A Tool for the Automatic Generation of Test Cases and Oracles for Simulation Models Based on Functional Requirements

Aitor Arrieta
Electronics and Computer Science Dpt.
Mondragon University
Mondragon, Spain
aarrieta@mondragon.edu

Joseba A. Agirre
Electronics and Computer Science Dpt.
Mondragon University
Mondragon, Spain
jaagirre@mondragon.edu

Goiuria Sagardui
Electronics and Computer Science Dpt.
Mondragon University
Mondragon, Spain
gsagardui@mondragon.edu

***Abstract*—Simulation models are frequently used to model, simulate and test complex systems (e.g., Cyber-Physical Systems (CPSs)). To allow full test automation, test cases and test oracles are required. Safety standards (e.g., the ISO 26262) highly recommend that the test cases of systems like CPSs are associated to requirements. As a result, typically, test cases that need to cover specific requirements are manually generated in the context of simulation models. This is, of course, a time-consuming and non-systematic process. However, the current practice lacks tools that generate test cases by considering functional requirements for simulation-based testing. In this short paper we propose a Domain-Specific Language (DSL) for specifying requirements for simulation-based testing in an easy manner. These files are later parsed by an automatic test generation algorithm, which generates test cases that follow the ASAM-XiL standard. The tool was integrated with two professional tools: (1) SYNECT from dSPACE and (2) xMOD from FEV. An initial validation was also performed with an industrial simulation model from YASA motors.**

***Index Terms*—Simulation-based Testing, Functional Requirements, Test Case Generation**

I. Introduction

Simulation-based testing is the driving technology to verify and validate complex dynamic systems, such as Cyber-Physical Systems [1], [2]. In many domains, safety standards require test cases to be associated to requirements (e.g., the ISO 26262 standard on the automotive domain) [3]. As a result, a common practice is to derive test cases directly from requirements. This is a typically manual process, which is a tedious, time-consuming and non-systematic activity. Subsequently, automated test generation tools are required to reduce verification and validation efforts. Besides test generation, to allow full-automation, test oracles are also necessary [4], [5].

A problem with test generation based on requirements is that the latter are usually specified in natural language. However, natural language-specified requirements are usually difficult to process by machines and are often ambiguous. To solve this problem, in this paper we propose a Domain Specific Language (DSL) that enables engineers (1) to easily specify requirements for simulation-models, (2) to reduce the ambiguity of requirements and (3) to easily parse requirements for automated test generation. For this last point, we adapted the well-known Adaptive Random Testing (ART) algorithm to this context for the purpose of generating requirements-specific test cases [6].

The main contributions of this paper include (1) a novel language that enables to specify requirements for simulation-based testing and (2) the adaptation of the ART algorithm [6] for generating test cases for simulation-based testing following requirements. These scientific contributions are complemented with the following advances in the state-of-the-practice: (1) integration with two professional tools (SYNECT [7], a data management and collaboration software for automated testing from dSPACE and xMOD [8], a co-simulation tool from FEV) and (2) generation of ASAM-XiL compliant test cases, which allows for the re-use of test cases at different test levels and with different simulation tools.

The rest of the paper is structured as follows: Background of the context is given in Section II. The DSL for test generation is presented in Section III. The tool support and the test generation workflow is explained in Section IV. We present the performed preliminary evaluation in Section V. We position our tool with other approaches in Section VI. Lastly, we conclude the paper and summarize future work in Section VII.

II. Background

Developers of CPSs rely on Model-Based Design workflows and simulation environments, where graphical models are used to test the systems at design-time [9]. Simulation tools (e.g., Simulink) are data-flow models, where each model contains a set of blocks that accept data through their inputs and may pass output through their output ports after performing a set of operations (e.g., mathematical or logical) [9]. To allow for a hierarchical organization, these models can be structured into several subsystems. Consider as an example the simulation-model depicted in Figure 1. This model includes six inputs (enable, brake, set, speed, inc and dec) and two outputs (throt and target), and is structured into two hierarchical levels, permitting engineers to organize their models [10]. Each input and output of the simulation-model is a signal (i.e., a function of time), which is stored as a vector where elements are indexed by time [2]. In the context of simulation-based testing, a test case is a signal that stimulates the highest level subsystem of the System Under Test (SUT) (i.e., for the case of the example shown in Figure 1, six signals stimulating the input of the subsystem named as "Controller").

