Towards an automated testing framework to manage variability using the UML Testing Profile

Beatriz Pérez Lamancha
Software Testing Centre (CES)
School of Engineering
University of the Republic
Julio Herrera y Reissig 565, 11300
Montevideo, Uruguay
bperez@fing.edu.uy

Macario Polo Usaola, Mario Piattini Velthius
Alarcos Research Group
Information Systems and Technologies Department
University of Castilla-La Mancha
Paseo de la Universidad/4, 13071
Ciudad Real, Spain
{macario.polo,mario.piattini}@uclm.es

Abstract

This paper proposes an extension to the UML Testing Profile to manage variability in testing artifacts for software product lines. The proposed extension has two main points: (i) Defining an extended architecture for the UML Testing Profile to deal with variability in the test models, and (ii) Defining the behavior to include variation points in the SPL. To this aim, this work focuses on the test case behaviour represented by sequence diagrams and defines an extension to UML interactions for SPL.

1. Introduction

A software product line (SPL) is a set of software-intensive systems sharing a common, managed set of features which satisfy the specific needs of a particular market segment or mission and which are developed from a common set of core assets in a prescribed way [1]. The commonalities consist of a structured list of assumptions that are true for each member of the SPL, while the variabilities are a structured list of assumptions on how family members differ [2]. Variability management plays a central role in software product lines and constitutes a new challenge if compared to classical software engineering. Firstly, additional work is inevitable in the beginning. Secondly, the choice of a notation to model the variability following the company domain is quite tricky. Thirdly, the variability must be explicitly managed during the whole development process and not only during the implementation phase where the code is produced [2].

A variation point locates a variability and its possible bindings by describing several variants. Each variant is one way to realize a particular variability label and binds it in a specific way. The variation point itself locates the insertion point for the variants and determines the characteristics (attributes) of the variability. Cardinalities of variation points indicate the minimum number of variants which have to be selected for this variation point and the maximum number of variants which can be selected for this variation point [2].

Testing in the context of SPL includes the derivation of test cases for the line and for each specific product, exploiting the possibilities of traceability to reduce the cost of creating both the test model and the line test cases, which includes their instantiation to test each product. Model-based testing requires the systematic and possibly automatic derivation of tests from models [3]. In our proposal, a methodology is defined to describe test cases using models, extending both UML 2.0 and the UML Testing Profile (UML-TP) to support variability and to allow model transformation. Thus, this work has two main points:

1. The inclusion of extensions in UML and UML-TP for managing variability.
2. The definition of the behavior of test cases to manage variation points in the SPL.

This paper focuses on the behavior of test cases, represented by sequence diagrams. The proposed extension conforms to the UML-TP for SPL testing. Our current works include the automation of test case generation from UML design models to test models, based on the Query/View/Transformation (QVT) [4] model transformation language.

It is important to note that, although several researchers have proposed techniques to deal with testing in SPL [5-10], none of them has been developed in the framework of the UML 2.0 OMG standard, which is the basis for the latest versions of software development tools. In SPL, the best practices for analysis, design, coding, etc., are intensively applied and the use of standards and tools becomes essential. In this respect, this paper makes a significant contribution, since the proposal is completely framed within well-known

978-1-4244-3711-5/09/$25.00 © 2009 IEEE

standards. The paper is organized as follows: Section 2 presents related work; Section 3 describes the SPL used to exemplify the proposal; Section 4 presents the extension proposed to manage variability in testing models using the UML Testing Profile; and Section 5 shows a test case behaviour example. Finally, section 6 draws some conclusions.

2. Related Works

This section summarizes two proposals that are used in this paper: the UML 2.0 Testing Profile and the Orthogonal Variability Model. It also includes a brief description of the main works that discuss SPL testing.

2.1 The UML 2.0 Testing Profile

The UML 2.0 Testing Profile (UML-TP) defines a language for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of test systems. It extends UML 2.0 with test specific concepts for testing, grouping them in test architecture, test data, test behaviour and test time. Being a profile, the UML-TP seamlessly integrates into UML.

It is based on the UML 2.0 specification and is defined by using the metamodeling approach of UML [11]. Figure 1 presents the UML-TP metamodel for the test architecture and test behaviour concepts [11].

