DEVS Models Design and Test using AGILE-based Methods with DEVSimPy

Timothée Ville, Laurent Capocchi, Jean-François Santucci

UMR SPE University of Corsica

ville@univ-corse.fr, capocchi@univ-corse.fr, santucci@univ-corse.fr

Keywords: Discrete event simulation; DEVS; behavior test; AGILE

Abstract

Validation and test of DEVS models at the early phases of the Design process is a crucial topic when dealing with complex DEVS models. Based on Software Engineering test methods, we present in this paper a new approach which integrates Agile methods in the process of the simulation in order to design and test DEVS models. We propose an implementation in Python language based on the use of aspect programming concept (patch, mocking objects and decorators). This implementation is performed in the framework of the DEVSimPy environment with the definition of a plug-in dedicated to the automatic generation and execution of test scenario. Two pedagogical example have been used in order to point out the feasibility of the approach.

1. INTRODUCTION

This paper deals with validation and test of DEVS models at the early phase of the design process. DEVS (Discrete Event System specification) is a widely used formalism in the framework of simulation of complex systems. The validation of models is traditionally a step which is relegated at the end of the design process: once the models have been defined and coded, experiments are conducted in order to validate them using simulation. However this traditional way to perform validation of models is often an expensive and time-consuming activity and the resulting quality of the models is still poor. Consequently, new approaches for coping with these challenges are necessary. The same remarks can be formulated when dealing with Software Testing. Considering software testing, one emerging trend is stronger integration of testing as early as possible in the design process of a program. For that reason software engineering has proposed new design and test as Agile methods which include Test Driven Development (TDD) [Fraser et al., 2003] and Behavioral Driven Development (BDD) [Solis and Wang, 2011] methods.

In order to go on with the analogy between modeling and simulation and Software Engineering, one can imagine applying BDD and TDD Agile methods to the Design and Test of DEVS models. Our main objective is to develop an approach that is able to use different Software Testing techniques stemming from Software Engineering (Agile methods to be more specific) that are applied to DEVS models design in order to improve quality assurance of the resulting DEVS models while proposing inexpensive and no time-consuming activity.

Our approach consists in applying the BDD method for the design of DEVS models. In order to achieve this goal, we have carefully performed a correspondence between the BDD method when applied to software and the BDD method which has to be applied to DEVS models design.

The problem is to perform a BDD method when defining DEVS models. Defining such a method in the DEVS Modeling and Simulation context requires the resolution of the following basic problems: (i) to define a semi-formal format for the behavioral specification of the test for any atomic models involved in a DEVS model, (ii) to define how to generate parameters for a test from a specification document of the tests of DEVS models, (iii) to define how to perform the previously defined tests using simulations. To solve these problems we proposed to: (i) to define a semi-formal format from the natural specifications of FDDEVS as proposed by B.P. Zeigler [Zeigler, 1976], (ii) to use the specificity of the Python Language to generate a test (by means of the decorator programming concept), (iii) the tests are executed by combining software engineering programming concepts such as decorators, patch, mocking objects in order to perform the tests using DEVS simulations in the framework of the DEVSimPy [Capocchi et al., June] environment. DEVSimPy is a collaborative general user interface implemented in Python language allowing us to experiment new approaches inside the DEVS formalism. To ensure this, we use DEVSimPy plug-ins in order to be more generic.

The next part of the paper gives the background of the paper: the DEVS formalism and the DEVSimPy framework are briefly introduced before the presentation of the main notions involved in Agile methods. In section 3. an overview of the proposed approach is briefly presented. Section 4.1. is devoted to the definition of the semi-formal format chosen for the user to write the behavioral specification of the tests as required in a BDD method. In section 4.2. we describe how we have been able to generate the test parameters which will be used to validate the models. Section 4.3. presents how the tests are performed using simulations within the DEVSimPy framework by integrating decorators, patches and mocking objects into DEVS models. The last part will permit to conclude and to give a brief overview of future work we envision.

