Variability Management in Testing Architectures for Embedded Control Systems

Goiuria Sagardui, Leire Etxeberria and Joseba A. Agirre
Computer and Electronics department
Mondragon Goi Eskola Politeknikoa
Loramendi 4, Mondragón (Gipuzkoa), Spain
Email: {gsagardui, letxeberria, jaagirre}@mondragon.edu

Abstract— **In recent years, embedded systems have substantially increased their presence both in industry and in our everyday lives. Hence, more and more effort is being dedicated to the development of such systems. Since embedded systems involve computation that is subject to physical constraints, the development and validation of software for such systems becomes a challenge. Moreover, the validation of the embedded system within the environment increases the complexity and cost of testing, so many efforts are being devoted to perform testing activities from early phases of the development. Testing by simulation of the system and its environment is one of the most promising approaches to reduce testing costs. In this paper, we present a proposal based on model-based testing and variability management and integrated in Simulink for ensuring the correctness of a embedded control software. Variability management of configurations helps managing different simulation environments and allows less costly and time-consuming testing.**

Keywords - testing architecture; variability management; simulation.

I. Introduction

Embedded systems are engineering artifacts involving computation that is 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]. Concentrating on software, embedded system software characterizes itself, among others, by heterogeneity, distribution (on potential multiple and heterogeneous hardware resources), ability to react (supervision, user interfaces modes), criticality, real-time and consumption constraints [2]. The need to consider all these factors in concert makes the development of software for embedded systems a complex endeavour.

However, not only development poses a significant challenge. Due to its complexity, the validation of embedded software also becomes a cumbersome task. Embedded software needs to cater for the variability on both the physical environment and the physical platform it is executed on apart from testing the software itself.

Moreover, when we consider that embedded systems are often part of safety-critical systems (e.g., aviation or railway systems), the validation of the software becomes essential [3], which also raises testing cost.

Model Driven Engineering (MDE) is a paradigm that promises a reduction in testing efforts. Models become the central asset of the development so testing can be started from early phases. Model-, software-, processor-, and hardware-in-the-loop (MiL, SiL, PiL, and HiL) tests; called X-in-the-loop tests provide four testing configurations [4]. “The model, software, processor, and hardware terms refer to the different target system configurations in the testing environment, each of which adds value to the verification process” [4].

The MiL tests the model along with the plant model that simulates the physical environment signals. For SiL testing, the model of the MiL is replaced with the corresponding software code. This source code can be autogenerated from the model. PiL tests the source code executed on the target processor machine. For HiL testing, the software is integrated with the real software infrastructure and deployed in the hardware processor or microcontroller. The environment around the system is still a simulated one, but the plan model is replaced by a dedicated hardware setup specially designed for the simulation [4].

Each of the configurations has a different focus from the validation point of view and following them allows detecting errors early when they are easier to correct and to validate incrementally different aspects of the system (functionality, performance, etc.). Functionality and system behavior can be tested at MiL and SiL level. Tests on PiL level can reveal faults that are caused by the target compiler or by the processor architecture [5]. HiL level is to reveal faults in the low-level services and in the I/O services [5]; and to confirm the real-time functionality and performance [4].

Embedded software for control systems usually has to run in different environment conditions, and has to control different number or/and types of sensors and actuators. This increases the complexity of testing even in early phases of the development. Testing the control system in different real scenarios is very costly and time-consuming.

Taking into account variability in different aspects of the validation from early testing architectures allows reducing the testing complexity by considering all the possible variants both in software, tests and the environment from simulation. This ensures an increased coverage of the testing in early phases of the development and a correct selection of the most risky scenarios for testing the final system.

This paper proposes a systematic approach to X-in-the-Loop validation considering the variability in the testing architecture. The proposed variability management can be reused along the testing process (MiL, SiL and PiL).

Simulink [6] was chosen as the simulation framework to simulate the real environment in which software should be integrated.

The paper is structured in the following way: Section II presents the background and the state of the art, Section III discusses about variability in testing architecture, Section IV presents the variable simulation model and, to finish, conclusion and future work are stated in Section V.

II. BACKGROUND

This section provides a brief introduction to the background.