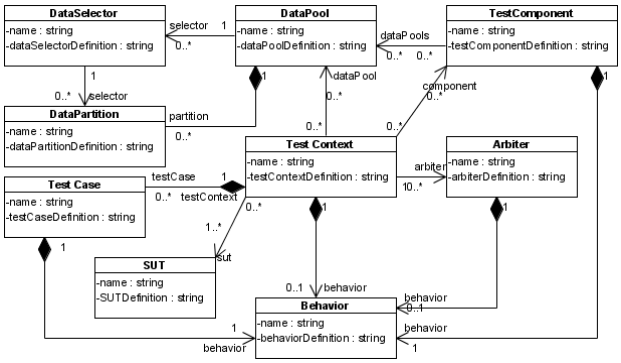

Figure 1. UML-TP Metamodel [11]

The test architecture in UML-TP is the set of concepts to specify the structural aspects of a test situation [3]:

- *TestContext* is a stereotyped class that contains the test cases (as operations) and whose composite structure defines the test configuration. The classifier *Behaviour* of the test context provides the test control, which determines the order of execution of test cases (see Figure 1).
- *TestConfiguration* is the composite structure of the test context showing the communication structure between the *TestComponents* and the system under test.
- *SUT* represents the system under test.
- *TestComponents* are those objects within the test system that can communicate with the *SUT* or other components to realize the test behaviour.
- *Arbiter* provides a means for evaluating test results derived from different objects within the test system in order to determine an overall verdict for a *TestCase* or *TestContext*.

The test behaviour specifies the actions and evaluations necessary to evaluate the test objective, which describes what should be tested. The test case is the major concept in a test model. The test case behaviour in UML-TP is described using the *Behavior* concept and can be shown using UML interaction diagrams, state machines and activity diagrams. The UML-TP concepts used to describe the test behaviour can be summarized by [3]:

- *TestObjective*: allows the designer to express the intention of the test.
- *TestCase* is a specification of one case to test the system, including what to test it with, the required input, result and initial conditions. It is a complete technical specification of how a set of *TestComponents* interacts with an SUT to realize a *TestObjective* to return a *Verdict* value. The implementation of *TestCases* is specified by a *testBehavior*, while the semantics of test cases are given by the semantics of the *Behaviour* that realizes it [11].
- *Verdict*: is a predefined enumeration specifying possible test results, such as pass, inconclusive, fail and error.
- *ValidationAction*: is performed by a *TestComponent* to indicate that the *Arbiter* is informed of the test component's test result.
- *DataPool*: contains a set of values or partitions that can be associated with a particular test context and its test cases. A data partition is used to define equivalence classes and data sets, and a data selector defines different selection strategies for these data sets.

This work focuses on test cases, whose behavior is represented with UML sequence diagrams.

2.2 Orthogonal Variability Model (OVM)

An orthogonal variability model (OVM) [12] defines the variability of a system family, i.e., the variability information is in a separate model in the form

of variation points (VPs) and variants. It associates the VPs and variants defined with other software development models such as design models or component models. The core concepts of the OVM language are variation points and variants. Each variation point offers at least one variant. Additionally, the constraints-associations between these elements describe dependencies between variable elements [12].

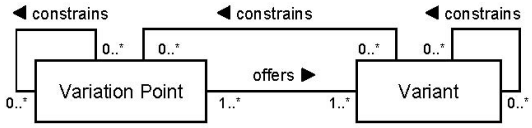

Figure 2. Excerpt of OVM metamodel [12]

While the OVM approach can be used for the construction of a complete product line, this paper is only concerned with the way that this approach metamodels the variability: the way in which the relationship between variation points and variations are represented is inspired by OVM.

2.3 Testing in SPL

There are different methodologies to derive test cases for SPL; all the works define their own models to represent the testing artifacts and variability in the test model. None of them uses the UML Testing Profile; some works were written prior to UML-TP.

Nebut et al. [8] propose a method supported by a tool set, in which test cases for each of the different products of an SPL are generated from the same SPL functional requirements. The requirements are documented with high-level sequence diagrams. They define a behavioral test pattern (behTP) as a set of generic scenario structures representing a high-level view of some scenarios which the system under test may engage according to its specification. These scenarios are used to automatically generate test cases specific to each product.