2. BACKGROUND

2.1. DEVS Formalism and Tools

The Discrete EVent system Specification (DEVS) formalism introduced by Zeigler [Zeigler, 1976] provides a means of specifying a mathematical object called a system. Basically, a system has a time base, inputs, states, outputs, and functions for determining next states and outputs given current states and inputs. The DEVS formalism is a simple way in order to characterizes how discrete-event simulation languages may specify discrete-event system parameters. It is more than just a means of constructing simulation models. It provides a formal representation discrete-event systems capable of mathematical manipulation just as differential equations serve this role. Furthermore by allowing an explicit separation between the modeling phase and simulation phase, the DEVS formalism is one of the best ways to perform an simulation of systems using a computer. In the DEVS formalism, one must specify: (i) basic models from which larger ones are built, and (ii) how these models are connected together in hierarchical fashion. An atomic model allows specifying the behavior of a basic element of a given system. Connections between different atomic models can be performed by a coupled model. A coupled model, tells how to couple (connect) several component models together to form a new model. This latter model can itself be employed as a component in a larger coupled model, thus giving rise to hierarchical construction. An atomic DEVS model can be considered as an automaton with a set of states and transition functions allowing the state change when an event occur or not. When no events occurs, the state of the atomic model can be changed by an internal transition function called δ_{int}. When an external event occurs, the atomic model can intercept it and change its state by applying an external transition function called δ_{ext}. The life time of a state is determined by a time advance function called t_a. Each state change can produce output message via an output function called λ. A simulator is associated with the DEVS formalism in order to exercise instructions of coupled model to actually generate its behavior. The architecture of a DEVS simulation system is derived from the abstract simulator concepts [Zeigler, 1976] associated with the hierarchical and modular DEVS formalism.

Concerning the natural language specifications FDDEVS proposed by Zeigler in [Zeigler and Sarjoughian, 2012]. FDDEVS gives us many advantages; in particular it offers a support for XML translation or graphic representation. Here is a basic example of a generator DEVS model specification.

```
to start hold in generate for time 10!
after generate output Job!
from generate go to generate!
when in generate and receive Stop then go to
    passive!
passivate in passive!
```

Listing 1. DEVS specification of the Generator model.

The Figure 4.1. show what are information are involved by the specification of Listing 1.

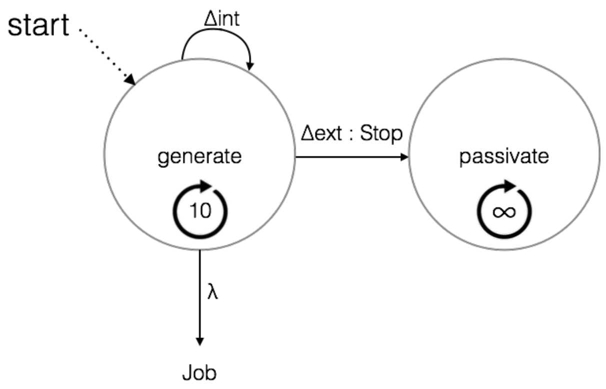


Figure 1. State automaton of model from Listing 1.

There are many tools which provide a user interface dedicated to help the user to define DEVS models and to perform simulations. A non exhaustive list can be done: PowerDEVS, DEVSim++, DEVSJAVA, VLE, DEVSimPy, CD++Builder, ATOM3, MS4Me, etc. Special attention will be given to DEVSimPy (stand for DEVS simulator in Python language) which is a collaborative modeling and simulation (M&S) software.

DEVSimPy (Python Simulator for DEVS models) [Capocchi et al., June] is a user-friendly interface for collaborative modeling and simulation of DEVS systems implemented in Python language. Python is a programming language known for its simple syntax and its capacity to allow modelers to implement quickly their ideas [Langtangen, 2005]. The DEVSimPy project used the Python language and provides a GUI based on PyDEVS [Bolduc and Vangheluwe, juin 2001] API in order to facilitate both the coupling and the re-usability of PyDEVS models. This API is used in the excellent multi-modeling GUI software named ATOM3 [de Lara and Vangheluwe, 2002] which allows to use several formalisms without focusing on DEVS. DEVSimPy is an open source project under GPL V3 license and its development is supported by the University of Corsica Computer Science research team. It uses the wxPython graphic library which is a wrapper of the most popular WxWidgets C library. DEVSimPy can be downloaded at http://code.google.com/p/devsimpy.

