TESTING IN SOFTWARE PRODUCT LINES
A Model based Approach

Pedro Reales Mateo, Macario Polo Usaola
Alarcos research group, Institute of Tecnologies and Information Systems
University of Castilla-La Mancha, Ciudad Real, Spain

Danilo Caivano
Department of Computer Science, University of degli Studi, Bari, Italy

Keywords: Model-based, Test Generation, Software Product Lines, Metamodel, Transformation.

Abstract: This article describes a model-driven approach for test case generation in software product lines. It defines a set of metamodels and models, a 5-step process and a tool called Pralíntool that automates the process execution and supports product line engineers in using the approach.

1 INTRODUCTION

Over the last few years, the construction of software based on the principles of product lines has emerged as a new development paradigm. According to Clements and Northrop (Clements and Northrop, 2002), 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". In SPL development, organizations work on two levels: (1) Domain Engineering, where both the common characteristics of all the products, as well as their variation points, are described and (2) Product Engineering, where specific products are built, transferring the common characteristics (described on the top level) to them and appropriately applying variability. Thus, variability management plays a central role in SPL and constitutes a new challenge when compared to classic software engineering.

Due to the nature of SPL (the complexity inherent to variability management, need for the future reuse of design artefacts, etc.), most works dealing with SPL require an intensive use of models (Czarnecki et al., 2005), which must be appropriately annotated with variability labels. At some time, some kind of transformation must be applied, both to deal with artefacts on the domain engineering level (translating, for example, a design model into test models), as well as to obtain product engineering artefacts from the domain engineering ones.

This article describes an approach for SPL design and test case generation. It is made up of:

- A set of models and metamodels that were built almost from scratch;
- A 5-step process that guides the Product Line Engineer in SPL modelling and test case generation;
- A tool, PralínTool, that automises the process execution.

The goal is to automate the derivation of test cases in SPL contexts. This approach offers agility due to the complete control of metamodels and algorithms, which can be quickly adapted or modified to incorporate the implementation of new ideas. One important obstacle in testing research is oracle automation. In order to solve the oracle problem, the approach supports the use of states and special notations, which would allow the partially automatic generation of oracles. Since SPL requires some effort to guarantee reuse, traceability, the adoption of tools and the application of other good practices, the context is excellent to investigate the possibilities of model-driven techniques to achieve

Reales Mateo P., Polo M. and Caivano D..
TESTING IN SOFTWARE PRODUCT LINES - A Model based Approach.
DOI: 10.5220/0003476200460054
In *Proceedings of the 13th International Conference on Enterprise Information Systems* (ICEIS-2011), pages 46-54
ISBN: 978-989-8425-55-3

its automation, since these techniques implicity assure characteristics such as traceability, reuse, automation and other good chracteristics for the engineering process.

The following section analyses the most significant works relating to testing in SPL and the oracle problem. Section 3 presents an example, which will be used to illustrate the proposal. After that, the approach is presented in Section 4. Finally, we draw our conclusions and present future lines of work.

2 RELATED WORK

Testing in the context of SPL includes the derivation of test cases for the line and for each specific product, exploiting the possibilities of variability to reduce the cost of creating both the test model and the line test cases. This includes their instantiation to test each product. In general, testing artefacts are derived at domain engineering level, and they are transformed for specific products afterwards. Almost all the proposals to generate tests for SPLs define their own models to represent the testing artefacts and variability in the test model.
The next paragraph summarises the most important works.