Bertolino et al. [5] propose the PLUTO (Product Line Use Case Test Optimization) methodology for planning and managing the tests cases of an SPL. It is based on the requirements expressed in Use Cases to deal with specifications of PL requirements, called PLUCs (Product Line Use Cases). Kang et al. [7] use an extended sequence diagram notation to represent use case scenarios and variability. The sequence diagram is used as the basis for the formal derivation of the test scenario given a test architecture. Reuys et al. [10] present ScenTED (Scenario-based Test case Derivation), whose key idea is the creation of reusable test scenarios in domain engineering, which are later used in application engineering to obtain concrete test scenarios. Activity diagrams are used as test models from which test case scenarios are derived. The test case scenarios are specified in sequence diagrams without specifying concrete test data. A test case specification refines a test case scenario and comprises detailed test inputs, expected results, additional information and test scripts relevant to this scenario. Test case scenarios can be generated automatically, but test case specifications are developed manually.

Olimpiew el al. [9] use the PLUS method (Product Line UML-based Software engineering): customizable test models are created during software product line engineering in three phases: creation of activity diagrams from the use cases, creation of decision tables from the activity diagrams, and creation of test templates from the decision tables. Test data would then be generated to satisfy the execution conditions of the test template.

Dueñas et al. [6] define a metamodel that can cope with variability in testing based on the UML Testing Profile, but they do not use the profile. The variation points are formally defined by an algebraic expression, which is defined outside the UML limits.

Ziadi et al. [13] propose a UML profile for software product lines for class and sequence diagrams. Their extension introduces three elements: optionality, variation and virtuality, which are applied to interactions and lifelines. The main difference from our proposal is that theirs does not use the expressiveness brought for the Combined Fragment to define the variability in SPL. Moreover, this extension is applicable to SPL, whilst ours is an extension to be used in SPL testing.

3. Example: Lottery Software Product Line

"Lottery SPL" manages the bets and payments for different lottery-type games. There are many versions of lotteries. In this example, the types considered are: instant lottery, lotto and keno. Instant lottery is typically played using a scratch card, whose participants rub or scratch it to remove a coating that conceals one or more playing game pieces and related cash prize amounts. Generally, instant lottery tickets are printed on heavy paper or cardboard. The "Lotto" type is played by selecting a predetermined quantity of numbers in a range: depending on the right numbers, the prize is greater or lower. For example, one chooses six numbers from 1 to 49. The "Keno" type is basically played in the same manner, although it differs from "Lotto" games in that (i) the population of playing game pieces is even larger, e. g., integers from 1 to 80; (ii) participants can choose the quantity of numbers

that they want to match; and (iii) the number of winning game numbers, e. g., twenty, is larger than the number of a participant's playing numbers, e. g. two to ten. One example of the Keno type is Bingo. This SPL has several variation points, but for illustrative purposes this paper only analyzes the "Game of Chance" variation point for these three games.

4. Managing variability with the UML Testing Profile

This section first describes the proposed extension to manage variability in UML sequence diagrams, which is used to represent the Test Case Behavior. Then, it presents the corresponding extension to the UML Testing Profile.

4.1 Extension to UML 2.0 Interactions

When testing is performed, the traces of the system can be described as interactions and compared with those of earlier phases. The most visible aspects of an interaction are the messages between the lifelines. The sequence of the messages is considered important for the understanding the situation. Figure 3 shows the relations between Interaction and CombinedFragment in UML 2.0: a *CombinedFragment* defines an expression of interaction fragments and is defined by an interaction operator and the corresponding interaction operands. Through the use of *CombinedFragment,* the user will be able to describe a number of traces in a compact and concise manner. The notation for a CombinedFragment in a Sequence Diagram is a solid-outline rectangle. The operator is shown in a pentagon in the upper left corner of the rectangle. The semantics of a *CombinedFragment* depend on the *InteractionOperator* and can be: Alternative (alt), option (opt), paralell (par), negative (neg), etc. [14].

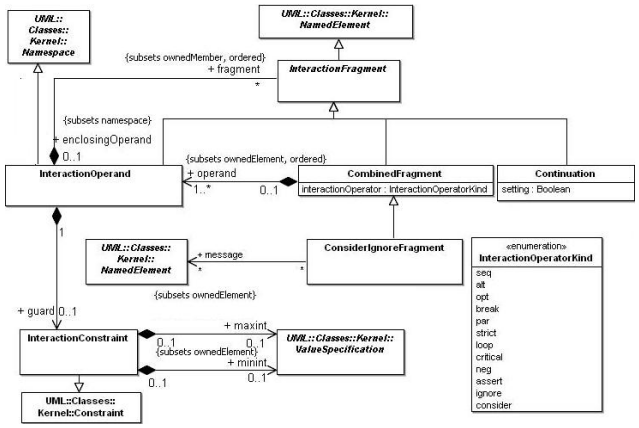

Figure 3. CombinedFragment in UML 2.0 [14]

To understand the way in which the extension is made, Figure 4 shows an example of how to use the extension proposed for UML interactions in the Lotteries Product Line described in Section 3.

