Testing Software Product Lines Using Incremental Test Generation

Engin Uzuncaova Daniel Garcia Sarfraz Khurshid
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
{uzuncaov,gdaniel,khurshid}@ece.utexas.edu

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

Abstract

We present a novel specification-based approach for generating tests for products in a software product line. Given properties of features as first-order logic formulas, our approach uses SAT-based analysis to automatically generate test inputs for each product in a product line. To ensure soundness of generation, we introduce an automatic technique for mapping a formula that specifies a feature into a transformation that defines incremental refinement of test suites. Our experimental results using different data structure product lines show that incremental approach can provide an order of magnitude speed-up over conventional techniques.

1 Introduction

The goal of software product lines is the systematic and efficient creation of products. Features are used to specify and distinguish products, where a *feature* is an increment in product functionality. Each product is defined by a unique combination of features. As product line technologies are applied to progressively more complex domains, the need for a systematic approach for product testing becomes more critical.

Specification-based testing [6, 12] provides an effective approach for testing the correctness of software in general. The importance of using specifications in testing was realized at least three decades ago [12], and approaches based on specifications are widely used today. A typical approach generates test inputs using an input specification and checks the program 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 outputs [15, 20].

For programs written in object-oriented languages, which are in common use for developing software product lines, a suitable specification language is Alloy [14]—a declarative, first-order language based on relations. Alloy's relational basis and syntactic support for path expressions enable intuitive and succinct formulation of structurally complex properties of heap-allocated data structures, which pervade object-oriented programs. The Alloy Analyzer [13]—a fully automatic tool based on propositional satisfiability solvers—enables both test generation and correctness checking [15]. Given an Alloy formula that represents desired inputs, the analyzer solves the formula using a given bound on input size and enumerates the solutions. Test inputs are generated by translating each solution into a concrete object graph on which the program is executed. Correctness of the program is then checked using another Alloy formula that represents the expected relation between inputs and outputs.

While the analyzer provides the necessary enabling technology for automated testing of programs with structurally complex inputs, test generation using the analyzer at present does not scale and is limited to generating small inputs (e.g., an object graph with less than ten nodes). To enable systematic testing of real applications we need novel approaches that scale to generation of larger inputs. The need is even greater for software product lines due to the current lack of support for analytical approaches for testing in this domain as well as due to the combinatorial nature of feature compositions [7].

This paper presents a novel approach for efficient test generation by combining ideas from software product lines and specification-based testing using Alloy. The novelty of our work is two-fold. First, each product is specified as a composition of features, where each feature is specified as an Alloy formula. An Alloy property of a program in a product line is thus specified as a composition (conjunction) of the Alloy formulas for each of the program's features. Second, the *conventional* use of the analyzer solves a complete specifica-

1071-9458/08 $25.00 © 2008 IEEE
DOI 10.1109/ISSRE.2008.56

tion of a program to generate tests. Instead, we use the Alloy Analyzer to perform test generation *incrementally*; that is, we execute the analyzer more than once but on *partial* specifications, which are ideally easier problems to solve. To ensure soundness of generation, we introduce an automatic technique into our tool for mapping a formula that specifies a feature into a transformation that that defines incremental refinement of test suites. We present experimental results on a variety of data structure product lines that show incremental test generation can provide an order of magnitude speed-up over the conventional use.

In a poster paper last year [24], we illustrated the use of Alloy for product line testing in the conventional fashion where tests are generated from a complete specification using a single execution of the analyzer. This paper makes the following new contributions:

- **Incremental test generation.** We introduce the notion of incremental generation of tests for testing products from a product line;
- **Mapping.** We define a mapping from a feature specification to a transformation among test suites and show how to perform it automatically;
- **Implementation.** Our prototype implementation uses the AHEAD and Alloy tool-sets to automate testing of product lines; and
- **Evaluation.** Experiments using a variety of data structure product lines that have intricate specifications show that our approach provides significant speed-ups over conventional techniques.

2 Example

This section illustrates a simple product line of data structures. We use AHEAD [4] and Alloy [14] notations to explain our ideas. Section 5 presents a more sophisticated example.

