UNIT TESTING PLATFORM TO VERIFY DEVS MODELS

Kevin Henares, José L. Risco-Martín, José L. Ayala and Román Hermida
Dept. of Computer Architecture and Automation
Complutense University of Madrid
Calle Prof. José García Santesmases, 9
28040 Madrid, Spain
{khenares, jlrisco, jayala, rhermida}@ucm.es

ABSTRACT

The natural progression from classic Model-Based Systems Engineering (MBSE) methodologies to Modeling and Simulation-Based Systems Engineering (M&SBSE) brings the need for more flexible and powerful validation tools. Completing the structure and descriptions of static models with self-diagnosis greatly facilitates the development, implementation, and validation of complex models. However, current simulation development environments and libraries often lack providing complete tools to validate these complex models straightforwardly and with the proper level of detail. In this article, we tackle this issue transferring one of the traditional software testing methods, unit testing, to the modeling and simulation field. To this end, we integrate a unit-testing method to the DEVS methodology, allowing the specification of expected states, outcomes, and behaviors of the simulation modules in an XML-based syntax, in all the levels of the hierarchical design. As a result, this methodology enables the generation of powerful and easy-readable verification files.

Keywords: DEVS, Verification, Validation, Unit Testing

1 INTRODUCTION AND RELATED WORK

Modeling plays a crucial role in the design of complex systems. Through it, it is possible to check their correctness and performance before final implementations. It also reduces development times and the overall costs, and facilitates the detection of bugs in any project. Because of it, Model-Based Systems Engineering (MBSE) has become a common perspective when implementing complex systems, having an incremental presence both in research and industrial environments (Shkarupylo 2016) (Hutchinson et al. 2011). This methodology focuses on the creation of models describing the final product, extending and improving them over the different phases of the design flow. Some of these models only detail static aspects of the domain, as their entities structure. Others allow us to model the dynamic interactions between components of the system. However, although these models are clearly useful, their definitions and descriptions have to be reproduced later in the software implementation. To facilitate this transition it is also common the use of executable models. They provide more complex mechanisms for defining dynamic system behaviors and simplifying the related implementation. This practice goes hand by hand with the principles of the Modeling and Simulation-Based Systems Engineering (M&SBSE) (Gianni, Daniele and D'Ambrogio, Andrea and Tolk, Andreas 2014).

Once the model has been developed, there are several ways to verify its behavior. One of the most common techniques is testing. Testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior

(Utting and Legeard 2010). This procedure usually implies the generation of representative sets of test cases. They include the necessary inputs to complete the system under tests, the expected results, and prefix and postfix values (to put the system into an appropriate state to receive the input or to prepare the system for next tests). Test cases can be defined based on the knowledge of domain experts or can be generated following different coverage criteria. These criteria are usually based on graphs of the behavior of the system, logic relation between its entities, input-space partitioning, or syntax-based methodologies (Ammann and Offutt 2016). To contrast these test cases against the actual system, several techniques have been used over the years in the software industry, usually applying different techniques in each activity of the development flow. Some of them imply monitoring the internal states and data flows inside the artifact under testing, and others apply black box testing. As a result, there exist a great variety of techniques and criteria (Jovanović 2006). However, although some authors have brought these methods closer to the simulation field (Hollmann et al. 2012) (da Silva and de Melo 2011), they are not well-established yet. In this paper we focus on one of the most used techniques, unit testing (Runeson 2006) (Huizinga and Kolawa 2007). Unit testing performs a low-level assessment of the units produced in the implementation phase. It allows finding bugs in early stages, which helps to increase confidence in the software artifact, and facilitates the integration of subsystems.

Verification techniques widely used in the software industry have little presence in the M&S field. Ad-hoc techniques are generally used to verify simulations and most of the simulation engines do not have complete and robust tools for validating the generated models. Bringing these software techniques to the M&S field would help to automate the verification of simulation in a straightforward way.