The main goal of this environment is to facilitate the modeling of DEVS systems using the GUI dynamic libraries and the drag and drop functionality. With DEVSimPy, models can be stored in a dynamic library in order to be reused and shared. The creation of dynamic libraries composed with DEVS components is easy since the user is coached by dialogs and wizard during the building process. With DEVSimPy, complex system can be modeled by a coupling of

DEVS models and the simulation is performed in a automatic way. Moreover, DEVSimPy allows the extension (or the overwrite) of their functionality in using special plug-ins managed in a modular way. The user can enabled/disabled a plug-in using a simple dialog window.

2.2. Agile Test Methods

Agile methods [Cockburn, 2002] is the increasingly common practice throughout the lifecycle to develop a software iteratively. These methods may include: the Test Driven Development (TDD) [Fraser et al., 2003] method and its extension/revision, the Behavior Driven Development (BDD) [Solis and Wang, 2011] method. TDD is a software development methodology which essentially states that for each unit of software, a software developer must: (i) define a test set for the unit first, (ii) then implement the unit, (iii) finally verify that the implementation of the unit makes the tests succeed. BDD is a specialized version of TDD which focuses on behavioral specification of software units. It is based on: (i) the use of examples to describe the behavior of the application, or code units; (ii) automating those examples to provide quick feedback and regression testing; (iii) finally, the use of “Mocks ” [moc] replacing the code modules which have not yet been written.

The main steps of the BDD method can be summarized by the following points:

1. Behavior-driven development specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit; usually the desired behavior should be specified using a semi-formal format for behavioral specification.

2. Then a specification document has to be read and each scenario of the document is breaking up into meaningful clauses. Each individual clause in a scenario is transformed into some sort of parameter for a test.

3. The framework then executes the test for each scenario, with the parameters from that scenario.

In this paper, we present a set of main BDD characteristics which are used to implement the test of DEVS model.

3. PROBLEM DESCRIPTION

The context of the proposed work relates to the cycle of software development. Traditionally software development corresponds to a logical and intuitive approach described in Figure 2.

The basic approach of such traditional cycle is: we code first and then we perform the tests.

As explained in the introduction, we have developed an analogy between Software Engineering design process and DEVS models design process. From Software Engineering point of view, the structure of the software is first defined, the implementation of the related classes and functions is performed and finally the implementation is executed in order to find errors (as it can be seen in the engineering part in Figure 2). The same kind of development cycle is used for DEVS models development: the structure of models is first defined, models are then implemented and the obtained implementation is simulated in order to find potentials errors (as it can be seen in the DEVS part in Figure 2). If errors are raised during simulation, they have to fixed one by one and verified each time by simulation. We also can deduce numerous problems from a validity and productivity points of view from the Figure 2:

- The models cannot be prepared for all the situations and the predictions have a reliability decreased because of the visible uncertainty of the produced results.
- The behavior of the developed models has big chances to be erroneous and it will certainly be necessary to adjust certain parameters or certain functions.
- The maintenance, the re-factorization or the evolution of a model guaranteed not the preservation of the behavior.
- The same model implemented in various environments is completely different. It can raise problems in the optics of a standardization.

