Automating Test Case Definition Using a Domain Specific Language

Kyungsoo Im
School of Computing
Clemson University
Clemson, SC 29634
kyungsi@cs.clemson.edu

Tacksoo Im
School of Computing
Clemson University
Clemson, SC 29634
tim@cs.clemson.edu

John D. McGregor
School of Computing
Clemson University
Clemson, SC 29634
johnmc@cs.clemson.edu

ABSTRACT

Effective test cases are critical to the success of a development effort but their creation requires large amounts of critical resources such as domain expertise. This study explores an approach to automating test case definition in the context of applying a model driven approach to the development of a software product line. In this study, test cases are automatically extracted from use cases, which are specified using a domain specific language (DSL). DSLs are easier for domain experts to use than formal specification languages and are more narrowly focused than natural languages making it easier to build tools. The task is further simplified by restricting the DSL to the scope of the software product line under development. The structure of the DSL and proven patterns of test design provide the clues necessary to be able to automatically extract the test cases. A chain of model-driven tools is used to automate the system test process, which begins with a use case model and ends with automatic execution of system tests.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design, Verification

Keywords

domain specific language, software testing, ontology, software product line

1. INTRODUCTION

Defining and executing tests consume large amounts of valuable resources. Depending upon the criticality of the application and the quality of the development process, estimates of the costs of testing range from thirty percent to two hundred percent of the cost of development [3]. Much progress has been made on automating the execution of test cases but not as much progress has been made on automatically defining tests. In the investigation reported in this paper we explored a technique for defining test cases that rely on the constrained scope of a domain specific language (DSL) [21].

Testing is a search for faults. An exhaustive search is seldom feasible and usually impossible. A large number of techniques have been developed to guide the search so that a partial search can be effective. Test patterns have been used to capture some of the effective ways to construct tests [5] [16] [12]. A test pattern solves a testing problem in a specific context. The pattern makes it possible to define a template, or templates, that captures the pattern and can be used to generate an implementation.

Testing a piece of software involves three essential phases. First, the tests are planned. Exactly what to test and how is determined. Second, the tests and test infrastructure are constructed. An environment in which the software under test will be executed is assembled or built. Finally, the tests are executed and the results are evaluated. Much work has focused on the final step and environments such as JUnit have largely automated the execution and evaluation phase. This phase is repeated often to check changes in the implementation.

Our focus is on the first and second phases where tests are planned and physically constructed. The test planning and construction phases are repeated less often than the execution and evaluation phase but particularly in iterative development these phases will be repeated often. Our goal is to reduce the effort of these two phases.

A software product line is an ideal environment for this experiment. A software product line is a collection of products that share a common set of features. This commonality includes the domain of application. Domains have characteristics that lend themselves naturally to certain designs, e.g. state machines for real-time systems. We are exploiting this relationship by creating a domain specific language and relating statements in that language to specific designs and hence to specific test patterns.

The definitive scope of a product line allows us to define very clear boundaries for the vocabulary of the domain specific language and the set of features. These constraints make possible a number of simplifying assumptions that can be used to make automatic test case generation feasible. The multiple products in the product line provide a context in

which the higher initial costs of automation can be amortized over the entire set of products. A number of issues regarding testing products in a software product line have been identified in [13] [14].

The contribution of this work is an illustration of the efficacy of applying domain specific languages to automating test case definition. We describe the tool chain needed to implement our technique. We also provide examples based on a software product line developed for pedagogical purposes.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction to those topics needed to understand the work. Section 3 provides a discussion of the technique we have developed. Section 4 describes some of the examples we have constructed to evaluate the technique. Finally, section 5 summarizes our work and describes where we go from here.

2. RELATED WORK

Our technique relies on work from several fields. In this section we briefly describe some essential concepts that underlie our work.