Several popular and well-known formalisms can be used to simulate complex systems and to integrate these testing methods. One of the most used ones is Discrete Event System Specification (DEVS)(Zeigler et al. 2000). This formalism encapsulates system functionalities in modules and hierarchically connects them until reaching the top view of the system. This approach facilitates reusability, sharing, and validation of the individual components of the system. Moreover, the definition of the DEVS formalism makes possible the execution of the resulting models. This paper takes advantage of these convenient features of DEVS to integrate a unit testing framework in one of the DEVS M&S engines of the state-of-art. Through this framework it is possible to easily inject test cases input data into the models, capture the states and outputs of all their internal components, and compare them against the expected behaviour in a straightforward way. For injecting and checking values in the model, the concept of experimental frame is already available (Rozenblit 1991). This concept, very common in the M&S field, consists of the generation of stimulus in the input of a model through generator entities. The resulting variables are collected by transducers, that analyze these variables. Then, several constraints or checking tests are applied over this collected data to study or validate the behavior of the system. This approach can help when implementing unit tests in a simulation environment.

Regarding the state of the art, other works have implemented Verification and Validation (V&V) techniques on DEVS models, but using a different perspective. It is worth to mention some of them. (Olsen and Raunak 2015) discuss several V&V methodologies compatible with the DEVS formalism to perform structural and behavioral validation, proposing a way to measure their confidence. In (Wainer et al. 2002), authors used the concept of the experimental frame to present a tool capable of checking the presence of specific values in input and output ports of DEVS components. In (Henares et al. 2019) a different approach is presented, introducing constraints over the outcomes produced in different DEVS components and generating alerts when a condition is not met.

In our V&V process proposed in this paper, test cases can be represented in several ways. They can be specified manually in the code, be stored in databases or be defined in a standard serialization syntax (as JSON or XML). We chose the XML standard for the representation of all the necessary inputs, outcomes, and components' internal states. In the simulation field, other authors have already used these formats quite

often, but for other purposes. For instance, they have been used to represent events and state-traces (Li, Vangheluwe, Lei, Song, and Wang 2011) or to propose simulation meta-languages (Janoušek, Polášek, and Slavíček 2006).

The paper is organized as follows: Section 2 presents the unit testing validation tool and describes the format of the input XML files. Section 3 describes the implementation of the testing framework. Section 4 shows an example of a complex system where the tool was applied. Finally, Section 5 summarizes the contribution and present the main conclusions.

2 DEVS UNIT TESTING FRAMEWORK

In this section, we first give an overview of the DEVS formalism to show in the following how to define test cases over this formalism, explaining the possibilities of the chosen format.

2.1 Introduction to DEVS formalism

DEVS is a general formalism for discrete event system modeling based on set theory (Zeigler, Praehofer, and Kim 2000). The DEVS formalism provides the framework for information modeling which gives several advantages to analyze and design complex systems: completeness, verifiability, extensibility, and maintainability. Once a system is described in terms of the DEVS theory, it can be easily implemented using an existing computational library. The parallel DEVS (PDEVS) approach was introduced after 15 years as a revision of Classic DEVS. Currently, PDEVS is the prevalent DEVS, implemented in many libraries. In the following, unless it is explicitly noted, the use of DEVS implies PDEVS.

DEVS enables the representation of a system by three sets and five functions: input set (X), output set (Y), state set (S), external transition function (δ_{ext}), internal transition function (δ_{int}), confluent function (δ_{con}), output function (λ), and time advance function (ta). DEVS models are of two types: atomic and coupled. Atomic models are directly expressed in the DEVS formalism specified above. Atomic DEVS processes input events based on their model's current state and condition, generates output events and transition to the next state. The coupled model is the aggregation/composition of two or more atomic and coupled models connected by explicit couplings. Given the recursive definition of coupled models, they can be a part of a component in a larger coupled model system giving rise to a hierarchical DEVS model construction.