The sequence diagram in Figure 4 represents a player who wants to know if his/her previously purchased ticket is a prizewinner. The *CombinedFragment* is stereotyped with a <<Variation Point>> alternative. This means that only one option can be chosen. Each variation in the *CombinedFragment* is stereotyped with <<Variation>>, representing each type of lottery: Keno, Instant or Lotto. In order to model the variability in the interaction diagrams, the UML concept of *CombinedFragment* is extended.

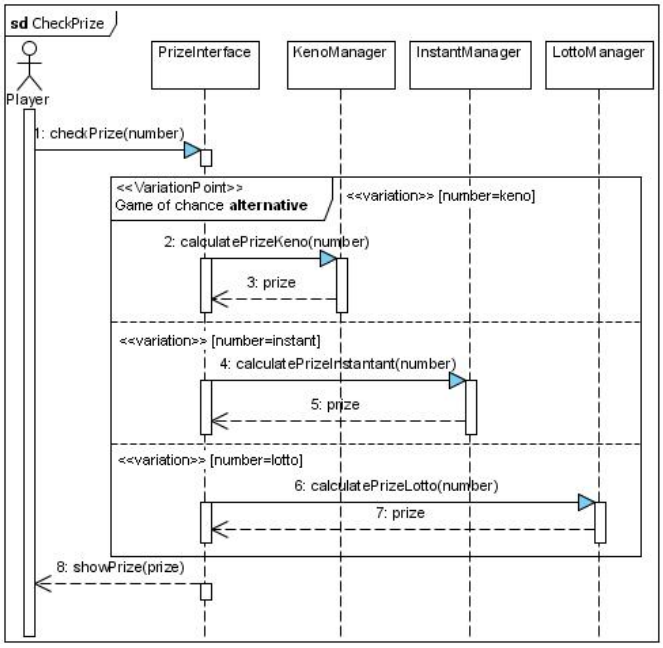

Figure 4. Example using the proposed profile

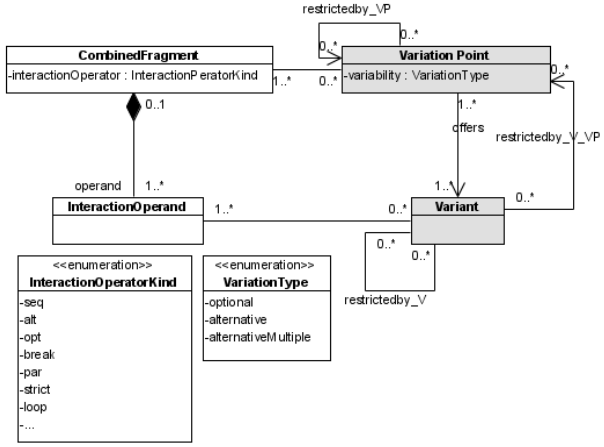

Figure 5. Proposed extension to UML 2.0 metamodel to manage variability

Figure 5 shows the proposed extension to the UML 2.0 metamodel to deal with variability in Combined Fragments. The way in which the relationship between variation points and variations are represented is inspired by OVM and shown in Figure 2.

Since the extension is made to be used in the UML Testing Profile, the addition of stereotypes is required. The stereotype <<Variation Point>> can be applied in *CombinedFragment* and the <<Variant>> stereotype can be applied to *InteractionOperand*.

Figure 6 shows the profile defined for UML Interactions.

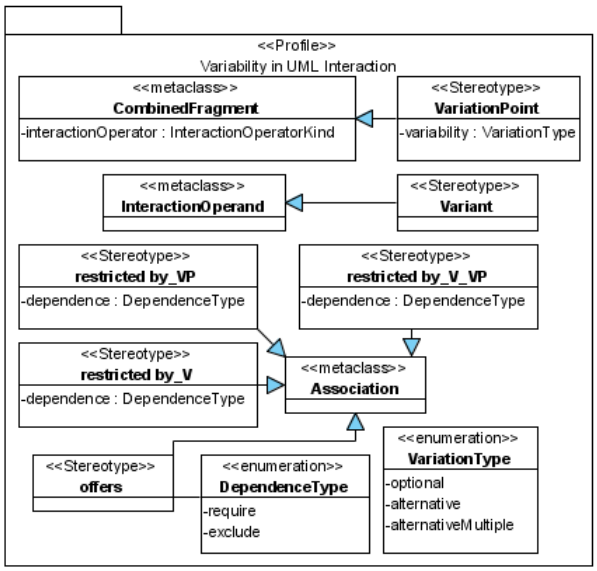

Figure 6. Profile proposed to UML Interactions

The semantic for the stereotypes defined in the profile proposed in Figure 6 is the following:

- A *CombinedFragment* with the <<Variation Point>> stereotype represents a variation point in the SPL and may be *optional*, *alternative* (i.e., XOR) or *alternativeMultiple* (i.e, OR).
- An *InteractionOperand* with the <<Variant>> stereotype represents a variant in the variation point (defined in its associated *CombinedFragment*).
- The *offers* relationship between the Variation Point and its Variants is represented in the profile with the <<offers>> stereotypes (applied to the *Association* class).
- The self relationship of Variant is represented in the profile with the <<restrictedby-V>> stereotype, applied to the Association class. The *DependenceType* indicates whether the Variant requires or excludes the other Variant in the relationship.
- The relationship between *VariationPoints* is represented in the profile by the <<restrictedby-VP>> stereotype, applied to the Association class. The *DependenceType* indicates whether the *VariationPoint* requires or excludes the other *VariationPoint* in the relationship.
- The relationship between *Variant* and *Variation Point* is represented in the profile with the <<restrictedbyV-VP>> stereotype, applied to the Association class. The *DependenceType* indicates whether if the *Variant* requires or excludes the *VariationPoint.*

Table 1 shows the constraints for the use of each stereotype defined in the proposed extension.

Applied to	Stereotype	Constraints
Combined-Fragment	Variation-Point	Applicable to InteractionOperatorKind alt or opt. When it is *alt, VariationType* must be *alternative* or *alternativeMultiple;* when *opt, VariationType* can be *optional.*
Interaction Operand	Variant	When applied to an *InteractionOperand,* the *VariantPoint* stereotype must be applied to the associated *CombinedFragment*
Association	offers	Only applicable to associations between *CombinedFragment* (stereotyped with *VariationPoint)* and *InteractionOperand* (stereotyped with *Variant)*
Association	restrictedby-V	Only applicable to associations between *InteractionOperand* and *InteractionOperand*, being both stereotyped with *Variant*
Association	restrictedbyVVP	Only applicable to associations between *InteractionOperand* (stereotyped with *Variant)* and *CombinedFragment* (stereotyped with *VariationPoint)*
Association	restrictedbyVP	Only applicable to associations between CombinedFragment and CombinedFragment, with both being stereotyped with VariationPoint

Table 1. Constraints for the profile proposed for UML interactions

4.2 Extension to the UML Testing Profile

In order to take full advantage of the UML extension described in the previous subsection and automate the case generation process, the extension of the UML Testing Profile is also required. This extension must include the mechanisms to describe the behavior of test cases, as well as the elements needed to support variability.

Thus, the UML-TP metamodel, which was shown in Figure 1, must be slightly modified to support variability, leaving it as in Figure 7. The way to deal with the relationships between variation points and variations is inspired by OVM (Figure 2).

To be used as a UML profile; the stereotypes shown in Figure 8 are defined to manage the variability in UML-TP.

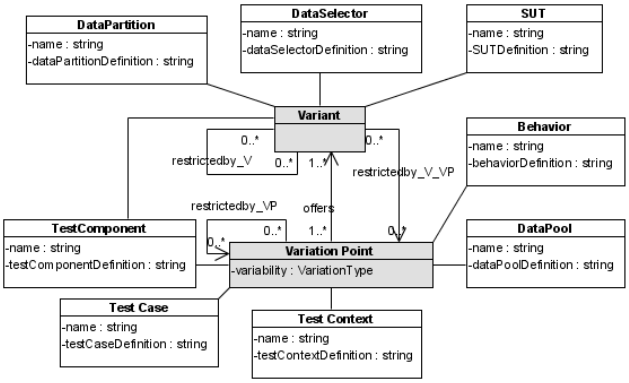

Figure 7. Proposed extension to the UML-TP metamodel

Using as example the *CheckPrize* functionality, whose sequence diagram was shown in Figure 4, a test model is constructed following the extension shown in Figure 8.

