Towards the Verification and Validation of DEVS Models

Yvan Labiche
Carleton University
Systems and Computer Engineering Dept.
1125 Colonel By Drive
Ottawa, ON K1S5B6, Canada
+1 613 520 2600 ext. 5583
labiche@sce.carleton.ca

Gabriel Wainer
Carleton University
Systems and Computer Engineering Dept.
1125 Colonel By Drive
Ottawa, ON K1S5B6, Canada
+1 613 520 2600 ext. 1957
gwainer@sce.carleton.ca

ABSTRACT
The creation of a simulation model, like the creation of any software product, is guided by principles and procedures that have been reasonably well established within the software engineering community. In the context of a simulation, we need to be able to characterize the dynamic behavior of the system, and such characterization should be expressed in a format that is as clear and unambiguous as possible. Wherever feasible, formal approaches should be used. One of these formal techniques, the DEVS formalism, has gained popularity in recent years. Although some efforts have been dedicated to the Validation and Verification (V&V) of DEVS models, this is an open research area with interesting opportunities for application of advanced software engineering techniques. Indeed, it appears that thanks to the characteristics of DEVS models and the fact that DEVS models can be executed (e.g., the CD++ toolkit allows the use of DEVS and Cell-DEVS formalisms) well-known software testing techniques are worth investigating for the V&V of DEVS models. In this article, we show these similarities and discuss open research paths in the field of DEVS modeling Verification and Validation by means of testing.

KEYWORDS
DEVS, Cell-DEVS, CD++, Verification & Validation, Testing

1. Introduction

In recent years, several efforts have been devoted to define modeling paradigms, improving the analysis of complex dynamic systems through simulation. The creation of a simulation model, like the creation of any software product, is guided by principles and procedures that have been reasonably well established within the software engineering community. A key prerequisite is the specification of a set of requirements, which, in a simulation context, means the characterization of the dynamic behavior of the System of Interest (SoI). It is essential that this characterization be expressed in a format that is as clear and unambiguous as possible. Wherever feasible, formal approaches should be used: e.g., Petri nets, Bond Graphs, Partial/Ordinary Differential Equations, etc. A formalism that gained popularity in recent years is called DEVS (Discrete Event systems Specification). It allows modular description of models that can be integrated using a hierarchical approach [Zeigler et al. 2000]. Different environments have been built to support modeling and simulation of discrete-event systems using DEVS, and a current effort is trying to standardize modeling environments [DEVSTD 2004]. One of them, the CD++ toolkit [Wainer 2002], allows the use of DEVS and Cell-DEVS [Wainer and Giambiasi 2001] formalisms. CD++ provides both graphical and programmatic (C++) ways of building simulation models [Christen et al. 2004].

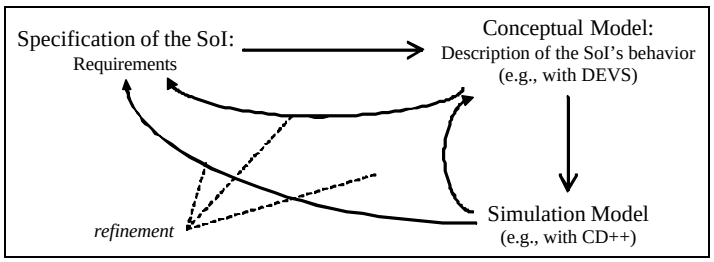


Figure 1 Deriving a simulation model: Life cycle

These activities can be organized in a life cycle (Figure 1), which begins with the definition of the requirements and follows with the definition of the conceptual and simulation models. As any software life cycle, this process is cyclic, allowing refinement. Reasons for refinement include but are not limited to: changes to the level of granularity of the model to better address the goals of the simulation, modification and maintenance of the model(s) to fix defects.

Validation and Verification (V&V) of these different steps has already been identified as paramount for users of such technology [Sargent 2000]. A simulation model can be useful and effective only to the extent that there is enough confidence in the results which it generates. *Validation* is concerned with the fundamental issue of whether or not the simulation model is an acceptable representation of the SoI, when viewed from the perspective of the stated goals of the intended simulation study. Typically, the question to be answered is: "Are we simulating, or have we simulated, the right system?" As the simulation model is an evolution of the conceptual model (together with aspects of the requirements specification), it follows that the validation process needs to focus on the conceptual model. On the other hand, *Verification* is concerned with the transformation of the conceptual model into a simulation model. Here, the question one tries to answer is: "Are we simulating, or have we simulated, the system right?"

V&V activities do not constitute a particular phase of the life cycle, but are an integral part of it, and Validation and Verification (V&V) techniques have to be used throughout this life cycle. V&V activities must be as systematic as possible and must be documented correctly to increase our confidence into simulation results.

Testing, that is the execution of a system with actual values (as opposed to symbolic execution) is one well known technique, among others, that has been used to verify and validate software systems. Given that simulation models such as DEVS models have the interesting property of being "executable" (thanks to environments such as CD++) we are interested in using (possibly adapting) traditional software testing techniques to perform V&V of discrete-event models based on the DEVS formalism.

A first attempt at adding V&V capabilities to CD++ by means of testing was described in [Wainer et al. 2002], where a specific infrastructure for the execution of test cases on DEVS models is described. Specifically, test case data provided by the tester is translated into a DEVS model (a DEVS *Generator* model) that represents a component of an Experimental Framework created for testing purposes, and it is connected to the model under test to generate test input(s). Another DEVS model (called the *Transducer*) is created to collect the output(s) of the model under test and verify whether they correspond to the expected ones, i.e., whether there is a failure. Experiments did show that such infrastructure could indeed help the designer to find defects in the model(s).

Although these facilities provide basic infrastructure to carry out the tests, a systematic approach for the derivation of test cases is required: we want to trust the simulation models. Shortcomings uncovered during the execution of the simulation model imply either shortcomings in the requirements specification or shortcomings in the verification process. In order to ensure trust, we need to address different issues: How has this been achieved? What was the testing strategy followed? (if any) Are the results simply empirical evidence? Are there any measures of code coverage when executing a set of models (benchmarks?).