Nebut et al. (Nebut et al., 2003) propose a pragmatic strategy in which test cases for each of the different products of an SPL are generated from the same SPL functional requirements. Source artefacts are parameterised use cases annotated with contracts (written in 1st-order logic) that represent pre- and post-conditions. Bertolino et al. (Bertolino et al., 2004) propose a methodology based on the category-partition method named PLUTO (Product Line Use Case Test Optimisation), which uses PLUCs (Product Line Use Cases). A PLUC is a traditional use case with additional elements to describe variability. For each PLUC, a set of categories (input parameters and environment description) and test data is generated. Kang et al. (Kang et al., 2007) 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. (Reuys et al., 2005) present ScenTED (Scenario-based Test case Derivation), where activity diagrams are used as test models from which test case scenarios are derived. Olimpiew and Gomma (Olimpiew and Gomaa, 2006) describe a parametric method, PLUS (Product Line UML-based Software engineering). Here, customisable test models are created during software product line engineering in phases.

Whether in the context of software products lines or in the traditional context, one of the most important tasks in software testing is the definition of the oracle, which is the mechanism provided for a test case to determine whether it has found a fault. According to Baresi and Young (Baresi and Young, 2001), all the methods for generating tests depend on the availability of oracles, since they are always required to determine the success or failure of the test. For Bertolino (Bertolino, 2007), an “ideal oracle” realistically is an engine/heuristic that can emit a pass/fail verdict over the observed test outputs. Thus, the automation of the oracle is one the most important difficulties in testing research (Offutt et al., 2003), since there is no a known method for its generic description and, in practice, it must always be manually described. The work by Baresi and Young (Baresi and Young, 2001) (published in 2001) is a complete analysis of the state–of-the-art about the oracle problem. Most of the proposals they analyse refer to the insertion of assert-like instructions in the source code. Later, other works have made proposals to solve this problem using other techniques such as artificial neural networks (Jin et al., 2008) or metamorphism (Mayer and Guderlei, 2006). The automation of the oracle has special significance in SPL, since meaningful portions of the system are analysed and developed at the domain engineering level, which includes the definition of their tests. Therefore, it is quite interesting to apply reusing and traceability not only to develop artefacts, but also to test them, including oracle descriptions.

In summary, software engineering communities have everything ready to work intensively with model driven approaches in SPL, but none of existing proposals for testing in SPL automate their transformations. Also, the problem of oracle automation still remains and it is important to propose ideas for its resolution. The SPL context offers an excellent opportunity to improve classic software engineering practices, development methods and techniques for testing. The joint use of SPL and model-based testing is very propitious for improving test and oracle automation. In the approach presented in this document, a set of metamodels and a process of seven steps have been developed with three goals: 1) generate test artefacts automatically at domain level with variability, 2) remove variability and generate automatically executable tests for specific products, and 3) support the generation of oracles through states and special

