A Testing Framework for DEVS Formalism Implementations

Xiaobo Li[1,2] , Hans Vangheluwe[2,3], Yonglin Lei[1], Hongyan Song[3], Weiping Wang[1]

[1]**Dept. of Syst. Eng.**
Natl. Univ. of Defense Technology
Changsha, PRC

[2]**Dept. of Math. and Comp. Science**
University of Antwerp
2020 Antwerp, Belgium

[3]**School of Computer Science**
McGill University
H3A 2A7 Montréal, Canada

{**lixiaobo, yllei, wangwp**}**@nudt.edu.cn,** {**hv, hsong9**}**@cs.mcgill.ca**

Keywords: DEVS implementation testing framework, DEVS tools, DEVS standardization, DEVS standardized trace representation

Abstract

The Discrete-Event system Specification (DEVS) is a widely used formalism for discrete-event modelling and simulation. A variety of DEVS modelling and simulation tools have been implemented. Diverse implementations with platform-specific characteristics and often tailored to specific problem domains need to be tested to ensure their compliance with the precise and formal DEVS formalism specification. Such compliance allows for meaningful exchange and re-use of models. It also allows for the correct comparison of simulator implementation performance and hence of specific implementation optimizations. In this paper, we focus on testing "correctness" and "preciseness" of DEVS implementations and propose a testing framework. Our testing framework combines black-box and white-box testing approaches. We start with the proposal of a standard XML representation for event- and state-traces (also known as segments). We then systematically derive a suite of concrete test cases covering all possible DEVS constructs and their combinations. We apply our testing framework to PythonDEVS and DEVS++, two concrete implementations of the Classic DEVS formalism. Analysis of the test results reveals candidate items for improvement of the two tools. Finally, insights gained into DEVS standardization are discussed.

1. INTRODUCTION

The Discrete EVent System specification (DEVS) [1] is a widely accepted formalism for discrete-event modelling and simulation. DEVS has been used extensively, both in fundamental modelling and simulation research and in a plethora of application domains. Various modelling and simulation tools implement either the basic DEVS formalism or its variants. The sharing and re-use of DEVS models is hampered by the often subtle differences in the DEVS modelling and simulation tools. To address this problem, the DEVS standardization group [2] was founded to develop a DEVS standard.

The modelling and simulation process heavily relies on the supporting tools. A variety of tools implement DEVS (or were adapted to incorporate DEVS) with platform-specific and problem-relevant features. Currently there are three main types of DEVS implementations: 1. DEVS tools such as DEVSJAVA [3], DEVS++ [4], and PythonDEVS [5] were built from scratch. 2. DEVS has been embedded in existing modelling and simulation languages and environments such as Modelica/DEVS [6], Matlab(Simulink)/DEVS [7], and DEVS/UML [8]. 3. DEVS is combined with other simulation model specifications and protocols with a focus on simulation interoperability, model reuse and composability as is the case with DEVS/HLA [9], DEVS/SOA [10], and DEVS/SMP2 [11].

These tools need to be tested to ensure their compliance with the DEVS formalism specification. Firstly, different implementations are often based on a developer's interpretation of DEVS. Secondly, DEVS implementations based on different platforms/tools or different protocols/specifications may introduce implementation-specific artefacts. Testing is needed to find and correct these. Thirdly, even if ad hoc additions do not conflict with the DEVS formalism, they also need to be tested to ensure that meaningful DEVS tool interoperation and model reuse are not affected. Apart from different implementation methods for DEVS, there are many DEVS variants such as classic DEVS, parallel DEVS, cell-DEVS etc. In this paper, we focus on the classic DEVS formalism and propose a framework to guide compliance testing. The framework is applied to two classic DEVS simulation tools.

The rest of the paper is organized as follows. Section 2 presents an overview of related work. Fundamentals of our testing framework are discussed in section 3. We test classic DEVS implementations in the tools PythonDEVS and DEVS++ in section 4. Conclusions are given in Section 5.

2. RELATED WORK

In [12], a benchmarking technique called DEVStone is used to generate DEVS models of different size and structure to study the efficiency of five simulation engines provided by a DEVS M&S tool–CD++. The main purpose of this paper is to test the simulator performance of tool, not the compliance of DEVS formalism implementations. In [13],

the authors analyze the feasibility and advantages of using software testing techniques for DEVS model V&V, and clarify some fundamental issues of a testing-based model V&V approach. "Trustworthiness of a DEVS simulator" is also mentioned in this paper as one necessary condition for applying the approach. The authors also state that given that the simulation engines are built using DEVS abstract simulation concepts in [1], only three errors have been reported. In work, we propose a framework to test the "trustworthiness of a DEVS simulator" as well as model specification. In [14], the authors present a theoretical foundation and a test development framework for service testing at the I/O behaviour level in a service-oriented computing environment. The system theoretic foundation for I/O behaviour testing is based on hierarchical system specifications of DEVS systems theory [1] and the model implementations are DEVS-compliant. This approach has inspired our behaviour testing. The DEVS Standardization group [2] is an international team trying to develop standards for a computer processable representation of the DEVS formalism. There are three Task Groups, on DEVS tools interoperation, DEVS Kernel definition, and DEVS Atomic models specification language. Our research is closely related to the mission of the DEVS tool interoperation Task Group.