A. *Embedded Systems Engineering*

The function of Systems Engineering is to guide the development of complex systems, understanding system as a set of interrelated components working together toward some common objective [7]. Embedded systems are a particular type of system, where the system is embedded in its enclosing device (e.g., elevators). There is an essential difference between embedded and other computing systems that makes their engineering particularly challenging. Since embedded systems involve computation that is subject to physical constraints, the separation of computation (software) from physicality (platform and environment) does not work for embedded systems. Instead, the design of embedded systems requires a holistic approach that integrates hardware design, software design, and control theory in a consistent manner [1].

B. *Model-based System Engineering*

“Model-based Systems Engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases” [8]. “MBSE is part of a long-term trend toward model-centric approaches adopted by other engineering disciplines, including mechanical, electrical and software” [8]. In the particular case of software, MBSE can be seen as part of Model Driven Engineering (MDE), a software development paradigm where models are the central element in the development process [9]. Hence, following MDE, systems software does not only serve as documentation, but can also be used to generate code or be executed for validation purposes.

C. *Variability Management*

Variability is the ability to change or customize a system [10]. Variability can also be understood as modifiability (to allow variation or evolution over time) and configurability (variability in the product space) to get a set of related products or different configurations [11]. Variability and its management are key aspects not only in software product lines, but in other systems such as embedded systems. Many variability modeling techniques have been developed. Several of the approaches are based on feature modeling, one of the most used technique for variability modeling: [12][13][14], etc. There are other approaches that are based on use cases [15] or approaches that use both feature models and use cases such as [16] and [17]. Other approaches model variation points such as [18][19] and [20]. There are also approaches that integrate variability in ADLs (Architecture Description Languages) such as Koalish [21] and [22]. Several techniques use UML (Unified Modelling Language) that is the de facto notation standard in industry for software modelling. UML profiles or extensions to UML are proposed to introduce variability [23][24][25], etc.

[26] presents an approach for managing variability in Simulink models using Pure:variants for Simulink. Another approach that addresses variability in Simulink models is the approach for model-based embedded software product lines of [27][28][29]. [30] also addresses variability in Simulink.

All these approaches address variability in Simulink models. However, the focus is on managing the variability in model-based embedded systems and product lines.

Our approach is more oriented towards testing and how to manage variability in a test architecture; a Simulink model is used for implementing a test architecture. In addition to the variability in the simulation environment represented in the Simulink model, variability in the Software under test and test specifications is also considered.

Regarding variability management during validation, [31] defines a software product line for validation environments, to support variability in those environments and to be able to test different applications in different domains and technologies.

III. VARIABILITY IN TESTING ARCHITECTURES

Testing architectures are the base for a systematic testing process. By defining the testing architecture from initial phases, we ensure a correct definition of the tests and a reutilization of them along the lifecycle.

We have defined the testing architecture in Simulink [6]. Simulink is a commercial tool for modeling, simulating and analyzing multidomain dynamic systems that is integrated into the MatLab programming environment. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries [6]. Simulink block diagrams define time-based relationships between signals and state variables. Signals represent quantities that change over time and are defined for all points in time between the block diagram's start and stop time. The relationships between signals (input and output) and state variables are defined by a set of equations represented by blocks [32].

A testing architecture can be structured in four key elements; each element and the implementation of those elements using a Simulink model is explained:

-*Sources*: the inputs are the test cases to execute on the system under test. A test case is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not. In Simulink, test cases will define the set of signals that determine the simulation of the environment in which the system has to run (plant). Usually a mixture of both signals and plants ensures a correct X-In-The-Loop simulation

-System-Under-Test (SUT): at early phases of the development, the SUT is a model of the system. Then the code of the system can be simulated (S-Function in Simulink) and in final stages of development, the code can be tested within the running platform. The software can be developed following an Software Product Line (SPL) methodology or as a single system. A Software Product Line is a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [33]. In this type of development, variability of the software is instantiated at design time so the final software will have the functions for the concrete configuration in which the software will run. When the development follows a single system development, it usual to have configurable software. In this case, the software has all the functions in all configurations, but by defining the values of some parameters, the software will execute as expected in each configuration.

-Metrics: the metrics automatically analyze the test results for each test case. In Simulink, one can use verification blocks associated with the output signals to decide automatically on the correctness of the results of the test.

-Test Control: In Simulink, it is a block that controls the order of the test cases.