There are many manners to test a DEVS model (atomic or coupled model). In [Li et al., 2011] the authors focus on the validation of DEVS formalism implementation. The approach described in [Byun et al., 2009] concerns the correctness of a given simulation. It proposes a framework allowing to check all the possible paths involved in a given simulation model. In [Hu et al., 2007], test agents are presented. This paper proposes a definition of a test agent which is connected to the I/O of a given model and is used to point out the behavioral information concerning the model. The test is performed using this information. However the proposed approach is not a generic one since test agent should be specific to a given model. In the same idea, we have developed a DEVSimPy plug-in to manage an universal test agent using the Behave Framework tool [beh, Ville, 2013]. However an im-

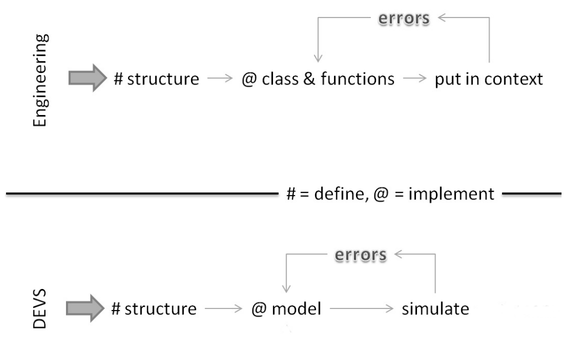


Figure 2. Traditional cycle of model development.

portant drawback was the fact that the simulation has to be interrupted in order to manually performed the tests. In the next sub-section we give an overview of the developed solution in order to propose an embedded mechanism allowing to automatically take into account the test part at the beginning phase of the design process of DEVS models.

4. PROPOSED SOLUTION

As described before, the traditional cycle for DEVS model design raises numerous problems. The proposed approach is sumarized in Figure 3.

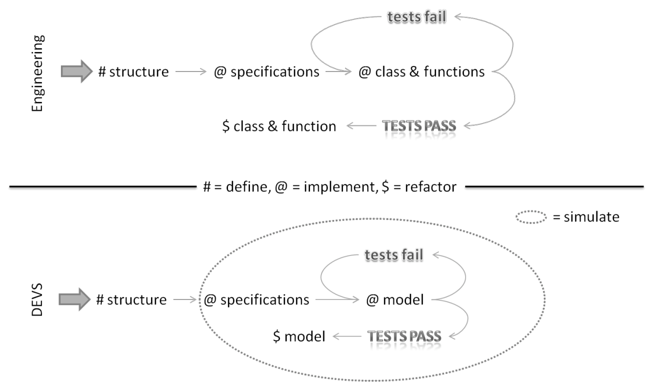


Figure 3. Proposed behavior driven development cycle.

We have developed a solution by analogy with the BDD approach defined in the Software Engineering domain. As shown in Figure 3 (engineering part) this approach consists in firstly define the structure of the software to be designed and immediately after to write the test specifications. Then the designer can implement and at the same time test the class and functions of the software to be designed. The same kind of development cycle is proposed for the design of DEVS models as it is shown on the DEVS part of Figure 3. The only difference is that the specifications writing and the model implementation are performed using a modeling and simulation framework. The simulation engine is used in order to perform the tests.

Implemented behavior in a DEVS model are directly tested. This brings numerous advantages:

1. Produced code is reliable.
2. The basic elements of the DEVS model can be tested one after the other.
3. Even if there is an evolution of the implementation of the DEVS model, the behavior remain the same.

The previously presented approach has been implemented in the framework of the DEVSimPy environment. The design and test process has been introduced in this Python Language oriented simulation environment. In order to propose a generic implementation we choose to define a DEVSimPy plug-in dedicated to: (i) the automatic generation of test scenario and (ii) the execution of the test scenario using the simulation kernel.

Figure 4 describes explicitly how a user is going to develop the Design and Test proposed methodology.

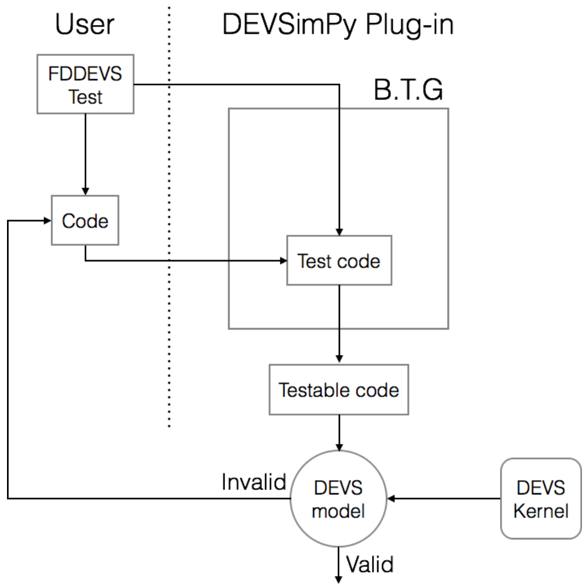


Figure 4. DEVS design and test proposed methodology.

