

An Automated Model-based Testing Approach in Software Product Lines Using a Variability Language

Boni García, Rodrigo García-Carmona, Álvaro Navas, Hugo A. Parada G.,
Félix Cuadrado, Juan C. Dueñas

Departamento de Ingeniería de Sistemas Telemáticos
ETSI Telecomunicación - Universidad Politécnica de Madrid
Avda. Complutense 30, 28040 Madrid, Spain
{bgarcia, rodrigo, anavas, hparada, fcuadrado, jcduenas}@dit.upm.es

Abstract. This paper presents the application of an automated testing approach for Software Product Lines (SPL) driven by its state-machine and variability models. **Context:** Model-based testing provides a technique for automatic generation of test cases using models. Introduction of a variability model in this technique can achieve testing automation in SPL. **Method:** We use UML and CVL (Common Variability Language) models as input, and JUnit test cases are derived from these models. This approach has been implemented using the UML2 Eclipse Modeling platform and the CVL-Tool. **Validation:** A model checking tool prototype has been developed and a case study has been performed. **Conclusions:** Preliminary experiments have proved that our approach can find structural errors in the SPL under test. In our future work we will introduce Object Constraint Language (OCL) constraints attached to the input UML model.

Keywords: Automated Testing, Model Checking, Variability, Software Product Lines.

1 Introduction

Software Product Lines (SPL) refer to engineering techniques to create a set of similar software systems from shared assets using common means of production. The variability of a SPL means the state or characteristic of that SPL which can be adjusted or changed in order to generate all the elements which composes the product family. Testing is the most important method to ensure the quality of software. Automation is desirable, because manual testing takes a lot of effort and is error prone.

This article presents the application of an automated testing approach for SPL driven by different models. The input in our method will be the state machine of the System Under Test (SUT) in the form of an UML diagram and the variability model. The variability model will be described using the Common Variability Language (CVL), which is an initiative of SINTEF, University of Oslo, and the European

project ITEA-MOSIS¹. At the time of writing, CVL is in Request For Proposal (RFP) phase by the OMG² consortium since December 2009.

The remainder of this paper is structured as follows: Section 2 introduces the context in which this work has been performed, i.e. Model-Based Testing (MBT) and variability in Software Product Lines. Section 3 describes in detail the proposed testing method. Section 4 discusses the validation of the work by means of a case study. Finally, section 5 presents the reached conclusions and the future work.

2 Context

Our work has as main topics the Model-Based Testing paradigm for the testing of software, and the Variability for addressing the challenges of Software Product Line systems.

2.1 Model-Based Testing

Software testing is the main technique to ensure quality and finding bugs. Automation is desirable, because manual testing is usually a complex and time-consuming task. Model-Based Testing (MBT) is a testing method in which the test cases are automatically generated from a model which describes the behaviour of the System Under Test (SUT). MBT uses models of different nature to drive test generation. Utting et al. [1] have identified four different MBT techniques: i) generation of test data from a domain model; ii) generation of test cases from an environmental model; iii) generation of test cases with oracle from a behaviour model; iv) generation of test cases from abstract tests.

The first kind of MBT model is the information about the domains of the input values. In this model, the test generation involves selection and combination of a subset of those values to produce test input data. The second kind of MBT employs a different meaning of model, which describes the expected environment of the SUT. An example of this kind of MBT is described in [2], in which the model is a statistic description of the expected usage of the SUT. The third meaning of model describes the expected behaviour of the SUT, such as the relationship between its inputs and outputs. The fourth meaning of MBT employs an abstract description of a test case, such as an UML sequence diagram. That abstract test case is transformed into a low-level executable test case.

Another key topic in MBT is the model checking. A model checker is a tool which takes as input an automaton based model of a system and a temporal logic property, and then explores the entire state space of the system in order to determine if the model violates the property or not. The actual implementation of the model is influenced by the environment, such as the platform, compiler, and so on. For this reason, any kind of testing is required. The main challenge of testing with model checkers is to force the model checker to systematically create sets of

¹ <http://itea-mosis.org/>

² <http://www.omgwiki.org/variability/doku.php>

counterexamples as test cases, which can then be used as a complete test suite. Callahan et al. [3] and Engels et al. [4] initially proposed the use of model checkers for the automated generation of test cases. The most commonly employed model checkers are the explicit state model checker SPIN [5] (Simple Promela Interpreter), the Symbolic Analysis Laboratory SAL [6], which supports both symbolic and bounded model checking, and the symbolic model checker SMV [7] and its derivative NuSMV [8], which both support symbolic and bounded model checking.