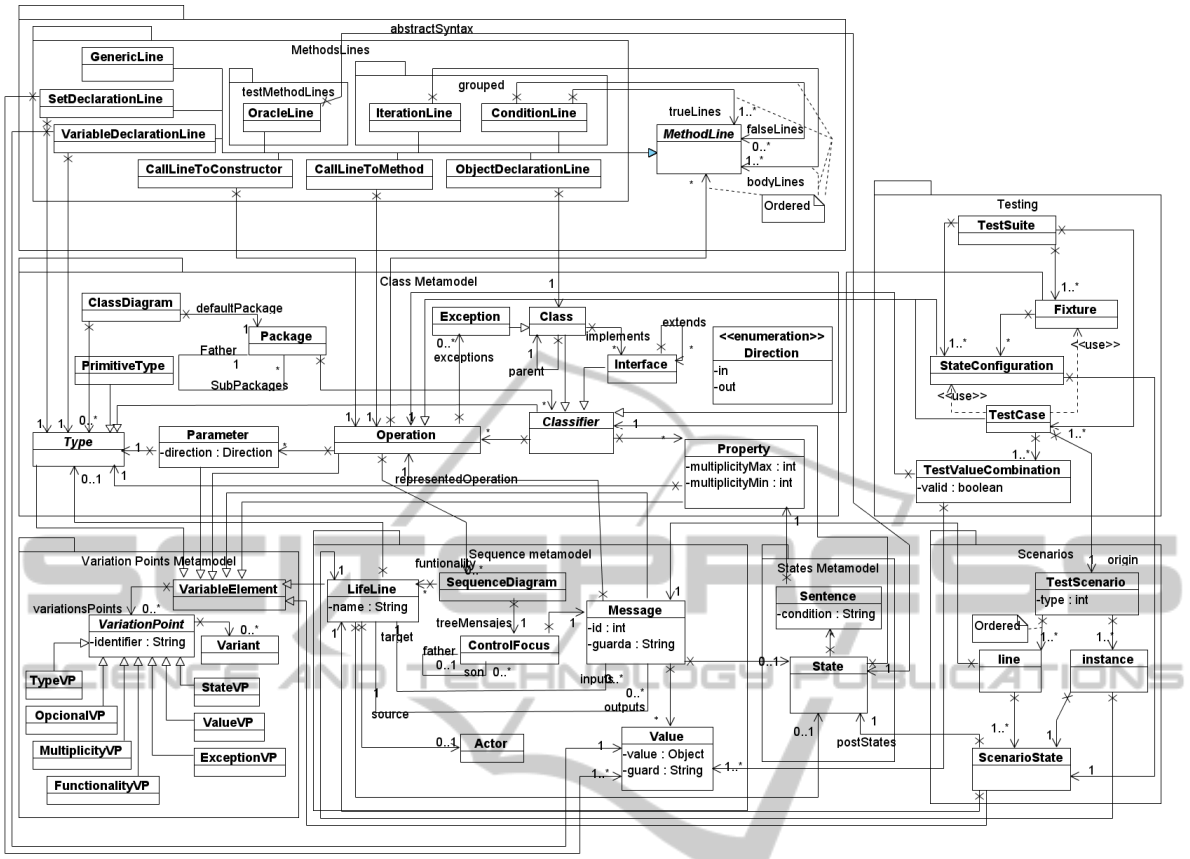

Figure 1: Metamodel for designing software product lines.

notations in the models. The main advantage of this technique is that a complete framework is presented (from the design of the line to the execution of tests) and this framework supports the automatic generation of oracles, thanks to special notations.

3 PROPOSED APPROACH

The proposed approach is made up of:

- A **metamodel** that was built almost from scratch;
- A 5-step ***process*** that guides the Product Line Engineer in PL modelling and test case generation;
- A **tool**, called PralínTool, that automates the process execution.

It uses class and sequence diagrams as the main element for representing SPL and, later, for generating test cases. In order to enable the automation (and the future extension) of both the process and the supporting tool, these elements were represented by means of metamodels.

3.1 SPL Metamodels

Figure 1 shows the defined metamodels. Here, Variability is provided by the *Variation Points Metamodel* package, with the element *VariableElement* and its specialisations. A *VariableElement* has a collection of *VariationPoint*, which defines the type of variability and the range of possibilities through its *Variant*.

This metamodel makes it possible to represent all the elements required to design a product line according to our principles, removing the complexity inherent to the UML 2.0 standard metamodel (OMG, 2007), although also losing part of its expressiveness. Currently, the metamodel only supports one type of event in sequence diagrams, instead of the wide variety of messages allowed by UML 2.0 (call, creation, signals, etc.). In fact, our metamodel was built from scratch and, thus, it is not completely based on the UML specification. This can be a thread in a real context, but due to the

flexibility of our metamodels, they can be adapted to new modelling requirements.

3.2 The Process

The process guides the Product Line Engineer in PL modelling and test case generation. Its general structure is shown in Figure 2 and it is implemented in PralínTool (Section 3.3).

It is made up of 5 different steps:

1) Step 1: Product Line modelling;
2) Step 2: Sequence diagram enrichment;
3) Step 3: domain-level test scenario generation;
4) Step 4: domain level test case generation;
5) Step 5: product level test case generation and automation.

