transformation by stating formal rules for mapping a DEVS model into a Z specification. We offer also the automation of the formal verification of the DEVS model via applying Z tools to the resulting specification. The formal verification consists of proving determinism and completeness. A summary of this discussion is shown in Figure 2.

2.2 Checking errors before simulation phase

The properties of the DEVS model we intend to verify are:

—**Determinism:** the behavior of a system is unambiguously defined for each combination of inputs and current state. An ambiguity occurs when there are several conditions which hold simultaneously, but each one implies a different transition. In this case, the model does not know which transition to choose.
Example:
$property_i$: IF system_temperature ¿ 20 THEN the cooling system has to be activated.
$property_j$: IF system_temperature ¿ 40 THEN the fire alarm has

Approach	Goal	Advantages	Drawbacks
DEVS-TL (Hong 1994)	Checking if DEVS model satisfies properties written in TL	Flexibility	high cost of integration
DEVS-¿Z-Object (Traoré 2006)	Formal reasoning on DEVS models	Formal verification of some properties	Transformation not subjected to formal rules (completely manual)
DEVS-¿TLA+ (Cristia 2007)	A logic semantics for DEVS models	Transformation subjected to formal rules	Verification process not elaborated
DEVS-¿Z (Our approach)	Checking the consistency of a DEVS model	Automatic transformation and verification	Sub-class of DEVS models and properties

Fig. 2. Main DEVS approaches integrating FM

to be released.
The question here, knowing that the system does just one task at a time, which task will it do when system_temperature ¿ 40? Activate the cooling system or the fire alarm? Thus, there is an ambiguity.
We are interested in the determinism property for simulation reasons. In fact if the model is non-deterministic, a simulation can not be achieved, at least in our DEVS simulator.

—**Completeness:** The behavior of a system is defined for each combination of inputs and current state. Incompleteness occurs when there are several conditions held on transitions released after a state but these conditions are incomplete. Thus, there is a possible scenario which is not specified.
Example:
$property_i$: WHEN 0 < car_speed ¡ 50 THEN the electric engine is activated.
$property_j$: WHEN 50 ¡ car_speed ¡ 220 THEN the thermal engine is activated.
The question here which engine will work when the car_speed = 50? Thus there is incompleteness.

3. REVIEW ON DEVS

DEVS [4, 14] is a modular formalism for discrete events systems. It allows for the modelling of the behavior of systems. A DEVS model has a time base, inputs, states (with functions of transition from one state to another) and outputs. Complex models are built from atomic and coupled models connected together in a hierarchical fashion. Interactions are mediated through input and output ports, which allows for modularity.

3.1 Specification of a DEVS model

According to the literature on DEVS [4], the specification of a discrete event model is a structure, M, given by:

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

where X is the set of the external input events:

—X={(p,v) — p∈InPorts, v $\in X_p$}, the set of input ports and values

Y, the set of the output events:

—Y={(p,v) — p∈OutPorts, v∈ Y_p}, the set of output ports and values

S, the set of sequential states:

—S=$V_{s_1} \times V_{s_2} \times ...V_{s_n}$, where V_{s_k} is the domain of the state variable s_k and n is the number of state variables.

λ is the output function:

—λ: S $\rightarrow$ Y

D is the lifetime function of the states:

—D: S $\rightarrow R^+ \cup \infty$. For a given state, s, D(s) represents the time interval during which the model will remain in the state s if no external event occurs.

A state may be viewed as passive when its lifetime is assumed to be infinite or active when the lifetime interval is assumed to be a finite real positive number, Zeigler [4] introduces the concept of total states, Q, of a model as:

$$Q=\{(s,e) — s\in S, 0\leq e\leq D(s)\}$$

where e represents the elapsed time in state s. The concept of total state is fundamental in that it permits one to specify a future state based on the elapsed time in the present state.
δ_{int} is the internal transition function:

—δ_{int}: S $\rightarrow$ S defines the state changes caused by internal events,

and δ_{ext} is the external transition function:

—δ_{ext}: (Q,X) $\rightarrow$ S specifies the state changes due to external events.