3. FRAMEWORK FUNDAMENTALS

3.1. What to Test?

The first and foremost issue of testing is to specify *what to test*. The test object in our work is the implementation of the DEVS modelling specifications and their abstract simulator algorithms from [1].

3.2. Test Requirements

There are two kinds of test requirements: structural compliance and behavioural preciseness. The former forms the foundation for the latter.

3.2.1. Structural Compliance

Structural compliance refers to the implementation of three aspects of the DEVS formalism: model specification (abstract syntax), constraints (static semantics), and simulation algorithm (operational semantics). A DEVS implemenation (possibly source code) with platform is analyzed to check whether all elements are implemented . This is a static check without strict proofs. Further steps (such as building reference models) could be taken to check whether the tool works properly. Investigating the tutorial examples with enough behavioural aspects can produce insights, especially when they are elaborate. Another effective way is to use the debugging mode of the platform to trace the simulation process.

Structural compliance is the foundation for behavioural preciseness. According to the principle of universality and uniqueness of DEVS, "Every DEVS-like system is a homomorphic image of a DEVS I/O system" [1]. So when testing DEVS formalism implementation, the structural compliance should be checked first to make sure that the behaviour is produced by a DEVS-compliant systems, not by a DEVS-like systems. We need to bear in mind that different structures may produce almost the same behaviours. For example, a simulation model and its response surface model (also called surrogate model).

3.2.2. Behavioural Preciseness

Dynamic behavioural preciseness needs to be tested, building on static structural compliance. The essence of simulation is to produce the temporal behaviours of a System under Study correctly and efficiently. Whether we have built the correct model depends on whether the model could produce exactly the correct state and I/O behaviour. We use the term "preciseness" instead of "correctness" because different implementation of DEVS introduces intrinsic platform-specific deviations even when the DEVS implementation is 100% correct. Pragmatically, if the deviation is smaller than a set tolerance, the tool will be said to "pass" the test.

Behavioural preciseness should be tested using *deterministic* models, in which no random factors are considered. Only in this way can strict statements about behavioural preciseness be made. Although stochastic models are widely used in DEVS applications, the difference among random number generators of different platforms and the difficulty of producing a test oracle make it impractical to test the behavioural preciseness based on stochastic models. Most importantly however, the DEVS formalism is deterministic and stochasticity is only introducted through sampling from distributions using pseudo-random number generators (deterministic in their own right).

3.3. Test Case Derivation

We adopt a white-box approach for test case derivation from the requirements. In this approach, the knowledge of DEVS formalism and the implementation details of the tool are used to derive a systematic suite of abstract cases to test each aspect of the DEVS formalism implementation. The abstract test cases should be sufficient and exhaustive to test all requiriments. If there are some ad hoc characteristics of the implementation which may influence the compliance with the DEVS formalism, this information should be used to derive corresponding test cases.

3.4. Test Case Concretization

The test cases derived from the requirements are abstract and need to be concretized. A black-box approach is adopted in our research to specify I/O behaviours of the specified

DEVS models for test case concretization. We need to construct corresponding abstract DEVS models and implement them in the testing tool to test the requirements of the abstract test case. This approach can provide a test oracle which is based on the comparison between I/O behaviours produced by simulating the executable models using the tool and expected behaviours specified by deduction from the abstract DEVS models.

The DEVS models to concretize the abstract test cases play a fundamental role in testing the behavioural preciseness. They should satisfy the following requirements to support the black-box approach. Firstly, they should be as simple as possible to avoid faults introduced by model implementation in the tool and accidental complexity caused by complex model structure or algorithm. Secondly, they should be representative and sufficient for the abstract test case, namely, the test requirements can be tested by specifying a set of I/O behaviour of the models.

3.5. Test Oracle

To test the behavioural preciseness of a DEVS formalism implementation, a *test oracle* checks whether the model produces the same I/O behaviour as expected. One method is to compare the output trace produced by the model with the expected trace derived from reasoning about the behaviour of DEVS models. Sometimes robustness testing is needed to check how the system under test reacts to wrong inputs. In this case, trace comparison is not sufficient, and the test must check whether the tool produces appropriate exceptions.