2.2 Software Product Lines and Variability

A Software Product Line (SPL) is a set of software-intensive systems sharing common features and satisfying some specific needs, and developed from a common set of domain artifacts. The domain is defined as the space of knowledge driven by business requirements and characterized by some concepts and terminology understood by specific stakeholders [9].

In a product line with a large number of products and requirements, managing variability can become a problematic task. Consequently, the management of variability plays a crucial role in successful Software Product Line engineering. A solution to that problem is the variability modeling, which captures the essence of how one product is similar, but still different from another. The competence of domain experts of one or more product line can be synthesized into a Domain Specific Language (DSL).

CVL (Common Variability Language) is a language for specifying variability in a way that is common to DSLs [10]. The main concepts in CVL are substitutions. Model elements are related by means of references. The CVL model points out model elements of the base product line model and defines how these model elements shall be manipulated to yield a new product model. There are three kinds of substitutions: value substitution, reference substitution and fragment substitution.

CVL has been defined as a metamodel. The element root is named CVLModel. It contains a variability specification, i.e. the variability model, and one or more resolution models. The variability specification can be divided into two groups: i) executable primitive, which specifies the CVL execution; ii) declaration, which specifies elements on which the executable primitives work. The executable primitives can also be divided in two groups: composite variability and iterator. While all executable elements in a composite variability will be executed, the iterator represents a choice, where the resolution model can make a selection of the contained executable primitives.

The semantics of CVL is described by a model-to-model transformation that relates to the actual base language through reflection mechanisms. This transformation has been implemented in an extended version of MOFScript³ which, in addition to model-to-text generation capabilities also supports model-to-model transformations.

³ <http://www.eclipse.org/gmt/mofscript/>

3 Method

The aim of this paper is the application of a MBT approach for SPL driven by UML and CVL models. This approach can be classified within the fourth kind of MBT techniques described in section 2.1. This approach has been implemented by means of a model checker tool prototype, which is composed of the following components, as seen in Fig. 1:

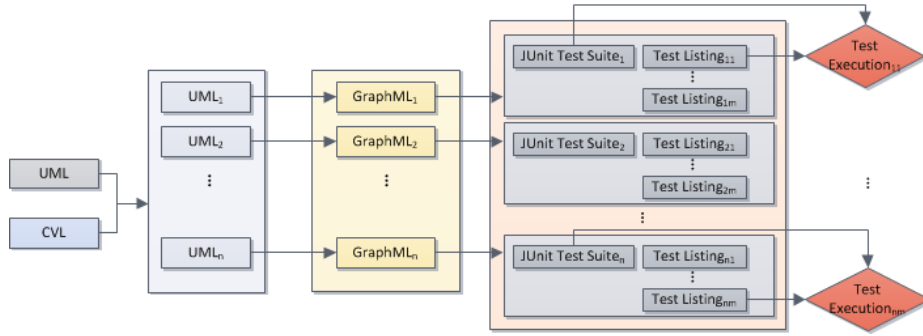









Fig. 1. Tool chain components.

First of all, we create a UML state diagram, which models the behavior of the base system. We also create, using CVL-Tool [11], a CVL model expressing the variability that can be applied to the state diagram. The result of the combination of these two models, again using CVL-Tool, is a set of UML state diagrams, each one of them representing a product of the whole product line derived from the original UML state diagram. The elements handled by the CVL-Tool are described in the following table:

Table 1. Summary of the CVL elements.

Element	Symbol	Description
CompositeVariability		An ordered set of Variability Specifications.
Invocation		Instantiation of a type, execution of the executable primitives of its body, and yielding as result a ReplacementFragmentReference.
Assignment		Assign an instance of a replacement fragment to a ReplacementFragmentReference.
AND		Boolean operator AND.
OR		Boolean operator OR.
IMPLIES		Boolean operator IMPLIES.
NOT		Boolean operator NOT.
ExecutablePrimitiveTerm		Boolean ?
Choice		Multiplicity, XOR, OR.
ReferenceSubstitution		A substitution that will change a reference attribute in the base model.
PlacementObject		A fragment of the base model that will be replaced by a ReplacementFragment during the variability

ReplacementObject		transformation. Model element that references attribute will denote after a reference substitution.
FragmentSubstitution		Substitutes a fragment of the base model with another fragment of the base model.
PlacementFragment		A fragment of the base model that will be replaced by a ReplacementFragment during the variability transformation.
ReplacementFragment		A fragment of the base model that will be used as replacement for some placement fragment of the base model.
ValueSubstitution		A substitution that will change the value of an attribute of a base model element.
PlacementValue		A placement value represents a value-typed attribute of the model element denoted by targetObject, an attribute that by a ValueSubstitution may get a new value represented by a ReplacementValue.
ReplacementValue		A value to replace the value of an attribute represented by a PlacementValue.