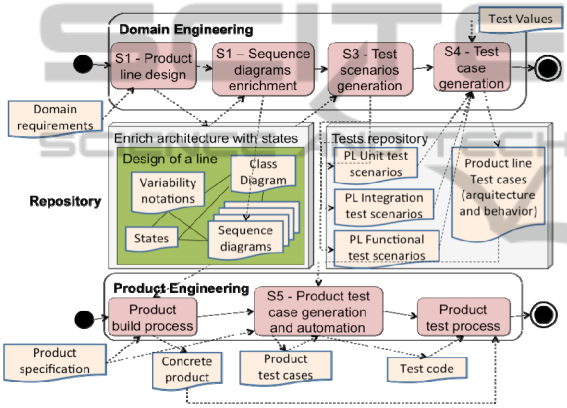

Figure 2: General description of the process.

3.2.1 Step 1: Product Line Modelling

At the domain engineering level, the software engineer models the software product line, which includes class structure and sequence diagrams. These models also contain variability specification.
In order to illustrate this and the following steps, a product line consisting of a distributed, client-server system for playing board games has been developed. These kind of games share a broad set of characteristics, such as the existence of a board, one or more players, possibly the use of dice, the possibility of taking pieces, the presence or absence of cards, policies related to the assignment of turns to the next player, etc. Currently, this SPL works with four types of board games: Chess, Checkers, Ludo and Trivial. Since it is impossible to show the whole system in this paper, it instead shows some of the variation points and variants identified for this SPL, which are described according to the Orthogonal Variability Model (OVM) graphical notation (Pohl et al., 2005). This notation identifies each variation point with a triangle and each variant with a rectangle. Arrows are used to include restrictions.

Figure 3 shows four variation points: Game, corresponding to one of the possible games supported (Chess, Checkers, Ludo or Trivial); Opponent indicating whether the player is playing against the computer or another online human player; Players where the minimum number of players is 2, but some games have the option of being played by more players; and Type, which depends on whether the games use dice or quiz the player.

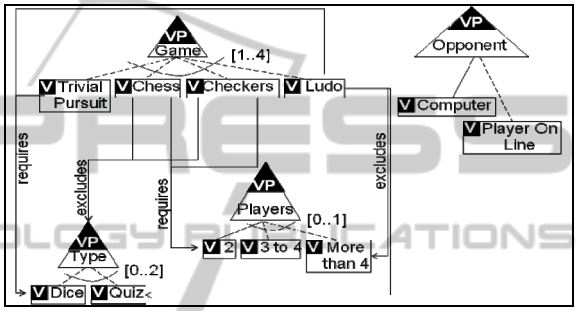

Figure 3: Variation points and variants.

In this system, one of the clearly variable use cases is “Piece movement”, which is executed on the server when a client sends a message corresponding to the movement of a piece. Figure 4 shows the sequence diagram that describes the functionality of “Piece movement”. This sequence diagram has special notations, which are explained in the following section, which describes the treatment given to the modelling, development and generation of test cases given to this SPL with our approach, which is illustrated using the functionality “Piece Movement”.

3.2.2 Step 2: Sequence Diagram Enrichment

In this proposal, a test scenario is a sequence of method calls, which must be executed to perform a test. Test scenarios are generic, in the sense that they belong to the domain engineering level of the SPL design. Since an interesting aspect is the generic description of oracles (for later inclusion in specific products, in the form of specific oracles), test scenarios must include the state of the different objects involved in the scenario before and after each method call. For this, both the messages and the objects are annotated with information about the states.

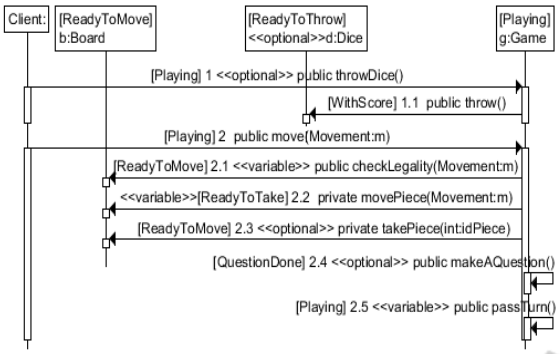

Figure 4: Normal event flow of Piece movement for the line, draw with PralínTool.