For the test cases which should produce precise behaviours, we adopt a "standardized trace representation" approach proposed by Song [15], which is based on XML to record all event trace of the simulated model. It offers a standardized way to record DEVS simulation output and a DTD to validate the XML trace files. The trace is composed of a set of events (and states) produced by the simulated DEVS models. In each event, the following elements are recorded: the fully qualified model name in which the event occurs, the time of the event, the event kind and port, the state information of the model when the event occurs.

There are several benefits to this approach. Firstly, it offers an event-based way to completely record the behaviours. Secondly, the XML format of the trace file provides a standardized trace, which can be used to compare traces produced by different DEVS models implemented on different platforms. Thirdly, it is feasible and easy to implement. The test oracle can be implemented by a simple "compare" of the produced trace file and the "expected" trace file.

Song [15] implements an output trace generation method in PythonDEVS, which is used in a case study on PythonDEVS. However, if the same method is implemented in other tools and traces are produced by different tools, we should pay attention to the computation precisions of different platforms and set a reasonable tolerance to eliminate their influence.

3.6. A Behavioural Testing Framework

Based on the discussion above, we propose a behavioural testing framework as shown in Figure 1. The framework consists of the following steps: 1. Specify testing requirements from the DEVS formalism. 2. Extract relevant information of the DEVS implementation from the candidate tool to be tested (test the stuctural compliance). 3. Derive a systematic suite of test cases from the test requirements and implementation information. 4. Construct abstract DEVS models to concretize the test cases. 5.Build Executable DEVS models to implement abstract DEVS models using the tool. 6. Import the inputs and oracles (I/O behaviour) into the executable model from the test cases. 7. Simulate the DEVS models using the tool and collect simulation output traces. 8. Compare output traces with test oracles and produce test results.

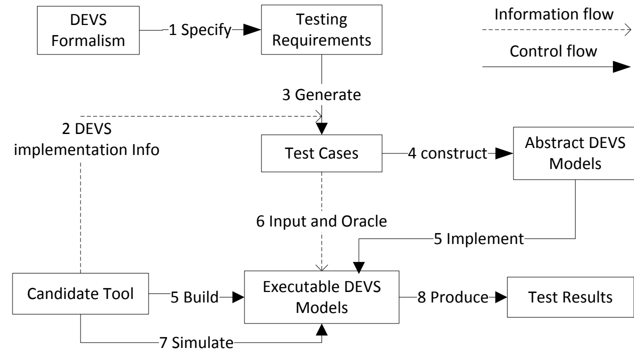

Figure 1. DEVS Implementation Testing Framework

4. CASE STUDY

In this paper we choose implementations of the classic DEVS variant as the test object for the following reasons. Firstly, classic DEVS is the core of the family of DEVS formalisms. Research results of classic DEVS are a useful foundation for other DEVS formalisms. Secondly, its conciseness makes it easy to test the implementation.

4.1. The Classic DEVS Formalism

4.1.1. Classic DEVS Introduction

The atomic DEVS formalism is a structure describing the different aspects of the discrete-event behaviour of a system [1]. Ports and Portvalues were introduced in [1] to make modelling easier and should also be considered in testing. The Coupled DEVS and Abstract Simulator algorithms we will test are as described in [1].

4.1.2. Test Requirements

We specify a set of requirements from three aspects of the classic DEVS formalism: model specification (abstract syntax), constraints (static semantics) and simulator algorithm (operational semantics). For structural compliance, we should check whether all the elements of the model specification and abstract simulator algorithm are appropriately implemented. This is done statically and no model executions are required. The implementation details are collected and used for test case derivation. For behavioural preciseness, we identify the following nine testing requirements. 1. **Non-negativeness of the time advance function(*ta*)**: *ta* should return a non-negative value. 2. **Correct output function**: *the output function* should occur before and only before the internal transition. 3. **Event passing instantaneity**: Event passing between ports should cost no simulation time. 4. **Precise time granularity**: the simulation should be run at a given time granularity. 5. **Event time synchronization**: the time of occurrence of one event should be the same in all relevant (connected) models. 6. **Correct event sequence**: all events should be processed in to the right temporal sequence. 7. **Correct tie-breaking**: simultaneous internal event occurrences should be handled correctly using the select function. 8. **Correct event dispatching**: events should be correctly dispatched from output port(s) to input port(s). 9. **Event independence**: All event instances should be unique (no references to each other nor to model states). Requirements 1-3 pertain to static semantics and 4-9 are simulator considerations. These requirements are not completely independent as some are based on other, more basic ones. Model executions are needed to test the above requirements.