The user has first to write the FDDEVS test specifications. Then he has to code part of (or completely) the DEVS model corresponding to the previously written specifications. The DEVSimPy plug-in will allow to:

- Generate testable code from the FDDEVS specification (own to the Behavior Test Generation).
- Integrate the testable code into the DEVS model already defined by the user.
- Simulate the integration of the testable code and the already defined DEVS model part.

If the result of the simulation points out an invalid DEVS model definition, the user has to rewrite the DEVS model and again execute the plug-in.

4.1. Test Scenario Specification

Specification is an important point when dealing with a BDD approach for DEVS formalism. It gives a manner to describe the test patterns associated with the behavior of a DEVS model which has to be defined and implemented. Instead of defining a new language we choose to select an already defined language which allows to describe DEVS modeling scheme under a semi-formal natural language. The pseudo-natural language proposed by Prof. Zeigler which is inspired from the grammar detailed in [Hong and Kim, 2006] has been adapted in order to offer a language for specifying test patterns of a future DEVS model. In order to use this specification language, we have defined a parser for the FDDEVS grammar with simpleparse [sim] tool helping. The proposed grammar follows the hierarchy depicted in figure 5.

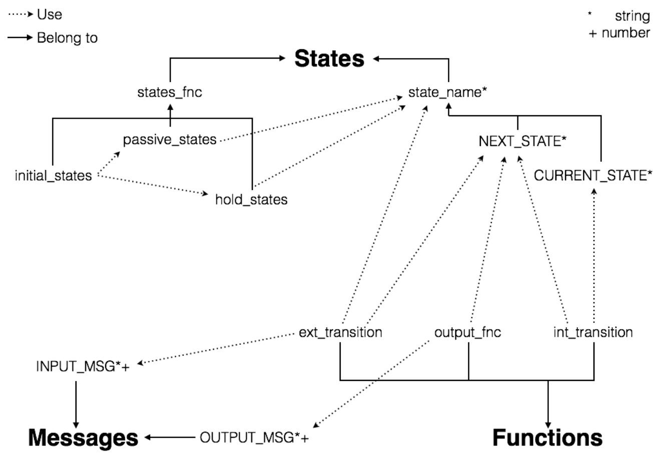
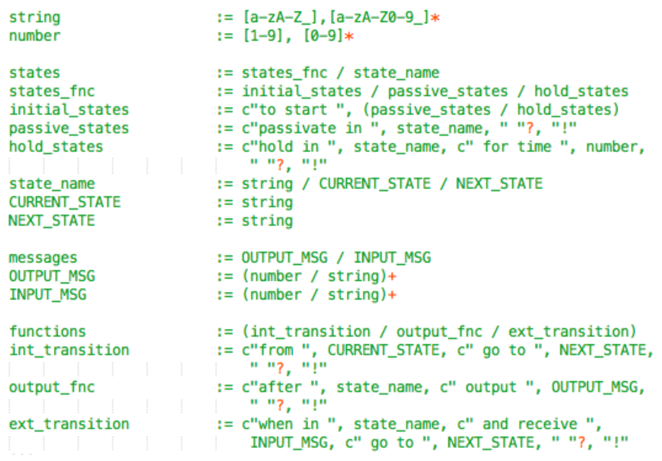