Figure 4 shows the sequence diagram corresponding to the normal flow of events of the Piece movement use case. It includes information about the expected states of each object (within square brackets) as well as variability labels (as stereotypes). For example, before entering this scenario, the d:Board instance must be ReadyToMove (see the annotation between the square brackets in the instance). These annotations are supported by the *States Metamodel* package.

States are described by means of Boolean expressions, written as a function of the fields and methods in the corresponding class. Since some of these expressions will affect some products but not others, states can be variable too, which will require different types of processing when specific test cases must be produced (this transformation is presented in Section 4.4). Table 1 shows an example of a state for the *Game* class: when the game is in *Playing*, the instance must have a board, the number of clients must be greater than zero, a player must have the turn, there is no still winner and, depending on the product, dice may exist.

Table 1: Description of a state in the Game class.

Playing	this. clients.size() > 0 this.pWithTurn != null this.clients.contains(this.pWithTurn) == true this.winner == null this.dice !=null <<*Optional*>>

3.2.3 Step 3: Domain-level Test Scenario Generation

The elements to model test scenarios are provided by the *Scenarios* package. The main element is *TestScenario*, which has an ordered set of *lines* that represents a sequence of *messages* that will be executed and a set of *instances* that represents the elements that will execute each message. All these elements include states for the future generation of oracles.

For generating scenarios, three transformations have been developed (due to a lack of space, the pseudocode of the transformations is not shown):

1. Unit test scenarios consider the messages producing a single object in the sequence diagram. The scenarios only keep a method, together with the states which annotate the instance (pre-state) and the message (post-state). With the goal of having all the objects in the correct state, the test scenario also knows the pre-state of all the objects involved in the method execution.
2. Integration test scenarios test the interactions between any two connected objects (i.e., one instance sends a message to the other). The scenario saves: (1) the method of the first instance whose execution produces its interaction with the second one; (2) the post-states of both instances. As with the unit test algorithm, the pre-states of all instances involved in the scenario must be taken into account to ensure that the scenario is, in fact, reproducible.
3. Functional test scenarios test the system from an actor's point of view. Thus, the scenario executes the messages arriving from an actor to the system, which is considered as a black box. In addition to these messages, the scenario must also hold the corresponding state annotations, both in the instances and in the events.

Table 2 groups the elements of the functional test scenario generated with the third transformation corresponding to the sequence diagram of figure 4; it holds all the instances involved in the execution of the functionality. Note that, in this example, since it is a functional scenario, only the messages arriving from an actor client appear in the message sequence (in the sequence diagram, only *throwDice* and move come from an actor). As can be seen, pre- and post-states are also saved. Note that this kind of description is supported by the metamodel.

Table 2: Functional test scenario of *Piece movement.*

LifeLines (instances)	Pre-state
g:Game	Playing
d:Dice «Optional»	ReadyToThrow
b:Board	ReadyToMove
Messages	**Post-state**
1:throwDice «Optional»	g:Game Playing d:Dice WithScore
6:move	g:Game Playing b:Board ReadyToMove «Variable»

Up to now, the process has produced test scenarios (representations of interesting test situations at the domain-engineering level). Now, these scenarios must be translated into specific test cases for the line and for the specific products of the line. This requires two steps: (1) still on the domain level, a test architecture must be generated, which takes into account variability, and the behaviour of each test case must be obtained; and (2) on the product-engineering level, variability must be resolved for each product, and the test architecture and behaviour must be translated into specific product test cases.