In this article, we address some of these questions, and provide general ideas on how to solve them in the long term, showing some open research questions and possible solution paths. Very recent efforts [Traoré and Zeigler, 2004] tried to formalize the concepts of Experimental Frameworks and their relationship with DEVS models. A discussion about the relation between these two efforts is outside the scope of this paper, and it could be addressed in future work. In our particular case, we are not focused on the formalization of the Experimental Framework concept, nor in the meta-modeling methods involved. Instead, we want to explore new techniques to incorporate automated testing facilities that are well-known in the Software Engineering community to the testing framework based on Experimental Framework originally presented in [Wainer et al. 2002].

In the following sections, we shortly describe the DEVS formalism and discuss some basic facilities already built into the CD++ toolkit (Section 2), and introduce some software testing terminology (Section 3). We then discuss how traditional software testing techniques pertain to the V&V of simulation models described in DEVS (Section 4). Conclusions are drawn in Section 5.

2. The DEVS formalism and CD++

The DEVS formalism was originally defined in the '70s as a discrete-event modeling specification mechanism. It is a systems theoretical approach that allows the definitions of hierarchical modular models that can be easily reused [Zeigler et al. 2000]. A real system modeled with DEVS is described as a composite of submodels, each of them being behavioral (atomic) or structural (coupled). Each model is defined by a time base, inputs, states, outputs and functions to compute the next states and outputs. It has been proved that DEVS models are closed under coupling, that is, when observing the I/O trajectories of a given atomic model, we can always find a coupled model that is semantically equivalent, although having different structure.

This allows coupled models to be integrated to a model hierarchy, allowing model reuse. As a result, the security of the simulations is enhanced, testing time reduced, and productivity improved.

A DEVS atomic model is formally described by:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

Each atomic model can be seen as having an interface consisting of *input* (**X**) and *output* (**Y**) *ports* to communicate with other models. Every *state* (**S**) in the model is associated with a *time advance* (**ta**) function, which determines the duration of the state. Once the time assigned to the state is consumed, an internal transition is triggered. At that moment, the model execution results are spread through the model's output ports by activating an *output function* (l). Then, an *internal transition function* (d_{int}) is fired, producing a local state change. Input external events (those events received from other models) are collected in the input ports. An external transition function (d_{ext}) specifies how to react to those inputs, using the current state (S), the elapsed time since the last event (e) and the input value (X).

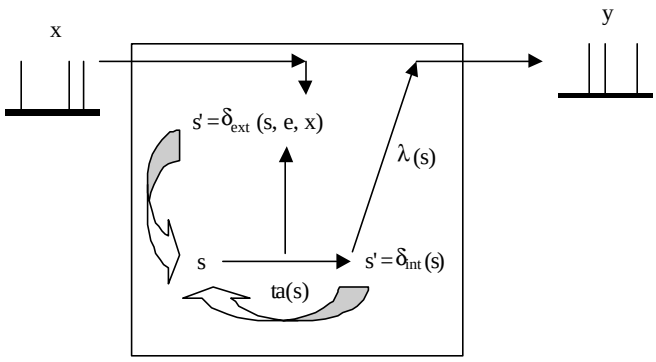


Figure 2. Informal description of an atomic model.

A DEVS coupled model is composed of several atomic or coupled submodels. They are formally defined as:

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\} >$$

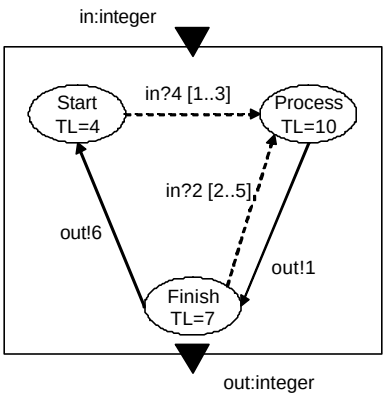
Coupled models are defined as a set of basic components (**D**), which are interconnected through the model's interfaces (***X***, ***Y***). The translation function ($\mathbf{Z_{ij}}$) is in charge of converting the outputs of a model into inputs for the others. To do so, an index of influencees is created for each model ($\mathbf{I_i}$). This index defines that the outputs of the model $\mathbf{M_i}$ are connected to inputs in the model M_j, where j is an element of I_i.

CD++ is a modeling and simulation tool that implements DEVS and Cell-DEVS theories. It allows to define models according to the specifications introduced in the previous section [Wainer 2002]. A set of independent applications related with the tool allows the user to have a complete toolkit to be applied in the development of simulation models.

The tool is built as a hierarchy of models, each of them related with a simulation entity. Atomic models can be described simply graphically or programmed in C++. CD++ provides a framework of CD++ classes with that purpose. A specification language allows to define the model's coupling, including the initial values and external events. Graph-based notations have the advantage of allowing the user to think about the problem in a more abstract way. Therefore, we have used an extended graphical notation to allow the user define atomic models behavior [Christen et al. 2004], based on the DEVS-graphs notation suggested in [Zeigler et al. 1996]. On the other hand, C++ classes increase flexibility, e.g., to define complex output functions. Figure 3 (a) represents a simple model using all the constructions available for the DEVS Graph notation in CD++.