After this step, we apply a transformation to these UML state diagrams and create a GraphML⁴ model from each of them. GraphML is an XML-based file format for graphs specified in [12]. The following stage generates the test listing from the finite-state machine in GraphML format. This work is carried out by the MBT implementation tool by Tigris⁵. A test listing is an ordered enumeration of the states and transitions that are traversed in a particular execution of the modeled system. Several listings can be produced by the MBT tool modifying the invocation parameters. This way, each test listing can have a particular purpose. Examples of this include a listing which traverses all the states of the diagram at least once, a listing which crosses a particular state at least twice or a listing which does not to traverse a set of transitions. Our tool prototype generates the test suite based on pluggable JUnit templates. This suite contains a test case for each of the states of the diagram, transitions, or both. The test case created is currently a simple skeleton that could be filled by a human tester.

Finally, we take each of the listings and invoke the corresponding test cases for each of the states and transitions contained in it in order. This execution is fully carried out by our tool. A verdict for each listing is produced, informing if the tests have been passed or failed.

This process has been automated with a set of Ant scripts. This way, the user does not need to know the inner workings of all the components and only needs to provide basic configuration parameters.

⁴ <http://graphml.graphdrawing.org/>

⁵ <http://mbt.tigris.org/>

4 Validation

To discuss the validity of our approach we have developed a tool chain prototype. This prototype aims to provide at least some basic functionality that enables us to study the consequences of using MBT with a product line. With this tool, we have created a sample product line using a UML state diagram and a CVL model, and generated and executed a set of tests for it.

With the tool developed, we can now generate and execute tests in a sample scenario. This scenario is composed of a UML state diagram representing a call processing system for a telecommunications company. This state machine has been created using the UML2 modeling platform by Eclipse⁶. This diagram shows the basic functionality of the system and is depicted in the following figure:

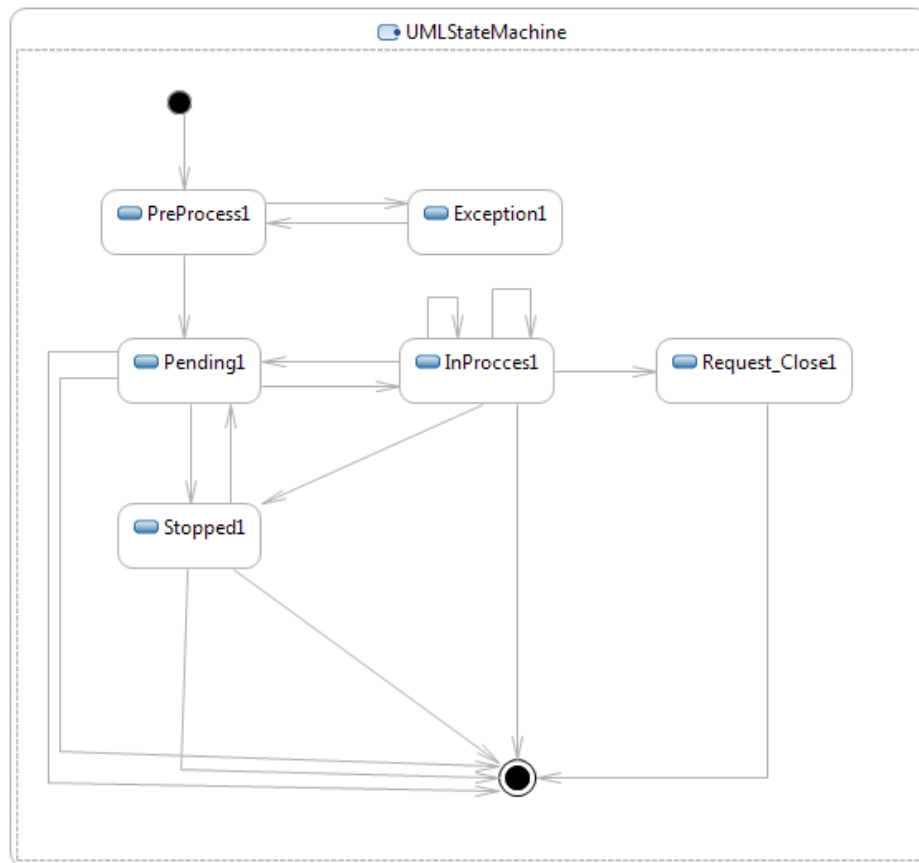


Fig. 2. State diagram of a call processing system.