3.2.4 Step 4: Domain Level Test Case Generation

In this step, specific test cases are obtained from the previously obtained test scenarios. The main difference between test scenarios and test cases is that the latter describe specific execution situations and, therefore, have specific test data and oracles. Moreover, the test suite is part of the test architecture of the system. Since the package Scenarios of the metamodel supports the three types of scenarios described, only a single transformation is required to obtain specific test cases from the three types of scenarios.

The architecture of the tests (supported by the package Testing) has a *TestSuite* as its basic element, which contains all the required elements to compose the tests: it has a set of *Fixture* elements (representing types under test or required types), a set of *TestCase* elements and a set of configuration methods (*StateConfiguration*, which is used to put the objects in the required state).

An instance of the test architecture stores the information required to generate specific test cases. A transformation for generating the test architecture has been developed. The transformation takes the set of test scenarios and the class diagram as inputs and produces a *TestSuite*. It adds the required fixtures to represent the lifelines, as well as the *StateConfiguration* elements required to put each Fixture into the specific required state. Finally, it obtains the specific test cases corresponding to each test scenario passed as a parameter, also combining the test data, and generates methods to check if the post-states of the fixtures are correct after the test executions. This transformation makes use of the combination algorithms implemented in *testooj* (*All combinations*, *Each choice* and several kinds of *Pair wise* (Polo et al., 2007)). Figure 5 shows the architecture generated from the sequence diagram in the figure 4.

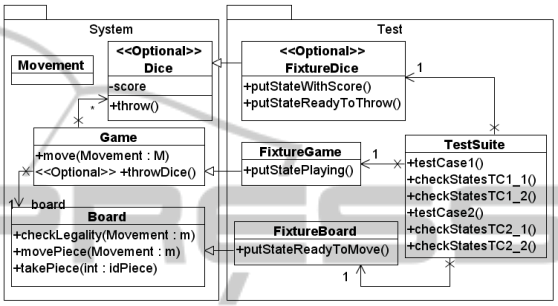

Figure 5: Test classes generated from the test scenario.

Once the test architecture has been instantiated, a further step obtains the functionality of each test case. As other authors have shown (i.e., (Baxter et al., 1998, Khatchadourian et al., 2007)), our metamodel contains a package for object-oriented source code (package *AbstractSyntax*), which actually represents the abstract syntax tree of the test cases. Thus, a new transformation has been developed to translate the ordered sequence of method of each test scenario into an abstract syntax tree for each test case.

This transformation analyses the generated scenario and, for each test case, includes in the abstract syntax tree a message to put each fixture in the correct pre-state. Then, the transformation adds the sequence of messages of the scenario with a concrete combination of test values, and after each message of the sequence, a message to check the post-states is added (these messages are considered the oracles of the test). Figure 6 shows the generated functionality of a test case (Note that the argument of message 6 is a concrete value. The metamodel supports this but it cannot be represented graphically yet).

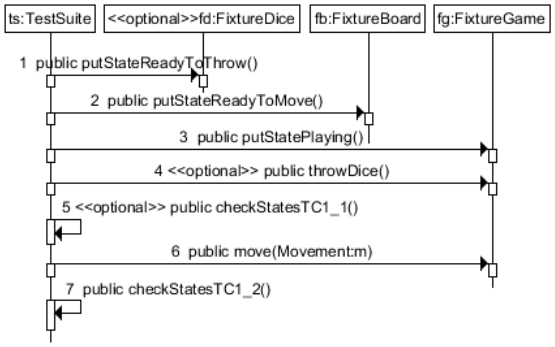

Figure 6: Functionality of test case 1.

3.2.5 Step 5: Product Level Test Case Generation and Automation

In this step, executable test cases are obtained from their respective models (actually, instances of the metamodel), and executed in two substeps (Figure 7): (1) generation of cases for a specific product, which involves the removal of variability; (2) generation of executable code for the desired technology.

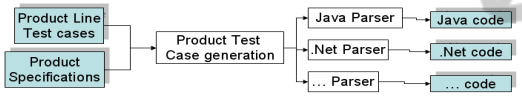

Figure 7: Schematic view of executable test case generation.