Each state is represented by a bubble including an identifier and the duration for the state (TL). This specification allows defining the pair (state, duration) associated with internal transition functions. For instance, Figure 3 (a) shows a state called "Start", whose duration is 4 time units. Each model includes an interface with input/output ports, represented as arrowheads associated to a model definition. Internal transition functions are represented by arrows connecting two states. Each of them can be associated to pairs of ports with values (p,v) corresponding to the output function. The syntax for the output function values is **`p!v.`** For instance, the model in the figure represents a state change from *Process* to *Finish* after 10 time units. In that moment, the output function will execute, sending the value 1 through the port *out*. External transition functions are represented graphically by a dashed arrow connecting two states. The notation used to represent ports and expected values is similar to the one used for external transition, but replacing the exclamation mark by a question mark: **`p?v [`**$\mathbf{t_i}$**`..`**$\mathbf{t_f}$**`].`** Here, **`p`** and **`v`** represent the input port for the event and the value needed to trigger the associated input behavior, while $\mathbf{t_i..t_f}$ represent the initial and final expected external event times for the external transitions. These values represent the

elapsed time value, as the external transitions can produce different state changes according to the elapsed time when an external event arrives. The model in Figure 3 (a) can be formally defined as illustrated in Figure 3 (b).



Server = < X, S, Y, δ_{int}, δ_{ext}, λ, ta >

X = { in / in ∈ Z };

Y = { out / out ∈ Z };

S = { Start, Process, Finish };

δ_{int} (Process) = Finish, δ_{int} (Finish) = Start;

δ_{ext} (Start, e, in) = { i = = 4 && 1 ≤ e ≤ 3 } ?
⇒ Process,

δ_{ext} (Finish, e, in) = { i = = 2 && 2 ≤ e ≤ 5 } ?
⇒ Process

λ(Finish) = (out = 6), λ(Process) = (out = 1);

ta (Start) = 4, ta (Process) = 10, ta (Finish) = 7.

```
[Server]
in: in
out: out
state: Start Process Finish
int: Process Finish out!1
int: Finish Start out!6
ext: Start Process  in 4 [1..3]
ext: Finish Process in 2 [2..5]
Start: 4
Process: 10
Finish: 7
```

(a) **(b)** **(c)**

Figure 3. Definition of an atomic model

CD++ defines an analytical notation for DEVS graphs, which can be executed by a model's interpreter. Figure 3 (c) shows the analytical specification for the model presented earlier in Figure 3 (a): Four lines (starting in "int:" or "ext:") define the state changes according to internal and external transition functions.

- After each atomic model is defined, they can be combined into a multicomponent model. Coupled models are defined using a specification language specially defined with this purpose. The language was built following the formal definitions for DEVS coupled models.

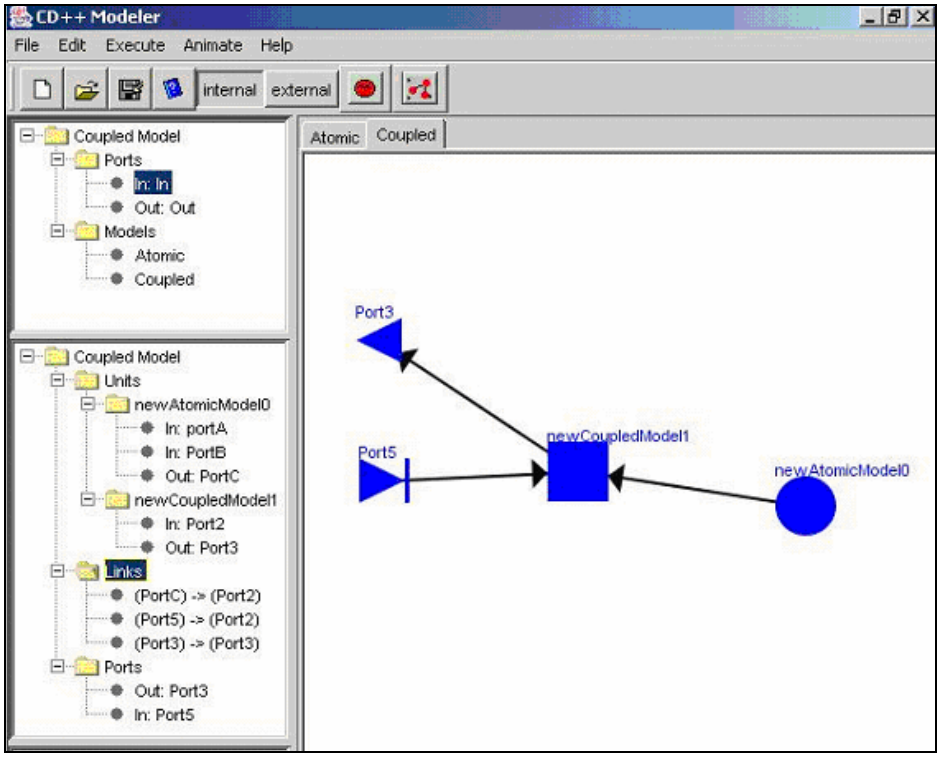


Figure 4. Analytical definition of Figure 3 (a) in CD++.

3. Software Testing Terminology

When testing a software system, the tester usually builds a *model* representing the software. The model can represent the structure of the software (e.g., the control flow graph of a procedure), in which case the testing technique is called *white-box* (or *structural*), or its behavior (e.g., a finite state machine) in which case the technique is called *black-box* (or *functional*) [Beizer 1990]. A *criterion* is then selected among a set of possible criteria to drive the construction of test cases from the model. A criterion usually indicates how much the elements of the model should be exercised (we say covered) by the test set composed of test cases. The *coverage ratio* of a test set is the proportion of the elements of the model that are covered by the test set. A tester usually tries to achieve a 100% coverage ratio, in which case the test set is said to be *adequate* for the selected criterion. A typical criterion for the control flow graph and finite state machine models are the all-statements and all-transitions criteria, respectively [Beizer 1990, Binder 1999]. The decision of choosing a specific criterion rather than another can be driven by the so called subsumes relationship between criteria [Binder 1999]: We say that criterion C1 *subsumes* criterion C2 when, for every program, any adequate test set for C1 is also adequate for C2. For instance, in the context of a state machine, the all transitions criterion (requiring that all the transitions be exercised) subsumes the all states criterion, since whenever a test set covers all the transitions, it also covers all the states. It is then possible to establish a subsumes hierarchy, that is to analytically compare criteria. A subsumes hierarchy usually indicates that some criteria are more expansive (e.g., in terms of number of test cases in an adequate test set) than others. However, it is not informative as to whether one criterion is more effective at finding faults than another, even when the former subsumes the latter.