⁶ <http://www.eclipse.org/modeling/>

To represent the different services that can be offered by the company, this scenario is also composed of a CVL model (in Fig. 3) which represents the variability that can be applied to the base model. The CVL model defines three variations of the UML model. The meaning of the CVL icons can be checked in Table 1. The first variation is basically the same that the original one but changing the name of one state. The second one changes a fragment of the UML model. The fragment is defined in CVL by a set of boundary elements. In this case study a fragment means a collection of states: *InProcces1* and *Request_Close1*. Finally the third variation of the model changes an object of the UML, the *Request_Close1* state. Therefore, with these two models (UML and CVL) our SPL is fully defined.

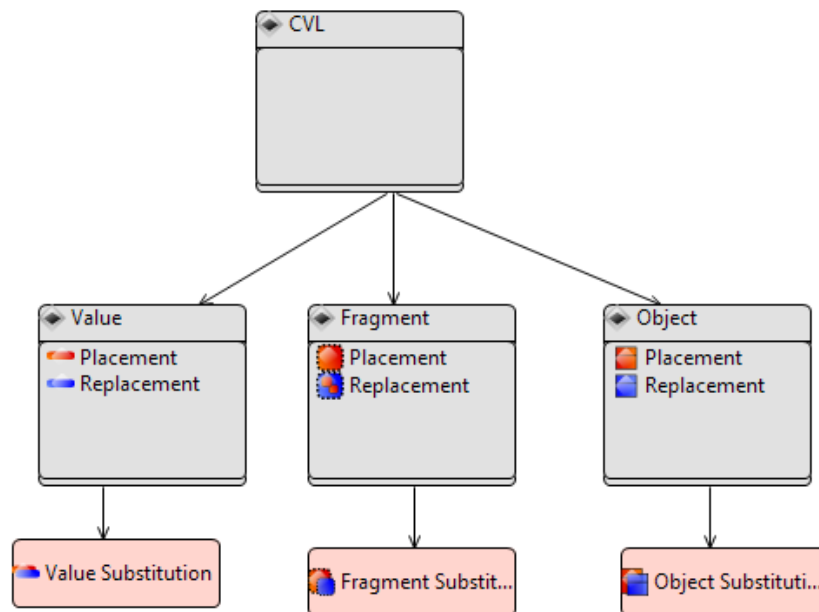


Fig. 3. CVL model.

Using these models as input for our tool chain we generated and executed several test listings for each product, experiencing a huge increment in efficiency from a manual approach. Below is a fragment of one of the test listings produced in the case study:

```

...
Pending1
Stop1
Stopped1
Close1
Closed1
e_fin
Start

```

```

Process1
PreProcess1
Error1
Exception1
Resolve1
PreProcess1
Error1
Exception1
Resolve1
PreProcess1
OK
Pending1
...

```

And here is the skeleton of the test case corresponding to the *Stop1* state:

```

/**
 * This method implements the Edge 'Stop1'
 */
public void testStop1()
{
    log.info( "Edge: Stop1" );
    throw new RuntimeException( "The Edge: Stop1 is not
implemented yet!" );
}

```

Although the test cases still needed to be completed manually, the process of selecting the way in which each state diagram needs to be traversed was completely automated. The manual definition of these paths to test certain requirements would have taken a huge amount of time. Moreover, this time grows exponentially with the number of states and transitions in the diagram, making it impractical without automation for some situations.

In addition to these results, two important concerns have arisen from the execution of this process. First we noticed that, during the test listing generation phase of the process, in some cases an error prevented some test listings from being created. This was due to the fact that after the variability has been applied, some of the produced UML state diagrams were left with some states that were unreachable under some conditions, from which no end state could be reached, or with other kind of structural problems. Therefore, the generation of a test listing discovered errors in the structure of some state diagrams. If instead of automating the traversing of each of the produced diagrams a manual approach would have been used, the discovery of some of these errors would have taken a huge effort. Or in some cases they might not even be uncovered at all. This advantage is more noticeable the bigger the product line is.

