Welcome

Plenary Talks

Technical Program

Committees

Reviewers

Author Index

Session Chair Index

Help

IEEE-ICIT'12

Athens - Greece

19-21 March 2012

International Conference on Industrial Technology

Sponsor: IEEE Industrial Electronics Society

IEEE Cat. No: CFP12CIT-CDR
ISBN: 978-1-4673-0341-5

ICIT 2012 CO-CHAIR'S WELCOME MESSAGE

Welcome to ICIT'2012

It is our pleasure to welcome all participants to ICIT'2012, the 2012 IEEE International Conference on Industrial Technology, Athens, Greece, which takes place at the Divani Apollon Palace & Spa Hotel, Athens.

ICIT'2012 is co-sponsored by the IEEE Industrial Electronics Society (IEEE-IES), the University of Patras and supported by the Industrial Systems Institute (ISI/RC ATHENA), Greece.

Founded in 1964 the University of Patras is the third Greek academic institution in terms of numbers of students, professors and other staff, and academic departments. It represents a dynamic academic education and research centre with approximately 22.000 undergraduate students, 2.000 post-graduate students, 700 teaching staff, 400 administrative personnel and 400 teaching and research assistants. The University of Patras is a major international centre for highest tertiary education with a proven high track record not only in teaching but also in research. Its twenty-two Departments offer a wide range of undergraduate courses as well as an expanding range of taught and research-based postgraduate degrees which reflect a balanced academic environment comprising science and technology as well as health sciences and humanities. Facilities for academic work are excellent and opportunities for social life are numerous and exciting. The University of Patras participates in a large number of European and international educational and research programmes and its forefront scientific research has been recognized internationally.

The Industrial Systems Institute (I.S.I.) is a dynamically growing research institute founded in 1998, located in Patras, Greece. It is a research institute of Research Center ATHENA and operates under the supervision of the Greek Ministry of Development. I.S.I. has strong ties with the University of Patras. Its activities and expertise covers all aspects of ICT that are used in the industrial / enterprise environment.

The scope of the ICIT'2012 being very wide, here we have a rare opportunity to interact with a wide spectrum of technical experts and we are sure you will find this conference very useful. ICIT'2012 features an excellent technical program, which includes keynote and plenary presentations delivered by leading authority from the industry and academia. We are fortunate to have outstanding keynote speakers, who kindly agreed to contribute to ICIT'2012 Program: Prof. Gerard-Andre Capolino, Prof. Constantinos Sourkounis and Prof. Armando Walter Colombo. The keynote addresses will present the state of-the-art and challenges on specific areas of research.

The quality of the program is a combination of quality submissions and diligent work of the members of the International Program Committee. Despite the current world economic crisis, this conference received more than 327 papers, 233 were provisionally accepted, and 194 were included in the program. It was a considerate effort to conduct the peer review process in a very short time. The meticulous and thorough review process was conducted according to IEEE Industrial Electronics Society's standard. ICIT'2012 has created records on almost all fronts. The 194 papers from 40 countries contained in the final program cover all continents and with more than 88% papers from outside Greece is proved that this conference to be truly international.

The final program of ICIT'2012 consists of a total of 38 sessions, covering the 8 regular Tracks and 4 Special Sessions. The subjects of the 8 Tracks are Control Systems and Computational Intelligence, Factory Automation and Industrial Informatics, Robotics and Mechatronics, Embedded and Cyberphysical Systems in Industrial Applications, ICT for Smart Grids and Renewable Energy and Power Systems, Electrical Machines and Drives and Sensors, Instrumentation and Signal Processing. Added to this is a set of four Special Technical Sessions (Condition Monitoring & Diagnostics, Smart Sensors & Microsystems for Industrial Applications, Model Based Testing and Engineering, From Data to Information: Methods and Industrial Technologies, Control & Optimization).

In addition, the conference program includes a welcome reception in the evening of Monday, March 19. The conference banquet will take place on Tuesday, March 20, at the "Vorres" folk and modern art museum, with a collection that covers 3000 years of Greek history.

We would like to thank all the authors of submitted papers and all the members of the Program Committee as well as additional reviewers, who spared their valuable time to review all submitted papers in a timely manner. Also, we would like to acknowledge the contribution of all members of the several committees that contribute to putting together such an exciting program. Our sincere thanks to all the Track Chairs, who offered their best professional support. Special thanks to Prof. Gerard-Andre Capolino, Prof. Constantinos Sourkounis and Prof. Armando Walter Colombo for their valuable keynote addresses to ICIT 2012 participants.

Finally, we are grateful to all members of the Local Organizing Committee who have spent their time generously to help in the organization of the event. In addition, an event of this size cannot be organized without the help of a large number of volunteers, who we thank warmly.

The success of any conference depends on the quality of the program and participation of people. We thank you all for being here. We trust that you will find the technical program intellectually stimulating and your stay in Athens really enjoyable. We are certain that the atmosphere of Athens, one of the most ancient European cities, with its historical centre, which is classified as World Heritage, will provide a stimulating environment for ICIT'2012 participants, allowing exchange of scientific ideas, updating information about new developments, and increasing international collaboration and friendship. Hopefully, it will be possible for several of the attendants to add some extra days for sightseeing and recreation in Athens.

We welcome you in a technically and intellectually stimulating conference. We also wish you a nice stay in Athens enjoying the rich Greek culture and the famous Greek hospitality!

We are sure you will find ICIT'2012 memorable!

Stavros Koubias and Luis Gomes

ICIT'2012 General Co-Chairs

Dimitrios Serpanos, Juergen Jasperneite, and Yousef Ibrahim

ICIT'2012 Program Co-Chairs

John Gialelis and Maria Ines Valla

ICIT'2012 Special Session Co-Chairs

Monday, 19.03.2012

SSMBTE1	**Model Based Testing**
Time:	14:30-16:00
Room:	D
Chair(s):	Juergen Jasperneite

SSMBTE1.1 **"Software Product Line Testing: a Feature Oriented Approach"**
Beatriz Pérez, Oscar Díaz, Maider Azanza, Macario Polo

SSMBTE1.2 **"Practical Model-Based Testing of User Scenarios"**
Vitaly Kozyura, Sebastian Wieczorek, Matthias Schur, Andreas Roth

SSMBTE1.3 **"Externalizing Business Rules from Business Processes for Model Based Testing"**
Sujithra Sriganesh, Chandrashekar Ramanathan

SSMBTE1.4 **"Experiences in Setting up Domain-Specific Model-Based Testing"**
Teemu Kanstrén, Olli-Pekka Puolitaival, Veli-Matti Rytky, Asmo Saarela, Janne Keränen