3.2 Graphical representation of a DEVS model

Song [15] represented the behavioral description of an atomic model by a state transition diagram, which consists of nodes and two-colored edges (see Fig. 3). Each node represents a state, a dotted arc denotes an internal transition, and solid arc, an external transition. An output event is specified on a dotted line by an output port followed by a message name with output operator '!'. Similarly an input event is specified on a solid line by an input port followed by a message name with input operator '?'. Optionally, a transition condition can be specified after an input or output event with a separator notation '@'. It is natural that a time advance in a state be attached to the state node because it represents a sojourn time to fulfill its activity.
In our approach, we keep this notation for representing graphical atomic DEVS models and we extend it by making explicit the update of values of state variables and representing an active state with a dotted node.

4. REVIEW ON Z

4.1 The Z specification language

Z [16] is a state-based formal specification language based on the established mathematics of set theory and first-order logic. The

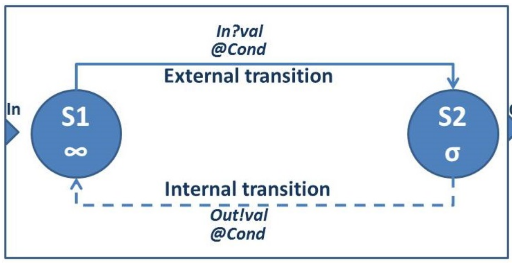


Fig. 3. Graphical notation of a DEVS model

set theory used includes standard set operators, set comprehension, Cartesian products and power sets. Z has been used to specify data and functional models of a wide range of systems, including transaction processing systems and communication protocols. In Z, mathematical objects and their properties are collected together in schemas: patterns of declaration and constraint. The schema language [17] is used to structure and compose descriptions: collating pieces of information, encapsulating them, and naming them for reuse. A schema contains a declaration part and a predicate part. The declaration part declares variables and the predicate part expresses requirements about the values of the variables. There are two forms of notation of the Z schema, vertical and horizontal. The vertical notation is the following: Schema name declarations (state space) predicates and the horizontal notation is the following:

schema [declarations — predicates]

Example. The following schema encapsulates the state information of a light object. Light dim : 0..100
on : **B** dim¿0⇔ $on = true$ The declaration part contains the declaration of two variables. The variable *dim* represents the illumination of the light object, which is a value in between 0 and 100 (in percent) and *on* is a Boolean variable indicating whether the light object is on or not. The predicate part, referred to as the state invariant, places a constraint on the values of the two variables, i.e., the *dim* is nonzero if and only if the light object is on.
Example. The operation schema defines the operation Adjust by how the state variables of the Light schema are updated. Adjust $\Delta Light$
$dim? : 0..100 on = true \wedge dim' = dim?$ The schema name after Δ indicates the state schemas to be updated by the operation. The variable *dim?* is an input from the environment. The state-update is expressed using a predicate involving both primed and unprimed state variables. In particular, the operation can only be applied when the light is on and, after the operation, the light level is set to be *dim?*. If *dim?* is zero, the state variable *on* will be set to false because of the state invariant.
A Z package contains one state schema, one initial schema which identifies the initial valuation of the state schema, and a number of operation schemas that may update the state schema. In other words, a Z package identifies the state space of a set of objects.