Figure 5. Diagram of the specification language grammar

From this diagram the grammar shown in figure 6 can be generate in EBNF format.



```
string                  := [a-zA-Z_],[a-zA-Z0-9_]*
number                  := [1-9], [0-9]*

states                  := states_fnc / state_name
states_fnc              := initial_states / passive_states / hold_states
initial_states          := c"to start ", (passive_states / hold_states)
passive_states          := c"passivate in ", state_name, " "?, "!"
hold_states             := c"hold in ", state_name, c" for time ", number,
                            " "?, "!"
state_name              := string / CURRENT_STATE / NEXT_STATE
CURRENT_STATE           := string
NEXT_STATE              := string

messages                := OUTPUT_MSG / INPUT_MSG
OUTPUT_MSG              := (number / string)+
INPUT_MSG               := (number / string)+

functions               := (int_transition / output_fnc / ext_transition)
int_transition          := c"from ", CURRENT_STATE, c" go to ", NEXT_STATE,
                            " "?, "!"
output_fnc              := c"after ", state_name, c" output ", OUTPUT_MSG,
                            " "?, "!"
ext_transition          := c"when in ", state_name, c" and receive ",
                            INPUT_MSG, c" go to ", NEXT_STATE, " "?, "!"
```

Figure 6. EBNF specification language grammar

4.2. Test Scenario Generation

The purpose of the test scenario generation is to transform the previously specification into test scenario and integrate them into the DEVS simulation models. In order to realize this transformation we have defined a Behavior Test Generator engine (called BTG in the following) which is a parser. Its goal is to transform the specification (which are expressed using the language presented in sub-section 4.1. and named "Spec "in Figure 7) into an adapted test code. Figure 7 describes how the parser is used when a user has to develop the code corresponding to a DEVS model (an atomic model in this case).

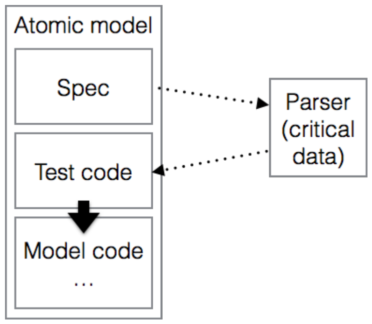
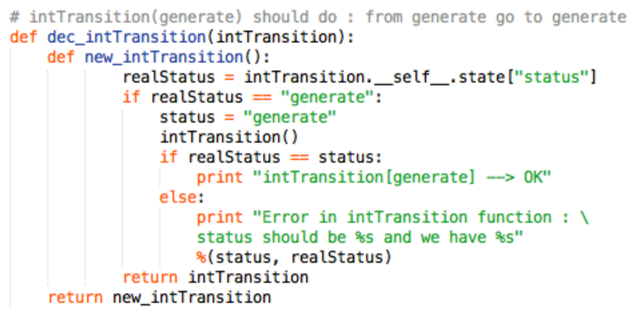


Figure 7. The parser role is to transform specification into test code.

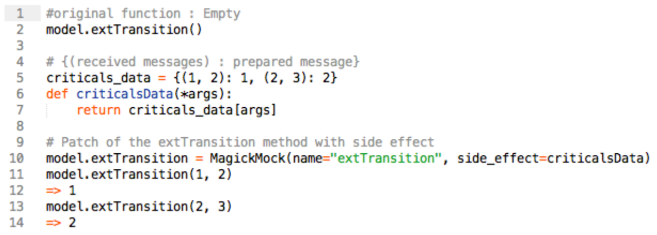
The transformation is based on important information deduced from the specification and called "critical data ". These "critical data "are determined by the BTG and injected as test code into the DEVS model to be implemented. Two cases have to be considered: (1) if the DEVS model has been already coded by the user, the injection will be performed using the software engineering decorator notion [dec]; (2) if the DEVS model has not already been implemented, the injection will be performed using the mocking objects [moc] notion. The two cases are illustrated by the following two pedagogical examples. An example of a resulting test code corresponding to case 1 is given in Figure 8. In this example the user has already written the code of an atomic DEVS model. It corresponds to the behavior which has been specified in Listing 1 (see sub-section 4.1.). The example highlights how the internal transition is modified using decorators in order to implement the test scenarios. In the presented example the test scenario which has been injected corresponds to the behavior involved by the line 3 of the Listing 1 in the generator DEVS model specification: "from generate go to generate ". This means that the injected test scenario allows to check that the state remains the same after the execution of the internal transition when the model is in the "generate "state.



```
# intTransition(generate) should do : from generate go to generate
def dec_intTransition(intTransition):
    def new_intTransition():
            realStatus = intTransition.__self__.state["status"]
            if realStatus == "generate":
                status = "generate"
                intTransition()
                if realStatus == status:
                    print "intTransition[generate] --> OK"
                else:
                    print "Error in intTransition function : \
                    status should be %s and we have %s"
                    %(status, realStatus)
            return intTransition
    return new_intTransition
```