2.1 Software Product Lines

"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission [7]" and that are developed from a common set of core assets in a prescribed manner. Product lines are often used to make software production more economical by achieving strategic levels of reuse. These reused items are called core assets. Core assets can be anything from the architecture to design documentation to the testing infrastructure or, of course, the code. Commonality among products from the product line leads to commonality in the assets used to build them, such as test cases. A domain specific language is a means by which the commonality can be captured in a notation that product designers can use to define products and testers can use to design test cases.

2.2 Use cases

A use case contains scenarios about using the product, that is divided into external stimuli and the system's response to the stimuli. Use cases show the uses of a system from the user's perspective. For this reason, use cases serve as valuable specifications for designing system-level test cases. There have been a number of approaches in generating test cases based on use cases [18][6][8][2]. Some of these approaches transform the use case into a state chart or other graphical notations manually to derive the test cases.

2.3 Domain specific languages

Domain specific languages are "languages tailored to a specific domain [17]". They provide "notations and constructs for an application domain, trading generality for expressiveness which results in substantial ease-of-use [17]". DSLs have been used in many application domains such as web computing [1], user interface construction [19] and software testing [20]. The effort required to design a domain specific language might not be justifiable in a one-off product setting but it can be economically justified in a product line context. There has been some effort in using a domain specific language to generate test cases [20] but we have found no work in using a DSL to automate test cases from a use case based approach. Sirer et al specified a production grammar using a domain specific language instead of focusing on use cases.

2.4 Test patterns

Test patterns can be leveraged to make testing easier for product lines. Test patterns are "design patterns for testing" [16]. For each design pattern there is a corresponding test pattern that is used to test it. However, test patterns encompass more than just design patterns. Test patterns can be used to describe test cases that follow a pattern for each product in a product line. For example, when checking for a "save game" option in a product line of games we can safely assume that the steps that will be taken to test that feature will be very similar for every game in the product line. Previous efforts in testing a product line such as [4], [9] describe how use cases can be used as a basis for creating the test cases for products in a product line.

3. OVERVIEW OF TECHNIQUE

Our test automation technique is divided into two threads of activities: the product line thread and the product specific thread. In a software product line organization, the product line team creates the core assets, assets that are used on multiple products in the product line. The product line core asset development team creates an infrastructure including the domain specific language, test templates, and then creates that portion of the use case model and test suite that apply to multiple products. Product specific teams develop assets that are unique to their assigned products using the core assets. The product team then builds their assigned product using the core assets and product specific assets.

3.1 Product line activities

The activities of the product line core asset team result in the artifacts illustrated in the product line box in Figure 1. A domain specific language, which must be sufficiently expressive to support the description of all of the products in the product line, is created first. Both the use cases and test templates are created using the DSL and can be developed in parallel.

3.1.1 Domain Specific Language

The technique we use to create the DSL combines the domain concepts and relationships captured in domain analysis with the user-visible properties of the products identified during feature analysis.

The domain analysis identifies the concepts and relations among the concepts in the application domain and captures them in an ontology. We process a number of documents written about the domain using a tool that identifies nouns and verbs in sentences [10]. This captures a starting vocabulary for the domain. Domain experts assist in identifying the relationships between the nouns and the actions on those concepts, the verbs. The ontology is represented in the Web Ontology Language (OWL) so that it can be manipulated automatically.

The feature analysis uses the vocabulary defined in the domain analysis to describe, in the feature model, the features of products to be developed. Feature modeling provides the opportunity to denote which features are included in every product (mandatory) and which features are only included in some products (optional). Feature modeling uses a stan-

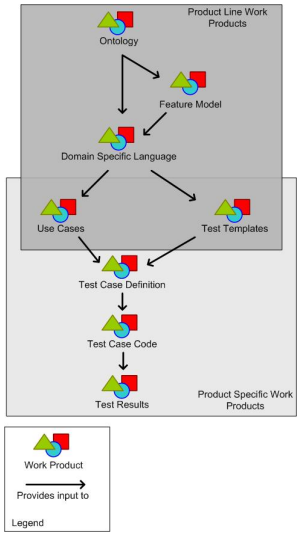

Figure 1: Artifact dependencies

dard notation that captures the features, cardinality of each feature, and constraints among the features.