SSMBTE1.5 **"Quality Model based on ISO/IEC 9126 for Internal Quality of MATLAB/Simulink/Stateflow Models"**
Wei Hu, Tino Loeffler, Joachim Wegener

SSMBTE1.6 **"Flexible Debugging of Behavior Models"**
Alexander Krasnogolowy, Stephan Hildebrandt, Sebastian Wätzoldt

Software Product Line Testing: a Feature Oriented Approach

Beatriz Pérez Lamancha*, Oscar Díaz†, Maider Azanza† and Macario Polo*
*Alarcos Research Group, Castilla-La Mancha University, Ciudad Real, Spain
Email: beatriz.plamancha@uclm.es, macario.polo@uclm.es
† Onekin Research Group, University of Basque Country, San Sebastina, Spain
Email: oscar.diaz@ehu.es, maider.azanza@ehu.es

***Abstract*—*Software Product Lines (SPLs)* are not intended to create one application, but a number of them: a product family. In contrast to one-off development, SPLs are based on the idea that the distinct products of the family share a significant amount of assets. This forces a change in how software is developed. Likewise, software testing should mimic its code counterpart: product testing should also be produced out of a common set of assets. Specifically, this paper addresses how model-driven testing, used for one-off development, can be moved to an SPL setting. We focus on feature-oriented software development as the SPL realization technique. UML sequence diagrams are used to represent the common and feature scenarios. This models are transformed through model transformations to obtain test cases that conform to the UML Testing Profile.**

I. Introduction

Software Product Lines (SPLs) emerge as a reuse approach when a set of software applications overlap in their functionality. Clements *et al.* define an SPL as *"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"* [1]. This approach distinguishes between (1) the developmet of the product line as such (i.e. the core assets) which defines and realizes the commonality and variability of the product family, and (2) the development of the product *per se*, which is derived from the core assets based on the features to be exhibited by this product. This two-step process highlights pre-planned reuse: a domain study is first conducted that provides a set of components one knows for sure will be reused for different products of the family.

On the other hand, preventing mismatches between the application and its testing counterpart calls for an alignment of the application and the testing development processes. SPL setting is characterized by (1) a potentially broad number of artifacts, (2) tangling relationships among them, (3) lasting lifecycles, and (4), neat distinction between artifact producers (i.e., domain engineers) and artifact consumers (i.e., application engineers). Since SPL products are developed out of core assets components, the testing of SPL products should be based on the tests which exist from those core components. Likewise, if SPL products are built by composing core assets then, the tests for such products should also be delivered through composition.

This paper presents a model-driven testing approach to SPLs. The final aim is to mimic the development of its product counterpart. The approach focuses on functional testing, where functional specifications are described for the SPL at design level using UML models. Specifically, UML use cases and UML sequence diagrams are used for define base and feature functionalities. This work presents two main contributions, first a way to define base and feature functionalities at design level following a feature-oriented approach for SPLs and, second, a model-driven testing approach from these models, where test models are realized using the UML Testing Profile [2]. These models are later traduced to executable test code for the base functionalities and the feature functionalities. Then, their are composes to obtain test code for the products in the line. The paper begins with a brief summary of Software Product Lines.

II. A Brief on Product Lines

SPLs are not intended to build a single application, but a number of them: a product family. This forces a change in the engineering process where a distinction is made between *Domain Engineering* and *Application Engineering*. Domain Engineering (a.k.a. core asset development [1]) determines the commonality and the variability of the SPL. On the other hand, Application Engineering (a.k.a. product development [1]) produces concrete products out of the core assets. Doing so, the construction of the reusable assets (i.e., core assets) and their variability (a.k.a. variants) is separated from the production of the concrete products that form the family. Variability is a central concept in product family development. Adequately managing the variability among the family members is what permits the generation of the different products by reusing core assets. Variability is captured through **features** (i.e., increments in program functionality that customers use to distinguish one application from another [3]). The set of features of an SPL defines its scope. Such set is captured through a **feature model** (i.e., the specification of all legal compositions of features in a product line [3]) during domain engineering. This feature model gets instantiated (a.k.a. **configuration model**) during application engineering. This configuration model describes the concrete features to be exhibited by each single product.

As a running example, consider the playing-board game SPL (Game SPL). These kind of games share a broad set of

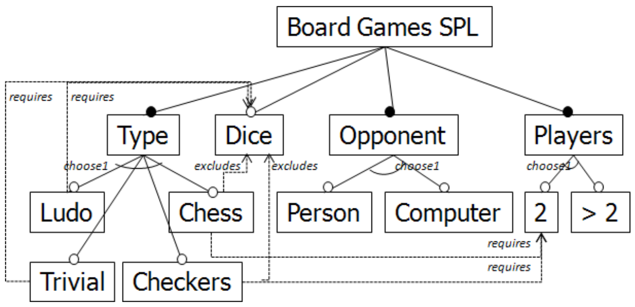
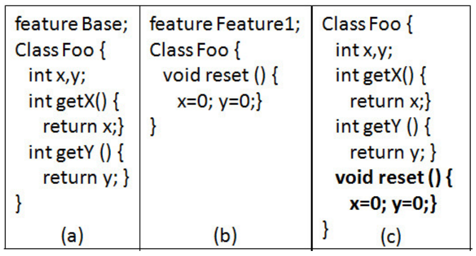


Figure 1. Feature Model for the Game SPL.



```
feature Base;
Class Foo {
  int x,y;
  int getX() {
    return x;}
  int getY () {
    return y; }
}
        (a)
```

```
feature Feature1;
Class Foo {
  void reset () {
    x=0; y=0;}
}
        (b)
```

```
Class Foo {
  int x,y;
  int getX() {
    return x;}
  int getY () {
    return y; }
  void reset () {
    x=0; y=0;}
}       (c)
```

Figure 2. Adding a Feature to a Class

characteristics, such as the existence of a board, one or more players, maybe the use of dice, the presence or absence of cards, policies related to the assignment of turns to the next player, etc. Figure 1 depicts the feature model for the Game SPL using the FODA notation [3]. Variations are admitted along four aspects, namely:

- *Type:* which corresponds to one of the possible variations. Each one corresponds to one of the supported four games (Chess, Checkers, Ludo or Trivial).
- *Opponent:* The player can play against either the computer or another online human player.
- *Players:* The minimum number of players for these games is 2, but optionally some games can be played by 3, 4, or more than 4 players. Chess and Checkers exclude this option.
- *Dice:* Ludo and Trivial require throwing the dice prior to moving, but Chess and Checkers exclude this feature.