Figure 8. FDDEVS to decorator example.

Figure 9 gives an example of the use of mocking objects which corresponds to case 2.



```
1   #original function : Empty
2   model.extTransition()
3
4   # {(received messages) : prepared message}
5   criticals_data = {(1, 2): 1, (2, 3): 2}
6   def criticalsData(*args):
7       return criticals_data[args]
8
9   # Patch of the extTransition method with side effect
10  model.extTransition = MagickMock(name="extTransition", side_effect=criticalsData)
11  model.extTransition(1, 2)
12  => 1
13  model.extTransition(2, 3)
14  => 2
```

Figure 9. Simple method patching example.

In Figure 9 we suppose that the external transition of the DEVS model to be tested has not been already implemented. From the specification we have been able to deduce the critical data which are expressed as follows: for inputs 1 and 2, the output should be 1 while for inputs 2 and 3, the outputs should be 3. The use of patches and mocking objects is highlighted by the introduction of a MagicMock object (from line 10 to 14 of Figure 9). The external transition ("extTransition "function is patched with the previously mentioned MagicMock object). This example points out how the behavior of DEVS functions can be defined using mocking objects when a DEVS model has not been totally implemented.

4.3. Test Execution

Figure 10 describes how the DEVSimPy plug-in: (i) is integrated into the simulation kernel, (ii) is able to select between the injection of decorators or patches (mocking objects).

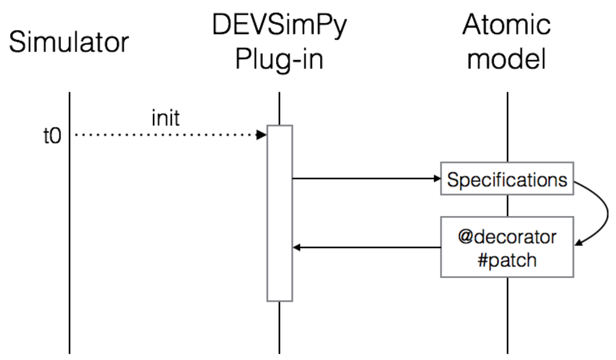


Figure 10. Sequence diagram of the DEVSimPy plug-in.

In order to select which kind of injection has to be performed during the simulation, we use an important property of the Python programming language called dynamic introspection: it is possible to know if a python object is completely implemented or not at any time of the execution of a given python script. Figure 10 details how this property is used in order to dynamically (during the simulation phase) detect if an given DEVS function has been (or has not been) already implemented. At time t0 (initialization of the simulation phase), the DEVSimPy plug-in is executed:

1. It refers to the specification associated with the DEVS model being implemented and tested.
2. It executes the BTG engine which allows to transform the previous specification into decorators or patches according to model introspection.
3. It ends by returning into the DEVSimPy simulation kernel in order to go on with the simulation.

The definition of this DEVSimPy plug-in as described above permits a user to perform design and test of DEVS models as presented in Figure 4.