2.1 A product line of binary trees

Consider a family of binary trees. 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 `base` product is an acyclic binary tree, which can be extended using a combination of three independent features: `size`, `parent`, and `search`. We denote the collection of the base program and its features as an AHEAD model `BT = {base, size, parent, search}`.

A tree is defined by an *expression*. For example, the expression `p = parent•base`, where '`•`' denotes feature composition, defines a tree with parent pointers,

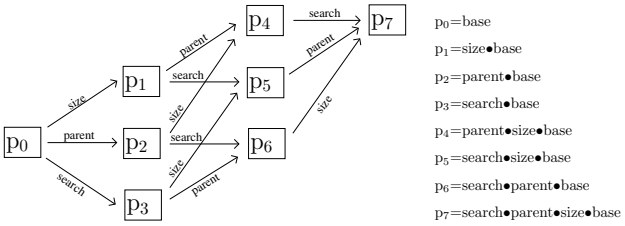

Figure 1. Family of binary trees.

and similarly, the expression `s = search•base` defines a binary search tree(BST). Syntactically different expressions may be equivalent, e.g., `size•parent•base = parent•size•base` since `size` and `parent` are independent (i.e., commutative). Figure 1 characterizes the eight distinct products of the `BT` family.

2.2 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 classes:

```
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 a `left` and a `right` child. 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 [16].

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).

AHEAD provides a veneer, Jakarta, on Java to facilitate development of product lines. The following Jakarta code uses the keyword `refines`, which denotes extension, to introduce the state that represents the feature `size` and the refinement of the invariant:

```
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. When this refinement is applied to our original definition of `BinaryTree`, the `size` field is added to `BinaryTree` and the the new invariant is the conjunction of the original invariant with the `size` refinement.

Similarly, we extend the base to introduce the state representing the feature `parent` by refining class `BinaryTree` and its invariant, and adding a new member to class `Node`:

```
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 element; }
```

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 its element and those in the right-subtree larger.

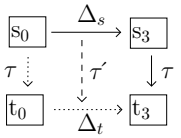

Figure 2. BST commuting diagram.

2.3 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 `size` method in product p_5 = `search•size•base`:

```
    // returns the number of nodes in the tree
   int size() { ... }
```

The method takes one input (the implicit input `this`). Generating a test input for method `size` requires solving p_5's class invariant, i.e., acyclicity, size, and binary search constraints (from Figure 1). 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 p_5's invariant, the analyzer takes 62.06 seconds on average to generate a tree with 10 nodes: This represents the conventional use of the analyzer.

We use *incremental* solving to generate a desired test (Section 4). The *commuting diagram* in Figure 2 illustrates how our approach differs from the conventional approach. The nodes s_i represent specifications for test generation for the corresponding products, e.g., s_0 represents the base specification—the acyclicity constraint. The nodes t_i represent the corresponding sets of test inputs. The horizontal arrow Δ_s represents a refinement of the class invariant, i.e., the 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`; Δ_t is computed from Δ_s and t_0 using the analyzer (Section 4). To generate tests t_3, the conventional approach follows the path $\tau \bullet \Delta_s$. Our approach follows the alternative (but equivalent) path $\Delta_t \bullet \tau$ (dotted arrows).

Given p_5's invariant, we invoke the analyzer thrice. The total time it takes to generate a tree with exactly 10 nodes is 1.13 seconds on average, which is a 55× speed-up. Since our approach re-uses tests already generated for another product, when testing each product in a product line, the overall speed-up can be significantly greater. Detailed results are presented later in Section 5.2.

3 Feature Orientation

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 [1]. In the following sections, we sketch the basic ideas of feature orientation by a progression of models.