The domain specific language evolves naturally as the nouns and verbs found in the ontology are combined in phrases and used to describe product features as they are added to the feature model. The language is captured in the form of phrases that are useful for the next step - use case definition. In our case we develop standard phrases that are suitable for the stimulus and response portions of use case descriptions.

3.1.2 Use cases

The specification of each product is constructed as a set of use cases. The stimuli and responses in each use case are written using the phrases in the domain specific language. A custom editor is generated from the DSL for use in constructing the use cases. The editor guides the use case definer by presenting the set of available phrases for stimuli and responses, and allowing the person defining the use case to select the appropriate phrases.

The product line set of use cases applies to multiple products that will be produced in the product line. The set of use cases is a reusable asset that product specific teams will use as the starting point for the specification of their products. Some product specific use cases are created by simply having a product line use case used "as is" while other product specific use cases will be defined by "extending" product line use cases or by "using" existing use cases.

3.1.3 Test templates

A set of test templates is created from test patterns. The test patterns are associated with phrases in the DSL and are chosen based on the DSL description and the type of testing being planned. System testing will use patterns that are based on how the product will be used, as described in the use cases, rather than on how the product is constructed. The test templates derived from the patterns are associated with requirements. A phrase in the DSL is associated with a template.

For example, the DSL phrase "responds within XXX seconds" would be associated with the test pattern "Test Case with Time Specification" [11]. The pattern would correspond to a template that inserts statements into the test harness that start a timer prior to the system performing the operation under test and then reads the timer when the operation is completed.

3.1.4 Test cases

The test cases created by the product line team are those that apply to all of the products. Test cases are generated from the use cases and test templates. As stated previously the phrases in the DSL are associated with test templates. This mapping is embodied in a test case editor that is generated automatically. The tester uses the test editor to create the test cases needed to fulfill the testing objectives.

At the product line level, the objective is to have at least one test case for every mandatory functional feature in the feature model. Test cases may also be created for each of the optional features. The feature model also includes non-functional features. The non-functional features often are represented by phrases in the DSL such as the "responds within " phrase. Rather than being a complete use case, a non-functional feature is often tested by clauses in a larger test case.

3.2 Product-specific activities

When a specific product is built, several activities are carried out. The use case model for the product is created using the product line use case model as the starting point. New uses are added to cover any product-specific features and product line use cases that correspond to optional features not selected for the product are removed. Use cases specific to the product are developed in one of three ways:

- some of the product line use cases are used "as is" as product use cases
- new use cases are defined via the extends relation with an existing use case
- new use cases are defined and the uses relation is used to compose existing definitions into the new definition

Product-specific test cases are also created from the product-specific use cases and the test templates. These are combined with the product-line test cases that are applicable to this product to form the product-specific test suite.

The product specific test suites are created in the same way as the use case model. That is, we hope that the vast majority of product specific test cases will come from the product line test cases. Test templates will only be added at the product specific level if new use cases have been created. The test cases and test templates are then used to generate test code. The test code is executed when needed.

3.3 Tool chains

The steps in the technique described above are embodied in the tool chains described below.

The product line tool chain consists of:

- Ontology modeler - An Eclipse plugin for the Web Ontology Language (OWL) provides the standard editor shown in Figure 2. Domain experts will use that editor to define the entities that will comprise the DSL. Each element in the diagram, such as OWL Class Sprite, shown in editor will become a noun in the DSL.

