FSE 2007

A Specification-based Approach to Testing Software Product Lines (Poster Paper)

Engin Uzuncaova
Dept. Of Electrical And Computer Engineering
The University of Texas at Austin
uzuncaov@ece.utexas.edu

Daniel Garcia
Dept. Of Electrical And Computer Engineering
The University of Texas at Austin
gdaniel@ece.utexas.edu

Sarfraz Khurshid
Dept. Of Electrical And Computer Engineering
The University of Texas at Austin
khurshid@ece.utexas.edu

Don Batory
Dept. Of Computer Sciences
The University of Texas at Austin
batory@cs.utexas.edu

ABSTRACT

This paper presents a specification-based approach for systematic testing of products from a software product line. Our approach uses specifications given as formulas in Alloy, a first-order logic based on relations. Alloy formulas can be checked for satisfiability using the Alloy Analyzer. The fully automatic analyzer, given an Alloy formula and a *scope*, i.e., a bound on the universe of discourse, searches for an *instance*, i.e., a valuation to the relations in the formula such that it evaluates to true. The analyzer translates an Alloy formula (for the given scope) to a propositional formula and finds an instance using an off-the-shelf SAT solver. The use of an enumerating solver enables systematic test generation.

We have developed a prototype based on the AHEAD theory. The prototype uses the recently developed Kodkod model finding engine of the Alloy Analyzer. We illustrate our approach using a data structure product line.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.2.4 [**Software Engineering**]: Software/Program Verification—*programming by contract*

General Terms: Reliability, Verification, Design

Keywords: Product-lines, specification-based testing, test data generation, GenVoca, AHEAD, Alloy

1. INTRODUCTION

Software product lines are rapidly gaining prominence as a means of automating software development. By defining a set of *features*—increments in program functionality—an implementation with desired functionality can automatically be synthesized by selecting the desired combination of features. Existing tools provide the ability to combine complex features and synthesize efficient implementations. As product lines become more popular and they are used in larger and more complex domains, developing techniques that can efficiently and systematically test the generated implementations emerges as a critical need. Moreover, combinations of certain features may introduce complex interactions among the features that must be thoroughly tested for correctness.

Specification-based testing provides an effective approach for testing correctness of software in general. The importance of using specifications in software testing was realized at least three decades ago [10], and approaches based on specifications are widely used today. A typical approach generates test inputs using an input specification and checks program correctness using an oracle specification—correctness criteria. Several existing approaches can automatically generate test inputs from a specification as well as execute the program to check its correctness.

Recent developments on tighter integration of specifications with programs via annotations [12] have made writing specifications intuitive and even more appealing. Moreover, recent advances in constraint solving and automated theorem proving [1] have enabled specification-based approaches to handle a large class of programs, including those that use advanced constructs of modern languages, and to scale to real applications and test them systematically [14].

This paper presents a specification-based testing approach for software product lines. Our approach uses specifications of properties given as formulas in Alloy [11], a first-order logic based on relations. Alloy formulas can be checked for satisfiability using the Alloy Analyzer. The fully automatic analyzer, given an Alloy formula and a *scope*, i.e., a bound on the universe of discourse, searches for an *instance*, i.e., a valuation to the primary variables (relations) in the formula such that it evaluates to true. The analyzer translates an Alloy formula (for the given scope) to a propositional formula and finds an instance using off-the-shelf SAT technology [15].

SAT solvers, e.g., mChaff [15], that can enumerate solutions enable the use of the Alloy Analyzer for systematic testing of Java programs: each solution represents a test input [14]. An Alloy formula that constrains relevant object fields to describe desired inputs is solved to enumerate instances. Each instance is translated into a Java object graph that forms a test input. The program is executed on each input, and its output is checked using another Alloy formula that characterizes the correctness of the program.

We present a novel use of Alloy to generate inputs for systematic testing of implementations synthesized from a product line. Each program in a product line is specified as a composition of features, where each feature represents an Alloy formula. The Alloy formula of a program is the composition of the formulas for each of the program's features. Tests are generated by solving the resulting formula.

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
ACM 978-1-59593-811-4/07/0009.

The contribution of this paper is to show how specification-based testing can be applied to product lines in a *traditional* fashion, where tests are generated for a product from its *complete* specification. In ongoing work, we are investigating an *incremental* approach, where previously generated tests are *refined* using changes in specification.

2. BACKGROUND

Categorical models of product-lines

A *feature* is an increment in program functionality. A *software product-line* (SPL) is a family of programs where no two programs have the same combination of features.

Every program in an SPL has multiple representations or models (e.g., source, documentation, etc.). Adding a feature to a program refines each of the program's representations. Furthermore, some representations can be derived from other representations. These ideas have a compact form when cast in terms of metaprogramming and category theory. We show below how this is done by a progression of models: GenVoca [6], AHEAD [7], and FOMDD [3,19].