A typical testing infrastructure entails the system under test, a *driver* in charge of executing the test cases[1] (it drives the tests), and *stubs* simulating the behavior of any software or hardware device that is not yet available (e.g., not yet developed) during the testing campaign. In the end, the tester needs a mechanism to decide whether a test case has passed or not. In the latter case, the test case has revealed a failure. This mechanism is known as the test *oracle*, and it is usually assumed that the tester is able to build one: This is known as the oracle assumption [Howden 1987]. An oracle requires (1) a mechanism to compute the expected value of a test case execution from the specifics of the test case, and (2) a mechanism to compare the actual result with the expected one [Beizer 1990, Binder 1999]. Unfortunately, building an oracle is rarely an easy task. It may even be impossible to build an oracle at reasonable costs (i.e., a cost that is lower than the cost of building the system under test): This violates the oracle assumption. When this happens, alternatives have to be found [Weyuker 1982] (e.g., partial oracle that only checks that the output is within range).

One advantage of testing, which likely explains that this has been the main software V&V technique used in practice, is that it does imply the execution of the software system under test. A major drawback, though, is that one can prove the presence of bugs but never their absence with testing techniques [Dijkstra 1972]. It is then paramount to be systematic when testing (thus the use of criteria) in order to increase our (tester, user) confidence in the system. Such a use of a model and associated criterion is indeed encouraged and required for the certification of specific software systems (e.g., avionics systems [RTCA 1992]).

4. Verification and Validation of DEVS Models: Techniques and Issues

When one wants to model and simulate a discrete event system, a possible approach is to model the system using the DEVS formalism [Zeigler et al. 2000] and simulate it using the CD++ toolkit [Wainer 2002], as described in Section 2, and illustrated by the left swimlane in the UML activity diagram [Booch et al. 1999] of Figure 5 (also referred to as Figure 5(a)). The *DEVS model* is defined and the *CD++ toolkit* is used to produce *Simulation results*. If one wants to perform V&V activities for the discrete event system, several questions have to find an answer, following the terminology introduced in Section 3 (see Figure 5(b)): What model and associated criteria to be used (Section 4.1); What are the specifics of the testing infrastructure (Section 4.2); How to address the oracle problem in our specific context (Section 4.3); What additional issues can be considered (Section 0). In this section, we intend to describe existing testing strategies that pertain to these aspects in the context of the verification and validation of DEVS models.

[1] Note that this may require that the system under test possess (or be extended by) mechanisms to control and observe it (e.g., set it state, observe its outputs) [Binder 1999]. These mechanisms are usually used by the test driver.

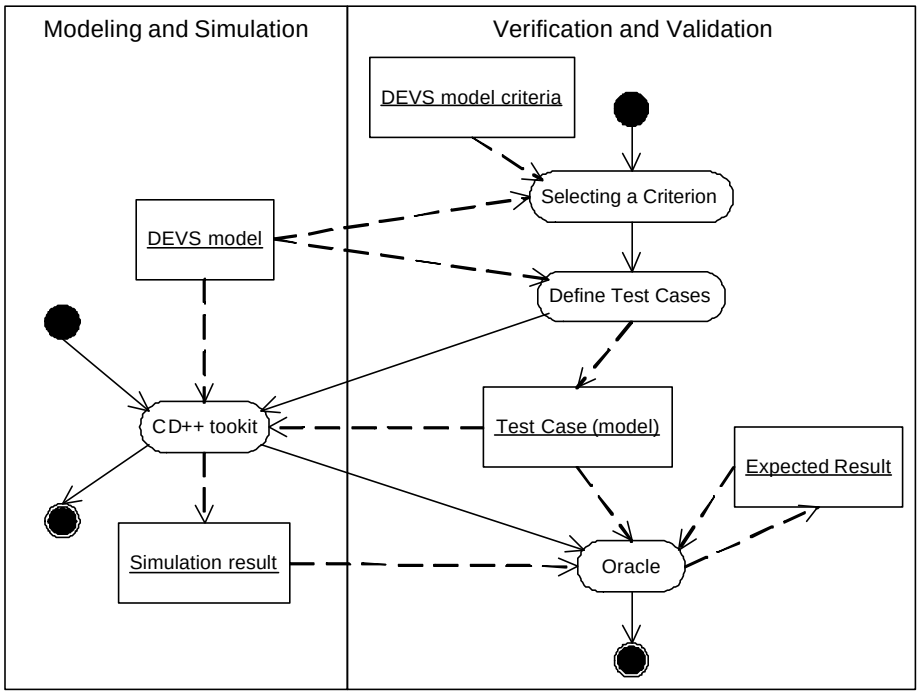


Figure 5 Modeling & Simulation (a) and Verification & Validation (b) activities

4.1. Model(s) and Associated Criteria

First, the model to be used to derive test cases is obviously the DEVS model that one is interested in validating/verifying. There likely exist a number of testing criteria that can be defined for DEVS models and it is important to decide which ones are pertinent (Section 4.1.1). Additionally, there exist two orthogonal black-box testing strategies (i.e., they can likely be combined with criteria identified in Section 4.1.1) that also seem to pertain to our context, namely testing from Boolean expressions and Category-Partition (Section 4.1.2).

4.1.1 Finite State Machines and Extensions

Since the DEVS models we want to validate and verify are based on the notation presented in Figure 3, which is essentially based on communicating finite state machines, it is paramount to investigate whether existing techniques for testing (extended) finite state machines or communicating finite state machines [Lee and Yannakakis 1996] can be used directly or should be adapted. Several directions for further investigations can be identified.