4.2 The Z package

Generally, the structure of Z specification package consists of [16]:

(1) Declaration of free types used lately into the specification.
(2) Definition of the global abstract state of the model: State declarations of the variables describing the
state of the model predicates (constraints on states variables)
(3) Definition of an initial state of the model: InitializingState State Initialization of states variables
(4) List of operations, each one is presented by the following schema: Operation $\Delta State(\Delta : to\ say\ that\ the\ state\ of\ the$
$system\ is\ changed)\ OR$
$\Xi State(\Xi : to\ say\ that\ the\ state\ of\ the$
$system\ is\ the\ same)$
$Eventual\ declaration\ of\ input\ variables$
$('?'\ has\ to\ be\ placed\ after\ each\ input\ variable)$
$Eventual\ declaration\ of\ output\ variables$
$('!'\ has\ to\ be\ placed\ after\ each\ output\ variable) Pre-$
$condition(condition\ on\ values\ of\ state$
$variables\ and\ eventual\ input\ variables$
$before\ the\ operation)$
$Post-condition(values\ of\ state\ variables$
$and\ eventual\ output\ variables$
$after\ the\ operation)$

5. FROM A DEVS MODEL TO A Z SPECIFICATION

We have elaborated formal rules to transform a DEVS model into a Z specification. In this transformation we used three assumptions:

(1) Basically Z does not specify real and rational numbers, thus we transform DEVS models of which variables are not real and rational numbers.
(2) Basically Z does not specify the time, thus the elapsed time variable and the life time function will be excluded from the transformation.
(3) Only deterministic DEVS models are covered by this transformation. Thus we can exploit the works of Hwang and Ziegler [18, 19] which deal with the abstraction of infinite behaviors of DEVS models into finite-vertex isomorphic graphs, called reachability graphs when using a sub-class of DEVS called Finite Deterministic DEVS. However, we don't exclude infinite behavior of DEVS models.

At a high level of abstraction, the transformation is divided into two parts: set transformations and function transformations.
The structure of the DEVS model is:

M = ¡X, Y, S, λ, D, δ_{int}, δ_{ext}¿

Set transformation:

means: "is transformed into."

—X X_Z — X_Z is the set of all inputs of Z package operations.
—Y Y_Z — Y_Z is the set of all outputs of Z package operations.
—S Z abstract state.

Function transformations:

—δ_{int} et δ_{ext} schema operations.
—λ output values of operations.

Formal Rules

(1) *Free type declaration*
Let V be a finite set of variable values of a DEVS model, V = $val_1, ..., val_n$, V is traduced by:
V::=$val_1|val_2|...|val_n$
(2) *Establishing the abstract state*
Let M be the name of a DEVS model and S the set of states, let $s \in S$:
M [s : S]

(3) *Establishing the initializing operation schema*
Let s_0 be the initial state of the DEVS model, $s_0 \in S$:
InitM [M'— $s' = s_0$]

(4) *Establishing operation schemas*
—External transition function
$\delta ext(source, e, (in?x)) = target\ IF\ cond_{src-trg}$
$source - target\ \ [\Delta M; in? : Domain_{in?} | in? = x;$
$s = source; \quad cond_{src-trg}; s' = target]$
where $Domain_{in?}$ is the domain of the input port variable $in?$, $cond_{src-trg}$ is the condition on the source when true, the model can cross the transition from the source state to the target state, $cond_{src-trg}$ must be written in the form of Z notation.
—Internal transition function and output function

$$\begin{cases} \delta int(source) \ = \ target\ IF\ cond_{src-trg} \\ \\ \lambda(source) \ = \ (out!y)\ IF\ cond_{src-trg} \end{cases}$$

$source - target\ \ [\Delta M; out! : Domain_{out!} | s = source;$
$cond_{src-trg}; \quad s' = target; out! = y]$
where $Domain_{out!}$ is the domain of the output port variable $out!$. It is possible that the internal transition does not hold any output. In this case, lines containing the variable $out!$ are deleted from the schema.

6. FORMAL VERIFICATION OF A DEVS MODEL

6.1 Verification of the determinism

The determinism is the absence of ambiguity in the model. This property is important because we use deterministic DEVS models in our approach.