Once the SPL is characterized along its feature model, the challenge rests on how to engineer artifacts (i.e. the core assets) for variability along this feature model. A first distinction can be made between the subtractive and additive approaches [4]. **Subtractive approaches** are successors of the conditional sentences found in code artifacts, where instruction execution is conditioned through checks of a feature's presence/absence. A core artifact is then "customized" by not executing all those instructions whose conditions are not met by the current configuration model. By contrast, the **additive approaches** strive to realize each feature in a separate artifact, the so-called **delta artifacts**. Base artifacts account for the commonality of the SPL while delta artifacts leverage the base with the feature at hand. At production time, base artifacts are composed with those delta artifacts according to the configuration model. *Feature Oriented Software Development (FOSD)* follows this second approach [5].

A. Feature Oriented Software Development

FOSD dictates that a complex program is developed from a simple program by adding features incrementally using function composition (where • denotes such composition):

$i \bullet x$ // adds feature i to base program x

$j \bullet x$ // adds feature j to base program x

Figure 2a shows a base artifact *Foo* defining variable members (x and y), and methods (*getX* and *getY*). This base artifact can now be incrementally extended by adding a new method *reset()* that extends the functionality of the base with a new feature *Feature1*. Figure 2b shows such extension using the *Jak* language [5]. The expression *Feature1•Base* returns a *Jak* artifact which holds feature *Feature1* (see Figure 2c). Likewise, *Feature2•Feature1•Base* stands for the base being enhanced with features *Feature1* and *Feature2*, where the order of feature composition (i.e., from right to left) can matter.

Core assets stand for either **base artifacts** (i.e. those providing the commonality) or **delta artifacts** (i.e. those realizing the variations). Delta artifacts realize features. That is, they encapsulate the set of changes that should be accomplished unitedly to leverage the base with an identifiable feature functionality (i.e. deltas encapsulate the changes that realize the feature, document the feature, test the feature, etc.). A formalization of deltas of models can be found in [6].

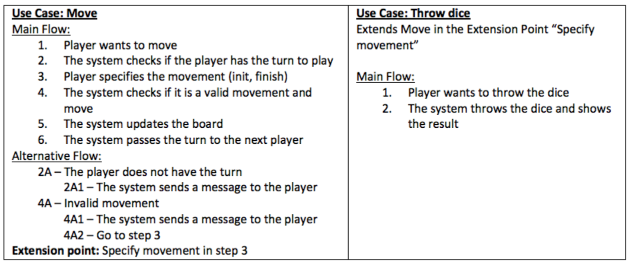
The challenge rests on applying FOSD approach to artifacts other than code. This however is most important to ensure uniformity so that all artifacts no matter their type follow the same development approach. In this way, a small number of operators can be use to manipulate all artifacts, ad-hoc complexity is reduced and better scalability is ensured [5].

III. A DESIGN APPROACH FOR SPLS

This paper advocates for SPL testing to mimic SPL product development. An SPL end product just looks as any other software product. So, traditional testing techniques could be used. However, the difference stems from the development process. SPLs differ from one-off development in that products are obtained from a common infrastructure (a.k.a. core assets). SPLs introduce a sharp distinction between artifacts for reuse (i.e., core assets) versus artifacts developed by reuse (i.e. the SPL products). Likewise, testing should follow similar practices. Effective reuse requires the existence of test core assets (i.e. tests for reuse) which are later composed to deliver full-fledged tests for the product at hand.

This section explains how the FOSD approach is extended to UML models used to design the SPL. Our approach focuses on functional testing level. In that sense, the UML models used to represent the system are: UML use cases and UML sequence diagrams.

UML **Use Cases** are used to capture the requirements of a system, that is, what a system is supposed to do. **Scenarios** concretize use cases by describing one path in the flow of the use case at hand. Each scenario provides the grounds for conducting functional testing and a **Test Scenario** accounts for the test requirements of a scenario. Figure 3 shows the *Move* and *ThrowDice* use cases. These use cases are part of the *Game* SPL.



Use Case: Move	Use Case: Throw dice
Main Flow: 1. Player wants to move 2. The system checks if the player has the turn to play 3. Player specifies the movement (init, finish) 4. The system checks if it is a valid movement and move 5. The system updates the board 6. The system passes the turn to the next player Alternative Flow: 2A – The player does not have the turn 2A1 – The system sends a message to the player 4A – Invalid movement 4A1 – The system sends a message to the player 4A2 – Go to step 3 **Extension point:** Specify movement in step 3	Extends Move in the Extension Point "Specify movement" Main Flow: 1. Player wants to throw the dice 2. The system throws the dice and shows the result

Figure 3. Move and ThrowDice Use Cases

UML's Interaction model is a common notation to describe scenarios [7]. An interaction is a composition of *messages* [7], a *message* defines a particular communication between lifelines. A *lifeline* represents an individual participant in the interaction. A *message* then relates two happenings in, normally distinct, lifelines. A common graphical notation to depict models of this metamodel are *Sequence Diagrams* (SD). In this work the terms sequence diagram and interaction are used indistinctly. Figure 4 illustrates the *Move Scenario*.

Back to SPLs, a distinction is made between *Base interactions* (i.e. interactions that contain the commonality of the SPL) and *Delta interactions* (i.e. interactions that account for a specific feature). The question is how deltas differ from traditional interactions. The main difference stems from deltas being inconclusive. Deltas do not exist in isolation but define increments on Base interactions. This fact is realized as an extension point to the Base interaction. This extension point is denoted through a UML *message*. Figure 5 shows *ThrowDice* interaction which partially describes the delta *Dice*. *ThrowDice* leverages *Move* with the ability of throwing dices. This enhancement is constrained to occur (1) after the player consults his turn and (2) before moving. To denote this location in UML terms, we resort to the notion of *gate*. A *gate* is a