2.1 GenVoca

GenVoca is a metaprogramming model that we have used to build feature-based product lines for over fifteen years [5–7]: base programs are values and features are functions that map programs to feature-refined programs. A GenVoca model `M = {f,h,i,j}` of a product-line is an algebra, where constants are base programs:

```
f      // a base program with feature f
h      // a base program with feature h
```

and functions are *program refinements*:

```
i•x    // adds feature i to program x
j•x    // adds feature j to program x
```

where • denotes function composition.

Note that even though we write the composition of features `a` and `b` as `a•b`, it really is an abbreviation of the expression `compose(a,b)`. We use • to simplify expressions.

The design of a program is a named expression, e.g.:

```
p1 = j•f      // p1 has features j and f
p2 = i•j•h    // p2 has features i, j, h
p3 = j•h      // p3 has features j and h
```

The set of programs that can be defined by a GenVoca model is its product-line. Expression optimization corresponds to program design optimization, and expression evaluation corresponds to program synthesis [4,17].

The use of one feature may preclude the use of some features or may demand the use of others. Tools that validate compositions of features are discussed elsewhere [2].

2.2 AHEAD

Every program has multiple representations or models: a program has source code, documentation, bytecode, makefiles, UML designs, etc. A vector of representations for a program is a GenVoca constant. Base program `f`, for example, has a statechart model c_f, a Java source code representation s_f derived from its statechart model, and a Java bytecode representation b_f derived from its source. Program `f`'s vector is $f = [c_f, s_f, b_f]$.

A GenVoca function maps a vector of program representations to a vector of refined representations. For example, feature `j` simultaneously refines `f`'s statechart model (to specify `j`), its source code (to implement `j`), and its bytecode (to execute `j`). If Δc_j is statechart refinement made by `j`, Δs_j and Δb_j are the corresponding refinements of source and bytecode, function j is the vector $j = [\Delta c_j, \Delta s_j, \Delta b_j]$.

The representations of a program, such as p_1, are synthesized by composing each base model with its refinement:

```
p1 = j•f                          // GenVoca expression
   = [Δc_j, Δs_j, Δb_j]•[c_f, s_f, b_f]
   = [Δc_j•c_f, Δs_j•s_f, Δb_j•b_f]
```

That is, the statechart of p_1 is produced by composing the base statechart with its refinement ($\Delta c_j \bullet c_f$), the source code of p_1's base with its refinement ($\Delta s_j \bullet s_f$), etc.

2.3 Feature oriented model driven design

AHEAD captures the lockstep refinement of program representations when a feature is composed with a program. But there are additional functional relationships among different representations that AHEAD does not capture. For example, the relationship between Java source s_f of program `f` and its bytecode b_f is expressed by `javac`. That is, `javac` is a transformation that maps s_f to b_f. Similarly, one can imagine a transformation τ that maps a statechart c_f to its Java source s_f. Unlike features that represent refinement relationships between artifacts, these transformations represent derivation relationships between artifacts.

All of these relationships are expressed by a *commuting diagram*, where objects denote program representations, downward arrows represent derivations and horizontal arrows denote refinements. These objects and arrows define a category [16]. Figure 1 shows the commuting diagram for program $p2 = i \bullet j \bullet h = [c_2, s_2, b_2]$.

$$\begin{tikzcd}
\boxed{c_h} \arrow[r, "\Delta c_j"] \arrow[d, "\tau"'] & \boxed{c_3} \arrow[r, "\Delta c_i"] \arrow[d, "\tau"] & \boxed{c_2} \arrow[d, "\tau"] \\
\boxed{s_h} \arrow[r, "\Delta s_j"] \arrow[d, "javac"] & \boxed{s_3} \arrow[r, "\Delta s_i"] \arrow[d, "javac"] & \boxed{s_2} \arrow[d, "javac"] \\
\boxed{b_h} \arrow[r, "\Delta b_j"] & \boxed{b_3} \arrow[r, "\Delta b_i"] & \boxed{b_2}
\end{tikzcd}$$

Figure 1: Commuting diagram.

A fundamental property of a commuting diagram is all paths between two objects produce equivalent results. For example, one way to derive the bytecode b_2 of program p_2 (lower right in Figure 1) from the statechart c_h of program `h` (upper left) is to immediately derive the bytecode b_h and refine to b_2, while another path immediately refines c_h to c_2, and then derives b_2:

$$\Delta b_i \bullet \Delta b_j \bullet \mathtt{javac} \bullet \tau = \mathtt{javac} \bullet \tau \bullet \Delta c_i \bullet \Delta c_j$$

There are $\binom{4}{2} = 6$ possible paths to derive the bytecode b_2 of program p_2 from the statechart c_h of program `h`. Each path represents a metaprogram whose execution synthesizes the target object (b_2) from the starting object (c_h).