DEVS conceptually separates models from the simulator, making it possible to simulate the same model using different simulators working in centralized, parallel or distributed execution modes. DEVS models can be encoded in different programming environments and simulated with a simple ad-hoc program written in any language. However, there exist many DEVS M&S engines around the world, like DEVSJAVA, CD++, xDEVS, aDEVS, etc. (Risco-Martín, Mittal, Fabero, Zapater, and Hermida 2017)

2.2 Test cases definition

Apart from the model definition, it is convenient to generate test cases that help to validate the behavior of the different components of the system. These tests are often defined after the model implementation. However, they can be developed in parallel or even before the model definition (test-driven development). This growing practice has been widely studied and usually results in defect reduction and quality improvements in the final software artifact (Janzen and Saiedian 2005).

In the following, we show the structure that we have defined for the M&S test case files used in the presented unit testing framework. As shown in Figure 1, we use XML files with two main sections: *Generators* and

```
<UnitTest accumulateOutputs="false">

    <Generators>
        <Generator name="generator_name" type="path.to.the.generator_class" port="oOut"
            connectTo="path.to.other.module_port" />
        <!-- ... -->
    </Generators>

    <States>
        <State time="1328227962">
            <Port name="comp1.comp2.comp3.out_port1">
                <[OutputType] attr1="val1" attr2="val2" />
                <[OutputType] attr1="val3" attr2="val4" />
                <!-- ... -->
                <[OutputType] attr1="val5" attr2="val6" />
            </Port>
        </State>

        <!-- ... -->

        <State time="1328235606">
            <Port name="comp1.out_port1">
                <OutputType attr1="val1" attr2="val2" />
            </Port>

            <Atomic name="comp1.comp2.comp3" phase="active" sigma="200" />
            <Coupled name="comp1.comp2.comp4" simple_attr="val1"
                obj_attr.simple_attr="val2"/>
        </State>
    </States>
</UnitTest>
```

Figure 1: XML-based syntax to specify test cases. It allows checking port outputs and internal attributes in all the components of the DEVS simulation.

States. In the *Generators* section, we define the different modules that are used to inject inputs into the system. Given the object oriented paradigm used by most of the DEVS simulation engines, these generators are defined as classes in the project structure and are dynamically instantiated in the testing procedure. However, it is worthwhile to mention that this method can be easily adapted to other non object oriented simulation engines. Additionally, since our target simulator is JAVA-based, each *Generator* element specifies the classpath of the generator module and the input port to inject the produced values. It is notable to point that several generators can be defined, even in different levels of the hierarchical design.

The *States* section include information about the variables and outputs of a given simulation time. Each *State* can incorporate port outputs and modules state variables. *Port* elements have to include in the name attribute the complete path of the port to monitor. This includes both the path of the module containing the port and the port name, in a fully qualified syntax: *component1.component2.componentN.portName* As seen in the last state in Figure 1, it is also possible to inspect the values of both Atomic and Coupled modules. It is worth mentioning that these variables can be checked even if they are private in the class design. The comparison will be made with a previous casting of the variable to a string (so objects are also allowed). Moreover, inspecting attributes inside other object attributes is also allowed, following a syntax like: *object1.object2.attribute_name*.

The *accumulateOutputs* of the root *UnitTest* element allows us to specify how to record the outputs in the transducers. If it is set to `false`, the verification occurs over the values generated at the precise moment specified in the state. On the contrary, if it is set to `true`, the transducers accumulate all the values generated since the previous state. Moreover, this flag can also be specified in individual *State* elements to overwrite the default behavior specified before.

3 IMPLEMENTATION

In this section, we describe the implementation of the testing framework. It includes the management of the test entities and the changes applied in the model to make the verification possible.

The proposed testing framework has been implemented in the Java branch of the xDEVS simulation engine. The methodology used is in line with the experimental frame. In this way, previous to the simulation phase, some additional modules are automatically added to the original design. These modules do not affect the behavior of the system and are only intended to inject data into the system and to obtain the resulting values. On the one hand, all the generators defined in the test file are instantiated and connected to the specified modules. These connections are made preserving the hierarchy levels, so each generator belongs to the same coupled module than the module to which is connected. On the other hand, a transducer is added for each monitored port specified in the test file. These transducers only receive the response values of the system to compare them with the expected ones.

Both the generators and the transducers can be connected to any module of the system to test, regardless of where it is in the module hierarchy. Hence, it can also be used to check the behavior of internal components while running the overall simulation.