5. CONCLUSION AND PERSPECTIVES

The paper introduced an approach allowing to perform the test of DEVS models at the very early phases of the Design. The main idea is to apply the concepts which have been defined in the Software Engineering domain in the framework of the Agile community. We described how the BDD (Behavior Driven Development) design and test methodology stemming from the software engineering domain can be applied to the design of DEVS models. We presented how we have adapted the three main steps of the BDD methodology: (i) definition of a semi-formal language allowing test specification using the DEVSSpecL language, (ii) generation of the test scenarios using the definition of a BTG using the notion of decorators and patches, (iii) execution of the test scenario using the concepts of dynamic introspection and mocking objects. The resulting Design and Test DEVS approach has been validated in the framework of the DEVSimPy environment. The three previous steps have been integrated into a DEVSimPy plug-in. We have validated the proposed solution on pedagogical examples which point out the feasibility of the approach. The future work will concentrate in the validation using the design of a more complex DEVS models.

REFERENCES

BDD python style. https://github.com/behave/behave.

Decorators for functions and methods. http://legacy.python.org/dev/peps/pep-0318/.

Mock - mocking and testing library. http://www.voidspace.org.uk/python/mock/index.html.

Simpleparse a parser generator for mxtexttools v2.1.0. http://simpleparse.sourceforge.net/.

J.-S. Bolduc and H. Vangheluwe. pythonDEVS : A modeling and simulation package for classical hierarchal DEVS. In *Rapport technique, MSDL, Universitè de McGill*, juin 2001.

J. H. Byun, C. B. Choi, and T. G. Kim. Verification of the DEVS model implementation using aspect embedded DEVS. In *Proceedings of the 2009 Spring Simulation Multiconference*, pages 151:1–151:7, San Diego, CA, USA, 2009. URL `http://dl.acm.org/citation.cfm?id=1639809.1655380`.

L. Capocchi, J. F. Santucci, B. Poggi, and C. Nicolai. DEVSimPy: A collaborative python software for modeling and simulation of DEVS systems. In *20th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 170–175, June. doi: 10.1109/WETICE.2011.31.

A. Cockburn. *Agile Software Development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-69969-9.

J. de Lara and H. Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In *Proceedings of FASE'02*, pages 174–188, London, UK, 2002. Springer-Verlag. ISBN 3-540-43353-8.

S. Fraser, K. Beck, B. Caputo, T. Mackinnon, J. Newkirk, and C. Poole. Test driven development (tdd). In *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering*, XP'03, pages 459–462, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40215-2. URL `http://dl.acm.org/citation.cfm?id=1763875.1763973`.

K. J. Hong and T. G. Kim. Devspecl: {DEVS} specification language for modeling, simulation and analysis of discrete event systems. *Information and Software Technology*, 48(4):221 – 234, 2006. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2005.04.008. URL `http://www.sciencedirect.com/science/article/pii/S0950584905000650`.

X. Hu, B. P. Zeigler, M. H. Hwang, and E. Mak. DEVS systems-theory framework for reusable testing of I/O behaviors in service oriented architectures. In *IRI*, pages 394–399. IEEE Systems, Man, and Cybernetics Society, 2007. URL `http://dblp.uni-trier.de/db/conf/iri/iri2007.html#HuZHM07`.

H. P. Langtangen. *Python Scripting for Computational Science (Texts in Computational Science and Engineering)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 3540294155.

X. Li, H. Vangheluwe, Y. Lei, H. Song, and W. Wang. A testing framework for devs formalism implementations. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 183–188, San Diego, CA, USA, 2011. Society for Computer Simulation International. URL `http://dl.acm.org/citation.cfm?id=2048476.2048500`.

C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383–387, Aug 2011. doi: 10.1109/SEAA.2011.76.

T. Ville. Méthodes de tests comportementaux de modèles devs et mise en oeuvre dans DEVSimPy. Technical report, SPE UMR CNRS 6134, University of Corsica, June 2013.

B. Zeigler and H. S. Sarjoughian. *Guide to Modeling and Simulation of Systems of Systems*. Springer, London, 2012. ISBN 978-0-85729-864-5. doi: 10.1007/978-0-85729-865-2.

B. P. Zeigler. *Theory of Modeling and Simulation*. Academic Press, 1976.