A. Case study: Door management control

The proposed approach has been applied in a door management control system of an elevator. This system controls the opening and closing of the doors (that include sensors, motors, etc.).

The behavior of the control is specified using a state machine where the states (Idle, Open, Opening, Closing, Closed) and actions to be applied in each state are defined. This state machine has been specified using Iar Visual State tool [34] and the code has been automatically generated. This code has been introduced in the Simulink model that implements the testing architecture as a block (an S-function, a computer language description of a Simulink block).

In Figure 1, the testing for the door management control is described:

- Test sequences indicate the values of signals over time. These sequences are automatically generated from abstract behaviour models of the software that are annotated with time aspects.
- Software-Under-Test is automatically generated from models in Iar Visual State and transformed to SFunction for integrating in Simulink.
- Model is simulated and results (output signals over the time) are obtained. These results are used to compare with expected ones.

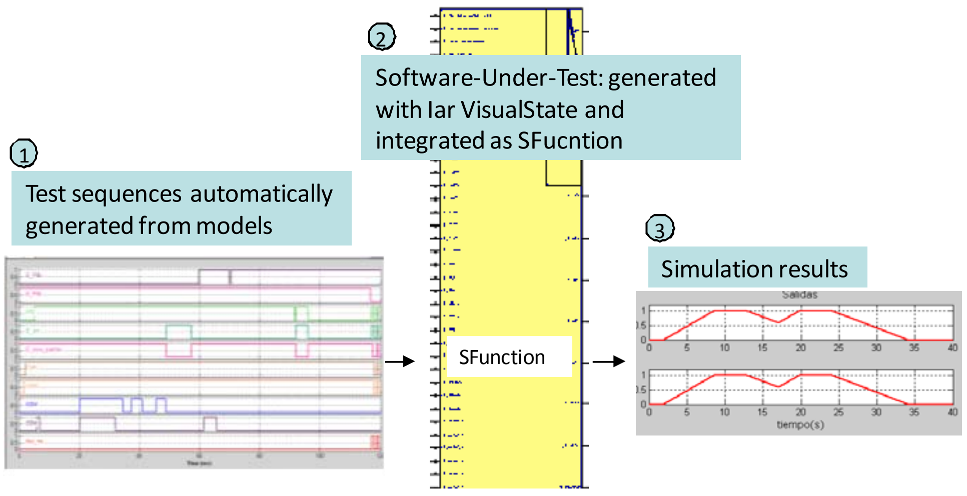

2
Software-Under-Test: generated with Iar VisualState and integrated as SFucntion
1
Test sequences automatically generated from models
SFunction
3
Simulation results
Salidas
tiempo(s)

Figure 1. Simulation Environment

The simulation environment (the plant model that simulated the physical environment signals) is valid only for one concrete configuration. One of the factors that add more complexity to testing of embedded software is the diversity of environments in which software can execute. Embedded software usually execute under different configurations. It can be connected to different number of devices, etc. There is a need to manage the variability in validation environment due to: number and type of sensors, number and type of actuators, communication mechanisms, etc.

In order to identify and model the environments in which software should be validated, a feature model can be used. A feature model is an and/or tree of different features. A feature as "a prominent or distinctive and user-visible aspect, quality, or characteristic of a software system or systems" [12]. Features can be mandatory, optional or alternative. Features are an effective way of identifying the variability (and the commonality) among different products in a domain. Moreover, features are a effective means of communication among stakeholders and are a intuitive way of expressing the variability [35] as features are distinctive characteristics or properties of a product that differ from others or from earlier versions.

The feature model contains the different elements that should be considered when validating the software depending on sensors, actuators, etc. of each configuration. Some of the most relevant are:

- Different types of doors: Software must be validated with different types of Doors: articulated, non-articulated, etc.
- Different number of doors: Software must be validated with one, two, three Doors. Doors can operate independently or not.
- Different floor configuration: floors can have different door configuration: depending on the floor, doors require different behaviour.
- Different sensors: Optional obstacle and presence sensor and optional limit switch.