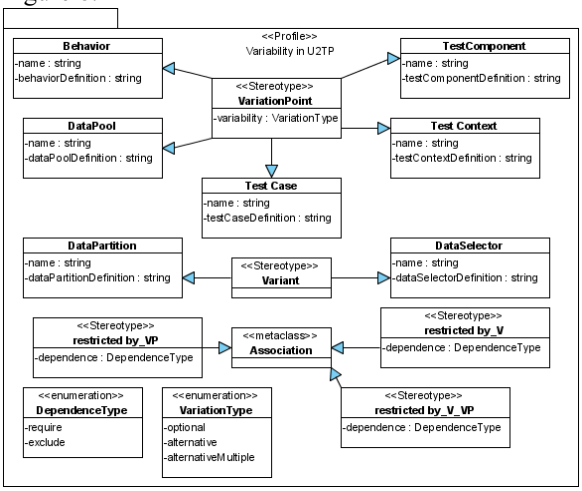

Figure 8. Proposed extension to UML-TP

Figure 9 shows the test architecture, which is part of the test model. The *TestContext* TCtxtCheckPrize groups the test cases for the *CheckPrize* functionality. It is stereotyped as <<*VariationPoint*>> because this functionality is variable and, therefore, its test cases will have to deal with its variability.

The testCheckPrize() *TestCase* tests the *CheckPrize* functionality and the TestPlayer *TestComponent* realizes its behaviour: to do so, it calls the testCheckPrizeGameOfChance() *TestCase*. This deals with variability and is stereotyped as <<*Variation Point*>>. To obtain the test data, the GameOfChance *DataPool* is stereotyped as <<*Variation Point*>>, which means that it handles the data for the variable part of the test case.

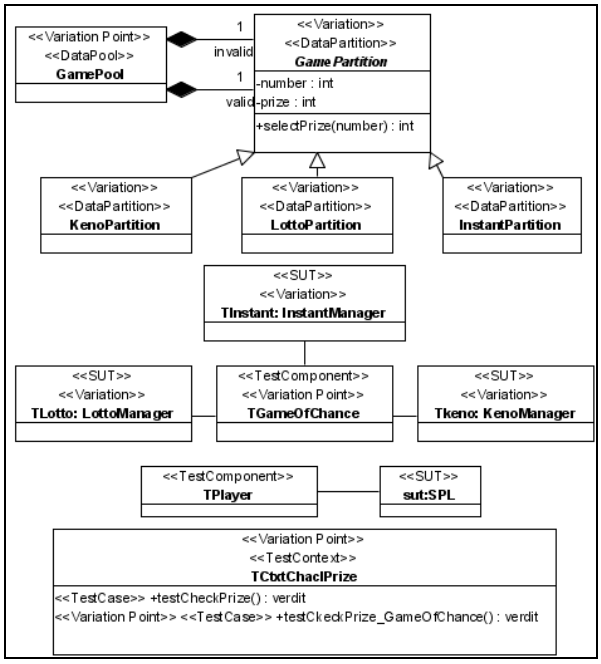

Figure 9. Test Architecture

Furthermore, three *DataPartitions* are defined to handle the data for each variant with the <<Variant>> stereotype.

Those *TestComponents* stereotyped with <<*Variation Point*>> realize the behavior for the testCheckPrizeGameOfChance() *TestCase;* to do so, they communicate with the *SUTs* stereotyped as <<*Variant*>>.

The TCtxtCheckPrize *TestContext* groups the test cases for the *CheckPrize* functionality. It is stereotyped with <<*Variation Point*>> because the functionality has variability, and its corresponding test cases must manage it.

These new stereotypes (see Figure 8) add a semantic to the existing stereotypes in UML-TP. The semantic of the defined extension is:

- *TestContext*: A testContext can be stereotyped with <<Variation Point>>, meaning that the testCases grouped in this testContext have variation points.
- *TestCase*: A testCase can be stereotyped <<Variation Point>>, which means that it tests a functionality with variability. The behaviour of this kind of test case was described in the previous section, and is modeled with a *CombinedFragment* stereotyped with <<Variation Point>>.
- *TestComponent*: A testComponent can be stereotyped with <<Variation Point>> or <<Variant>>, meaning that the testComponent can encapsulate the communication with the SUT, for the entire variation point or for one of its variants.
- *SUT*: The *SUT* can be stereotyped <<Variant>>, meaning that it realizes the functionality for this variant.
- *DataPool, DataPartition and DataSelector*: The *dataPool* can be stereotyped <<*Variation Point*>> and the *Data Partition* and the *DataSelector* as <<Variant>>. This means that the *dataPool* contains the specific data for a Variation point and that each *dataPartition* contains the data associated with one of its variants. The *dataSelector* selects the data in the *dataPartition* for a specific variant.