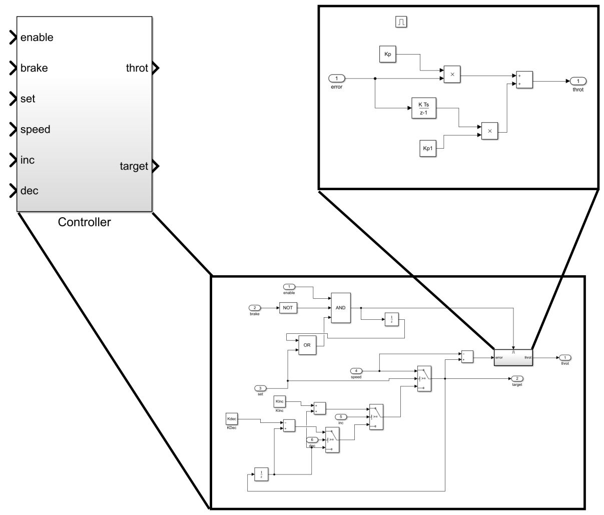


Fig. 1. Example of a simulation-model of a Cruise Controller of a car

Based on our discussions with industrial partners, two aspects should be considered when generating test oracles for simulation-based testing. Firstly, the transitory regime that dynamic models are exposed to. For some systems/subsystems (e.g., electrical engines), some signals are exposed to a transitory regime that is difficult to predict. For the industrial example used in this paper, consider the graph shown in Figure 2. As can be seen, this output signal has a transitory regime of around 0.1 seconds. After this time, the signal is stabilized. The transitory regime of simulation-models corresponds to the non-functional part of the simulation-model and thus, it is normally not considered by engineers when manually evaluating the functional requirements.

The second aspect when generating test oracles corresponds to the tolerance (i.e., bounds) given by the engineers to determine whether a signal meets certain requirements. This is because usually the simulation-models are modeled with complex mathematical models and it is difficult to predict the specific trajectory that the signal should have. To cope with this issue, test oracles must consider certain tolerance, which we solved by using a tolerance tube (i.e., certain bounds are given to a reference signal), as shown in Figure 3.

III. DSL for the Generation of Test Cases

We now provide an overview of the tool for generating test cases. Implementation and integration details, along with the detailed description of the DSL syntax is provided in a technical report [11] to keep the paper at a reasonable size.

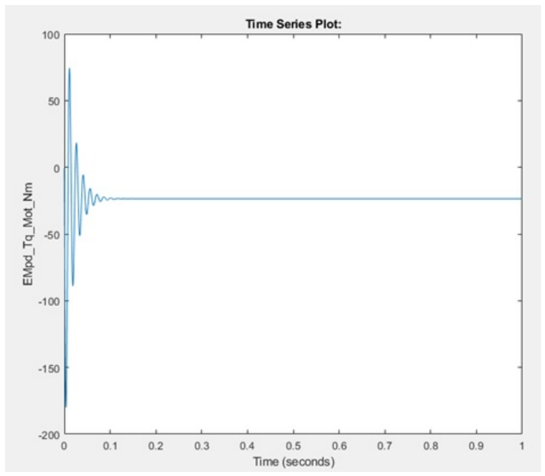


Fig. 2. Output signal example for the industrial case study used in this paper for a given test case generated by our tool