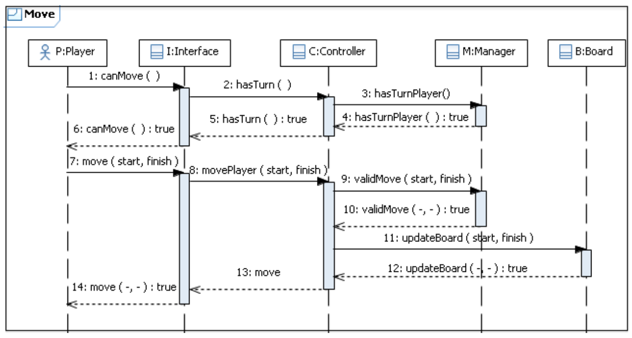

Move
P:Player
I:Interface
C:Controller
M:Manager
B:Board
1: canMove ( )
2: hasTurn ( )
3: hasTurnPlayer()
4: hasTurnPlayer ( ) : true
5: hasTurn ( ) : true
6: canMove ( ) : true
7: move (start, finish)
8: movePlayer (start, finish)
9: validMove (start, finish)
10: validMove (-, -) : true
11: updateBoard (start, finish)
12: updateBoard (-, -) : true
13: move
14: move (-, -) : true

Figure 4. *Move* scenario functionality

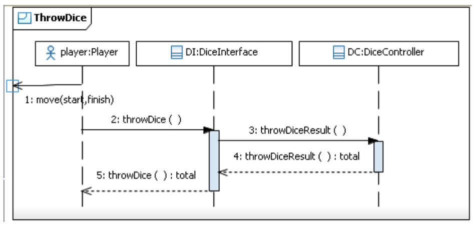
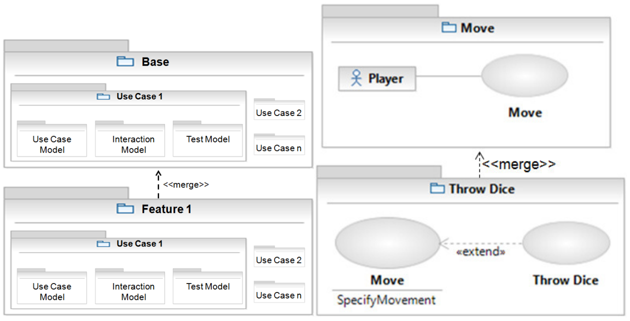


Figure 5. ThrowDice sequence diagram



(a) Package Structure (b) Move and ThrowDice Packages

Figure 6. UML Package Merge

representative of an *OccurrenceSpecification* that is not in the same scope as the *gate* [7]. A *gate* is a connection point for locating a message outside an interaction with respect to a *message* inside the interaction. Figure 5 shows such a gate for the *ThrowDice* delta. The gate indicates that the delta is to occur just above the namesake message in the base interaction.

At this point, it is worth noticing that artifacts are arranged into higher units along the feature criterium. For each feature in the SPL, one delta package exits. Both *Base packages* and *Delta packages* exhibit the same structure. For the purpose of this paper, artifacts are arranged along three main types: use cases, scenarios and tests. Figure 6(a) provides an example. Unlike *Base packages*, *Delta packages* can hold delta scenarios along with complete artifacts. This infrastructure is used during product generation. This is the topic of the next section.

IV. A TESTING APPROACH FOR SPL

An SPL infrastructure refers to the set of core assets that will be reused during the production of end products. This infrastructure includes testing. Figure 7 shows the main assets and their relationships. Hereafter, the term "test deltas" is used to denote these test core assets. The notion of "delta" aims to suggest the inconclusive definition of these tests that require to be composed to be fully operational.

In our particular setting, "functional artifacts" are to be complemented with "testing artifacts". The artifacts involved are depicted in the upper part of Figure 7. Besides Use Cases, Scenarios and Scenario Tests, now we introduce Delta Use Cases, Delta Scenarios and Delta Scenario Tests, which support features. In brief, deltas provide an increment in

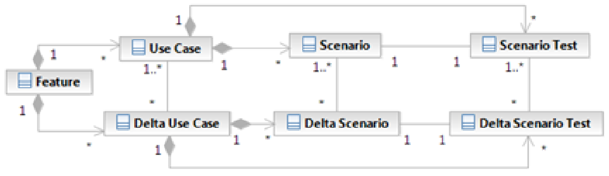


Figure 7. Artifact types and their content relationships.

functionality independent of the artifact that realizes such functionality (e.g. a use case, a scenario or a test).

The previous section addressed delta definition for design models . Delta development can be distributed among different developers. After all, deltas account for features. A feature is a meaningful increment in functionality which can be subject to a different evolution pace than its other feature companions. This makes features common units of workload distribution among developer teams. This implies that features tend to be developed (and tested) separately from the rest of the SPL infrastructure.

We then propose three-step for Testing in SPL (see Figure 8), namely: test base, test delta and test product functionalities .

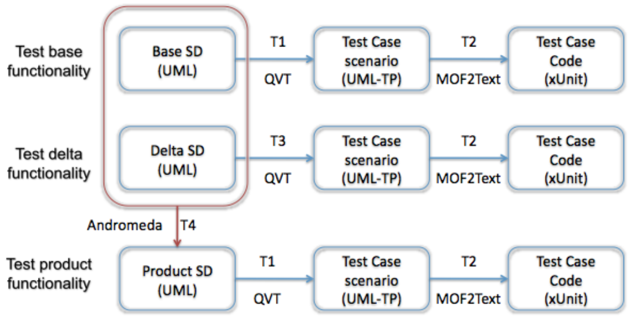


Figure 8. Metamodels and transformations for Testing in SPL

A. *Testing base functionality*

The base functionality is described through UML SD. In previous works [8], [9], we defined an automated approach to obtain executable test cases from scenarios described using UML SD. From a functional testing point of view, generating test cases scenarios implies that the system must be considered as a black box, and the stimulus from the actor to the system must be simulated, and vice versa. Figure 8 (above) describes the transformations for test base functionality. The first transformation (T1), converts a SD in a test case scenario. The test case scenario is represented also using UML SD and follows the UML Testing Profile *(UML-TP)* [2]. The second transformation (T2), converts test case behavior in test code. These transformations are described next:

UML SD to test case scenario (T1): This transformation is a model to model transformation, mapping from the UML Interaction metamodel (used for scenario description) to the

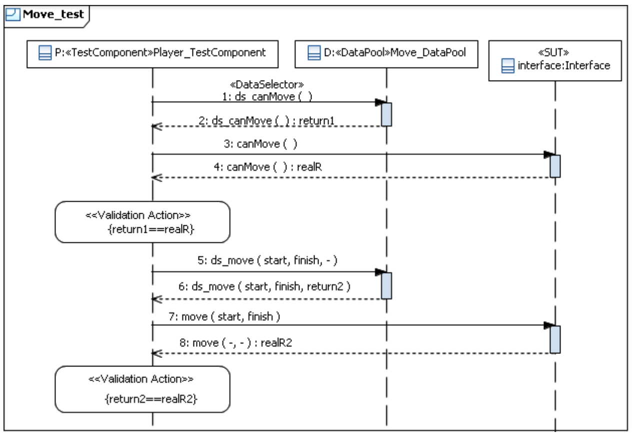

Move_test
P:«TestComponent»Player_TestComponent
D:«DataPool»Move_DataPool
«SUT» interface:Interface
«DataSelector» 1: ds_canMove ( )
2: ds_canMove ( ) : return1
3: canMove ( )
4: canMove ( ) : realR
<<Validation Action>> {return1==realR}
5: ds_move (start, finish, -)
6: ds_move (start, finish, return2)
7: move (start, finish)
8: move (-, -) : realR2
<<Validation Action>> {return2==realR2}