First, to what extent DEVS-like finite state machines have the appropriate characteristics to allow the use of the so called W-method [Chow 1978], that has been widely used in the telecom domain for protocol testing. The W-method requires the construction of a transition tree, for instance using a breadth-first or depth-first traversal of the finite-state machine. In the example state machine (DEVS model) in Figure 3(a), a breadth-first traversal[2] would lead to a tree with two paths: `Start-in-Process-in-Finish-in-Process` and `Start-in-Process-in-Finish-out-Start`, where the notation used to specify a tree path is a sequence of pairs `state-event`. The W-method then consists of two steps: The first one traverses the transition tree so that each path in that tree be covered by the test cases (this is the criterion), and the second one appends a state identification sequence (also known as characterization sequence) to each transition tree sequence (i.e., test case) in order to check the state that was reached. The characterization sequence is an input sequence that can distinguish between the behavior of every pair of states by simply observing the output(s) of the state machine. In other words, the output(s) produced by the finite state machine in response to the characterization sequence is unique to the states. The successful identification of the characterization sequence depends on the characteristics of the state machine (the interested reader is referred to [Chow 1978] for further details). Other similar techniques for state based testing from finite state machines exist (e.g., [Lee and Yannakakis 1996]), but they all test the same sequences (from the tree) and only differ with respect to the sequence added for the state identification problem. Note that the W-method, thanks to the characterization sequence, elegantly solves the oracle problem with respect to verifying the state reached at the end of a test case: One simply executes the characterization

[2] Using as a stopping criterion for the traversal the one suggested in [Chow 1978]: we stop traversing when a node in the tree corresponds to a state that is already present elsewhere in the tree.

sequence, and uniquely identifies the state reached at the end of the test case by simply observing the output(s); There is no need to add controllability and observability mechanisms to the system under test, which can really be considered as a black-box.

These strategies do not account for the timing aspects in DEVS models. Authors have thus extended the W-method for testing real-time systems, when the software behavior is specified as a timed input output automaton (TIOA) [En-Nouaary et al. 2002]. Such an automaton specifies behavior in terms of input and output actions that the machine receives from and sends to the environment, states (including an initial state), clocks and transitions. The TIOA formalism does have interesting similarities with DEVS formalism, although the mapping is not immediate, and some work has been done in finding equivalences between the two formalisms [Giambiasi et al. 2003]. Given that the Timed Wp-method seems to be applicable to communicating TIOA, it is thus worth investigating to what extent the Timed Wp-method can be applied (or adapted) to DEVS models.

Note that in a situation where the state-based behavior is not fully specified in the state machine (the DEVS model), it is necessary to complement any criterion explicitly based on the state-based behavior with the test of so-called sneak paths [Binder 1999]. For instance, it may happen that the response of a state to a specific event is not specified in the state machine, thus stating that the event is simply ignored (this is for instance a common practice when specifying UML statecharts). In such a situation, one would want to verify that when the system executes, is in that given state, and receives the specific event, the event is indeed ignored. If test cases are only derived from the behavior specified in the state machine (e.g., using a transition tree), this sneak path will not be exercised.

Last, in order to give insight to the DEVS designer/tester as to which criterion to use in a particular context, the identified criteria will have to be evaluated: see Section 4.4.

4.1.2 Boolean expressions and Category-Partition

Since, as pointed out in [Wainer et al. 2002] complex cellular models can be defined thanks to Boolean, relational and arithmetic operators, it seems clear that existing techniques for testing Boolean predicate expressions are relevant [Ammann et al. 2003, Weyuker 1994]. Different strategies to derive test cases from a Boolean formula exist (the interested reader is referred to [Ammann et al. 2003, Weyuker 1994] for details) and some are more cost-effective than others (i.e., less test cases and more fault detection capabilities). If a DEVS model involves complicated Boolean expressions, it will be necessary to complement the state-based criteria with such Boolean expression criteria. Similar combinations between Boolean expression testing and state-based testing criteria have been already suggested for UML statecharts [Offutt and Abdurazik 1999].

Another functional testing strategy that could be considered is Category-Partition [Ostrand and Balcer 1988]. It is the most well known extension of input domain partitioning and boundary value analysis[3] [Myers 1979]. The specification (e.g., a DEVS model) is decomposed into functional units to be tested independently (e.g., system operations or class public operations, depending on the context of application). Then, parameters and environment conditions affecting the function's behavior are identified. Categories for such parameters and conditions are defined such that they trigger a different behavior of the functionality, and are chosen to maximize the chances of finding errors. A category can be seen as a major property of the parameter or condition. Then, each category is partitioned into a series of distinct choices, i.e., partition classes, including all the possible values for the category. The set of categories and choices constitute the test specification from which it is possible to derive the test frames, that is a template to derive test cases. Each test frame is composed of a set of choices from all the categories, where each category contributes with, at most, one choice. Possible interaction among different choices belonging to different categories can be annotated in the test specification as constraints. A constraint indicates, for instance, that a choice belonging to a category cannot appear in the same test frame of another choice belonging to another category. This allows testers to reduce the number of possible test cases. Work has already investigated, with success, the combination of state-based criteria for UML statecharts and Category-Partition [Briand et al. 2004c].

Since there is evidence that traditional state-based criteria can be efficiently combined with Boolean expression testing or Category-Partition testing, it seems worth investigating the combination of state-based criteria for DEVS models (e.g., an adapted Wp-method) and those two black-box techniques. Again, an evaluation of the different strategies will be required, especially considering that in the case of combining state-based criteria with Boolean expression testing, initial empirical investigation showed the resulting combined criterion to have a low cost-effectiveness compared to other criteria (e.g., the W-method) [Briand et al. 2004a].

---

[3] A common approach to generating test cases from functional specifications is to partition the input domain of the function/method being tested into equivalent classes, and then to select an input for each class of the partition (and at the boundaries of the partitions), according to the principle that all elements belonging to an equivalence class are interchangeable for testing purposes.