The other concern is the test case reusability. Comparing the test executions for different state diagrams we observed that the test cases of which they were composed were the same in most executions. This way, although the order of the test cases and a small amount of them are different from one product to another, a lot of effort in the

definition of them can be saved with the creation of these tests from the original UML state diagram and the CVL model, instead from the UML state diagrams resulting from the combination of the two.

5 Conclusions

In this paper we have proposed an approach to automate the generation and execution of tests cases in software product lines. For this approach we have employed two different methodologies: variability to express the differences between the products instead of defining each one individually, and Model-Based Testing to automate the generation and execution of tests. The gap between these two methodologies has been filled with the CVL modeling language, which expresses variability by means of models.

The final result of the presented work is two-folded. On one hand, we have developed a method and a tool prototype for Model-Based Testing based on the usage of UML plus CVL as inputs. This prototype tool chain produces and executes tests from a UML state diagram representing the behavior of a system, and a CVL model representing the variability. It is based on Eclipse technologies, using the UML2 plugins and the CVL-Tool. The output of this prototype is a set of JUnit test suites for the UML family described by the input (UML plus CVL). On the other hand, we have validated this approach by means of a case study (a call processing system) and found that, on top of the reduced manpower that this approach needs compared to a traditional one, other benefits can be gained from it. One of these benefits is an improvement in the capability of detecting errors in the structural integrity of all the products of the product line. The other is the possibility of reusing individual test cases in several products and generating them from the original model plus the CVL model instead of the models of the whole product line.

This benefit is particularly promising and should be developed further. Our future work will be dedicated to extend this approach by introducing constraints within the input model. These constraints will be implemented using the declarative language Object Constraint Language (OCL) linked to the UML model [13]. Our prototype will map the OCL constraints to different JUnit assertions.

Acknowledgements

This research project has been performed in the context of the European project ITEA-MOSIS (project number 06035), under grant by Spanish Ministerio de Industria, Turismo y Comercio in the PROFIT program.

References

1. Utting, M., and Legeard, B.: Practical Model Based Testing: A Tools Approach, Morgan Kaufmann 1st ed., 2006. ISBN: 978-0123725011, 456p.

2. S. J. Prowell. JUMBL: A tool for model-based statistical testing. In Proceedings of the 36th Annual Hawaii International Conference on System Sciences, 331–345. IEEE, 2003.
3. Callahan, J., Schneider, F., Easterbrook, S. Automated Software Testing Using Model-Checking. In Proceedings 1996 SPIN Workshop, August 1996. Also WVU Technical Report NASAIVV-96-022.
4. Engels, A., Feijs, L., and Mauw, S.: Test generation for intelligent networks using model checking. In Ed Brinksma, editor, Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'97), volume 1217 of Lecture Notes in Computer Science, Enschede, the Netherlands, April 1997. Springer-Verlag.
5. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. Softw. Eng., 23(5):279–295, 1997. ISSN 0098-5589. doi: 10.1109/32.588521.
6. DeMoura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., and Tiwari, A. SAL 2. In Rajeev Alur and Doron Peled, editors, Computer-Aided Verification, CAV 2004, volume 3114 of Lecture Notes in Computer Science, pages 496–500, Boston, MA, July 2004. Springer-Verlag.
7. K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie-Mellon University, 1992.
8. Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. NUSMV: A New Symbolic Model Verifier. In CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification, pages 495–499, London, UK, 1999. Springer-Verlag. ISBN 3-540-66202-2.
9. Käkölä, T., Dueñas, J.C.: Software Product Lines: Research Issues in Engineering and Management. Springer, 2006. ISBN: 3540332529
10. Franck, F., Øystein, H., Birger, M., Gøran, O., Andreas, S., Xiaorui, Z.: A Generic Language and Tool for Variability Modeling. Cooperative and Trusted Systems. University of Oslo. SINTEF. ISBN 9788214044676
11. Øystein, H., Møller-Pedersen, O., Svendsen, O.: Adding standardized variability to domain specific languages”. In B. Geppert and K. Pohl, editors, Software Product Lines, 12th International Conference, (SPLC 2008) Proceedings, volume 1, pages 139–148, September 2008, Limerick, Ireland.
12. Brandes U. et al. 2002. GraphML Progress Report: Structural Layer Proposal. *Proceedings of 9th International Symposium of Graph Drawing (GD '01)*. LNCS 2265, pp. 501-512. Springer-Verlag.
13. Object Management Group (OMG); Object Constraint Language (OCL) Specification. Version 2.0. June 2006.