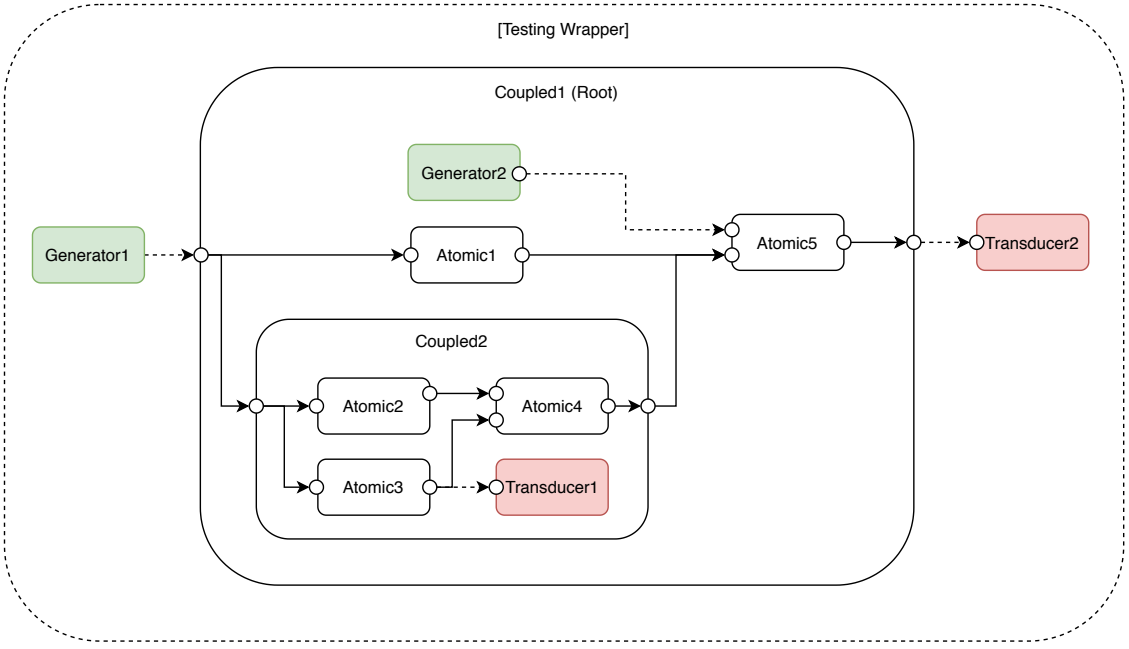

Figure 2: Experimental frame methodology to perform unit testing.

To perform the verification, the xDEVS library provides a *UnitTester* helper class. It only receives as arguments the root coupled module of the design and the XML testing file. Internally, this root component is allocated inside a *TestingWrapper* (as shown in Figure 2). This coupled module allows us to include generators and transducers at the root level and includes methods to facilitate the addition of these components

in all the levels of the hierarchy. After that, the states are processed sequentially. For each state, the time difference between the current and the next specified states is calculated. Then, the simulation advances according to this difference. Depending on the configuration stated above, the transducers contain either all the values generated from the previous state or the ones generated at the exact time specified in the current state. After each state, the values in the transducers are compared with the expecting ones. Moreover, the state of the different atomic modules of the system can also be checked. When some discrepancy is found, an exception is thrown indicating the reason, the simulation time, and the location where the problem occurs. Through the use of this methodology, the verification of the model can be automatized. This results in a more trustworthy implementation flow and in an increase in the model quality.

The whole verification process described above is summarized in Algorithm 1.