DEFINITION 1. *Determinism in a DEVS model is traduced by the following properties [20]:*

—*External transitions*
δ_{ext}*(src, e, ?x1) = trg1 if Cond1* $\wedge$
δ_{ext}*(src, e, ?x2) = trg2 if Cond2* $\wedge$
?x1 = ?x2 $\wedge$
trg1 $\neq$ *trg2*
$\Longrightarrow \neg$*(Cond1* $\wedge$ *Cond2) = vrai*
—*Internal transitions*
δ_{int}*(src) = trg1 if Cond1* $\wedge$
δ_{int}*(src) = trg2 if cond2* $\wedge$
trg1 $\neq$ *trg2*
$\Longrightarrow \neg$*(Cond1* $\wedge$ *Cond2) = vrai*

In other words, the model is said to be deterministic when external transitions (or internal transitions) released from each state hold distinct conditions (when some condition is true all the others are false). Otherwise, the model is said to be non-deterministic.

—δ_{ext}*(src, e, ?x1) = trg1 if Cond1* is traduced in Z:
$src - trg1\ \ [\Delta M; in? : Domain_{in?} | in? = x1; s = src;$
$Cond1; s' = trg1]$
—δ_{ext}*(src, e, ?x2) = trg2 if Cond2* is traduced in Z:
$src - trg2\ \ [\Delta M; in? : Domain_{in?} | in? = x2; s = src;$
$Cond2; s' = trg2]$

We apply the definition of determinism to the corresponding Z specification, thus:

$$[?x1 = ?x2 \wedge trg1 \neq trg2 \Rightarrow \neg(Cond1 \wedge Cond2) = vrai]$$
$$\Longleftrightarrow$$
$$[?x1 = ?x2 \wedge trg1 \neq trg2 \Rightarrow \neg(\textbf{pre}\ src - trg1 \wedge \textbf{pre}\ src - trg2)]$$

We summarize: checking the determinism of a DEVS model is performed by checking whether all the preconditions are pairwise distinct.
Formally this is traduced by [21]:
If the behavior of a component is defined as a set of operations $\{Op_1, Op_2, ..., Op_n\}$ over the inputs and state, then a conjecture on the determinism of the specification of that component can be formulated as follows:

$$\vdash \forall GlobalState, \forall Inputs, \forall i, j : 1..n | i \neq j \ \bullet$$
$$\neg(pre\ op_i \wedge pre\ op_j)$$

Z theorem proving tool verifies the operation two to two by writing the following theorem for each couple of operation (op_i, op_j):
Theorem *CheckingDeterminism*
$\forall GlobalState, \forall Inputs \bullet \neg(\textbf{pre}\ op_i \wedge \textbf{pre}\ op_j)$

In the case of internal transitions, the theorem is written as follows:
Theorem *CheckingDeterminism*
$\forall GlobalState \bullet \neg(\textbf{pre}\ op_i \wedge \textbf{pre}\ op_j)$
If the Z theorem proving tool returns *true* for all couples of operations, that means that DEVS transitions corresponding to op_i and op_j are deterministic, or else the Z theorem proving tool can determine non-deterministic operations and the corresponding DEVS model transitions can be localized. We can optimize the verification by applying the theorem only on the operations having the same source state.

6.2 Verification of the completeness

A model is complete when it describes all possible scenarios. This property guarantees that the model is not locked in a state during the simulation process.

DEFINITION 2. *We define completeness in DEVS by the following property:*

—*External transitions*
δ_{ext}*(src, e, ?x1) = trg1 if Cond1* $\wedge$
δ_{ext}*(src, e, ?x2) = trg2 if Cond2* $\wedge$
?x1 = ?x2 $\wedge$
trg1 $\neq$ *trg2*
$\Longrightarrow$ *Cond1* $\vee$ *Cond2 = vrai*
—*Internal transitions*
δ_{int}*(src) = trg1 if Cond1* $\wedge$
δ_{int}*(src) = trg2 if cond2* $\wedge$
trg1 $\neq$ *trg2*
$\Longrightarrow$ *Cond1* $\vee$ *Cond2 = vrai*

