Mondragon Unibertsitatea Biblioteka Biblioteca

biblioteka@mondragon.edu

This is an Accepted Manuscript version of the following article, accepted for publication in:

G. Sagardui, L. Etxeberria, J. A. Agirre, A. Arrieta, C. F. Nicolás and J. M. Martín, "A Configurable Validation Environment for Refactored Embedded Software: An Application to the Vertical Transport Domain," 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2017, pp. 16-19.

DOI: https://doi.org/10.1109/ISSREW.2017.9

© 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

A Configurable Validation Environment for Refactored Embedded Software: an Application to the Vertical Transport Domain

Goiuria Sagardui, Leire Etxeberria,
Joseba A. Agirre, Aitor Arrieta
Mondragon Unibertsitatea
Mondragon, Spain
Email: {gsagardui, letxberria, jaagirre, aarrieta}@mondragon.edu

Carlos Fernando Nicolás
Ik4-Ikerlan
Mondragon, Spain
Email: cfnicolas@ikerlan.es

Jose María Martín
Orona EIC
Hernani, Spain
Email: jmmartinc@orona-group.com

***Abstract*—As systems evolve, their embedded software needs constantly to be refactored. Moreover, given the different needs of different customers, embedded systems require to be customizable. The variability of these systems is large, and requires automated testing solutions. In this paper we propose a methodology that automatically generates validation environments for highly configurable embedded software that is being refactored. The method has allowed for systematically testing a real-world industrial case study involving the software in charge of controlling the doors of an elevator. Finally, we extract the lessons learned from its application.**

I. Introduction

Embedded systems are engineering artifacts involving computations that are subject to physical constraints. The physical constraints arise through two kinds of interactions of computational processes with the physical world: (i) reaction to a physical environment, and (ii) execution on a physical platform [1]. The software of these systems is constantly evolving, which makes maintaining it a time-consuming, error-prone and cost-intensive activity [2]. Moreover, embedded systems are becoming highly configurable, and their software has to deal with many parameters and configurations. Testing these systems is becoming a complex endeavor, which requires automated solutions to efficiently test and validate the different versions of the software programs.

Embedded systems in the vertical transport domain are subject to several factors that make the maintenance of the software specifically challenging. For instance, many legislation changes are enacted over the years, what requires the software to be constantly changed. Moreover, in each of country where an elevator has to be installed, the software in charge of controlling it must be compliant with the standard required in that country. In addition, in order to be competitive in the market, new elevator functionalities are required to be integrated within the software. Furthermore, given the fast technological evolution, the software has to be constantly adapted in order to be deployed in new technological platforms (e.g., new processors with higher computation capabilities).

Apart from all these factors, the variability of embedded systems in the vertical transport domain is huge. Notice that each elevator installation is different from one another depending on the number of floors of the building, number of doors that each elevator has, types of sensors, mechanical elements, etc. For this reason, the embedded software in charge of controlling the elevators contains a single embedded code base which is later configured and parameterized to be deployed in a specific installation.

In this paper we present an approach for testing refactored code that is highly configurable. Firstly, Model-Based Testing (MBT) is employed to automatically generate test cases [3]. In addition, feature models [4] are employed to manage the variability of the systems and to derive configurations. The test cases are executed employing simulation-based testing. Specifically, we automatically generate the simulation environment in MATLAB/Simulink for each of the configurations of the refactored code that needs to be tested. Moreover, the original software code is employed and used as a test oracle, which permits the automatic evaluation of the executed test cases.

The proposed approach was applied in the vertical transport domain. As a case study, we employed the refactorized code for the embedded software in charge of controlling the horizontal movements of the elevators (i.e., the elevator doors) from Orona, the leading elevator company in Spain. The refactorized code permitted reducing the complexity of the software, raising the abstraction level to ease the integration of new functionalities and improving the understandability as well as the maintainability of the code. The proposed approach permitted automatically testing the configurable refactored software code in a systematic manner.

The rest of the paper is structured as follows: Section II presents some background related to this study. Section III presents the proposed approach for testing variability-intensive and refactored embedded software. Section IV outlines the lessons learned from applying our approach in an industrial context. Finally, conclusions and future work are outlined in Section V.