In the first substep, variability is removed from domain specifications, proceeding in the same way as the variability is removed for generating a product (not a test case) from the line design. These specifications are made up of the different selected variants for each variation point in the line. As an example, table 3 shows all the variation points for the board games product line (left column), together with the selected variants for the specific product of *Chess*.

Table 3: Selected variants for the *Chess* product.

Class diagram	
Variation point	*Selected variant*
Game	chess
Players	2
Opponent	Player
Type	(*excluded*)

Figure 8 shows the functionality of the generated test case in figure 6 after removing the variability. It can be seen that the features related to Dice have been removed because there are no dice in chess.

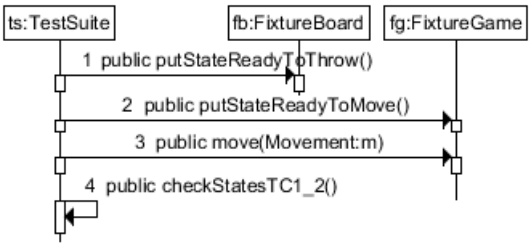

Figure 8: Test case without variability.

```
public class TestSuite{
  private FixtureGame fg;
  private FixtureBoard fb;
  public void testCase2(){
    Object v1 = new Movement(14,18);
    fg.putStateTC1();
    fb.putStateTC1();
    fg.move(v1);
    this.chackStatesTC1_2();
  }
  public void checkStatesTC1_2(){
     assertTrue(fg.clients.size()>0 &&
          fg.pWithTurn!=null &&
fg.clients.contains(fg.pWithTurn)==true &&
      fg.winner==null);
    assertTrue(fd.pieces.size() >0);
  }
  public void testCase2(){…}
  public void checkStatesTC2_2(){…}
}
```

Figure 9: Source code of a Java executable test case.

Finally, the executable code of the test cases is obtained from the transformation of the abstract syntax tree instances (package abstractSyntax of the metamodel), but now taking into account the specific characteristics of the selected technology, which can be either Java (Figure 9) or .NET.

3.3 A Short Overview of PralínTool

Figure 10 shows an aspect of PralínTool, the tool we developed to support test case generation in SPL. With the tool, it is possible to include capabilities for describing use cases with a structured template, which makes the almost automatic transformation of scenarios to sequence diagrams easy. States can be also defined for each class in the system, which are also specified in a hierarchical tree. The sequence diagram editor enables the annotation of the event flows with variability labels. The generation of test scenarios and test cases is supported by the implementation of the previously described algorithms.

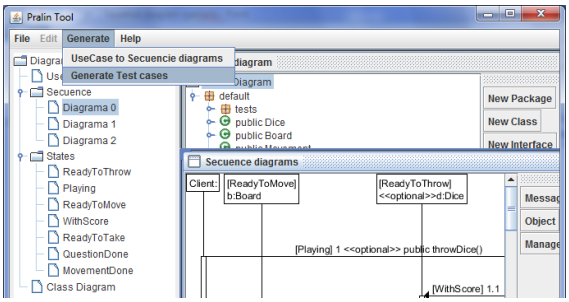

Figure 10: A view of PralínTool.

4 CONCLUSIONS AND FUTURE WORK

This paper has presented an approach for automating the generation of test cases in SPL. A set of metamodels to design class and sequence diagrams has been developed. These metamodels allow variability and can include special notations to generate oracles for the tests. The approach is a complete framework that makes it possible to design an SPL and to generate test models and executable tests. The entire process takes the oracle problem into account. To solve this, the developers can define states and relate them to sequence diagram messages . These relations (represented as special notations in brackets) are used to generate oracles for the tests.

However, the approach has some disadvantages, because only sequence and class diagrams (similar to UML) can be defined, which results in a loss of expressiveness. But, due to the flexibility of the metamodels and transformation algorithms, they can easily be modified and extended, so they can be adapted to new expressive necessities with no difficulties.