In other words, the model is said to be complete when external transitions (or internal transitions) released from each state hold complete conditions (one condition at least must be true). Otherwise, the model is said to be incomplete. The Z operations corresponding to "δ_{ext}(src, e, ?x1) = trg1 if Cond1" and "δ_{ext}(src, e, ?x2) = trg2 if Cond2" are given above. Formally, checking completeness is performed by checking the following theorem for each group of operations having the same source state [21]:
If the behavior of a component is defined as a set of operations $\{Op_1, Op_2, ..., Op_n\}$ over the inputs and the same source state phase, then a conjecture on the completeness of the specification of that component can be formulated as follows:

$\vdash \forall GlobalState | phase = SourcePhase, \forall Inputs \bullet$
$pre\ op_1 \vee pre\ op_2 \vee ...pre\ op_n$

Z theorem proving tool verifies the following theorem for each group of operations $\{op_{1..n}\}$:
Theorem *CheckingCompleteness*
$\vdash \forall GlobalState | phase = SourcePhase, \forall Inputs \bullet$
$pre\ op_1 \vee pre\ op_2 \vee ...pre\ op_n$

In the case of internal transitions, the theorem is written as follows:
Theorem *CheckingCompleteness*
$\vdash \forall GlobalState | phase = SourcePhase \bullet$
$pre\ op_1 \vee pre\ op_2 \vee ...pre\ op_n$
If Z theorem proving tool returns *true* for all groups of operations having the same source state, that means the model is complete. Otherwise Z theorem proving tool can determine the incomplete operations, and therefore their corresponding DEVS model transitions can be localized.

7. SOFTWARE DESIGN FOR THE Z TRANSFORMATION AND VERIFICATION PROCESSES

We developed a tool which implements the transformation and the verification of a DEVS model: we call this tool "DEVS-Compiler" (see Fig. 4). Once DEVS-Compiler reads a DEVS model saved in an XML file it generates another XML file containing the equivalent Z specification, 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 errors in the model, the DEVS user can fix them and verify the model again via DEVS-Compiler. Once the model is consistent (no errors in the model), the user proceeds to the simulation process via the DEVS simulator.

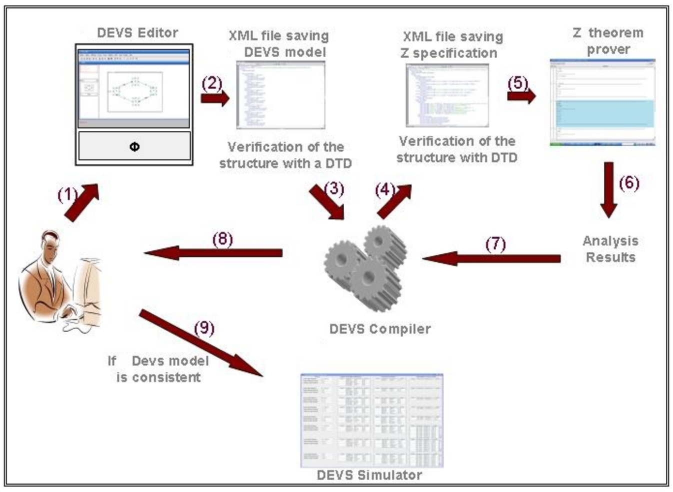

DEVS Editor
XML file saving DEVS model
XML file saving Z specification
Z theorem prover
(2)
(5)
Φ
Verification of the structure with a DTD
Verification of the structure with DTD
(1)
(3)
(4)
(6)
(8)
(7)
Analysis Results
DEVS Compiler
(9)
If Devs model is consistent
DEVS Simulator

Fig. 4. Formal verification of a DEVS model with Z notation

7.1 DEVS-Compiler design and interface

DEVS-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 DEVS model and the resulting XML file saves the equivalent Z specification.
The DEVS interface is shown in Fig. 5. We integrated our tool with LSIS_DME. Therefore, once the DEVS user establishes the model, he can compile the model by clicking on the "DEVS-Compiler" button. The compiling process is divided into two steps, the first one is the transformation process: the XML file saving the DEVS 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 [22] 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 errors in the DEVS model, i.e., this is traduced by writing "the model is consistent" within the analysis results box, the user can start the simulation process to analyze the behavior of the model.