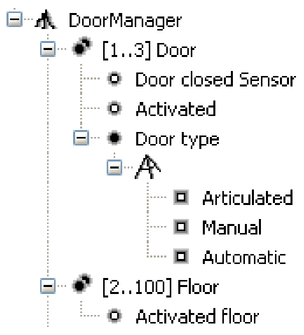


Figure 2. Feature model for validation

Feature model containing the variability can be modelled as a form in MatLab or using some specific tool for variability management, such as pure::variants for Simulink. To perform validation taking into account this variability, it is necessary to manage the variability in the following aspects of the simulation environment:

- Configuration of the Software-Under-Test. A .xml file is automatically generated from variability management form and indicates the initialisation of the SUT for a concrete configuration.
- Modelling of the simulation environment: In Simulink, variability is on relations and blocks that are required for simulation. Simulation elements are contained in a library and are connected automatically guided by the variability form in order to create the simulation model for a configuration.
- Tests' specification: As not all the configurations require the same requirements for testing, variability in tests should be taken into account too. Depending on the configuration some functionalities are not active or even, same functionalities could differ on required response time.

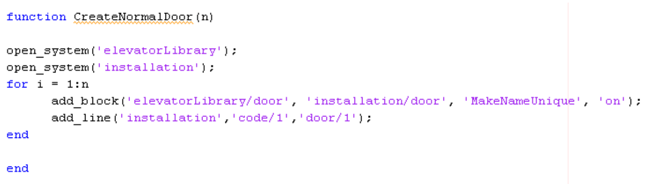
The next section details this variable simulation model.

IV. VARIABLE SIMULATION MODEL

Simulation model includes the simulation of both mechanicals elements and software that manages these elements. Including variability in the simulation models allows representing different configurations in which software will run. This way it is possible to validate the system taking into account different configurations in a less costly and time-consuming way. The software is integrated as a block in the model and is connected to the blocks representing the mechanicals elements. Running the model provides the simulation of the real system.

In order to get an effective simulation of different configurations, variability management has been included in Simulink. For this purpose, a variability form has been developed in MatLab asking for the information that represents the configurations: number and types of doors, etc.

This information is used to develop dynamically the simulation model of the configuration to be validated. In order to develop the simulation model dynamically we have created a library with the elements that can appear in the simulation model: doors, code block, etc. Code is executed for creating the simulation model with the features selected in the variability form. See Figure 3 for simulation model creation for a configuration containing two doors of NormalType.



```
function CreateNormalDoor(n)

open_system('elevatorLibrary');
open_system('installation');
for i = 1:n
      add_block('elevatorLibrary/door', 'installation/door', 'MakeNameUnique', 'on');
      add_line('installation','code/1','door/1');
end

end
```

Figure 3. MatLab code for dynamic simulation model creation

This way, by selecting values in the variability form, we obtain automatically simulation models that are specific for the configuration we want to prove (See Figure 4). The same test architecture is used and test cases may be also adapted and reused as test cases will be also developed taking into account variability. Thus, we obtain the advantage of getting simulation models for different configurations with a reduced cost. Therefore, tests could be easily performed in different configurations obtaining greater test coverage of the embedded system. In an initial development stage, the simulation models may be automatically generated in an exhaustive way to test all configurations. In later stages and during maintenance, the generation of simulation models may be used to test new configurations.

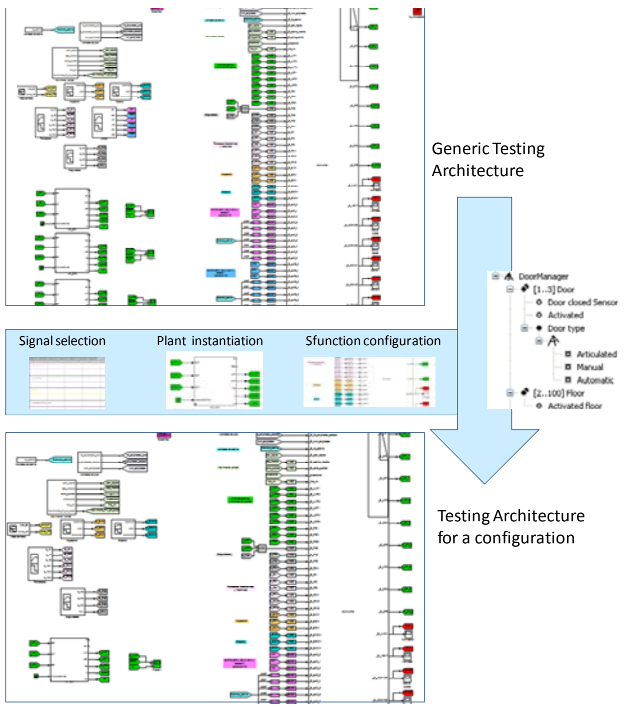


Figure 4. Instantiation of the testing architecture for an configuration

V. CONCLUSION AND FUTURE WORK

This paper has described a simulation environment for embedded software based on model based testing and variability management and using Simulink as simulation tool. As embedded software usually runs under different configurations, it is costly to test the software under real conditions. Variability management of configurations helps automating the simulation environment. This way, validation is simplified and intensive testing can be performed.