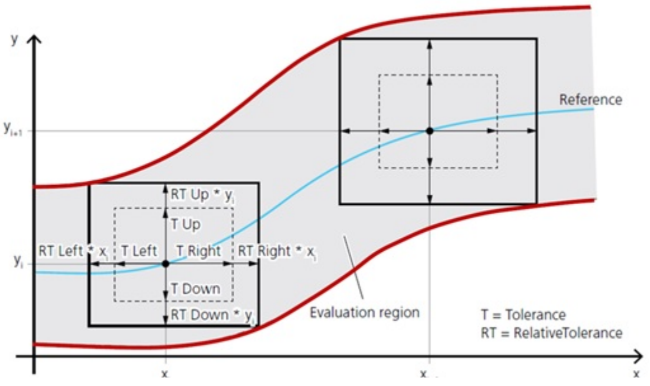


Fig. 3. Sample of a reference signal and its corresponding tolerance tube signals (figure courtesy of dSPACE)

A. Syntax of the DSL

A DSL is a programming language in charge of solving a particular problem of a specific domain. In this paper we propose a DSL for specifying system properties (such as the interface of the simulation-models and their requirements) to automatically generate test cases and oracles. The developed DSL supports all the characteristics for simulation-based testing explained in the Background section.

The DSL has been developed in Xtext [12] as it permits creating a language and generating a syntactic analyzer, an abstract model of classes for the syntax tree. Another advantage for Xtext is that it permits the development in the Eclipse environment, which eases the integration with other tools. Furthermore, this environment permits making small corrections and providing the end-user with the appropriate guidance not to introduce syntax errors in their files. The developed DSL has two main structures. The first one is intended to characterize the model (e.g., the interfaces and its parameters) of the models to be tested, whereas the second part is intended to specify functional requirements of the models. The second part makes use of the first part, which means that in the workflow, it is mandatory to first characterize the interface of the simulation-model under test.

1) Model characterization: This involves the characterization of the inputs, outputs and parameters of the SUT. This file is automatically generated from SYNECT, although some manual changes might be required depending on the SUT. When characterizing the inputs of a system, the following properties need to be specified: (1) name of the input signal, (2) datatype, (3) signal pattern (i.e., constant, PWM, sine, etc.), (4) unit, (5) maximum value and (6) minimum value (when the datatype is not boolean). All this information, with the exception of the unit, is required by either the functional requirement specification or the test generator. For the signal pattern, we have followed the signals specified in the ASAM-XiL standard. This is important as it helps delimit the search space when generating test cases.

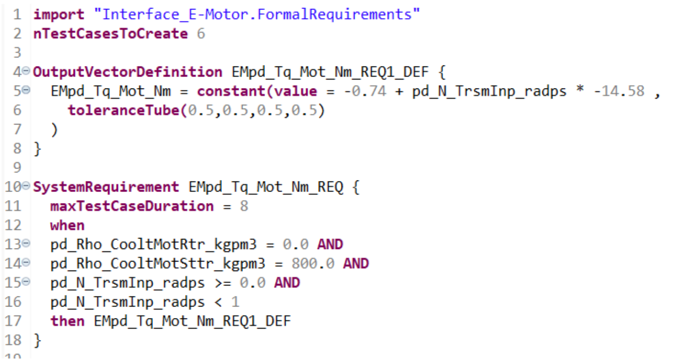
With regards to the outputs, the following properties are mandatory: (1) name of the output signal, (2) datatype and (3) unit. Besides this information, an optional property is the stabilization time, which can be set to those output signals that show a transitory time behavior (similar as shown in Figure 2). This information is processed by the test oracle generator. Other properties (e.g., signal pattern or maximum/minimum values) are not necessary when characterizing the outputs.

As an optional property, the interface file also allows to specify parameters. The name of the parameter is set and a default value is given. The specified parameters in this section can later be used to specify the functional requirements. This can be interesting, for instance, to model different parameters within PID controllers (e.g., a higher Kp will mean a higher ramp, etc.).