4.1.3. Abstract DEVS Model for Test Cases

In our framework, DEVS models are needed to concretize the abstract test cases. In Figure 2 we build a very simple DEVS system called "pipeline EventGPCSystem" (also called model 1), which contains three parts: an event generator ("Generator"), an event processor ("CoupledProcessor") and an event collector ("Collector"). Adapted from the generator, processor and pipeline examples in [1], it is illustrative, understandable and easy to implement.

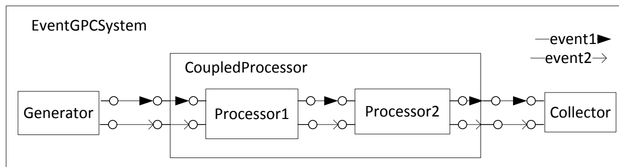

Figure 2. Model 1: Pipeline "EventGPCSystem"

The event generator is an atomic DEVS model and generates two events, "event1" and "event2", respectively before an internal transition from "gen_event1" to "gen_event2" and another internal transition from "gen_event2" to "gen_over", and outputs two events via port "send_event1_gen" and "send_event2_gen". Its initial state is "gen_event1". The state duration time for the states "gen_event1", "gen_event_2" and "gen_over", are "t_gen_event1", "t_gen_event2" and $+\infty$. "t_gen_event1" and "t_gen_event2" are two initialization parameters.

The event processor is a coupled DEVS model with two input ports ("rec_event1_P" and "rev_event2_P") and two output ports ("send_event1_P" and "send_event2_P"), via which "CoupledProcessor" receives the two events from the generator and sends them to the collector. There are two atomic DEVS models in "CoupledProcessor": "Processor1" and "Processor2", which have the same structure. The atomic processor model has the same I/O ports as the "CoupledProcessor". The atomic processor's initial state is "idle", and when it receives an event, an external transition is triggered to set the state to a processing status ("proc_event1" when receiving "event1" and "proc_event2" for "event2"). The processing time for "event1" and "event2" are correspondingly "t_event1_P1" and "t_event2_P1", which are two initialization parameters. After the processing time, the processed event is output via the corresponding output port and the state becomes "idle" again. A "pipeline" style is adopted for the "Processor" in model 1. The events are processed first by "Processor1", then by "Processor2", and are finally collected by "Collector".

The event collector has two input ports, "rec_event1_col" and "rec_event2_col", to collect the two events respectively. Its initial state is "idle". When an event arrives, the state transits to "event1_received" in the case of only having collected "event1", "event2_received" (only "event2") and "event1and2_received" (both events are collected). For simplicity, the two events have very simple data structure containing one float parameter ("eventSize"). By changing the generation time and processing time in "Generator", "Processor1" and "Processor2", we can derive different behaviours of "EventGPCSystem". So we use the six initialization parameters ("t_gen_event1", "t_gen_event2", "t_event1_P1", "t_event2_P1", "t_event1_P2" and "t_event2_P2") as the input of the abstract model and show how we adjust them to build different test cases in section 4.2.1.. Our base abstract model needs to be adapted for some test cases, but only the port coupling relationships are changed and the atomic models remain the same (model 4 in Figure 4 and model 5 in Figure 5). In the two coupled models, namely "EventGPCSystem" and "CoupledProcessor", we set the select function to choose the first one in the event list.

4.2. Case Study: PythonDEVS

4.2.1. Testing Procedures

PythonDEVS [5] is a classic DEVS M&S tool built by the MSDL in 2002. We test the classic DEVS implementation in PythonDEVS under the framework proposed in section 3.6..

Step 1: specify the test requirements based on the classic DEVS formalism. This has been done in section 3.2..

Step 2: check the DEVS implementation in PythonDEVS and extract relevant information. PythonDEVS is an implementation of the classic DEVS formalism. All the elements in the classic DEVS formalism are implemented except the PortValue. We can find in the model examples(tutorials) that the PortValue is implemented as a class. The tool sets a value of 1E-10 as the tolerance for time synchronization. These are useful tips for the test derivation and model building.

Step 3 and 4: derive test cases and concretize them based on the abstract models. These two steps are closely related(iterative) and we put them together. For most of the requirements we can build a test case (or a series of cases) based on the base abstract model(“Pipeline processor”, we call it model 1). For some requirements we need to revise the select function (requirement 7) or change the coupling relationships (requirements no. 8 and 9).