8. CONCLUSION

This paper contributes to works dealing with the improvement of the $V\&V$ of simulation models via the integration of FM. Thus, it establishes a bridge between $M\&S$ and FM. Our goal was to propose an approach for the formal verification of the DEVS models before proceeding to the simulation phase. The proposed approach is based on the integration of Z notation into the DEVS formalism in order to verify a DEVS model determinism and completeness before the simulation process. This integration is done in two steps:

—transforming DEVS formalism to a Z specification.

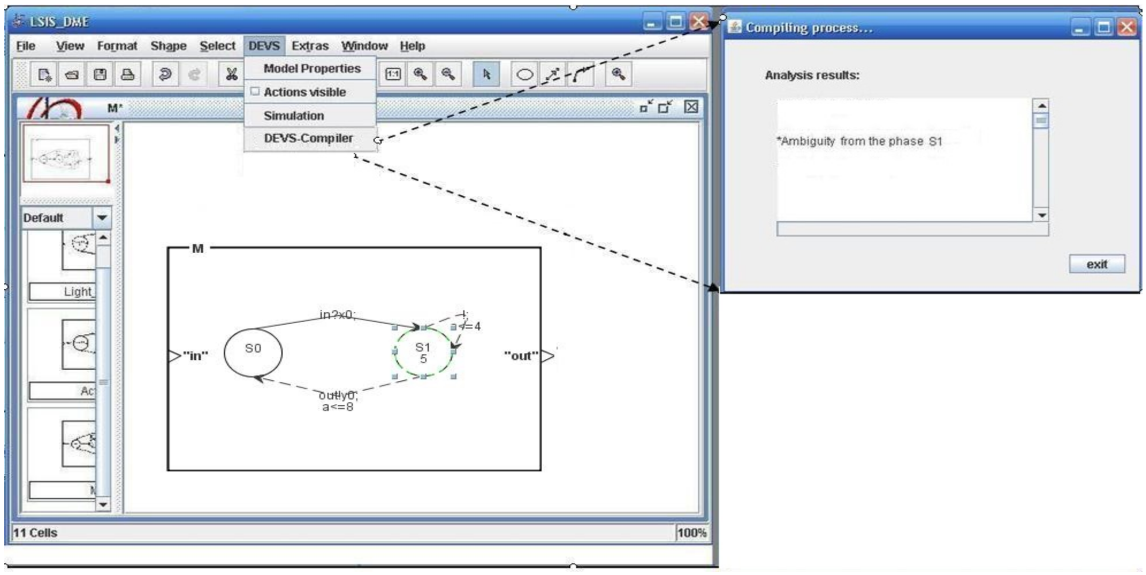
—verifying the resulting specification using a Z tool and drawing conclusions about the DEVS model.

Regarding the first step, we elaborated a set of rules translating a sub-class of DEVS models to Z formal specifications. In fact, a Z document consists of four parts: (1) the first one contains free types declaration, each type identifies a finite set of values, it is deduced from the finite set of values of a DEVS model variables (input, output, and state variables), (2) the second one contains the abstract state schema which includes all state variables of the system and constraints, it is deduced from DEVS model state variables, (3) the third one contains the initializing schema which is deduced from the initial state of the model and (4) the last one contains operation schemas performed by the system which are deduced from transition (internal and external) and output functions. Regarding the second step, the formal verification of a DEVS model consists of checking the following properties on the resulting Z specification:

—determinism: this property guarantees the absence of eventual ambiguities in the model.

—completeness: this property guarantees that the behavior of the system is defined for each combination of inputs and current state.