2) Functional requirements specification: After characterizing the model in the interfaces files, requirements need to be specified with our DSL. The idea is to formally specify which the signal behavior should be for certain inputs. To this end, each requirement file has two properties, as depicted in Figure 4: (1) system requirement and (2) output vector definition. The former specifies potential conditions that are required to be given to cover a requirement (i.e., the test stimulation). The latter specifies the expected output (i.e., the test oracle) for the given conditions in the system requirement part as well as the parameters. In addition, the tolerance tube for a given signal is specified in the output definition part, as can be seen in Figure 4. This is the information used to generate the test oracles.

B. Automated Test Generator

Along with the DSL we have developed an automated test generator. This test generator generates a set of test cases for each of the requirements specified within the DSL. The implemented test generator is a novel algorithm based on the Adaptive Random Testing (ART) test generation algorithm [6]. The ART algorithm relies on the hypothesis that the more diverse the test data is, the higher the probability of detecting faults. We selected ART because its implementation is simple, it is a relatively fast algorithm, it is intended for black-box testing and its focus is on detecting faults. Nevertheless, it has some limitations that we have addressed in our tool.



```
import "Interface_E-Motor.FormalRequirements"
nTestCasesToCreate 6

OutputVectorDefinition EMpd_Tq_Mot_Nm_REQ1_DEF {
  EMpd_Tq_Mot_Nm = constant(value = -0.74 + pd_N_TrsmInp_radps * -14.58 ,
    toleranceTube(0.5,0.5,0.5,0.5)
  )
}

SystemRequirement EMpd_Tq_Mot_Nm_REQ {
  maxTestCaseDuration = 8
  when
  pd_Rho_CooltMotRtr_kgpm3 = 0.0 AND
  pd_Rho_CooltMotSttr_kgpm3 = 800.0 AND
  pd_N_TrsmInp_radps >= 0.0 AND
  pd_N_TrsmInp_radps < 1
  then EMpd_Tq_Mot_Nm_REQ1_DEF
}
```

Fig. 4. Sample of a functional requirement file specified with the DSL

The first limitation is that ART is designed for unit testing of code rather than simulation-based testing. To cope with this limitation, the algorithm we have developed has two main features. The first feature includes a parser to determine the signal pattern of each input, which enables later to generate test cases with these patterns. The second feature consists on a novel heuristic to measure the distance among test cases, which was based on the well-known Hamming Distance. However, we adapted it to the context of signal-based testing, by considering different features (i.e., not only the amplitude of the signals but also frequencies, etc.), depending on the type of signals that the test generator aims to generate.

The second limitation is that ART does not consider requirements. This means that if pure random is used to generate the candidate set of test cases, there is no control over the covered and uncovered requirements. To overcome this issue, we have opted to individually generate a set of test cases for each requirement. A parser has been implemented to extract which are the specifications provided by the requirements in order to set certain constraints when generating the set of candidates. After the algorithm generates the test cases, these are converted into a format that follows the ASAM-XiL standard and stored with a *.sti extension, which is a simple XML file.

IV. Tool Support and Work-flow

The DSL has been integrated with two professional tools: SYNECT, a tool from dSPACE intended to manage simulation-models data for automated testing, and xMOD, a co-simulation tool from FEV that allows for the execution of test cases. The integration with SYNECT has been performed through Python scripts, whereas the integration with xMOD has been done via the XiL-API that follows the ASAM-XiL standard.

The test generation and execution workflow has five steps. In the **first step**, a model to be tested is selected in SYNECT from a specific workspace. In SYNECT, the model can be characterized, which later allows for the automated generation of the interface file explained in Section III-A1. The **second step** is to edit the requirements with our DSL, which is opened by triggering it through SYNECT. When our tool is opened, it is possible to edit and/or create functional requirements for the selected simulation-model. When this process has been finished, the editor is closed and the changes can be saved in SYNECT. The **third step** is to generate test cases. The test generator explained in Section III-B is triggered through SYNECT, test cases generated and stored back in SYNECT. The **fourth step** corresponds to the test execution. In SYNECT it is possible to select which test cases need to be executed and when. When this is selected, the tests are executed, where, for each test to be executed the XIL-API is invoked and the tests executed through xMOD. When the test execution has finished, the simulation results files are obtained (which have an *.mdf format), and evaluated through our oracles, which determine whether the test has *passed* or *failed*. The **last step** is to store all the results in SYNECT. The overview of all these steps is available in [11].