In the case study, a variability management form has been developed in MatLab. This option has been adequate for our purpose, but as complexity increases it is recommended to use a tool specific for variability management. Pure::Variants is a tool integrated with Simulink that could be adequate for this purpose [26]. Or, the variability management approach for Simulink proposed by [27][28][29] can be also used for plant instantiation part. Those approaches can be used in a complementary way for variability management and instantiation of the Simulink models (plant). Those approaches are not oriented to manage variability in validation architectures, but in general in Simulink models, so they do not cover some specific needs such as the configuration of Sfunctions in Simulink, generation of test sequences, etc., that our approach covers.

It is always difficult to establish the coverage of the tests, more when multiple configurations have to be validated. Although tests cover 100% of transitions we can not ensure that all configurations have been tested. In this case, variability instantiation has been done manually. In order to get a greater coverage, it is highly recommended to automate the variability selection, generating the simulation environment for all the configurations sequentially. Next steps include analysing the feasibility of this option and the coverage that is got this way.

The paper has focused on the simulation phase. However, once a system has been validated in Simulink, software is integrated in the real system. Test architecture and variability management should be reused in subsequent phases. Our next actions will consider the generation of tests from the model for running in the software using python test scripts [36].

ACKNOWLEDGMENT

This work is co-supported by the Basque Government under grants UE2011-4 (COMODE Project). The project has been developed by the embedded system group 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, volume 4085 of Lecture Notes in Computer Science, pp. 1–15. Springer, 2006.

[2] OMG. “UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems”. Formal Specification, November 2009. Online at: http://www.omg.org/spec/MARTE/1.0/PDF. [retrieved: September, 2012].

[3] J. A. Stankovic. “Strategic Directions in Real-time and Embedded Systems”. ACM Computing Surveys, 28, pp. 751–763, December 1996.

[4] H. Shokry and M. Hinchey. “Model-Based Verification of Embedded Software”, IEEE Computer, vol. 42, no. 4, pp. 53-59, April, 2009

[5] E. Bringmann and A. Krämer. “Model-based Testing of Automotive Systems” pp. 485–493, 2008 International Conference on Software Testing, Verification and Validation, ICST, 2008.

[6] Simulink webpage. http://www.mathworks.com/products/simulink/. [retrieved: September, 2012].

[7] A. Kossiakoff and W. N. Sweet. Systems Engineering. Principles and Practice. Addison Wesley, 2003.

[8] International Council on Systems Engineering (INCOSE). “Systems Engineering Vision 2020”. Technical Report INCOSE-TP-2004-004-02, INCOSE, September 2007.

[9] R. France and B. Rumpe. “Model-Driven Development of Complex Software: A Research Roadmap”. In Workshop on the Future of Software Engineering (FOSE 2007), at ICSE 2007, Minneapolis, Minnesota, USA, pp. 37–54, 2007.

[10] J. Van Gurp, J.Bosch, and M. Svahnberg. “On the notion of variability in software product lines”. In WICSA ’01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA’01), Washington, DC, USA, 2001. pp. 45-54. IEEE Computer Society.

[11] S. Thiel and A. Hein. “Systematic integration of variability into product line architecture design”. In SPLC 2: Proceedings of the Second International Conference on Software Product Lines, pp. 130–153, London, UK, 2002. Springer-Verlag.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. “Feature-oriented domain analysis (foda) feasibility study”. Technical Report CMU/SEI-90-TR-21, November 1990.

[13] K. Czarnecki and U. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, June 2000.

[14] K. Czarnecki, S. Helsen, and U. W. Eisenecker. “Staged configuration using feature models”. In Robert L. Nord, editor, SPLC,

volume 3154 of Lecture Notes in Computer Science, pp. 266–283. Springer, 2004.