As we can see from Table 1 we list the test cases derived from each requirement and concretize them on an appropriate DEVS model. The number “RxTy” means it is the y test case for requirement x. The “Model” number specifies which DEVS model the test case concretizes. “Input” sets the input parameters for each test case. We give the results for both PythonDEVS (T1 in the table) and DEVS++ (T2 in the table), which we will discuss in the following sections. For **requirement 1**, though *ta* is implemented by users, we still demand that the M&S tool deals with the situation when users make a mistake and calculates a negative value. So we assign a negative value to *ta* and expext the tool to throw an exception saying that the value of *ta* is negative. This case is concretized to *R1T1* based on model 1. In this case, a “-0.5” is assigned to “t_event1_P1”. Considering that the tolerance for time synchronization is 1E-10, we design an extreme case *R1T2* which sets “t_event1_P1” to -1E-11.

No.	Model	Input	T1	T2
R1T1	1	(1.0, +∞, -0.5, +∞, 1.0, +∞)	FAIL	PASS
R1T2	1	(0.0, 2.0, -1E-11, 2.0, 0.0, 4.0)	FAIL	PASS
R2T1	1	(1.0,+∞, +∞, +∞, +∞, +∞)	PASS	N/A
R3T1	1	(0.0, +∞, 0.0, +∞, 0.0, +∞)	PASS	PASS
R4T1	1	(1.0, 2E-10, 1E-10, 2.0, 1.0, 4.0)	PASS	PASS
R4T2	1	(1.0, 2E-11, 1E-11, 2.0, 1.0, 4.0)	FAIL	PASS
R5T1	1	(1.0, 2.0, 1.0, 2.0, 1.0, 2.0)	PASS	PASS
R6T1	1	(1.0, 4.0, 1.0, 4.0, 1.0, 4.0)	PASS	PASS
R7T1	1	(1.0, 1.0, 1.0, 2.0, 1.0, 4.0)	PASS	N/A
R7T2	2	(1.0, 1.0, 1.0, 2.0, 1.0, 4.0)	PASS	N/A
R7T3	1	(1.0, 2.0, 1.0, 2.0, 3.0, 4.0)	PASS	N/A
R7T4	3	(1.0, 2.0, 1.0, 2.0, 3.0, 4.0)	PASS	N/A
R8T1	4	(1.0, +∞, 1.0, +∞, 2.0, +∞)	PASS	PASS
R8T2	5	(1.0, +∞, 1.0, +∞, 2.0, +∞)	PASS	PASS
R8T3	6	(1.0, +∞, 1.0, +∞, 2.0, +∞)	PASS	PASS
R9T1	4	(1.0, +∞, 5.0, +∞, 2.0, +∞)	FAIL	FAIL
R9T2	5	(1.0, +∞, 5.0, +∞, 2.0, +∞)	PASS	FAIL
R9T3	6	(1.0, +∞, 5.0, +∞, 2.0, +∞)	PASS	FAIL

Table 1. Test Cases and Results

For **requirement 2**, constraints of the output function we design a simple case *R2T1* also based on model 1. The test oracle is that “Generator” should produce an output first and then only make a transition. Both should occur at the same simulation time.

For **requirement 3** we build case *R3T1* which sets the generating and processing times of “event1” to 0. We then check whether “collector” can collect “event1” at time 0. From the input and model 1 we can derive that only “event1” is generated, processed and collected. All the events are at time 0.0. So we build a event trace file conforming to the “standardized trace representation” shown in Figure 3 and use it as the test oracle.

```
- <trace>
  + <event>
  - <event>
      <model>EventGPCSystem.Collector</model>
      <time>0.0</time>
      <kind>EX</kind>
    - <port name="rec_event1" category="I">
        <message>(Event1 3, eventSize 0.000000)</message>
      </port>
    - <port name="rec_event2" category="I">
        <message>None</message>
      </port>
    - <state>
      - <attribute category="P">
          <name>event</name>
          <type>string</type>
          <value>event1_collected</value>
        </attribute>
        <![CDATA[            event = event1_collected  ]]>
      </state>
    </event>
  </trace>
```

Figure 3. Standardized event trace file

From **requirement 4** we derive 2 cases (based on model 1) that two internal transitions in two atomic models happen closely enough to each other and test whether the simulator can differentiate between them. We set the time interval equal to the time synchronization tolerance in *R4T1* and smaller than the time synchronization tolerance in *R4T2*. The test oracle is that the tool should differentiate the temporal sequence of these two events and not treat them as collisions.

Requirement 5 is very fundamental since it guarantees that coupled DEVS models maintain the event lists correctly and triggers events at the correct time in their imminent children. We build a test case *R5T1* where in each DEVS model

there are events to test whether time synchronization exceptions are thrown and the simulation produces the right trace file.