This step is performed by elaborating the formal theorems describing these properties and using a Z theorem proving tool (Z/EVES) to check these theorems on the resulting specification.
We developed a tool automating our approach, we call this tool $DEVS - Compiler$. It performs the two steps of the integration of Z notation into atomic DEVS models in a systematic way.
As known in software engineering, assertions are used to avoid programs to use insignificant tests that increase running time [23] [24]. Logically, it seems that our apporach which checks assertions using $DEVS - Compiler$, reduces significantly the simulation time.



Analysis results:

*Ambiguity from the phase S1

Fig. 5. DEVS-compiler interface

9. FUTURE WORK

In one hand, we plan to extend our approach in order to verify other properties, such as safety property which means: ”something (bad) will not happen” and liveness property which means ”something good must happen” [25].
In other hand, we are applying our approach for the improvement of cooperative systems modelling.
In addition, we are working on enlarging the sub-class of DEVS models by integrating real numbers. In fact we will use Z extensions [26, 27] and other Z tools.
Some experimentations will be done to prove that the $DEVS - Compiler$ has a great effect on reducing time simulation since it permits to avoid simulation deadlock in the case of ambiguity or incompleteness.

10. REFERENCES

[1] A. Maria. Introduction to modeling and simulation. In *Proceedings of the 1997 Winter Simulation Conference*, 1997.

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

[5] M. K. Traoré. Combining devs and logic. In *OICMS*, 2005.

[6] M. K. Traoré. Analyzing static and temporal properties of simulation models. In *Proceedings of the 2006 Winter Simulation Conference*, pages 897–904, 2006.

[7] M. K. Traoré. Making devs models amenable to formal analysis. In *DEVS/HPC/MMS’06*, 2006.

[8] M. Cristia. A tla+ encoding of devs models. In *International Modeling and Simulation Multiconference 2007. Buenos Aires, Argentina*, 2007.

[9] M. Cristia. Formalizing the semantics of modular devs models with temporal logic. In *7ème Conférence Internationale de Modélisation, Optimisation et Simulation des Systèmes MOSIM 08. Paris, France*, 2008.

[10] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[11] G. P. Hong and T. G. Kim. The devs formalism: A framework for logical analysis and performance evaluation for discrete event systems. *IEEE*, 1994.

[12] J. S. Ostroff. *Temporal Logic for Real-Time Systems, Advanced Software Development Series*. Research Studies Press, 1989.

[13] S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, July 1984.

[14] B. Zeigler, H. Praehofer, and T. Kim. *Theory of Modelling and Simulation. Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.

[15] H. S. Song and T. G. Kim. The devs framework for discrete event systems control. In *Conference Proceedings AI, Simulation and Planning in High Autonomy Systems*, Gainesville, FL, USA, 1994.

[16] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[17] J. Sun and J. S. Dong. Design synthesis from interaction and state-based specifications. *IEEE Transactions on Software Engineering*, June 2006.

[18] M. H. Hwang and B. P. Zeigler. A modular verification framework using finite and deterministic devs. In *Proceedings of 2006 DEVS Symposium, Huntsville, AL, USA*, 2006.

[19] M. H. Hwang and B. P. Zeigler. Reachability graph of finite and deterministic devs networks. *IEEE Transactions on Automation Science and Engineering*, 2009.

[20] N. Giambiasi, J. L. Paillet, and F. Chaane. From timed automata to devs models. In *Proceedings of the 2003 Winter*

Simulation Conference S. Chick P. J. Sanchez, D. Ferrin, and D. J. Uorrice, 2003.

[21] S. Burton, J. Clark, A. Galloway, and J. McDermid. Automated v&v for high integrity systems, a targeted formal methods approach. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, june 2000.

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

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

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

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

[26] S. H. Valentine. Putting numbers into the mathematical toolkit. *Springer Verlag*, pages 9–36, 1993.

[27] S. H. Valentine. An algebraic introduction of real numbers into z. *Harribas*, pages 183–204, 1995.