[15] G. Halmans and K. Pohl. “Communicating the variability of a software-product family to customers”. Software and System Modeling, 2(1): pp. 15–36, 2003

[16] M. Eriksson, J. Börstler, and K. Borg. “The pluss approach - domain modeling with features, use cases and use case realizations”. In J. Henk Obbink and Klaus Pohl, editors, SPLC, volume 3714 of Lecture Notes in Computer Science, pp. 33–44. Springer, 2005.

[17] T. von der Maßen and H. Lichter, “Requiline: A requirements engineering tool for software product lines”. In 5th International Workshop on Product Family Engineering, PFE5, Proceedings, 2003, pp. 168-180.

[18] H. Gomaa and D. L. Webber. “Modeling adaptive and evolvable software product lines using the variation point model”. In HICSS ’04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS’04) - Track 9, p. 90268.3, Washington, DC, USA, 2004. IEEE Computer Society.

[19] K. Pohl, G. Böckle, and F. J. van der Linden. Software Product Line Engineering : Foundations, Principles and Techniques. Springer, September 2005.

[20] M. Becker. “Towards a general model of variability in product families”. In Proceedings of the 1st Workshop on Software Variability Management, 2003.

[21] T. Asikainen, T. Soininen, and T. Männistö. “A koala-based approach for modelling and deploying configurable software product families”. In Frank van der Linden, editor, Software Product-Family Engineering, 5th International Workshop, PFE, Revised Papers, volume 3014 of Lecture Notes in Computer Science, pp. 225–249. Springer, 2003.

[22] A. van der Hoek. “Design-time product line architectures for any-time variability”. Sci. Comput. Program., 53(3), pp.285–304, 2004.

[23] H. Gomaa. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley, 2004.

[24] M.s Clauß. “Modeling variability with uml”. In Proceedings of GCSE2001. Young Researchers Workshop, 2001.

[25] T. Ziadi, L. Hélouët, and J.M.c Jézéquel. “Towards a uml profile for software product lines”. In Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, Revised Papers, pp. 129–139, 2003.

[26] C. Dziobek, J. Loew, W. Przystas, and J. Weiland. “Model diversity and variability - handling of functional variants in simulink-models”. Elektronik automotive, February 2008, pp.33-37.

[27] A. Polzer, D. Merschen, A. Botterweck, G. Pleuss, J. Thomas, S. Hedenetz, and B. Kowalewski. “Managing complexity and variability of a model-based embedded software product line”. Innovations in Systems and Software Engineering (ISSE), 8, pp.35–49, 2011.

[28] G. Botterweck, A. Polzer, and S. Kowalewski. “Using higher-order transformations to derive variability mechanism for embedded systems”. 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACESMB 2009) atMoDELS 2009, Vol-507, pp. 107 – 121, Denver, Colorado, USA, September 2009.

[29] G. Botterweck, A.s Polzer, and S Kowalewski. “Variability and evolution in model-based engineering of embedded systems”. In 6. Dagstuhl-Workshop Model-Based Development of EmbeddedSystems (MBEES 2010), pp. 87–96, Dagstuhl, Germany, February 2010.

[30] D. Beuche and J. Weiland. “Managing flexibility: Modeling binding-times in simulink”. In Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA ’09, pp. 289–300, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] B. Magro, J. Garbajosa, and J. Pérez. “A software product line definition for validation environments”. In Proceedings of the 2008 12th International Software Product Line Conference, SPLC ’08, pp. 45–54, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Modeling Dynamic Systems, web page, http://www.mathworks.es/es/help/simulink/ug/modeling-dynamic-systems.html. [retrieved: September, 2012].

[33] P. Clements and L. Northrop. Software Product Lines - Practices and Patterns. Addison-Wesley, 2001.

[34] IAR VisualState, Web page. http://www.iar.com/en/Products/IAR-visualSTATE/, [retrieved: September, 2012].

[35] K. Lee, K. C. Kang, and J. Lee. “Concepts and guidelines of feature modeling for product line software engineering”. In ICSR-7: Proceedings of the 7th International Conference on Software Reuse, pp. 62–77, London, UK, 2002. Springer-Verlag.

[36] Python official webpage. http://www.python.org/. [retrieved: September, 2012].