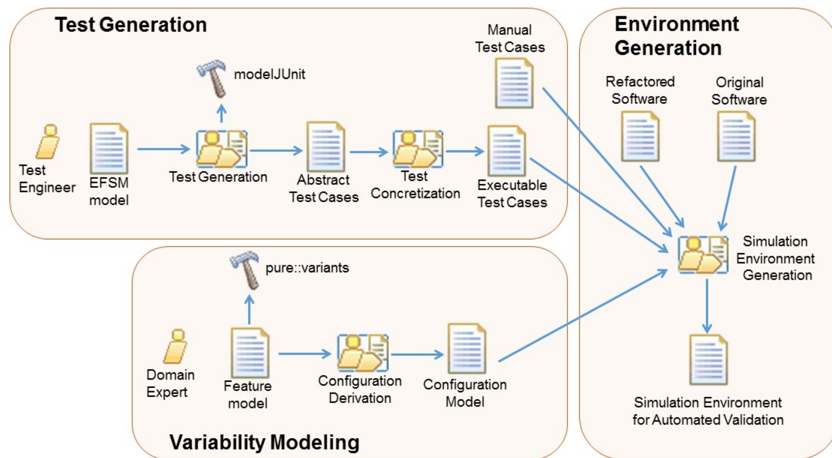

Fig. 1: General overview of the process for the generation of the simulation environment for the automated validation

II. Background

A. Simulation-Based Testing

Embedded systems are usually integrated with hardware components (such as electrical or mechanical actuators) that are typically complex or expensive, and the use of a real prototype is often too costly. For that reason, simulation is employed for testing embedded software, permitting several advantages such as (1) execution of larger test cases, (2) selection of critical scenarios, (3) specification of test oracles for the automated validation of the system and (4) replication of safety-critical functions (e.g., free fall of an elevator) [5][6]. Moreover, simulation-based testing permits testing the software at different test levels depending on the validation stage of the product (i.e., Model-, Software-, Processor and Hardware-in-the-Loop [7]).

B. Product Line Engineering

A product line is a set of related products sharing commonalities and variabilities, where a family of software products are reused to satisfy the customer's needs instead of developing each product individually [8]. Feature models are a hierarchical representation of the information of all possible products of a product line [8]. These models have been cataloged as the most used variability modeling notation in industry to manage variability of product lines[9].

A basic feature model is composed of various features and cross-tree constraints. These features can be cataloged in a feature model as "Mandatory" (when a feature appears in all products), "Optional" (when a feature can be optional in a product), "Alternative" (when only one feature among a set of child features can be selected) or "Or" (when one or more features among a set of child features can be selected). A feature model can also contain cross-tree constraints: the "Require" constraint (when a feature requires another feature in the product) and the "Exclude" constraint (when two features cannot be integrated in the same product).

III. Approach

Figure 1 depicts the overview of the different steps for the automatic generation of the simulation environment for the automated validation of refactored software. Specifically, it shows the Software Process Engineering Metamodel (SPEM), with three different stages. The first stage corresponds to the generation of test cases by using MBT. The second stage corresponds to variability modeling. The last stage corresponds to the automatic generation of the validation environment.

A. Systematic Generation of Test Cases

The systematic generation of test cases was achieved by employing the widely used MBT technique. MBT employs explicit behavior models that encode the intended behavior of a system, which is later used to automatically generate test cases [3]. Based on our needs, we selected the tool ModelJUnit [3]. ModelJUnit is an open source MBT tool that allowed us to use time annotations, transition-based notation as well as structural coverage metrics for test case generation. Moreover, it supports on-line as well as off-line test execution. In our case off-line test execution was required since tests are executed after generation employing simulation.

Three main steps were employed to generate test cases for testing the refactored embedded software employing simulation. The first step consisted of the elaboration of an Extended Finite State Machine (EFSM) model, that modeled the behavior of the System Under Test (SUT). The EFSM considered timing aspects, and thus, some timeouts were required to be specified in the transitions of the model. The second step consisted of the generation of the abstract test cases. To this end, the greedyTester algorithm provided by the ModelJUnit tool was employed to generate test cases that maximized the transition coverage. The last step consisted of the concretization of the abstract test cases in order for them to be executable into the Simulink models. To concretize these test cases, we implemented a test concretization program that obtained the abstract test cases provided by ModelJUnit, and it transformed them into MATLAB signals which were then directly executed into Simulink models. In our case, the

greedyTester algorithm was able to generate 47 test cases taking the transition coverage as a generation criteria.

Apart from the automatically generated test cases, domain knowledge is also employed to to manually generate realistic test cases. These test cases serve as complement of the automatically generated test cases and simulate realistic situations of elevator, which are also important to execute.