Requirement 6 is a basic requirement and lays the foundation for requirements 7-9. Based on model 1, we set the parameters in case *R6T1* to generate "event2" after collecting "event1". The test oracle is that the produced trace should have the same event sequence as expected.

Requirement 7 on events collision is a tough but important issue in DEVS. In the classic DEVS formalism if an internal transition and an external transition of the same atomic model occur at the same time, the external transition overrides the internal transition unconditionally. Additionally, in classic DEVS, there is a "select" function in coupled DEVS models to determine the sequence of events in case of collision between internal events. We need to test the usage of the "select" function to check whether collisions are handled correctly. We build the following 4 cases for requirement 7. Case *R7T1* and *R7T2* set the input (1.0, 1.0, 1.0, 2.0, 1.0, 4.0) to cause an external event on input port "rec_event2_P1" and an internal transition from "proc_event1" to "idle" at time 2.0 in "Processor1". *R7T1* is based on model 1, in which the "select" function of DEVS coupled model "EventGPCSystem" chooses the first event in the set of imminent models –implemented as an ordered list in most tools– ("Generator" generates "event2" which causes an external event on input port "rec_event2_P1"). *R7T2* is based on model 2 and chooses the second event in the list. Case *R7T3* and *R7T4* use the input (1.0,2.0,1.0,2.0,3.0,4.0) to test the event collisions inside the coupled model "CoupledProcessor" and its "select" function. As we can deduce from the abstract model, at time unit 5.0, "Processor1" sends "event2" to "Processor2" while "Processor2" has an internal transition from "proc_event1_P2" to "idle". Case *R7T3* is based on model 1 which selects "Processor1" and case *R7T4* is based on model 3 in which the "select" in "CoupledProcessor" chooses "Processor2". The test oracle is that in these test cases, external transitions override the internal transitions (which means the internal transition doesn't occur) when it is chosen to occur first and events which don't influence each other occur in the sequence as chosen.

For **requirement 8** we mainly test the event dispatching from one output port to two input ports. Model 1 needed to be revised to get this one-to-two port coupling relationship. As shown in Figure 4, model 4 adopts a parallel style which processes events in parallel to have an one-to-two External Input Coupling(EIC). Then model 5 (in Figure 5) and model 6 (in Figure 6) are built to test event dispatching in Internal Coupling (IC) and External Output Coupling. Test case *R8T1*, *R8T2* and *R8T3* are based on model 4, 5 and 6 respectively. The test oracle is that generated events should be dispatched to the two processors at the same time and with identical event attribute.

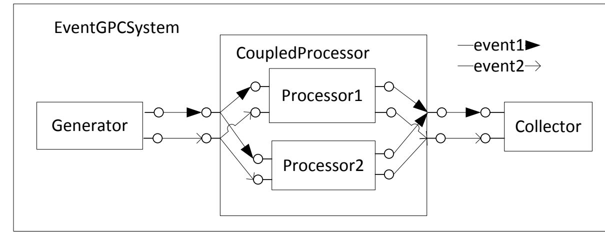

Figure 4. Model 4: Parallel "EventGPCSystem"

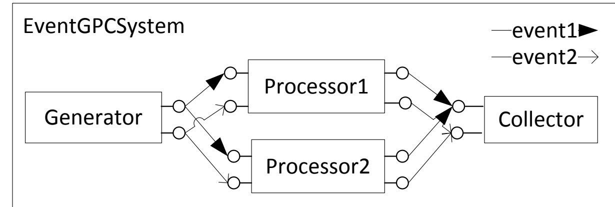

Figure 5. Model 5: Parallel "EventGPCSystem" without CoupledProcessor

For **Requirement 9** we need to test if we revise event attributes of two instances of the same event, whether they influence each other or not. We build three cases *R9T1* based on model 4, *R9T2* based on model 5 and *R9T3* based on model 6. We don't change the event attribute ("eventSize") in "Processor1" but revise it in "Processor2", and set the processing time of "Processor2" smaller than that in "Processor1". So the test oracle is that "eventSize" should remain unchanged when "Processor1" sends "event1" to "Collector". If "eventSize" is the same value as revised in "Processor2", then they are not independent.

Step 5: Executable DEVS models implementing abstract DEVS models are built in the tool. It is simple to implement models 1-6 in PythonDEVS. We implement the events as Python classes, which we learn from the examples provided by the tool.

Step 6: Import the I/O behaviour into the executable model from the test cases. We set the input parameters (in Table 1) of each case to configure corresponding the DEVS model. For each test case a trace file including the "correct" output trace is built confroming to the standardized trace representation format.

Step 7: Simulate the DEVS models using PythonDEVS and collect outputs. The simulation will produce a simulation trace file if no exceptions are thrown. If the simulation terminates abnormally, PyUnit catches this.