Algorithm 1 Unit testing validate process

```
root_entity ⇐ instantiate_root()
unit_tester ⇐ instantiate_unit_tester(root_entity)
for gen_path, gen_out_port, model_in_port ∈ input_generators() do
    generator ⇐ instantiate_generator(generator_info.path)
    unit_tester.add_generator(generator, gen_out_port, model_in_port)
end for
for out_port ∈ monitored_ports() do
    unit_tester.add_transducer(out_port)
end for
unit_tester.initialize()
last_time ⇐ 0
for state ∈ monitored_states() do
    time_diff ⇐ state.time − last_time
    if state.accumulative then
        unit_tester.simulate(time_diff)
    else
        unit_tester.simulate(time_diff − 1)
        unit_tester.clear_transducers()
        unit_tester.simulate(1)
    end if
    check_transducers()
    last_time ⇐ state.time
end for
```

4 USE CASE

In this section, the unit testing framework is applied over a validated design. Specifically, we use the DEVS-based SFIDE data center simulator (Penas et al. 2017). This simulator provides a customizable model to represent the architecture of data centers and measure their efficiency and performance. For that, it allows introducing custom allocation policies, cooling control strategies, and custom server models. Although it is a dynamic model and can generate multitude of data centers structure based on configuration files, we use a data center with 400 servers, 10 servers per rack and a 2 racks per In-Row Cooling Units (IRC). This results in a model of 466 atomic models (distributed as can be seen in Figure 3). The input of the model is a dataset containing the characterization of different compatational jobs, based on the data collected in a real

data center. The output of the model consists of a summary of the performance of the data center, including jobs allocations, and temperatures and energy power of different components.

In this section, we first introduce the SFIDE simulator. Then we describe the unit test cases applied over the system. Finally, we discuss the results generated by the testing framework.

4.1 Overview of the SFIDE simulator

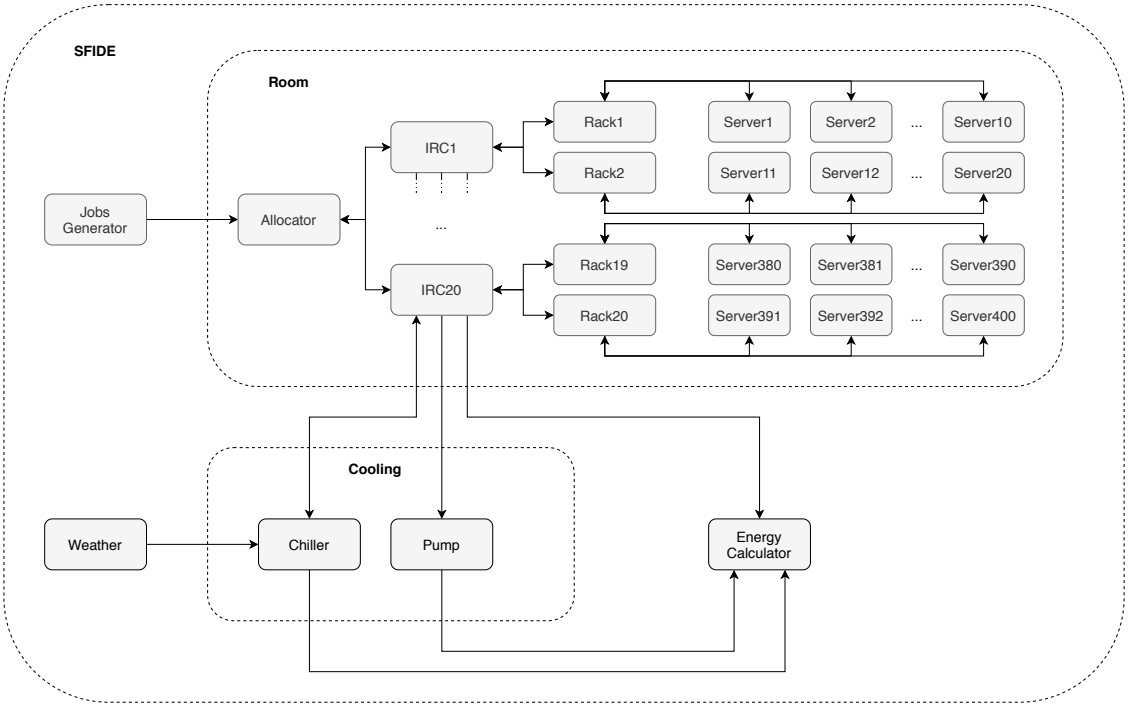

Figure 3: Root view of the SFIDE simulator model.