Figure 9. Test Case for Move Scenario

UML 2.0 Testing Profile (used for test case description) and was developed using OMG's *Query-View-Transformation language (QVT)* [10]. Along the *UML-TP*[2], actors are represented as *TestComponents, test data* is stored in the *DataPool* whilst the *System* is represented as the *System Under Test (SUT)*. Figure 9 depicted the Move test scenario transformed from Move Scenario (Figure 4). Each message between the actor and the *SUT* must be tested. For example, to test the message *Move(start,finish) in Move,* the following steps are necessary:

1) **Obtaining the test data**: The TestComponent asks for the test data using the *DataSelector* operation in the *DataPool*. Figure 9 shows the message *ds_Move()* stereotyped as DataSelector, that returns three parameters, the start and finish data and the expected result for the test case (return2).
2) **Executing the test case in the SUT**: The *TestComponent* calls the message to test in the SUT. The operation *Move(start,finish)* is tested with the input data returned by the *dataPool* as parameter and returns the actual result (*realR2*).
3) **Obtaining the test case verdict**: The *TestComponent* executes the *state invariant stereotyped as validation action*, comparing the expected and the real results. For the operation *Move(start,finish)*, if the *realR2* (actual value) is equal to the *return2* (expected value) the test case *pass* else *fail.*

Test case scenario to test code (T2): This is a model to text transformation, takes as input test case scenarios described through the UML-TP and returns as output executable test code using the xUnit test language. xUnit is a family of frameworks, which enable the automated testing of different elements (units) of software. The generated test code can be executed using the corresponding xUnit framework to test the system. This transformation was developed with the MOFScript tool[1]. To illustrate this transformation we use Java

[1]*http://www.eclipse.org/gmt/mofscript/*

and its corresponding test language JUnit[2]. The transformation takes as input the test case for Move (Figure 9), and transforms it into a JUnit test method (see Figure 11(a)). Basically, the test method asks for the test data to the dataPool. The dataPool returns a vector with the test data set, allowing test the same functionality with several data values. Then, extract each test data and call the operations to test. First, the *canMove()* is tested and the expected and the returned result is compared using the *assertTrue* sentence of JUnit. The process is repeated for the *Move(start, finish)* operation.

B. Testing delta functionality

This step focuses on the scenarios that apply uniquely to the feature at hand. The only difference between testing *Base interactions* and *Delta interactions* is that the latter hold a *gate*. Figure 8 shows transformation T3. This transformation

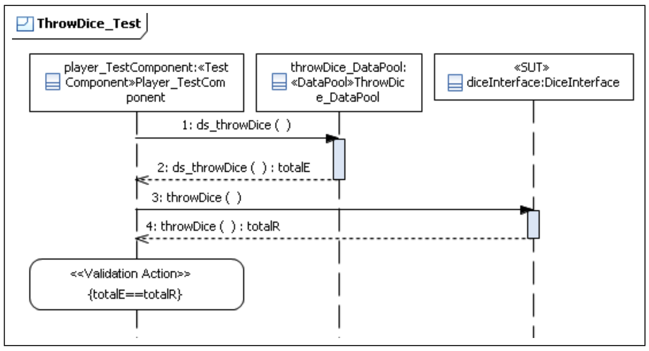

ThrowDice_Test
player_TestComponent:«Test Component»Player_TestComponent
throwDice_DataPool: «DataPool»ThrowDice_DataPool
«SUT» diceInterface:DiceInterface
1: ds_throwDice ( )
2: ds_throwDice ( ) : totalE
3: throwDice ( )
4: throwDice ( ) : totalR
<<Validation Action>>
{totalE==totalR}

Figure 10. Test case scenario for *ThrowDice*.

first just removes the gates and proceeds in the same way as transformation T1, as was described above. Figure 5 shows the *ThrowDice* interaction with a gate and Figure 10 shows the generated test case. To obtain the executable test code, the transformation T2 is also used.

C. Testing the composition between the delta and the base

The rationale behind this step is that a feature never occurs alone but at least the *Base* needs to be present. This implies that *Base* scenarios need to be first composed with *Delta* scenarios for the feature at hand. This corresponds to transformation T4 in Figure 8. Figure 11(b) shows the composition between *Move* and *ThrowDice*. Section V explains in detail how the composition was implemented.

Once the composition is obtained, the test case for that composition is obtained using first transformation T1 (see Figure V-B) , and then the test code is obtained using transformation T2, as was explained in section V.

V. COMPOSITION IMPLEMENTATION

SPL product generation starts by characterizing the product through its configuration model (i.e. the set of features to be hold by the product). This configuration model is then transcribed to a feature equation. For instance, the equation $trivialGame = trivial \bullet dice \bullet base$ stands for product $trivialGame$ as the result of composing features $dice$ and $trivial$ to the common $base$.

Implementation wise, both $base$ and $dice$ are realized by the namesake packages. This implies that feature composition is realized as package composition. A package is a containment hierarchy of artifacts (see Figure 12). A package can contain

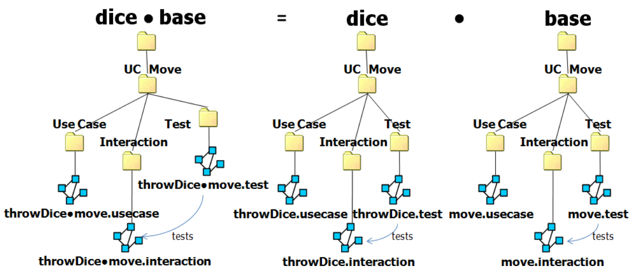


Figure 12. Package Composition

other packages, which are in turn composed, until atomic artifacts are reached. Package composition percolates along the containment hierarchy until atomic artifacts (the leaves) are reached. This highlights that the composition operator is polymorphic in the sense that its semantics depend on the type of the artifact being compound.