5. Example: Test Case Behavior

In this section, the test case behavior for the CheckPrize functionality is developed following the proposal defined in Section 4. Figure 10 shows the sequence diagram describing the test case behavior. This test case:

1) First, obtains the prize for the ticket from the SUT representing the entire SPL.
2) Second, calls the testCase *testCheckPrizeGameOfChance(ticket),* where the variability is tested. The prizes resulting from the first and second part are compared and the test case is considered to be successful if both prizes are equal.

Figure 11 shows the sequence diagram describing the behavior for the *testCheckPrizeGameOfChance* test case. This sequence diagram makes use of the extension to the UML interactions defined in Section 4.1.

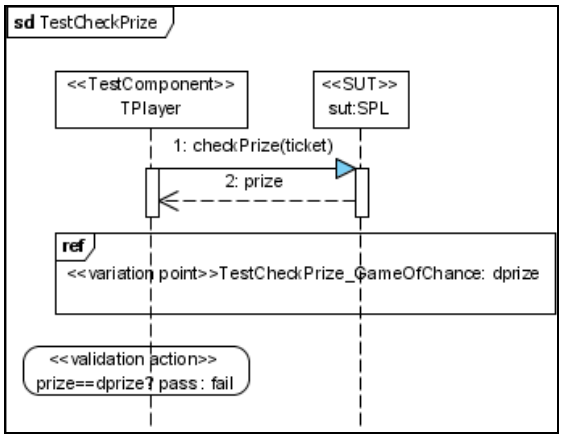

Figure 10. Test case *testCheckPrize*

The *testComponent* TGame-OfChance, stereotyped as <<Variation Point>>, realizes the test case behavior. To do this, and according to the rules mentioned in the previous section, it stereotypes a *CombinedFragment* GameOfChance as <<Variation Point>>, and defines three <<Variant>>, one for each game. If the game is Keno, then the *testComponent* calls the *SUT* TKeno (stereotyped as <<Variant>>), which provides the KenoManager functionality.

This SUT calculates the prize and returns it. The GamePartition *dataPartition* has the entrance data and the expected results for this test case. Then the *TestComponent* retrieves the expected result for this variant, which is shown as *dprize*.

The description is similar for other variants in the *CombinedFragment*. Finally, the *testCase* checks if the result returned for the SUT is equal to the expected one (the testCase will pass) or different (then, it will fail).

6. Conclusions

This paper defines a model to handle the variability in SPL testing, reusing the metamodel defined by the UML Testing Profile. Regardless of how the SPL was modeled, the test model can be defined to test the SPL functionality independently. In this approach, we take into account the modeling of the test case behavior with the sequence diagrams. An extension to the UML interactions to manage variability has been defined.

Currently, we are using these extensions to transform UML models (representing the SPL) to test models (represented by means of the UML-TP) automatically using Query/View/Transformation (QVT) [4], a model transformation language.