Traversing arrows of commuting diagrams has a cost. It is an optimization problem to determine the shortest (least-cost) path between two objects in a commuting diagram.

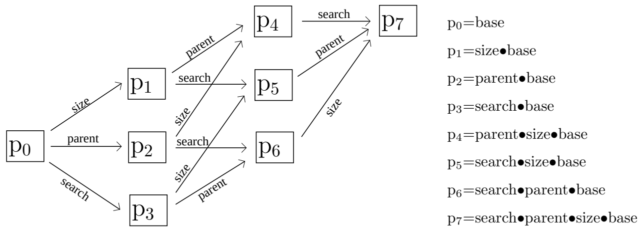

Figure 2: Family of binary trees. Nodes represent products. Arrows represent feature inclusion.

3. OUR APPROACH ILLUSTRATED

We describe our specification-based testing approach using an illustrative example. We present the example using AHEAD [8] and Alloy [11] notations, which we describe as we introduce them; details are available elsewhere [8, 11].

3.1 A product line of binary trees

Consider a family of binary trees [9]. While all trees in this family are acyclic, they are differentiated on whether their nodes have parent pointers, or whether they have integer values satisfying search constraints, or whether the trees cache the number of their nodes. The following AHEAD model describes this family:

```
BT = {base, size, parent, search}
```

where the `base` product is an acyclic binary tree [9], which can be extended using a combination of three independent (i.e., commutative) features:

- `size`: the tree caches the number of nodes in a field;
- `parent`: nodes have parent pointers;
- `search`: nodes have integer values, which satisfy the search constraints of a binary search tree

The model `BT` represents a product-line of different trees. A particular tree is defined by an *feature expression*. For example, the expression `p = parent•base`, where the operator '•' denotes feature composition, defines a tree with parent pointers, and the expression `s = search•base` defines a binary search tree. Syntactically different expression may define equivalent products, e.g., since `size` and `parent` are independent, `size•parent•base = parent•size•base`. Figure 2 characterizes the eight distinct products of the `BT` family.

3.2 Alloy annotated Jakarta code

We next describe the basic class declarations and specifications that represent the `BT` family. The following annotated code declares the base class:

```
class BinaryTree {
    /*@ invariant
      @ all n: root.*(left + right) {
      @   n !in n.^(left + right) // no directed cycles
      @   lone n.~(left + right) // at most one parent
      @   no n.left & n.right // left and right nodes differ
      @ }
      @*/

    Node root;
}

class Node {
    Node left, right;
}
```

A binary tree has a `root` node and each node has `left` and `right` children. The `invariant` annotation in comments states the *class invariant*, i.e., a constraint that a `BinaryTree` object must satisfy in any publicly visible state, such as a *pre-state* of a method execution [13].

The invariant is written as a universally quantified (keyword `all`) Alloy formula. The operator '`.`' represents relational composition; '`+`' is set union; and '`*`' is reflexive transitive closure. The expression `root.*(left + right)` represents the set of all nodes reachable from `root` following zero or more traversals along `left` or `right` edges. The invariant formula universally quantifies over all reachable nodes. It expresses three properties that are implicitly conjoined. (1) There are no directed cycles; (the operator '`!`' denotes negation and '`^`' denotes transitive closure; the keyword `in` represents set membership). (2) A node has at most one parent; (the operator '`~`' denotes relational transpose; the keyword `lone` represents a cardinality constraint of less than or equal to one on the corresponding set). (3) A node does not have another node as both its `left` child and its `right` child; (the operator '`&`' denotes set intersection).

The AHEAD tool-set provides a veneer, Jakarta, on Java to facilitate development of product lines [5]. The following Jakarta code uses the keyword `refines`, which represents extension, to introduce the feature `size`:

```
refines class BinaryTree {
    /*@ refines invariant
      @ size = #root.*(left + right)
      @*/

    int size;
}
```

Note (1) the new field `size` in class `Node` and (2) the additional invariant that represents the correctness of `size`: the value of `size` field is the number of nodes reachable from root (inclusive). The Alloy operator '`#` denotes cardinality of a set.

We extend the base to introduce `parent` as follows.

```
refines class BinaryTree {
    /*@ refines invariant
      @ no root.parent
      @ all m, n: root.*(left + right) |
      @   m in n.(left + right) <=> n = m.parent
      @*/
}

refines class Node {
    Node parent;
}
```

The correctness of `parent` is: (1) root has no `parent` node (i.e., `root.parent == null`); and (2) if node `m` is the `left` or `right` child of node `n` then `n` is the parent of `m` and vice versa.