- Feature modeler - The fmp Eclipse plug-in is used for feature modeling. The editor, with example output, is shown in Figure 3. The feature modeling notation uses closed circles to represent mandatory features, such as Player, and open circles to denote optional features, such as the scoreboard. The Game feature is a composite feature under which other features may be mandatory or optional.
- Editors - We use the JET technology, now a part of the Eclipse open source project, to generate the code for editors that work with a well-defined meta-model. Using the JET technology allows us to quickly regenerate editors when the input model is changed. This speeds the initial development and testing of the infrastructure.
- A use case editor is generated from the domain specific language. The editor, with example output, is shown in Figure 5. The input for the use case definition comes from the user of the editor selecting from menus of DSL elements. Constraints on the language are embodied in the editor which prevents the user from entering invalid language expressions. The stimulus section of the use case allows for stimuli coming from various actors although here we only show stimuli coming from a game user. The system's reaction is a list of actions on objects.
- The test case editor is shown in Figure 7. The figure shows a test case being edited for a specific value using the custom test case editor. The input to the test case editor comes from a transformation of each use case to a new meta-model.
- Template editor - Whenever a new template is required, a standard text editor is used to produce the code for the template. A segment of one of the test templates discussed below is shown in figure 4. The JET template definition language is similar to the Java Server Pages language.

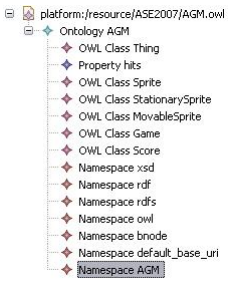

Figure 2: Ontology Editor

The product-specific tool chain consists of some of the same tools used by the product line team where the editors will be used to modify the product line assets. The product specific tool chain includes:

- Test case editor - The product specific test case editor is the same as the product line test case editor.
- Test execution environment - uses the test environment selected at test case construction time. JUnit, and the Abbot extensions, are two of the possible environments. Figure 6 shows the code for a test case

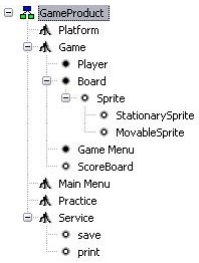

Figure 3: Feature Model Editor

```
... ... ...

tester.wait(new Condition() {
   public boolean test() {
   <c:iterate select="$currUseCase/stimulus/userActions"
      var="currAction">
      <c:choose
        select="$currAction/actionPerformed/@action">
        <c:when test="'PlayGame'">
            return (positionX !=
              brickles.getPuck().getPosition().x);
        </c:when>
        <c:when test="'HitBrick'">
            brickles.moveMouse(
              brickles.getPuck().getPosition());
            return
              (brickles.getBrickPile().getBrickCount()
              == <c:get select=
              "$currAction/attributeEffected/@value"/>);
        </c:when>
        <c:when test="'PuckLost'">
            return puckOrig != brickles.thePuck;
        </c:when>

... ... ...

<c:iterate select="$currUseCase/response/reactions"
       var="reaction">
    <c:choose select="$reaction/@reactingObject">
    <c:when test="'BrickCount'">
      assertTrue("New Brick Count",
            brickles.getBrickPile().getBrickCount()
            ==
            <c:get select="$reaction/@valueExpected"/>);
    </c:when>

... ... ...
```

Figure 4: Template Editor

for a product in the example discussed in the next section. This step is repeated any time the tester wishes to verify that the code still works as specified. The results of running the test cases are summarized by the execution environment and are available for the tester to use.

4. EXAMPLES

To better understand the effectiveness of our technique we have constructed several examples using the tool chain described previously and the core asset base for the Arcade Game Maker pedagogical product line [15]. In this section we first describe the product line which we used in the examples. We then describe the examples and lessons we learned from them.

The real power of automation is not in the executing test code but in the savings in resources needed to satisfy certain evolutionary scenarios. Rather than give static representations of dynamic execution, we describe examples in which a number of possible modifications are hypothesized, the actions required to accomplish the modifications are identified, and the implications of those actions examined. Due to space limitations we focus on modifications that affect system testing.

4.1 Pedagogical Product Line

We developed an example product line, intended for pedagogical purposes, for the Software Engineering Institute.