Step 8: Compare simulation output with the test oracle and produce test results. All the results are listed in Table 1.

We use the **PyUnit** [16] Python testing framework in step 6-8 to partly automate the testing process for our test cases.

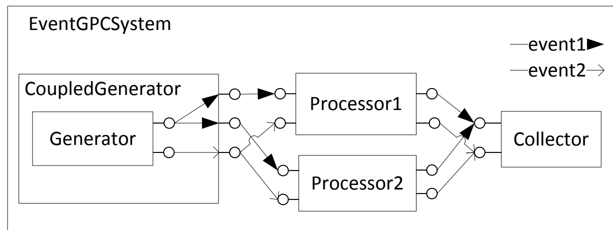

Figure 6. Model 6: Parallel "EventGPCSystem" with CoupledGenerator

After building the executable DEVS models, we build a test file for our test cases using PyUnit. We build a test unit for each test case and use the input parameters to instantiate a simulatable model. Then we simulate the model in each test unit and get a simulation trace file. We compare the trace file with the expected trace by means of Python's `filecmp.cmp`. We execute the (unit) test cases one by one and test result for each case is produced based on the value return by `filecmp.cmp`. We can do all the test cases at one time, which is very conveniently especially when there are a large number of test cases.

4.2.2. Test Results and Analysis

The test results are listed in Table 1. We discuss why PythonDEVS fails certain tests and propose some ways in which to improve conformance. In case *R1T1*, PythonDEVS throws an exception of "bad synchronization in processor2 on external transition", which is not the expected exception. In case *R1T2* we notice that no exceptions are thrown. So it is obvious that PythonDEVS does not have a proper mechanism to deal with negative ta values. Negative values can easily occur due to a user mistake. To fix this problem we only need to add an assertion for the value of *ta* which throws an exception when it negative. In case *R4T2*, PythonDEVS treats the two events as collision when the occurring time interval is smaller than the time event synchronization value. The fact that PythonDEVS passes case *R4T1* tells us the time granularity should be no larger than 1E-10. Upon closer examination of the PythonDEVS simulator we notice that the tool sets the time synchronization tolerance 1E-10 the same as the time granularity, which could be confusing to users. An improvement would be to use a distinct value for time granularity. In case *R9T1*, "event1" instance in "Processor1" is influenced by "event1" instance in "Processor2" and its attribute is the same as the attribute revised by "Processor2". PythonDEVS passes case *R9T2* and *R9T3* for IC and EOC. We dive into the PythonDEVS simulator and find that it uses a "deep copy" of event objects when dispatching events from one output port directly to multiple input ports (namely Internal Coupling) to ensure independence among these events. But for EIC in case *R9T1*, it dispatches duplicated events which have references to each other. So our suggestion to fix this problem is to also use "deep copy" for EICs.

4.3. Case Study: DEVS++

DEVS++ [4] is a DEVS tool based on C++. We test the classic DEVS implementation in DEVS++ by means of our framework. For space reasons we do not describe the testing procedures in detail, as it is almost the same as that described in 4.2.1.. In Table 1 we list the test results and discuss the failed and non-applicable cases. Testing requirement 2 is not applicable as the DEVS++ manual [4] specifies that the output function resides inside the internal transition function. So whether the output function is evaluated before the internal transition depends on the tool user's implementation. Testing requirement 7 is not applicable as there is no "select" function implemented in DEVS++. In case of collision, DEVS++ always chooses the first model in imminent list. To fully conform to the classic DEVS formalism, DEVS++ needs to provide a select function for tie-breaking. DEVS++ fails test *R9T1*, *R9T2* and *R9T3* because it uses the same pointer to deliver multiple event instances. This makes events dependent of each other. To avoid this problem different pointers can be assigned for different event instances to ensure mutual independence.

4.4. Discussion

One might argue that certain requirements can be satisfied in non-compliant tools by careful modelling by experienced users. For example, users can ensure requirement 2 is satisfied by putting the output function before the internal transition in the implementation of the internal transition function in DEVS++ or ensure event instance independence by performing a "deep copy" in their DEVS model implementations in PythonDEVS. A tool-builder should make maximum efforts to minimize the possibility for users to make mistakes. The flaws detected in the two tools originate partly from negligence (e.g., *R1T1* in PythonDEVS), partly from the platform-specific characteristics (e.g., *R9T1* and *R9T2* in DEVS++), and partly from variations of the basic DEVS formalism (e.g., no "select" function in DEVS++). All the flaws are hard to detect but easy to fix (in hours or days). DEVS implementation testing is a complicated task for an individual and needs collaboration of tool builders, users and M&S experts.