4.2. Testing Infrastructure

Regardless of the selected criterion, testing data for a DEVS model will entail: (1) a sequence of inputs (possibly sent to different ports) to the model under test, as well as (2) delays between the inputs. For instance, referring to the model in Figure 3 (a), a test case could be: supposing the model is in the *Start* state, send the input value 4 at time 2 through port *in*, wait for 13 time units, and send an input with value 2 through the input port *in*. As a result, we expect the model should go from *Start* to *Process* at time 2, produce an output of the value 1 through the output port *out* and change to *Finish* at time 12, and then be back to *Process* at time 13.

It is then clear that such a test case can itself be modeled using the DEVS formalism, as illustrated in Figure 6. Similarly, a DEVS model could be in charge of collecting the outputs of the DEVS model under test, and play the role of the oracle, i.e., directly report on failures if erroneous data are received on correct data are not received in time. Such a strategy has already been experimented with [Wainer et al. 2002]. Note that it may be necessary to add output ports to the model under test (and perhaps input ports too) to observe (and control) the behavior of the model under test. This will especially required if, because of the specifics of the model, we cannot find a characterization sequence (recall Section 4.1.1).

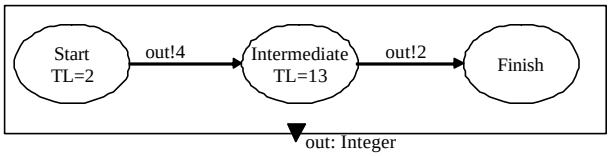


Figure 6 A test case for Figure 3(a) as an DEVS model

In terms of test scaffolding, that is test infrastructure, no stub seems to be required for the verification and validation of a DEVS model. CD++ can serve as the core of the testing infrastructure. Any other DEVS model simulator would do too. Note however that it is important that the simulator be able to execute without the user inputs (i.e., in a batch mode): we do not want the tester to stand by the computer while hundreds of test cases execute. Thanks to such environment, test cases generated according to the selected criterion can be transformed into a DEVS model that is connected to the input(s) of the model under test as discussed above.

4.3. Oracle

The oracle problem is acute in such V&V activities as in any other complex V&V activity (e.g., software), and there is no general solution: can the expected output be described (or computed) simply from the specifics of a test case or do we have to resort to the expertise of the modeler or any other expert in the problem at hand? Is it always possible to describe the correct result of the simulation as a formula or perhaps a DEVS model? The modeler (or expert) may not know the correct answer, and this might be the reason why s/he is performing a simulation. It is well known in the simulation community that interpreting simulation results is not an easy task, even for an expert. Nonetheless, there might exist simple cases for which the expected output is known, although the modeler is really interested in more complicated cases (e.g., longer runs): the model can then be tested against those simple cases and results can be extrapolated for the more complicated ones. If two different models can be built for the same problem, they can be used to test each other. (This is known as having a partial oracle). Executing a test case on each of them should produce the same result. The reader is referred to [Weyuker 1982] for a discussion on these alternative solutions to the oracle problem in the context of software testing. It is worth noting that as the complexity of the model under test increases, we can expect any interesting criterion (e.g., a cost-effective one) to produce a large number of test cases and it does not seem reasonable to ask the modeler or expert to look at them all and decide whether the outputs are correct. The tester should then strive for an oracle (even partial) as automated as possible.

An elegant solution to the oracle problem in a state-based context is the W-method, or its extensions, already introduced in Section 4.1.1. Recall that when the state machine has the right characteristics [Chow 1978] it is possible to use a characterization sequence to uniquely identify the state reached at the end of a test case. When possible, this simply eliminates the need to build an expensive oracle for the verification of the state reached. (Note that we still need an oracle to evaluate the outputs). However, as in the context of finite state machines for which the W-method can be applied, the machine may not possess those interesting characteristics. The W-method, i.e., the construction of test cases by traversing the stat machine, can still be used. It is the second part of the technique that cannot.

Another alternative to building expensive oracles has recently been suggested in the context of software testing. Contract, for instance defined according to the Design by Contract approach [Meyer 1997], can be transformed into code statements (called assertions), that are eventually inserted in the source code under test. Such assertions have been shown to be reasonable substitute for oracles [Briand et al. 2003], thus solving the oracle problem, and some work has already shown that contracts defined in OCL (in the context of UML) can be automatically instrumented in a Java implementation [Briand et al. 2004b]. Then, there is no need to build an oracle, which as we have seen before is not an easy task, and simply rely on the assertions to reveal failures. Choosing between oracles and assertions is a decision that can be driven for instance on the level

of confidence we would like to have in the simulation results. Indeed, those assertions are only reasonable substitute as experiments have shown that some faults, which are revealed by oracles are missed by assertions [Briand et al. 2003]. This seems an interesting avenue for further research though, especially since CD++ allows the designer to create C++ classes to describe complex behavior, i.e., to write code in addition to build a model, and combine the C++ classes with a DEVS model.

4.4. Other Considerations

Note that once a set of test cases have been identified when using a given criterion, such as the two examples described in Section 4.1.1 for the model in Figure 3(a), one issue still remains: How to identify real inputs (values for input ports and delays) for the DEVS model in order to exercise the set cases. For instance, what is a possible set of inputs to the DEVS model in Figure 3(a) that would trigger test case `Start-in-Process-in-Finish-in-Process`? As discussed previously, the following would do: send input 4 at time 2, wait during 12 time units, send input 2 after 1 time unit. Identifying actual input values to trigger a test case (i.e., a path in a model) is a well known issue in the software testing community, referred to as path sensitization [Beizer 1990], that often does not find an obvious answer: it is an un-decidable problem; no algorithm can solve this problem in all cases. People have thus investigated the use of heuristics techniques such as Genetic Algorithms (e.g., [Tracey et al. 1998]) .