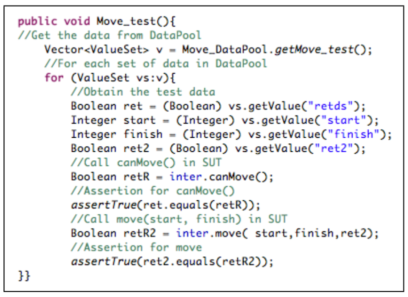
Therefore, we need to define the semantics of composition for two artifacts: UML packages and UML interaction diagrams. Since interaction diagrams realize scenarios and test cases, defining composition of interaction diagrams addresses how testing scenarios are obtained out of the composition of scenarios and scenario deltas (i.e. features). Next subsections define package composition and scenario composition. The bottom line is that the very same feature-based reuse mechanism is being defined for both functional artifacts (e.g. java classes) and test artifacts (i.e., scenarios). More to the point, test artifacts are obtained at the very same pace as their functional counterparts. The very same process that yields the *trivialGame* also obtains the scenarios to test this product.

A main premise of this work is that current UML 2.0 compliant tools must support the definition of deltas, avoiding developers the task of directly editing the XMI representation of delta interactions. This is the most important issue to ensure practitioners will embrace the approach. Both, Scenarios and Test Scenarios are described through UML sequence diagram models.

The current UML 2.0 compliant tools support the use of *gates*. Specifically, IBM Rational Software *Architect*[3] is used to obtain the diagrams in this work. This implies that base and deltas can resort to the same toolings. More to the point, the composition between base and deltas (see next section) can be achieved using XMI which in turn, ensures that the result can also be displayed through existing UML editors.

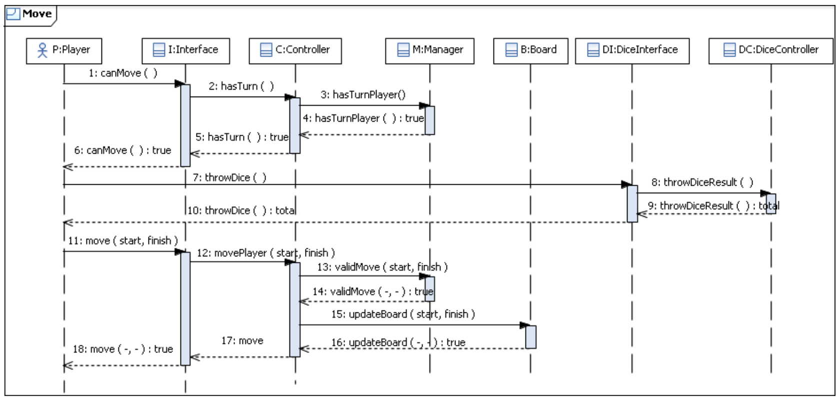
[2] *http://www.junit.org/*

[3] *http://www.ibm.com/developerworks/rational/products/rsa/*



```java
public void Move_test(){
//Get the data from DataPool
    Vector<ValueSet> v = Move_DataPool.getMove_test();
    //For each set of data in DataPool
    for (ValueSet vs:v){
        //Obtain the test data
        Boolean ret = (Boolean) vs.getValue("retds");
        Integer start = (Integer) vs.getValue("start");
        Integer finish = (Integer) vs.getValue("finish");
        Boolean ret2 = (Boolean) vs.getValue("ret2");
        //Call canMove() in SUT
        Boolean retR = inter.canMove();
        //Assertion for canMove()
        assertTrue(ret.equals(retR));
        //Call move(start, finish) in SUT
        Boolean retR2 = inter.move( start,finish,ret2);
        //Assertion for move
        assertTrue(ret2.equals(retR2));
}}
```

(a) Move test case in JUnit


Move
P:Player
I:Interface
C:Controller
M:Manager
B:Board
DI:DiceInterface
DC:DiceController
1: canMove ( )
2: hasTurn ( )
3: hasTurnPlayer()
4: hasTurnPlayer ( ) : true
5: hasTurn ( ) : true
6: canMove ( ) : true
7: throwDice ( )
8: throwDiceResult ( )
9: throwDiceResult ( ) : total
10: throwDice ( ) : total
11: move (start, finish)
12: movePlayer (start, finish)
13: validMove (start, finish)
14: validMove (-, -) : true
15: updateBoard (start, finish)
16: updateBoard (-, -) : true
17: move
18: move (-, -) : true

(b) ThrowDice Move scenario composition

Figure 11.

A. Composition of Packages

Composition of packages can be supported through the UML's *Merge* mechanism. A package merge is a directed relationship between two packages that indicates that the contents of the two packages are to be combined. It is very similar to *Generalization* in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both [7]. Laguna *et al.* [11] use the package merge mechanism to represent variability in an SPL setting.

Figure 6(b) shows the package merge for the *Move* functionality in the Board Game SPL. For the *Dice* feature, the *ThrowDice* use case extends the *Move* use case. In the UML specification, the semantics of package merge are defined by a set of constraints and transformations. The constraints specify the preconditions for a valid package merge, while the transformations describe its semantic effects. Different metatypes have different semantics, but the general principle is always the same: a resulting element will not be any less capable than it was prior to the merge. Explicit merge transformations are only defined for certain general metatypes (Packages, Classes, Associations, Properties, etc.). UML does not provide merge semantics for other kinds of metatypes. This is the case of interaction diagrams.

B. Composition of Interaction Diagrams

Back to our sample case, *trivialGame* is characterised through its feature equation *trivial* • *dice* • *base*. Therefore, scenarios for *trivialGame* are obtained through composition of *base* scenarios, *dice* scenarios and *trivial* scenarios. As an example, the composition of the *ThrowDice* scenario, which is part of the *dice* feature and the *Move* scenario, which corresponds to the *base* yields a scenario for *trivialGame* (see Figure 11(b)). This scenario can then be used to obtain a test scenario by running a model transformation that generates tests. This subsection addresses composition of scenarios described as interaction diagrams.

Interaction diagrams are models. Model composition has been defined as the operation M_{AB} = Compose (M_A, M_B, C_{AB}) that takes two models M_A, M_B and a correspondence model C_{AB} between them as input, and combines their elements into a new output model M_{AB} [12]. Delta composition is a special case, where both M_A and M_B conform to the *same* metamodel. Delta composition is performed by pairing objects of different models with the same name and composing them [13].