B. Variability Management

To manage the high variability of the system, feature models have been employed. The feature model includes all the different elements that should be considered when validating the control system and the rest of the environment (i.e., the different sensors, horizontal movement elements (i.e., doors), etc.). The most relevant elements include the following:

- Different types of access operators: the software must be tested with different types of horizontal movement elements, where each of them has different access operators.
- Different number of horizontal movement elements: from one to three horizontal movement elements that can operate independently or not.
- Different floor configurations: each floor can have different horizontal movement element configurations. Based on the floor the elevator is on, the horizontal movement elements can require a different behavior.

From the feature model a configuration model that includes all the features related to the SUT. Figure 2 depicts the part of the developed feature model for the case study.

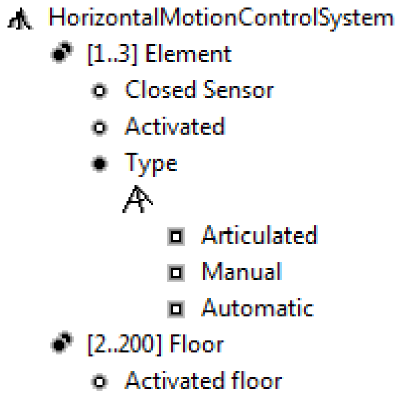

Fig. 2: Part of the developed feature model for the case study

C. Generation of the Simulation Environment

Since the refactored embedded software as well as its environment is highly exposed to variability, a configurable simulation environment was required. The idea behind this simulation environment was to allow the simulation environment to be automatically configured in order to test specific configurations of an installation related to an elevator horizontal movements.

To generate the simulation environment, a simulation environment generator was developed in MATLAB. This generator obtained (1) the original software, (2) the refactored code, (3) a configuration file derived from the feature model and (4) the executable test cases as inputs. The simulation environment generator works as follows. First, it parses the configuration model, which is an *.xml file. With this file, both, the refactored and the original software are configured. After the software is configured, the simulation environment generator creates a new Simulink file and both software codes are allocated in the form of S-Functions (i.e., executable code). Additionally, the environment generator allocates the required physical elements and it configures them based on the configuration file. At this point all the required electrical and mechanical elements are configured. As a last step, the executable test cases are obtained, their variability is pruned and they are allocated and integrated with the rest of the sources in the Simulink file. Once all these steps are undertaken, the model is ready to run and execute all the test cases. A sample model of the generated simulation environment for a two door configuration is shown in Figure 3.

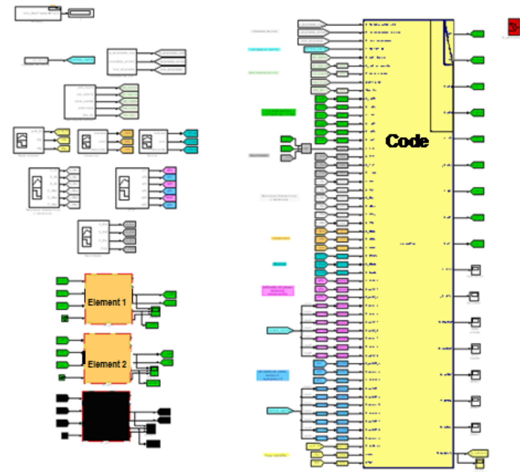

Fig. 3: Sample of the different elements on a generated simulation environment for a two door configuration

D. Automated Evaluation of Test Cases

Since the original software was thoroughly tested and even deployed in elevators, we assumed that the software was error free. In our case, we took advantage of the availability of the original software to automatically evaluate the refactored software by stimulating the inputs of both systems with the same test cases and comparing the outputs of both. Hence, both software units were simulated at the same time using Simulink and results were compared. When a test case was launched and the outputs of both systems did not match, domain experts analyzed the failing test case. Figure 4 shows the position of the two doors (0 means completely closed and 1 means completely opened), where the blue line indicates the position of the door controlled by the original code and the red one indicates the position of the door controlled by the refactored code. For the last part of the test case, it can be appreciated that the positions of the doors controlled by the refactored software do not match the positions of the doors controlled by the original software. This might be either because the refactored software is wrong, or because the new functionality indicates that the door should be opened in that case.