V. Preliminary Evaluation

A preliminary validation was performed with an industrial case study from YASA. The first objective was to see whether the syntax of the DSL had sufficient expressiveness to specify functional requirements of their simulation-model. They provided us with three requirements of their model specified in natural language and later written in our DSL. The results showed that the proposed DSL had sufficient expressiveness to write the specified model's requirements. This means that for a real-world industrial simulation-model, the proposed syntax is able to specify its requirements to later generate test cases and oracles.

After writing the requirements with our DSL, we wanted to know the practicality of the tool when generating test cases. To this end, we specified different number of test cases and oracles to be generated (i.e., from 15 to 300 test cases in total), and measured the time the tool took to generate test cases. Notice that the test generation time was triggered when the user orders the generation of test cases through SYNECT, and it was stopped when test cases were stored in SYNECT. This means that the overhead introduced by SYNECT was considered besides the time our algorithm took to generate test cases. As expected, the highest test generation time was when setting the tool to generate 300 test cases. This was, however, less than 40 seconds, a time that is, by far, affordable by engineers.

VI. Related Work

In the last few years, several studies have considered test case generation for simulation-based testing [2], [13]–[16]. Most of these studies follow search techniques when generating test cases. Despite some of them considering functional requirements for test generation (e.g., [13], [14]), none of them specify these requirements with a specific language for simulation-based testing. In addition, in contrast to these studies, the method proposed in this paper has been integrated with two professional tools. Other approaches focus on the generation of test cases based on falsification [17]–[20]. These approaches, however, require the execution of simulation-models, which might enlarge the test generation time, as opposed to our tool, which generates test cases within seconds.

In the context of test oracle generation for simulation-based testing there is much less work as compared with test case generation [21]. Our previous tool generated test systems for simulation-based testing of configurable CPS [22], which included test cases and oracles. However, this tool focused on the test system, which means that test cases and oracles needed to be generated beforehand following any state-of-the-art technique. A recent study proposed a DSL for the generation of test oracles for Simulink models, which is quite similar to what we propose in this paper [21]. The main difference between the study proposed by Menghi et al. and our approach is that they solely focus on test oracle generation whereas we focus on both, test case and test oracle generation. As for the test oracle generation, there are still some differences between the work presented in [21] and our work. One such difference is that their approach evaluates test cases on-line whereas our approach does it off-line. We used standards for doing this, such as the used *.mdf files generated by xMOD, which eases our oracles to be used by other simulation tools that generate these files. Another difference is that we consider aspects like the tolerance a signal can have or the transitory regime. On the contrary, we do not consider aspects like the degree of satisfaction or violation, which is considered in [21]. Nevertheless, although this could be easily included in our oracles by measuring how far a signal is from being out of the tolerance tube, SYNECT is not prepared for obtaining a quantitative degree and solely focuses on verdicts (i.e., PASS, FAIL, INCONCLUSIVE, etc.).

VII. Conclusion and Future Work

In this paper we have proposed a tool based on a DSL to automatically generate test cases and test oracles of simulation-models. A preliminary evaluation was performed with an industrial case study from YASA, showing the feasibility of the tool both to specify requirements and generate test cases and oracles. The future steps will have as main objective a more comprehensive evaluation with other industrial case studies. To this end, at the current stage, the tool has been transferred to other industrial partners (e.g., Idiada, Iveco, Ricardo, etc), and we are planning to get feedback from them with regards to the efficiency and usability of the tool.