DEVS implementation testing and standardization. DEVS standardization will certainly ease the DEVS implementation testing. Firstly, if a standard model library is built and the I/O behaviours are specified, we can easily use these models as the testing model, and no errors would be introduced by model building. Secondly, if a core DEVS formalism with minimum elements is established, we can focus on testing the core formalism and the work would be consid-

erably reduced. Testing of DEVS M&S tool requires us to think about the essence of the DEVS formalism and its implementation. By testing different tools and comparing the implementation details we obtain insights for standardization (like what differences are essential and need to be standardized and how to standardize), which will help us build the standard library, the core DEVS formalism and the interoperation mechanism between DEVS tools.

5. CONCLUSION

In this paper, we proposed a framework to test behavioural correctness of DEVS tools w.r.t. the DEVS formalism specification. In this framework, a series of abstract DEVS models are constructed to derive a systematic suite of concrete test cases. A "standard trace representation" is introduced to allow test oracle specification. We test the classic DEVS implementation in the PythonDEVS and DEVS++ tools to illustrate our methodology. Some flaws of these two tools were detected by testing. Based on analysis of the testing results we provide some advice to improve these two tools. From the case study we notice that one formalism can have very diverse implementations. So the testing-based approach seems a necessary and important enabler for tool interoperation and DEVS standardization.

Though extensive, we cannot claim completeness of our proposed set of test requirements for the classic DEVS formalism. One item of future work is to find and test more requirements emerging from DEVS standardization considerations and from specific application domain needs. In this paper we have tested the classic DEVS implementation category 1 (namely building DEVS tools from scratch) mentioned in Section 1.. Another item of future work is to test classic DEVS implementations in category 2 (e.g., Modelica/DEVS) and 3 (e.g., DEVS/SMP2). We will also extend and apply our framework to other DEVS variants such as parallel DEVS and cell-DEVS.

ACKNOWLEDGMENTS

This work is partly supported by the National Natural Science Foundation of China (No.60974073), the China Scholarship Council and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, 2nd Edition*. Academic Press, New York, 2000.

[2] DEVSSG. DEVS standardization group. `http://cell-devs.sce.carleton.ca/devsgroup/`.

[3] B.P. Zeigler and H.S. Sarjoughian. Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-based Simulation Models. Technical report, 2003.

[4] Moon Ho Hwang. *DEVS++*, v.1.4.2 edition, April 2009. `http://odevspp.sourceforge.net/`.

[5] Jean Sébastien Bolduc and Hans Vangheluwe. The modelling and simulation package PythonDEVS for classical hierarchical DEVS. MSDL technical report MSDL-TR-2001-01, McGill University, June 2001.

[6] Victorino Sanz, Alfonso Urquia, François E Cellier, and Sebastian Dormido. System modeling using the Parallel DEVS formalism and the Modelica language. *Simulation Modelling Practice and Theory*, 18(7):998–1018, 2010.

[7] Kyung Min Seo, Change Ho Sung, and Tag G. Kim. Realization of the DEVS Formalism in MATALB/Simulink. June 2008.

[8] Joseph Mooney. DEVS/UML - A Framework for Simulatable UML Models. Master's thesis, Arizona State University, May 2008.

[9] B. P. Zeigler, G. Ball, H. Cho, J. Lee, and H. Sarjoughian. Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions. In *1999 Spring Simulation Interoperability Workshop*, Orlando, FL, March 1999.

[10] S. Mittal, J. L. Risco-Martin, and B. P. Zeigler. DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process. *Simulation*, 85(7):419–450, 2009.

[11] Yonglin Lei, Weiping Wang, Qun Li, and Yifan Zhu. A transformation model from DEVS to SMP2 based on MDA. *Simulation Modelling Practice and Theory*, 17(10):1690–1709, 2009.

[12] E. Glinsky and G. Wainer. DEVStone: a Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environments. *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 265–272, 2005.

[13] Yvan Labiche and Gabriel Wainer. Towards the Verification and Validation of DEVS Models. *Proceedings of the Open International Conference on Modeling & Simulation*, 2005.

[14] Xiaolin Hu, Bernard P Zeigler, Moon Ho Hwang, and Eddie Mak. DEVS Systems-Theory Framework for Reusable Testing of I/O Behaviors in Service Oriented Architectures. In *Proceeding of the IEEE International Conference on Information Reuse and Integration*, 2007.

[15] Hongyan Song. Infrastructure for DEVS Modelling and Experimentation. Master's thesis, McGill University, 2006.

[16] S. Purcell. *PyUnit Testing Framework*, v.1.4.1 edition, 2001. `http://pyunit.sourceforge.net`.