The strict practices in SPL software development make it possible to obtain new and additional knowledge for software engineering. In particular, the intensive use of models and tools can enrich knowledge about MDA. In the case of testing, it is relatively easy to experiment with algorithms and ideas with self-metamodels, before passing them on to a standardised approach, whose elements and tools will likely be adopted by the industry soon. In our opinion, the solution to this problem, which has been the subject of research for many years, is now closer to being resolved, especially today, when significant effort is being devoted to the model-driven discipline. In general, our future work will continue to incorporate new techniques for model transformation and test automation in SPL, since it is easy to extrapolate the results obtained here to other contexts.

ACKNOWLEDGEMENTS

This work is partially supported by the ORIGIN project, *Organizaciones Inteligentes Globales Innovadoras* and FPU grant AP2009-3058.

REFERENCES

Baresi, L. & Young, M. (2001) Test oracles. Dept. of Computer and Information Science, Univ. of Oregon.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M. & Bier, L. (1998) Clone Detection Using Abstract Syntax Trees. *International Conference on Software Maintenance.*

Bertolino, A. (2007) Software testing research: Achievements, challenges, dreams. *International Conference on Software Engineering.* IEEE Computer Society Washington, DC, USA.

Bertolino, A., Gnesi, S. & di Pisa, A. (2004) PLUTO: A Test Methodology for Product Families. *Software Product-family Engineering: 5th International Workshop, PFE 2003, Siena, Italy, Nov. 4-6, 2003.*

Clements, P. & Northrop, L. (2002) Salion, Inc.: A Software Product Line Case Study. DTIC Research Report ADA412311.

Czarnecki, K., Antkiewicz, M., Kim, C., Lau, S. & Pietroszek, K. (2005) Model-driven software product lines. *Conference on Object Oriented Programming Systems Languages and Applications.* ACM New York, NY, USA.

Jin, H., Wang, Y., Chen, N., Gou, Z. & Wang, S. (2008) Artificial Neural Network for Automatic Test Oracles Generation. *Computer Science and Software Engineering, 2008 International Conference on.*

Kang, S., Lee, J., Kim, M. & Lee, W. (2007) Towards a Formal Framework for Product Line Test Development. *Computer and Information Technology, 2007. 7th IEEE Int. Conference CIT*, 921-926.

Khatchadourian, R., Sawin, J. & Rountev, A. (2007) Automated Refactoring of Legacy Java Software to Enumerated Types. *International Conference on Software Maintenance (ICSM 2007).* Paris (France).

Mayer, J. & Guderlei, R. (2006) An Empirical Study on the Selection of Good Metamorphic Relations. *Proceedings of the 30th International Computer Software and App. Conference (COMPSAC'06)-V 01.* IEEE Computer Society Washington, DC, USA.

Nebut, C., Pickin, S., Le Traon, Y. & Jezequel, J. (2003) Automated requirements-based generation of test cases for product families. *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, 263-266.

Offutt, A. J., Liu, S., Abdurazik, A. & Amman, P. (2003) Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 25-53.

Olimpiew, E. & Gomaa, H. (2006) Customizable Requirements-based Test Models for Software Product Lines. *International Workshop on Software Product Line Testing*.

OMG (2007) Unified Modeling Language: Superstructure. Version 2.0.

Pohl, K., Böckle, G. & Van Der Linden, F. (2005) *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer.

Polo, M., Piattini, M. & Tendero, S. (2007) Integrating techniques and tools for testing automation. *Software Testing, Verification and Reliability,* 17, 3-39.

Reuys, A., Kamsties, E., Pohl, K. & Reis, S. (2005) Model-based System Testing of Software Product Families. *Pastor, O.; Falcao e Cunha, J.(Eds.): Advanced Information Systems Engineering, CAiSE*, 519–534.