Acknowledgment

The work of this paper has been done inside the HIFI-ELEMENTS project (www.hifi-elements.eu). This project received funding from the European Union's Horizon 2020 research and innovation program under Grant Agreement no. 769935.

References

[1] L. Briand, S. Nejati, M. Sabetzadeh, and D. Bianculli, "Testing the untestable: Model testing of complex software-intensive systems," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. ACM, 2016, pp. 789–792.

[2] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, “Automated test suite generation for time-continuous simulink models,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: ACM, 2016, pp. 595–606.

[3] A. Mjeda and M. Hinchey, “Requirement-centric reactive testing for safety-related automotive software,” in *2015 IEEE/ACM 2nd International Workshop on Requirements Engineering and Testing*, Florence, Italy, 2015.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[5] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, “A survey on metamorphic testing,” *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[6] T. Y. Chen, H. Leung, and I. Mak, “Adaptive random testing.” in *ASIAN*, vol. 4. Springer, 2004, pp. 320–329.

[7] dSPACE. (2020) SYNECT. [Online]. Available: https://www.dspace.com/en/inc/home/products/sw/datenmanagement/synect.cfm

[8] FEV. (2020) xMOD - Co-Simulation Tool. [Online]. Available: https://xmod.fev.com/

[9] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner, “Automatically finding bugs in a commercial cyber-physical system development tool chain with SLforge,” in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 981–992.

[10] A. Arrieta, S. Wang, U. Markiegi, A. Arruabarrena, L. Etxeberria, and G. Sagardui, “Pareto efficient multi-objective black-box test case selection for simulation-based testing,” *Information & Software Technology*, vol. 114, pp. 137–154, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2019.06.009

[11] A. Arrieta, “Technical report of: “a tool for the automatic generation of test cases and oracles for simulation models based on functional requirements”,” Mondragon University, Tech. Rep., 2020. [Online]. Available: https://tinyurl.com/Arrieta-AMOST2020

[12] M. Eysholdt and H. Behrens, “Xtext: implement your language faster than the quick and dirty way,” in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2010, pp. 307–309.

[13] A. Arrieta, S. Wang, U. Markiegi, G. Sagardui, and L. Etxeberria, “Search-based test case generation for cyber-physical systems,” in *Evolutionary Computation (CEC), 2017 IEEE Congress on*, 2017, pp. 688–697.

[14] ——, “Employing multi-objective search to enhance reactive test case generation and prioritization for testing industrial cyber-physical systems,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 3, pp. 1055–1066, 2018.

[15] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, “Test generation and test prioritization for simulink models with dynamic behavior,” *IEEE Trans. Software Eng.*, vol. 45, no. 9, pp. 919–944, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2018.2811489

[16] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, “Testing vision-based control systems using learnable evolutionary algorithms,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18, 2018.

[17] S. Nejati, K. Gaaloul, C. Menghi, L. C. Briand, S. Foster, and D. Wolfe, “Evaluating model testing and model checking for finding requirements violations in simulink models,” *arXiv preprint arXiv:1905.03490*, 2019.

[18] C. Menghi, S. Nejati, L. C. Briand, and Y. I. Parache, “Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification,” *arXiv preprint arXiv:1910.02837*, 2019.

[19] A. Donzé, “Breach, a toolbox for verification and parameter synthesis of hybrid systems,” in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 167–170.

[20] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, “S-taliro: A tool for temporal logic falsification for hybrid systems,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 254–257.

[21] C. Menghi, S. Nejati, K. Gaaloul, and L. C. Briand, “Generating automated and online test oracles for simulink models with continuous and uncertain behaviors,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338920

[22] A. Arrieta, G. Sagardui, L. Etxeberria, and J. Zander, “Automatic generation of test system instances for configurable cyber-physical systems,” *Software Quality Journal*, vol. 25, no. 3, pp. 1041–1083, 2017.