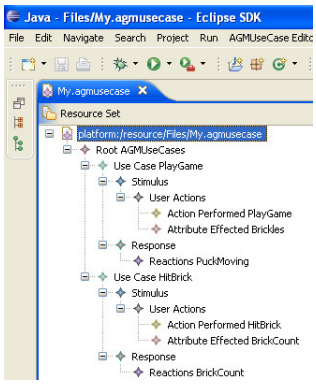

Figure 5: Use Case Editor

```
public void testTestMethod2_HitBrick() {
    //start the game
    brickles = new BricklesView();
    brickles.preinit(brickles);

    //wait until the stated condition is satisfied
    tester.wait(new Condition() {
        public boolean test() {
            brickles.moveMouse(
                    brickles.getPuck().getPosition());
            return (brickles.getBrickPile()
                    .getBrickCount() == 11);
        }
    });

    //make assertion based on expected result
    assertTrue("New Brick Count",
        brickles.getBrickPile().getBrickCount() ==  11);
}
```

Figure 6: Abbot Test Case Code

The product line is a set of video games produced by the fictitious company Arcade Game Maker (AGM) [15]. The product line is scoped to include three games: Brickles, Pong, and Bowling. Through the definition of several variation points, the three games will result in over nine different products.

The pedagogical product line has served as a testbed for a variety of examples. The example product line contains all of the major core assets that would be produced by a product line organization.

The examples in this section have been created from an implementation based on the J2SE SDK.

4.2 Examples of System Testing

Testing the completed system often requires the manipulation of a graphical interface rather than a programmatic interface. For this purpose we selected the Abbot GUI test tool for the test execution environment listed last in the product-specific tool chain described in section 3.3. We constructed several examples to observe the actions necessary to carry out several common activities. Here, we describe each example and list our observations.

4.2.1 Extension Example

In this example we investigated the effort required to add a new use case, either a new requirement or a refactored requirement, to the existing set. Specifically consider adding: "The game player modifies the game properties to use a different bitmap image for the puck". To add the new use case:

- The use case editor is used to open the existing use case model.
- A new use case is instantiated and completed. This includes adding a new identifier to designate the use case.

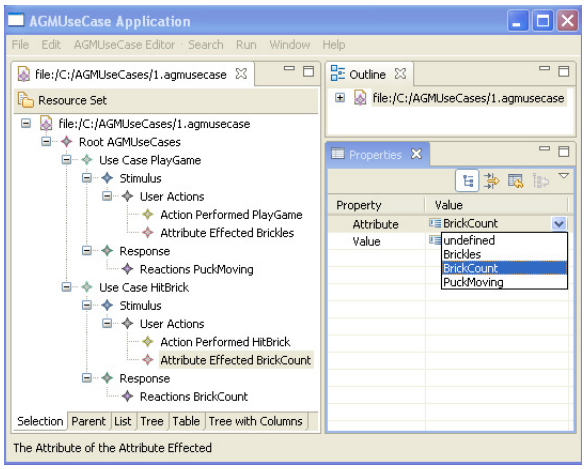

Figure 7: Generated Test Editor

- The test team examines the new use case to determine if it is already covered by a test case template.
- If no test case template exists that covers the circumstances described in the use case, a new test case template is created. This is done in the JET Template Editor.
- When the appropriate template is available, test cases are generated as source code using the test case generator.

We also investigated the effort required to add a new test case to the existing test suite. Consider adding a test case to test the performance of the game's simulation loop.

- The test case editor is used to instantiate a new test case.
- The tester selects the use case to be instantiated.
- The tester selects and enters the appropriate data for any variables for this test. The tester enters the identifier of the use case to maintain traceability between the requirements and the test suite.
- The code for the test is automatically generated and is ready to be run.

Finally, we investigated the effort required to extend the domain specific language. Consider that performance was not an explicit concept in the original game ontology. When performance is added:

- When a new concept is added to the ontology or a new feature is added to the feature model, appropriate phrases are added to the domain specific language. This is a manual operation.
- The addition of a new phrase to the DSL may trigger the need for a new test pattern if the new phrase implies an additional style of testing.
- The use case and test case editors must be re-generated before the new concept and related phrases can be used.
- New use cases are now possible and this would lead to the need for new test templates.