Inside the SFIDE simulation model, there are two main coupled models: (i) the *Room*, representing all computing infrastructure and the allocation policies, and (ii) the *Cooling*, controlling the temperature of the whole data center and establishing cooling policies. Apart from that, there are three atomic modules in the root module of the simulator: (i) the *JobGenerator*, that loads the characterization of computational jobs, (ii) the *Weather* module, that simulates the room temperature variances, and (iii) the *EnergyCalculator*, that groups the efficiency and performance stats to generate a report.

The jobs generated in the *JobsGenerator* goes to the *Allocator*, inside the Room. This module has the responsibility of allocating incoming jobs to specific servers. SFIDE has several *Allocator* modules, each one with a different allocation policy. It is also possible to implement custom *Allocator* modules when none of the available policies suit a specific use case. When a job is assigned, it goes through the corresponding *IRCs* (In-Row Cooling) and *Racks* until reaching the suitable *Servers*. There, the consumption is computed, changing parameters as CPU temperature or airflow accordingly. All the information about the status of the servers is recovered by the *Rack* modules and goes to the *Cooling* system. With this grouped information, it takes actions to stabilize the temperature if needed. For that, it dispose of two Atomic models: a *Chiller* and a *Pump*.

```
<UnitTest>
    <States>
        <State time="1328227960">
            <Atomic currentState.chillerPower="null" currentState.pumpPower="null"
                    currentState.totalPower="null" currentState.towerPower="null"
                    currentState.weatherTemp="5.0" name="environment.dc01.energyCalculator"/>
        </State>
        <State time="1328227961">
            <Port name="environment.dc01.room.ConsLinearAllocator.oJobirc0">
                <Job id="1001" ircName="irc0" rackName="rack0" serverName="s0"
                     workloadName="calculix128"/>
            </Port>
        </State>
        <State time="1328228060">
            <Atomic currentState.chillerPower="null" currentState.pumpPower="null"
                    currentState.totalPower="294633.6546870043" currentState.towerPower="null"
                    currentState.weatherTemp="5.0" name="environment.dc01.energyCalculator"/>
        </State>

        <!-- ... -->
    </States>
</UnitTest>
```

Figure 4: Extract of the test case file generated for the SFIDE simulator.

As stated before, all the data produced both in the Room and the Cooling modules are also directed to the *EnergyCalculator*. It generates a log with the status of the data center in each moment. In this way, it allows checking several allocation policies and architecture configurations straightforwardly, without having to test it directly in a real environment.

4.2 Generation of test cases

The test cases were generated using a workload characterization dataset (Curie High Performance Computing). These data were collected in CEA-Curie between March 3rd and October 20th and have been used previously in previous studies of the SFIDE simulator (Penas et al. 2017). The data center scenario consists of 20 racks, each one of them hosting 10 servers. The cooling equipment includes an in-row cooler for each couple of racks, cooled down using a chiller and a tower outside the server room.

A set of correct values was extracted performing a simulation with this validated dataset. Specifically, the expected values of job allocation, weather temp, and chiller, pump, tower, and total power were collected. Based on them, a new dataset was generated with the simulation results grouped by timestamps, resulting in 454 simulator states. During the simulation, 135 jobs are allocated and 366 power states are checked in the `EnergyCalculator` module (including the rest of the variables aforementioned). An extract of this test case file can be seen in Figure 4. In it, the expected behaviour is represented as a set of states (identified by their related simulation time). For checking the correct functioning of the *Allocator*, its expected outcomes are specified in the states matching the expected allocation time. This includes the information of the identifier of the job, the names of the assigned IRC, rack and server, and the workload type. On the other hand, for checking the values of the `EnergyCalculator` its *currentState* object is inspected in an *Atomic* XML element. The expected values of its relevant attributes are specified through the syntax aforementioned.

4.3 Results and dicussion

Once the test cases file was generated, we run the SFIDE simulator applying that constraints. To check the correctness of the simulation behavior, we manually introduce some errors. We contemplate two error scenarios: (i) a bad design in the Server module, and (ii) errors in the input configuration files.