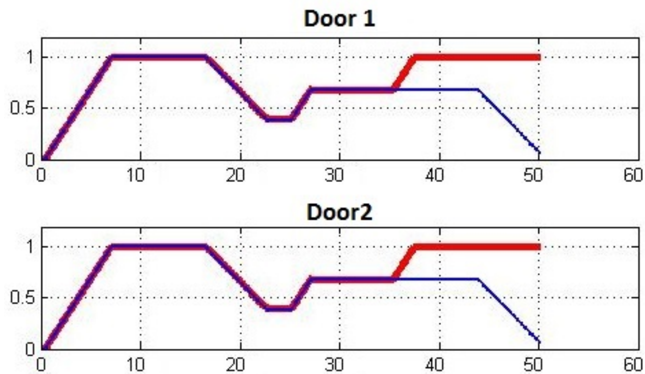

Fig. 4: Simulated position of the installation of two doors under the control of the refactored and the original code

IV. Lessons Learned

In this section we present the lessons learned when testing the refactored embedded software for the horizontal controller of an elevator. These lessons can be used as guidelines for practitioners from different domains but who are addressing similar problems.

A. Importance of Simulation-Based Testing

The deployment of the software in real systems is extremely expensive, especially in the vertical transport domain. Simulation permits several advantages when testing re-factored embedded code. First, it permits the execution of long test suites. Second, it allows testing hundreds to thousands of product configurations (involving different features, different parameters, etc.). Third, it eases the automated validation unlike with a real system. Lastly, it is possible to test safety-critical functionalities by using fault injection and other techniques without risks or damages (either economical or human-based).

B. Usefulness of the Original Software for the Automated Validation

Typically, the original (non-refactored) code has been throughly tested, and in many cases, it has been on production for a long time, as it was in our case. This means that most or all the errors have been corrected, and thus, it can be assumed it behaves correctly. This permits the comparison of the refactored code with the original code, which allows for the automated validation. It is possible to compare the outputs of both systems and inform domain experts if results are not the same.

C. Importance of Variability Management

The use of a variability management tool, such as pure::variants or FeatureIDE, from the early refactoring stages permits several advantages. One such advantage is the systematic and automatic generation of the validation environment. Another important advantage is the systematic derivation of the different configurations to test. This can be performed by employing different product generation criteria investigated by the product line engineering community. Another advantage we found when managing the variability is the possibility of incorporating variability in test cases.

D. Effectiveness of ModelJUnit for Test Case Generation

The generation of the test cases was performed by employing the MBT tool ModelJUnit and later executed in Simulink. To make the generated test cases executable in Simulink, we developed a test concretization program. Despite being an open source tool, the effectiveness of ModelJUnit combined with the execution of test cases in Simulink was high, permitting the systematic and automated generation of test cases with high transition coverage. Moreover, the use of MBT by means of ModelJUnit reduced development time and cost. Apart from ModelJUnit, automatically generated test cases were combined with manually generated ones to test realistic scenarios. A drawback of MBT is that in complex systems (e.g., Cyber-Physical Systems), it might be difficult to capture complex continuous dynamics and interactions between the system and its environment [5].

V. Conclusion and Future Work

This paper proposes a method for testing refactored embedded software that is subject to variability employing MBT and simulation-based testing. The approach was employed to test the refactored embedded software of an industrial case study involving the doors control of elevators. In the future, we foresee to integrate more features to the tool, such as automatic fault injection to test safety-critical functions of the system.

Acknowledgment

This work has been developed by the embedded systems group of Mondragon Unibertsitatea supported by the Department of Education, Universities and Research of the Basque Government.

References

[1] T. A. Henzinger and J. Sifakis, "The Embedded Systems Design Challenge," in *14th International Symposium on Formal Methods(FM 2006), Hamilton, Canada*, ser. Lecture Notes in Computer Science, vol. 4085. Springer, 2006, pp. 1–15.

[2] M. Lindvall, S. Komi-Sirviö, P. Costa, and C. Seaman, "Embedded software maintenance," Tech. Rep., 2003.

[3] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[4] K. Lee, K. C. Kang, and J. Lee, "Concepts and Guidelines of Feature Modeling for Product Line Software Engineering," in *7th International Conference on Software Reuse: Methods, Techniques, and Tools (ICSR 2002), Austin, TX, USA*, ser. Lecture Notes in Computer Science, vol. 2319. Springer, 2002, pp. 62–77.

[5] 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. New York, NY, USA: ACM, 2016, pp. 789–792.

[6] 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.

[7] H. Shokry and M. Hinchey, "Model-based verification of embedded software," *Computer*, vol. 42, no. 4, pp. 53 – 59, 2009.

[8] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615 – 636, 2010.

[9] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *Variability Modelling of Software-intensive Systems (VaMoS)*, 2013, pp. 7:1–7:8.