The order between the messages in a interaction diagrams is vital, and it should be preserved in the composition. We need to know where exactly the variable part (i.e., the delta) must be added to the common part (i.e. the base) to obtain the entire functionality. To define the composition we need to identify the point in the diagram where the delta must be added. When adding the functionality of a feature (i.e., a delta) to a common functionality (i.e. a base), three different cases appear: (1) the delta is added before the first message in the base, (2) the delta is added between two messages in the base and (3) the delta is added after the last message in the base.

UML Gates provide a solution for the three cases. For the second and third case, Figure 13 shows a gate in the diagram *Feature1*. It is represented as an arrow with the start in the border of the rectangle. This indicates that all messages in *Feature1* must be added after the message referenced in the gate. If the base has more messages, the feature must be added in between the message referenced by the gate and the next message in the base. The gate in *Feature1* is c2Op1 and it references the message with the same name in base. The SD *Feature1•Base* shows the composition. The solution for the first case, where the variable part needs to be added before the first message in the common part is shown in Figure 13. In this case, the arrow is represented with the start in a lifeline inside the Interaction and the finish in the border of the Interaction. This approach was realized using ANDROMEDA, a tool for model composition [13]. This solution allows for multiple composition: *Feature2•Feature1•Base* obtains a SD similar to *Feature1•Base* of Figure 13(a) with the message

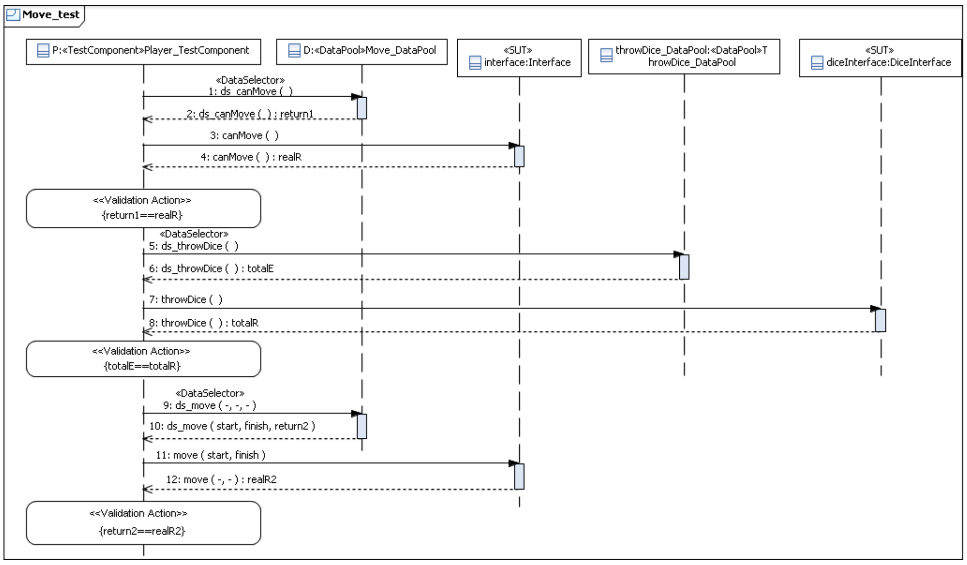

Move_test
P:«TestComponent»Player_TestComponent
D:«DataPool»Move_DataPool
«SUT» interface:Interface
throwDice_DataPool:«DataPool»ThrowDice_DataPool
«SUT» diceInterface:DiceInterface
«DataSelector» 1: ds_canMove ( )
2: ds_canMove ( ) : return1
3: canMove ( )
4: canMove ( ) : realR
<<Validation Action>> {return1==realR}
«DataSelector» 5: ds_throwDice ( )
6: ds_throwDice ( ) : totalE
7: throwDice ( )
8: throwDice ( ) : totalR
<<Validation Action>> {totalE==totalR}
«DataSelector» 9: ds_move (-, -, -)
10: ds_move (start, finish, return2)
11: move (start, finish)
12: move (-, -) : realR2
<<Validation Action>> {return2==realR2}

Figure 14. Test scenario for *ThrowDice.Move*

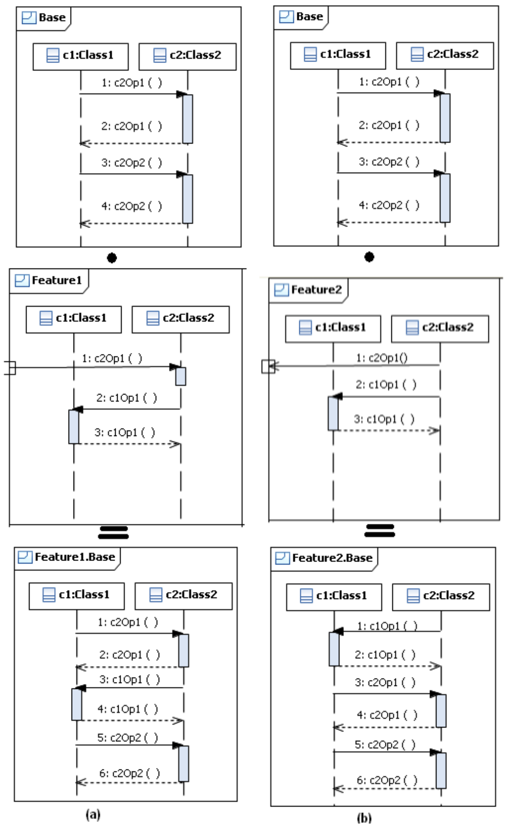


Figure 13. Model composition for UML Interactions

c1Op2() added as first message. Due to space limitations, implementation details have been omitted. We refer the interested reader to [14] for a more comprehensive description.

VI. Related Work

SPL testing includes the derivation of test cases for both the core assets and the products themselves. There exist different methodologies for deriving test cases and a summary of this methods is given in [15], [16], [17], the most cited are from: Nebut et al. [18], Bertolino and Gnesi [19], Reuys et al. [20], Olimpiew and Gomma [21], Kang et al. [22]. Some of them use UML 2.0 models to capture SPL functionality, but define their own models to represent testing artifacts and variability in the test model. By contrast, our work strives to be fully UML compliant: (1) UML-TP is used for testing scenarios, (2) package composition is introduced by using UML *merge* primitive, and (3), scenarios resort to the UML *gate* construct to describe partial scenarios for feature realization. Only when UML falls short (e.g. composition of interaction diagrams) our own semantics are introduced. Additionally, our approach advocates for a uniform treatment of artifacts no matter whether their realize functional or testing behaviour. This is most important for SPLs to scale up. Both, the number of artifacts and the complexity of the development process, are larger in SPLs than in one-off development. Uniform development (both code artifacts and scenarios follow the same composition principles) helps to face complexity [5]. We regard scalability as a main requirement for traditional testing methods to be applied to SPLs.