Server model error: the data center scenario described above was built with Solana server models. These servers are already implemented in the simulator and are specified through a model resulting from a characterization the physical server. In this case, we slightly increase the leakage of these servers. This modifies the expected values of energy consumption, increasing the total power consumed by the system. As a result, the unit testing module throws an exception indicating this situation.

```
java.lang.RuntimeException:
Attribute currentState.totalPower does not match
at simulation time 1.328261045E9
in element environment.dc01.energyCalculator'
(expected: 294854.86540038267, found: 294980.71480302897)
```

Figure 5: Error message thrown by the unit testing framework on test fail.

Configuration files: We introduce errors in two of the input configuration files of the SFIDE simulator. Specifically, we modify (i) the general configuration file and (ii) the workloads specification file. The general configuration file specifies the equipment that composes the data center and the initialization values (e.g. initial status of the computing and cooling equipment or initial temperature of the room). We have modified the value indicating the initial temperature. Consequently, an exception is thrown pointing out a discrepancy in the expected weather temperature (in the `EnergyCalculator` module). One of these exceptions can be seen in Figure 5. The exception message informs the user of the test fail cause, the simulation time, the modules involved, and the expected and actual values.

In an independent scenario, we also modified the workloads specification file. This file describes the resources consumed by each kind of server executing each specific task. Here, we modify the duration of one of the characterized tasks in the Solana servers. As a result, we get again an exception reporting an unexpected increase in the total power consumption.

With these two error scenarios, we pretend to reproduce the introduction of typical bugs always present in any development process. The modified values are not enough to cause an exception in the data center simulator for having exceeded some of the limits specified in some of their modules. As a result, they remain transparent in the simulation phase and generate unexpected values. However, with the unit testing framework presented in this paper, we are capable of detecting these kinds of situations specifying the expected evolution of relevant variables of the system.

5 DISCUSSION

In this work, a framework for performing unit testing based verification over DEVS simulations has been presented. Through it, we transfer a common verification technique widely used in the software industry to the simulation field. The framework describes an XML-based syntax that allows us to easily define the relevant states of the system and the specific aspects to validate the whole states' trajectory. These aspects include both the output generated by the different models and specific attributes regarding the internal state of the different components of the system.

This framework has been incorporated into the xDEVS library. Its implementation follows the principles of the experimental frame. According to the specified configurations, several generators and transducers

are transparently added to the original design during the initialization phase. This allows us to introduce stimulus in the entity to test and collect all the necessary outputs for further verification (even in different levels of the hierarchy design).

This unit-testing framework has been tested on a previously validated complex model, the SFIDE data center simulator. According to the input files and the expected outputs, test cases were generated. They included 454 states describing representative outputs of the *Allocator* and internal states of the *EnergyCalculator*, module that groups statistics of performance of the whole simulator.

The use of this framework allows adding a verification layer to the increasingly complex simulation models. Through the use of test cases, the correctness and behavior of the models can be checked. Moreover, it allows grouping the expected behaviors present in the modules of the different depth levels of the design in a standard XML syntax. These files are easily readable and can complement the documentation of the project. Finally, the unit testing framework presented here can be independently used without the use of specification files through the direct use of its testing auxiliary classes.

ACKNOWLEDGMENTS

This project has been partially supported by the Spanish Ministry of Science and Innovation and by the Education and Research Council of the Community of Madrid, under grants PID2019-110866RB-I00 and S2018/TCS-4423, respectively.

REFERENCES

Ammann, P., and J. Offutt. 2016. *Introduction to software testing*. Cambridge University Press.

Curie High Performance Computing. http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm. Accessed: 2019-12-10.

da Silva, P. S., and A. C. de Melo. 2011. "On-the-fly verification of discrete event simulations by means of simulation purposes". In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pp. 238–247.

Gianni, Daniele and D'Ambrogio, Andrea and Tolk, Andreas 2014. "Introduction to the modeling and simulation-based systems engineering handbook.".