4.2.2 Modification Example

In this example we investigated the effort required to modify an existing use case. The original games involved perfect collisions. Consider changes resulting from changing the requirement to include a small amount of jitter to each collision.

- The use case editor is used to modify the high-level description of the use case.
- Individual stimuli or responses can be deleted or replaced.
- The identifier remains the same.

4.2.3 Refactoring Example

In this example we investigated the effort required to combine two existing independent use cases into one general use case and two more specific use cases. Originally there were use cases about (1) saving the state of the game and (2) printing the current state of the game. These were to be combined into one use case that defines "use of a service" and then two that describe the save service and the print service.

- The use case editor is used to open the set of use case descriptions.
- The user determines a strategy such as simply creating a new use case or modifying one and deleting the other.
- The user carries out the modification of the use cases as described in the cases described above.
- The tester modifies the test suite to reference the remaining use case and updates test cases to include a complete scenario.

These examples showed that all maintenance could be performed at the model level rather than the code level. This results in a large increase in productivity and will keep the models up-to-date at no additional cost. The results also show that most actions can be done automatically.

5. CONCLUSIONS

Our investigation focused on establishing a tool chain to support a model-driven development process for creating test cases. This investigation was in the context of a software product line development effort. Our technique uses a domain specific language as the language in which use cases and test cases are written. Taking advantage of test patterns, templates are created that support the automatic generation of executable test cases. Our experiments illustrated that operations on the use cases and the test cases can be fully carried out at the model level. Further, the experiments showed that most of the effort is expended at the product line level and this effort is then amortized over all of the software products that are built in the product line.

Much work remains. Additional test patterns need to be incorporated and the templates for all of the patterns need to be generalized. As our knowledge of template definition grows we are incorporating more functionality into the editors. We plan to investigate further the relationship between domain specific languages and automation.

6. REFERENCES

[1] D. L. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: a domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346, 1999.

[2] F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite approach to planning and deriving test suites in UML projects. *Lecture Notes in Computer Science*, volume 2460, 2002.

[3] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1995.

[4] A. Bertolino and S. Gnesi. Use case-based testing of product lines. In *Proceedings of the 9th European software engineering conference*, pages 355–358, September 2003.

[5] R. V. Binder. *Testing Object-oriented Systems: Models, Tools, and Patterns*. Addison-Wesley, 1999.

[6] L. C. Briand and Y. Labiche. A UML-based approach to system testing. *Software and System Modeling*, 1(1):10–42, 2002.

[7] P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Professional. Addison-Wesley, 2001.

[8] P. Fröhlich and J. Link. Automated Test Case Generation from Dynamic Models. In *Proceedings of ECOOP 2000-object-oriented programming: 14th European Conference*, 2000.

[9] E. Kamsties, K. Pohl, S. Reis, and A. Reuys. Testing variabilities in use case models. In *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, November 2003.

[10] Linguistic Computing Laboratory. http://lcl2.uniroma1.it/termextractor/index.jsp, 2007.

[11] M. Lange. It's testing time! patterns for testing software. In *Proceedings of EuroPLOP01*, 2001.

[12] B. Marick. http://www.testing.com/test-patterns/patterns/, 2007.

[13] J. D. McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, 2001.

[14] J. D. McGregor. Reasoning about the testability of product line components. In *Proceedings of the Second International Workshop on Software Product Line Testing*, 2005.

[15] J. D. McGregor. http://www.sei.cmu.edu/productlines/ppl, 2007.

[16] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.

[17] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *CSURV: Computing Surveys*, 37, 2005.

[18] C. Nebut, F. Fleurey, Y. Le Traon, and J. M. Jezequel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 2006.

[19] K. A. Schneider and J. R. Cordy. AUI: a programming language for developing plastic interactive software. *HICSS. Proceedings of the 35th Annual Hawaii International Conference on System Sciences, 2002.*, pages 3656–3665, 2002.

[20] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In *Second Conference on Domain Specific Languages*, pages 1–13, 1999.

[21] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.