Focusing on *Feature Oriented Model Driven Development (FOMDD)* [23], Uzuncaova et al. [24] presents an incremental test generation approach for specifi-
cation-based testing of software product lines developed using

AHEAD tool and focusing on the use of Alloy for test generation. They perform test generation incrementally, mapping a formula that specifies a feature into a transformation that defines incremental refinement of test suites. These formulas then feed a SAT-based analyzer to generate test inputs for each product in SPL. This work uses Jakarta code (a variation of Java for feature implementation) as feature realizations which are then enhanced with annotation of invariants. Our work also uses a FOMDD approach but scenarios (rather than Java code) are used to set the test cases.

Our proposal defines model composition for UML SD in SPLs. The work of Apel et al. [25] is related to ours, they use superimposition as a model composition technique in order to support variability in the following UML diagrams: class, state and sequence. For UML sequence diagram they only take into account the case in that the feature is added to the end of the base sequence diagram. Jezequel[26] also works in model composition for UML sequence diagrams and relates it with aspect weaving. They implement the proposal with Kermeta using self metamodels to represent UML sequence diagrams. Our approach uses the OMG metamodel for UML and existing market tools to represent graphically UML sequence diagrams.

VII. Conclusions

This work addresses how current feature oriented practices for SPL development can be extended to testing. This has a twofold implication: first, defining a SPL testing infrastructure. Second, applying this infrastructure to obtain product tests out of delta tests, hence realizing the reuse benefits brought by SPL also to the testing realm. The specification of delta scenarios is UML compliant so that UML editors are applicable. Delta scenario reuse is achieved through scenario composition where model composition techniques are used. The approach is illustrated using the Game SPL. Current and future work includes evaluating the approach in a industrial software product line.

Acknowledgments. This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2008-06507-C02-01/TIN (MODELINE) and TIN2008-06507-C02-01/TIN (PEGASO/MAGO). Also, founding is acknowledge from DIMITRI (Ministerio de Ciencia e Innovacin, grant TRA2009_0131) and MOTERO (JCCM, PEII11-0366-9449). Beatriz Perez has a grant from "JCCM (Orden de 13-11-2008)".

References

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

[2] OMG, "UML Testing Profile Version 1.0," OMG Document Number: formal/05-07-07, 2005, status: revision underway.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie-Mellon University/Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, November 1990.

[4] M. Völter and I. Groher, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development," in *International Conference on Software Product Lines (SPLC 2007)*, 2007.

[5] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Transactions on Software Engineering (TSE)*, 2004.

[6] I. Schaefer, "Variability Modelling for Model-Driven Development of Software Product Lines," in *4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2010), Linz, Austria*, 2010.

[7] OMG, "Unified Modeling Language (UML)," OMG Document Number: formal/2007-11-02, November 2007.

[8] B. Pérez, P. Reales, I. Rodríguez, M. Polo, and M. Piattini, "Automated model-based testing using the UML testing profile and QVT," in *6th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa 2009), Denver, CO, USA*, 2009.

[9] B. Pérez, P. Reales, M. Polo, and D. Caivano, "MODEL-DRIVEN TESTING: Transformations from Test Models to Test Code," in *6th International Conference on Evaluation of Novel approaches to Software Engineering (ENASE 2011), Beijing, China*, 2011.

[10] OMG, "Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.0," OMG Document Number: formal/2008-04-03, April 2008.

[11] M. Laguna, B. González-Baixauli, and J. Marqués, "Seamless Development of Software Product Lines," in *6th International Conference on Generative Programming and Component Engineering (GPCE 2007), Salzburg, Austria*, 2007.

[12] J. Bézivin, S. Bouzitouna, M. Didonet, M. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R. Paige, "A Canonical Scheme for Model Composition," in *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13, 2006*, 2006.

[13] M. Azanza, D. Batory, O. Díaz, and S. Trujillo, "Domain-Specific Composition of Model Deltas," in *3rd International Conference on Model Transformations (ICMT 2010), Malaga, Spain*, 2010.

[14] M. Azanza, "Model driven product line engineering: Core asset and process implications," Ph.D. dissertation, University of the Basque Country, February 2011, available at: https://www.educacion.gob.es/teseo/ imprimirFicheroTesis.do?fichero=22952.

[15] P. da Mota, I. Carmo, J. McGregor, E. de Almeida, and S. de Lemos, "A systematic mapping study of software product lines testing," *Information and Software Technology*, 2010.

[16] E. Engstrom and P. Runeson, "Software product line testing-a systematic mapping study," *Information and Software Technology*, 2010.

[17] B. Pérez *et al.*, "Software Product Line Testing, A Systematic Review," in *ICSOFT*, 2009.

[18] C. Nebut, S. Pickin, Y. L. Traon, and J. Jézéquel, "Automated Requirements-based Generation of Test Cases for Product Families," in *18th IEEE International Conference on Automated Software Engineering (ASE 2003), Montreal, Canada*, 2003.

[19] A. Bertolino and S. Gnesi, "Use Case-based Testing of Product Lines," in *11th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2003) , Helsinki, Finland*. ACM, 2003, pp. 355–358.

[20] A. Reuys, E. Kamsties, K. Pohl, and S. Reis, "Model-Based System Testing of Software Product Families," in *17th International Conference Advanced Information Systems Engineering (CAiSE 2005), Porto, Portugal*, 2005.

[21] E. Olimpiew and H. Gomaa, "Customizable Requirements-based Test Models for Software Product Lines," in *International Workshop on Software Product Line Testing (SPLiT 2006), Baltimore, MD, USA*, 2006.

[22] S. Kang, J. Lee, M. Kim, and W. Lee, "Towards a Formal Framework for Product Line Test Development," in *7th International Conference on Computer and Information Technology (CIT 2007), Fukushima, Japan*, 2007.

[23] S. Trujillo, D. Batory, and O. Díaz, "Feature Oriented Model Driven Development: A Case Study for Portlets," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA*, 2007.

[24] E. Uzuncaova, S. Khurshid, and D. Batory, "Incremental test generation for software product lines," *Software Engineering, IEEE Transactions on*, vol. 36, no. 3, pp. 309–322, 2010.

[25] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model Superimposition in Software Product Lines," in *2nd International Conference on Theory and Practice of Model Transformations (ICMT 2009), Zurich, Switzerland*, 2009.

[26] J. Jézéquel, "Model Driven Design and Aspect Weaving," *Software and System Modeling (SoSyM*, 2008.