Henares, K., J. L. Risco-Martín, and M. Zapater. 2019. "Definition of a transparent constraint-based modeling and simulation layer for the management of complex systems". In *Proceedings of the Theory of Modeling and Simulation Symposium*, pp. 9. Society for Computer Simulation International.

Hollmann, D. A., M. Cristiá, and C. Frydman. 2012. "Adapting model-based testing techniques to DEVS models validation". In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, pp. 6. Society for Computer Simulation International.

Huizinga, D., and A. Kolawa. 2007. *Automated defect prevention: best practices in software management*. John Wiley & Sons.

Hutchinson, J., M. Rouncefield, and J. Whittle. 2011. "Model-driven engineering practices in industry". In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 633–642. ACM.

Janoušek, V., P. Polášek, and P. Slavíček. 2006. "Towards DEVS Meta Language". *ISC 2006 Proceedings*, pp. 69–73.

Janzen, D., and H. Saiedian. 2005. "Test-driven development concepts, taxonomy, and future direction". *Computer* vol. 38 (9), pp. 43–50.

Jovanović, I. 2006. "Software testing methods and techniques". *The IPSI BgD Transactions on Internet Research* vol. 30.

Li, X., H. Vangheluwe, Y. Lei, H. Song, and W. Wang. 2011. "A testing framework for DEVS formalism implementations". In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pp. 183–188. Society for Computer Simulation International.

Olsen, M. M., and M. S. Raunak. 2015. "A method for quantified confidence of DEVS validation.". In *SpringSim (TMS-DEVS)*, pp. 135–142.

Penas, I., M. Zapater, J. L. Risco-Martín, and J. L. Ayala. 2017. "SFIDE: a simulation infrastructure for data centers". In *Proceedings of the Summer Simulation Multi-Conference*, pp. 34. Society for Computer Simulation International.

Risco-Martín, J. L., S. Mittal, J. C. Fabero, M. Zapater, and R. Hermida. 2017. "Reconsidering the performance of DEVS modeling and simulation environments using the DEVStone benchmark". *SIMULATION* vol. 93 (6), pp. 459–476.

Rozenblit, J. W. 1991. "Experimental frame specification methodology for hierarchical simulation modeling". *International Journal Of General System* vol. 19 (3), pp. 317–336.

Runeson, P. 2006. "A survey of unit testing practices". *IEEE software* vol. 23 (4), pp. 22–29.

Shkarupylo, V. 2016. "A technique of DEVS-driven validation". In *2016 13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET)*, pp. 495–497. IEEE.

Utting, M., and B. Legeard. 2010. *Practical model-based testing: a tools approach*. Elsevier.

Wainer, G., L. Morihama, V. Passuello et al. 2002. "Automatic verification of DEVS models". In *Proceedings of the 2002 Spring Simulation Interoperability Workshop*. Citeseer.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation. Integrating Discrete Event and Continuous Complex Dynamic Systems*. 2 ed. Academic Press.

AUTHOR BIOGRAPHIES

KEVIN HENARES is a Ph.D. candidate at the Complutense University of Madrid (UCM). His work focuses on the development of robust modeling and simulation methodologies to study the behavior of complex systems. His email address is khenares@ucm.es.

JOSÉ L. RISCO-MARTÍN received his Ph.D. from Complutense University of Madrid, and currently is Associate Professor in the Department of Computer Architecture and Automation at Complutense University of Madrid. His research interests include computer aided design, and modeling, simulation and optimization of complex systems. He can be reached at jlrisco@ucm.es.

JOSÉ L. AYALA got his Ph.D. in Electronic Engineering from Technical University of Madrid and is currently an Associate Professor in the Department of Computer Architecture and Automation at Complutense University of Madrid. His research interests focus on IoT and edge solutions for personalized medicine approaches, including health monitoring, wireless sensor networks and disease modeling. His email address is jayala@ucm.es.

ROMÁN HERMIDA received his Ph.D. from Complutense University of Madrid, and is currently Full Professor in the Department of Computer Architecture and Automation at the same university. His research interests include design automation, computer architecture and embedded systems. He can be reached at rhermida@ucm.es.