In a standard process, when a modeler tests a DEVS model and results show failures, the model undergoes changes and the changed model should pass through a testing procedure too. One of the objectives is to verify that the changes haven't introduced new problems and this testing activity is known as regression testing. Regression testing techniques have been applied to procedural and object-oriented software as well as spread-sheets [Fisher II et al. 2002, Rothermel and Harrold 1997, Rothermel et al. 2000] and it will be interesting to see how they can be applied (or adapted) to DEVS models.

CD++ provides two main working approaches, as discussed in Section 2. The principles discussed so far apply to both approaches. In the case where the model is directly implemented in C++, additional strategies can be considered. First, the tester will be interested in structural coverage of the C++ source code [Beydeda et al. 2001, Binder 1999]. It is indeed usually admitted that functional and structural approaches are complimentary.

Another issue is that applying those testing techniques for the V&V of DEVS models assumes that the DEVS model simulator is trustworthy. How is this achieved is another challenge which seems to be similar to testing compilers [Boujarwah and Saleh 1997]. So far, given that the simulation engines are built using DEVS abstract simulation concepts [Zeigler et al. 2000], and that tool users did not report many problems related to the simulation engines[4], users have enough confidence in the simulation results. The strategy that has been adopted so far for the CD++ toolkit is to resort to a number of benchmarks models of varying complexities (which is very similar to compiler testing). Such a strategy may become an issue if DEVS models and CD++ are used to specify safety critical systems, such as an airborne system. Indeed, in such cases, not only users have to be convinced that the simulation results are trustworthy, but international certification organizations have to be convinced too. For specific such systems the tools used to develop them (in our case CD++) must be certified at the same level or criticality as the built system, and this requires a lot of effort [RTCA 1992].

Last, there is an increasingly stringent call for experimental evaluation of testing criteria. On the one hand, it may seem that if criterion C1 subsumes criterion C2, then a C1 adequate test set is guaranteed to find more faults than a C2 adequate test set for every program. This is unfortunately not true: there exist programs for which a C2 adequate test set finds more faults than a C1 adequate test set although C1 subsumes C2 [Frankl and Weiss 1993]. The analytical comparison of criteria (i.e., the subsumes hierarchy) is not sufficient and we have to resort to experimentations. For instance, it has been shown that the strategy suggested in [Chow 1978] for finite state machines and adapted by Binder to UML statecharts [Binder 1999] does not necessarily lead to interesting test sets [Briand et al. 2004a]. Similarly, combining state based criteria with Boolean testing, as suggested in [Offutt and Abdurazik 1999], has been shown to be very effective (more effective than other well-known state-based criteria), but at a very high cost [Briand et al. 2004a]. It will then be necessary, once some criteria have been identified for DEVS model, to empirically evaluate their cost and effectiveness.

5. Conclusion

Performing Verification and Validation (V&V) of discrete event system model (for instance specified in DEVS) has been identified as a paramount activity, as it can increase the confidence of the user in the simulation results and lead to the accreditation/certification of the simulated system. Because of the interesting capability of DEVS models to be executable, thanks to environments such as the CD++ toolkit, a natural approach is to consider software testing techniques to perform V&V of DEVS simulation models (testing requires the execution of the tested system).

---

[4] Since 1999, CD++ users reported only 3 errors related to the simulation engine, which were identified and fixed in a matter of hours.

As discussed in this article, DEVS models have many similarities with behavioral models that are used to specify and test software systems (i.e., finite state machines and extensions of it). It clearly follows that there are many avenues for research on applying (perhaps adapting) existing software testing techniques to the V&V of DEVS models.

In this article we tried to clarify what V&V of DEVS models, by means of testing, would entail in terms of applicable testing techniques and testing infrastructure. We identified a couple of testing techniques commonly used for software testing that seem to immediately apply to the testing of DEVS models. We also clarified some of the main issues that we will have to address in the future (e.g., the oracle problem). We showed that the DEVS formalism provides mechanism to help building a testing infrastructure. Our objective was not to be exhaustive, and we may have overlooked possible software V&V techniques (including testing) applicable to the V&V of DEVS models. Our findings in this preliminary investigation seem encouraging, although many issues remain to be investigated. Overall, we now have to do it and experiment the V&V of DEVS models by applying software testing techniques.

We have focused on testing techniques on purpose. Other software V&V approaches could be investigated too: symbolic execution, model checking, transformation of the model into other models amenable to automatic checks (such as Petri nets), etc.

Note that by trying to see how simulation can benefit from testing, we came to the conclusion that perhaps testing could also benefit from simulation. This will be investigated in the future as both authors get acquainted with each other's field.

ACKNOWLEDGEMENTS

Yvan Labiche and Gabriel Wainer are supported by NSERC operational grants.

REFERENCES

[Ammann et al. 2003] Ammann, P., Offutt, A.J. and Hong, H.S., "Coverage Criteria for Logical Expressions," *Proc. International Symposium on Software Reliability Engineering*, pp. 99-107, 2003.

[Beizer 1990] Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold, 2nd Ed., 1990.

[Beydeda et al. 2001] Beydeda, S., Gruh, V. and Stachorski, M., "A graphical class representation for integrated black- and white-box testing," *Proc. IEEE International Conference on Software Maintenance*, pp. 706-715, 2001.

[Binder 1999] Binder, R.V., *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Addison-Wesley, 1999.

[Booch et al. 1999] Booch, G., Rumbaugh, J. and Jacobson, I., *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[Boujarwah and Saleh 1997] Boujarwah, A.S. and Saleh, K., "Compiler test case generation methods: a survey and assessment," *Information and Software Technology*, vol. 39 (9), pp. 617-625, 1997.

[Briand et al. 2003] Briand, L.C., Labiche, Y. and Sun, H., "Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code," *Software - Practice and Experience*, vol. 33 (7), pp. 637-672, 2003.