We extend the base to introduce `search` as follows.

```
refines class BinaryTree {
    /*@ refines invariant
      @ all n: root.*(left + right) {
      @   all nl: n.left.*(left + right) | n.elem > nl.elem
      @   all nr: n.right.*(left + right) | n.elem < nr.elem
      @ }
      @*/

}

refines class Node {
    int elem;
}
```

The `search` constraint requires that the elements in the tree appear in the correct search order: all elements in the left sub-tree of a node are smaller than the element in the node (and those in the right-subtree larger).

3.3 Test generation

We next illustrate how to generate inputs for methods defined in implementations of the products in the binary tree family. Since an input to a (public) method must satisfy its class invariant, we must generate *valid* inputs, i.e., inputs that satisfy the invariant. To illustrate, consider testing the method "`int size() { ... }`", say in product $\texttt{p}_3$.

The method takes one input (the implicit input `this`). Generating a test input for method `size` requires solving $\texttt{p}_3$'s class invariant, i.e., acyclicity and binary search constraints (from Figure 2). Given the invariant in Alloy and a bound on the input size, the Alloy Analyzer can systematically enumerate all structures that satisfy the invariant; each structure represents a valid input for `size` (and other methods that take one tree as input). Given $\texttt{p}_3$'s invariant, the analyzer takes 2.83 seconds to generate 100 trees with exactly 10 nodes (1.8 GHz Pentium M with 512 MB RAM).

4. ONGOING WORK

Solving a constraint using a single invocation of the analyzer is inspired by its traditional use. We are currently investigating an *incremental* approach: the analyzer is invoked multiple times to generate a desired structure, where each invocation is on a SAT formula that is simpler than the formula that represents the complete constraint.

$$\begin{tikzcd}
\boxed{\texttt{s}_0} \arrow[r, "\Delta_s"] \arrow[d, dotted, "\tau"'] & \boxed{\texttt{s}_3} \arrow[d, "\tau"] \\
\boxed{\texttt{t}_0} \arrow[r, dotted, "\Delta_t"'] & \boxed{\texttt{t}_3}
\end{tikzcd}$$

Figure 3: Binary search tree commuting diagram.

The *commuting diagram* in Figure 3 illustrates how the incremental approach differs from the traditional approach. The nodes $\texttt{s}_i$ represent specifications for test generation for the corresponding products, e.g., $\texttt{s}_0$ represents the base specification, which is just the acyclicity constraint. The nodes $\texttt{t}_i$ represent the corresponding sets of test inputs. The horizontal arrow Δ_s represents the refinement of the class invariant, i.e., addition of search constraints. The vertical arrows τ represent test generation using Alloy Analyzer. Δ_t represents a transformation of tests for the base product into tests for `search•base`.

To generate tests $\texttt{t}_3$, the traditional approach follows the path $\tau \bullet \Delta_s$. The incremental approach follows the alternative (but equivalent) path $\Delta_t \bullet \tau$ (dotted arrows). A key problem in traversing this path is definition of the function Δ_t. One solution is to let the user provide the function. For example, the user could provide a *test refinement function* that takes a test for $\texttt{p}_0$ and transforms it into a test for $\texttt{p}_3$. An alternative solution is to automatically compute Δ_t, given Δ_s. We are investigating the use of the Alloy Analyzer for this computation: using $\texttt{t}_0$ as a *seed* that provides a valuation to relations in $\texttt{s}_0$, the analyzer solves the new constraints Δ_s to generate a valuation for the new relations in $\texttt{s}_3$, thereby generating test $\texttt{t}_3$. Preliminary results show that the incremental approach holds much promise.

Acknowledgments

This work was funded in part by NSF's Science of Design Awards #CCF-0438786 and #IIS-0438967.

5. REFERENCES

[1] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *CAV*, 2004.

[2] D. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, 2005.

[3] D. Batory. From implementation to theory in program synthesis. In *Keynote at POPL*, 2007.

[4] D. Batory, G. Chen, E. Robertson, and T. Wang. Design wizards and visual programming environments for genvoca generators. *Software Engineering*, 26(5):441–452, 2000.

[5] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proc. International Conference on Software Reuse*, 1998.

[6] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions of Software Engineering Methodology*, 1(4):355–398, 1992.

[7] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE*, 2003.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[10] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.

[11] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.

[12] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.

[13] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[14] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, 2001.

[15] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, 2001.

[16] B. C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, Cambridge, MA, 1991.

[17] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[18] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.

[19] S. Trujillo, D. Batory, and O. Diaz. Feature oriented model driven development: A case study for portlets. In *ICSE*, 2007.