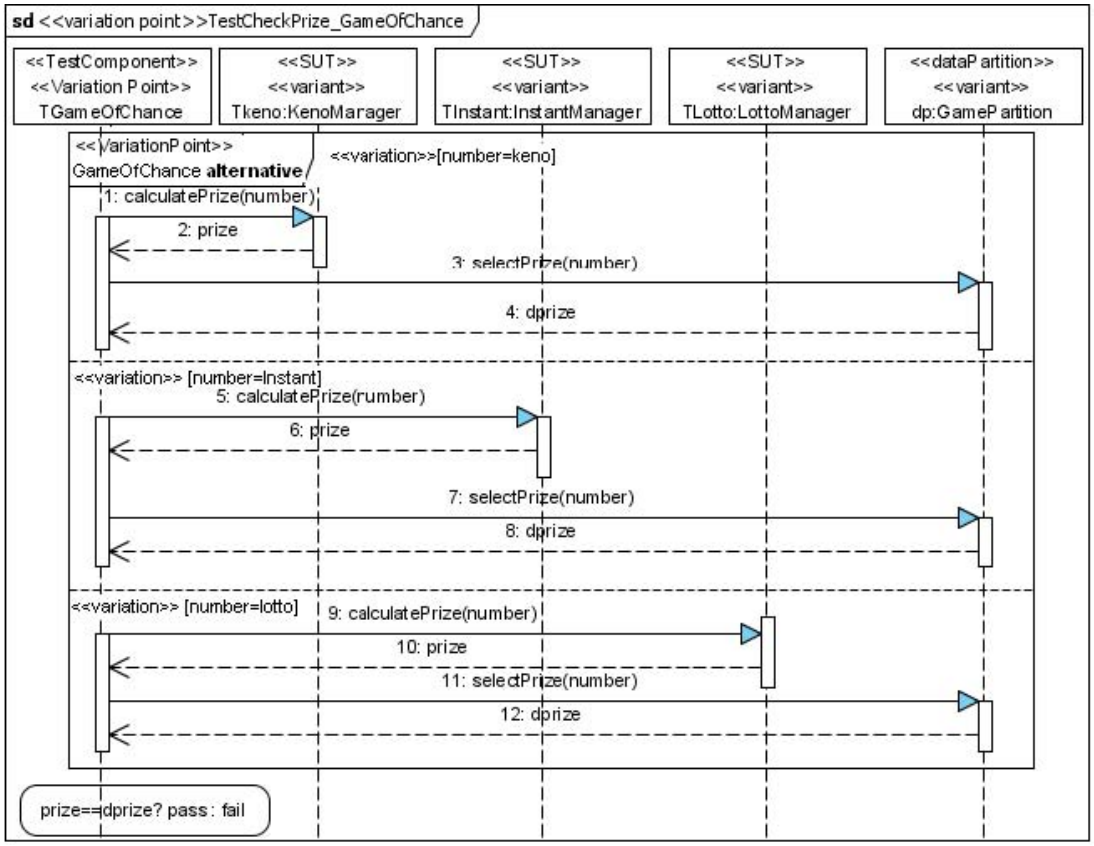

Figure 11. Test case testCheckPrizeGameOfChance

References

1. Clements, P. and L. Northrop, Salion, Inc.: A Software Product Line Case Study. 2002, DTIC Research Report ADA412311.
2. Trigaux, J. and P. Heymans, Modelling variability requirements in Software Product Lines: A comparative survey. EPH3310300R0462/215315, FUNDP–Equipe LIEL, Namur, 2003.
3. Baker, P., et al., Model-Driven Testing: Using the UML Testing Profile. 2007: Springer.
4. OMG, Meta Object Facility 2.0 Query/View/Transformation Specification. 2007.
5. Bertolino, A., S. Gnesi, and A. di Pisa, PLUTO: A Test Methodology for Product Families. Software Product-family Engineering: 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003: Revised Papers, 2004.
6. Dueñas, J., et al., Model driven testing in Product Family Context, in Model-Driven Architecture with Emphasis on Industrial Applications, U.o.T. CTIT Technical Report TR-CTIT-04-12, Editor. 2004, University of Twente: Enschede, the Netherlands,. p. pp 91-96.
7. Kang, S., et al., Towards a Formal Framework for Product Line Test Development. Computer and Information Technology, 2007. 7th IEEE International Conference on, 2007: p. 921-926.
8. Nebut, C., et al., Automated requirements-based generation of test cases for product families. Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on, 2003: p. 263-266.
9. Olimpiew, E. and H. Gomaa, Customizable Requirements-based Test Models for Software Product Lines. International Workshop on Software Product Line Testing, 2006.
10. Reuys, A., et al., Model-based System Testing of Software Product Families. Pastor, O.; Falcao e Cunha, J.(Eds.): Advanced Information Systems Engineering, CAiSE, 2005: p. 519–534.
11. OMG, UML testing profile Version 1.0, O.M. Group, Editor. 2005.
12. Pohl, K. and A. Metzger. Variability management in software product line engineering. in Proceedings of the 28th international conference on Software engineering. 2006: ACM New York, NY, USA.
13. Ziadi, T., L. Helouet, and J. Jezequel, Towards a UML Profile for Software Product Lines. 5th International Workshop on Software Product-Family Engineering (PFE'03, Proceedings), LNCS, 2004. 3014.
14. OMG, Unified Modeling Language, Superstructure, version 2.1.2. 2007.