[Briand et al. 2004a] Briand, L.C., Labiche, Y. and Wang, Y., "Using Simulation to Empirically Investigate Test Coverage Criteria," *Proc. IEEE/ACM International Conference on Software Engineering*, Edinburgh, pp. 86-95, May, 2004a.

[Briand et al. 2004b] Briand, L.C., Dzidek, W. and Labiche, Y., "Using Aspect-Oriented Programming to Instrument OCL Contracts in Java," Carleton University, Technical Report SCE-04-03, http://www.sce.carleton.ca/squall, 2004b.

[Briand et al. 2004c] Briand, L.C., Di Penta, M. and Labiche, Y., "Assessing and Improving State-Based Class Testing: A Series of Experiments," *IEEE Transactions of Software Engineering*, vol. 30 (11), pp. 770-793, 2004c.

[Chow 1978] Chow, T.S., "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4 (3), pp. 178-187, 1978.

[Christen et al. 2004] Christen, G., Dobniewski, A. and Wainer, G., "Modeling State-Based DEVS Models in CD++," *Proc. MGA, Advanced Simulation Technologies (ASTC)*, Arlington, VA, 2004.

[DEVSTD 2004] DEVSTD, DEVS Standardization Group, www.sce.carleton.ca/faculty/wainer/standard, [Last checked: December 2004]

[Dijkstra 1972] Dijkstra, E.W., "Notes on Structured Programming," in O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Eds., *Structured Programming*, Academic Press, 1972, pp. 1-82.

[En-Nouaary et al. 2002] En-Nouaary, A., Dssouli, R. and Khendek, F., "Timed Wp-Method: Testing Real-Time Systems," *IEEE Transactions of Software Engineering*, vol. 28 (11), pp. 1023-1038, 2002.

[Fisher II et al. 2002] Fisher II, M., Jin, D., Rothermel, G. and Burnett, M., "Test Reuse in the Spreadsheet Paradigm," *Proc. International Symposium on Software Reliability Engineering*, pp. 257-268, November 2002.

[Frankl and Weiss 1993] Frankl, P.G. and Weiss, S.N., "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," *IEEE Transactions of Software Engineering*, vol. 19 (8), pp. 774-787, 1993.

[Giambiasi et al. 2003] Giambiasi, N., Paillet, J.-L. and Châne, F., "From timed automata to DEVS models," *Proc. Winter Simulation Conference*, New Orleans, LA, 2003.

[Howden 1987] Howden, W.E., *Functional Program Testing & Analysis*, McGraw-Hill, 1987.

[Lee and Yannakakis 1996] Lee, D. and Yannakakis, M., "Principles and Methods of Testing Finite State Machines - A Survey," *Proceedings of the IEEE*, vol. 84 (8), pp. 1090-1123, 1996.

[Meyer 1997] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 2nd Ed., 1997.

[Myers 1979] Myers, G.J., *The art of software testing*, John Wiley & Sons, 1979.

[Offutt and Abdurazik 1999] Offutt, A.J. and Abdurazik, A., "Generating Tests from UML specifications," *Proc. 2nd International Conference on the Unified Modeling Language (UML'99)*, Fort Collins, CO, pp. 416-429, October, 1999.

[Ostrand and Balcer 1988] Ostrand, T.J. and Balcer, M.J., "The Category-Partition Method for Specifying and Generating Functional Test," *Communications of the ACM*, vol. 31 (6), pp. 676-686, 1988.

[Rothermel and Harrold 1997] Rothermel, G. and Harrold, M.J., "A Safe, Efficient Regression Test Selection Technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6 (2), pp. 173-210, 1997.

[Rothermel et al. 2000] Rothermel, G.,Harrold, M.J. and Debhia, J., "Regression Test Selection for C++ Software," *Journal of Software Testing, Verification, and Reliability*, vol. 10 (2), pp. 77-109, 2000.

[RTCA 1992] RTCA, "Software Considerations in Airbone Systems and Equipment Certification," Radio Technical Commission for Aeronautics (RTCA), European Organization for Civil Aviation Electronics (EUROCAE), Standard Document no. DO-178B/ED-12B, December, 1992.

[Sargent 2000] Sargent, R.G., "Verification, Validation, and Accreditation of Simulation Models," *Proc. Winter Simulation Conference*, pp. 50-59, 2000.

[Tracey et al. 1998] Tracey, N.,Clark, J.A.,Mander, K.C. and McDermid, J.A., "An Automated Framework for Structural Test-Data Generation," *Proc. IEEE Conference on Automated Software Engineering*, pp. 285-288, 1998.

[Traoré and Zeigler 2004] Traoré, M.; Zeigler, B. "Conditions for model component reuse". Materials of the Dagstuhl Seminar 04041, Component-Based Modeling and Simulation. 2004.

[Wainer and Giambiasi 2001] Wainer, G. and Giambiasi, N., "Timed Cell-DEVS: modeling and simulation of cell spaces," in H. Sarjouand and F. Cellier, Eds., *Discrete Event Modeling & Simulation: Enabling Future Technologies*, Springer, 2001.

[Wainer 2002] Wainer, G., "CD++: a toolkit to develop DEVS models," *Software - Practice and Experience*, vol. 32 (3), pp. 1261-1306, 2002.

[Wainer et al. 2002] Wainer, G.,Morihama, L. and Passuello, V., "Automatic Verification of DEVS Models," *Proc. SISO Spring Interoperability Workshop*, 2002.

[Weyuker 1994] Weyuker, E., "Automatically Generating Test Data from a Boolean Specification," *IEEE Trans. on Software Engineering*, vol. 20 (5), pp. 353-363, 1994.

[Weyuker 1982] Weyuker, E.J., "On Testing Non-testable Programs," *The Computer Journal*, vol. 25 (4), pp. 465-470, 1982.

[Zeigler et al. 1996] Zeigler, B.,Song, H. and Kim, T., "DEVS Framework for Modeling, Simulation, Analysis, and Design of Hybrid Systems," *Proc. Hybrid Systems IV*, 1996.

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