3.1 AHEAD

GenVoca is a model of software product lines that is expressed purely in terms of transformations. Base programs are values (0-ary functions) and features are unary functions (i.e., transformations) that map programs to feature-extended programs, where • denotes function composition:

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

AHEAD extends GenVoca [3] by revealing the internal structure of GenVoca values and function as tuples. Every program has multiple representations: a program has source code, documentation, bytecode, makefiles, UML designs, etc. A GenVoca value is a tuple of representations. Base program `f`, for example, has a statechart model $\mathtt{c}_f$, a Java source code representation $\mathtt{s}_f$, and a Java bytecode representation $\mathtt{b}_f$. Program `f`'s tuple is $\mathtt{f} = [\mathtt{c}_f, \mathtt{s}_f, \mathtt{b}_f]$.

A GenVoca function maps a tuple of program representations to a tuple of updated representations. For example, feature `j` simultaneously refines `f`'s statechart model (that adds new states and transitions, and extends existing actions), its source code (that adds new classes, new members to existing classes, and extends existing methods), and its bytecode (to execute `j`). If $\Delta\mathtt{c}_j$ is the statechart refinement made by `j`, $\Delta\mathtt{s}_j$ and $\Delta\mathtt{b}_j$ are the corresponding refinements of source and bytecode, function j is the tuple $\mathtt{j} = [\Delta\mathtt{c}_j, \Delta\mathtt{s}_j, \Delta\mathtt{b}_j]$.

The representations of a program, such as $\mathtt{p}_1$, are synthesized by tuple composition:

$$\begin{aligned}\mathtt{p}_1 &= \mathtt{j}\bullet\mathtt{f} \qquad \text{// GenVoca expression}\\ &= [\Delta\mathtt{c}_j, \Delta\mathtt{s}_j, \Delta\mathtt{b}_j]\bullet[\mathtt{c}_f, \mathtt{s}_f, \mathtt{b}_f]\\ &= [\Delta\mathtt{c}_j\bullet\mathtt{c}_f, \Delta\mathtt{s}_j\bullet\mathtt{s}_f, \Delta\mathtt{b}_j\bullet\mathtt{b}_f]\end{aligned}$$

That is, the statechart of $\mathtt{p}_1$ is produced by composing the base statechart with its refinement ($\Delta\mathtt{c}_j\bullet\mathtt{c}_f$), the source code of $\mathtt{p}_1$'s base with its refinement ($\Delta\mathtt{s}_j\bullet\mathtt{s}_f$), etc. This is the essence of AHEAD [4].

3.2 Feature-oriented model driven design

AHEAD captures the lockstep refinement of program representations when a feature is composed with a program. However, AHEAD does not capture derivation relationships among program representations. *Feature-Oriented Model Driven Development (FOMDD)* generalizes AHEAD to include derivation relationships [23]. To illustrate, the relationship between Java source $\mathtt{s}_f$ of program `f` and its bytecode $\mathtt{b}_f$ is expressed by `javac`. That is, `javac` is a transformation that derives $\mathtt{b}_f$ from $\mathtt{s}_f$. Similarly, one can imagine a transformation ϕ that derives the Java source $\mathtt{s}_f$ from its statechart $\mathtt{c}_f$.

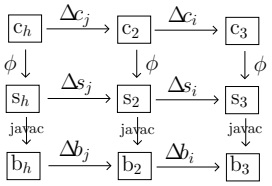

Figure 3. Commuting Diagram.

Refinement and derivation relationships are expressed by a *commuting diagram*, where objects denote program representations, downward arrows represent derivations and horizontal arrows denote refinements. Figure 3 shows the commuting diagram for program `p3` $= \mathtt{i}\bullet\mathtt{j}\bullet\mathtt{h} = [\mathtt{c}_3, \mathtt{s}_3, \mathtt{b}_3]$.

4 Our Approach

This section describes our specification-based approach for test generation for systematic testing of implementations synthesized from an SPL.

4.1 FOMDD model

For specification-based testing, the FOMDD models of our SPLs are defined as follows. Each program `p` of an SPL can be viewed as a pair: a specification `s` and a set of test inputs `t`, i.e., `p = [s, t]`. A feature `f` refines both a specification ($\Delta\mathtt{s}_f$) and its test suite ($\Delta\mathtt{t}_f$).

In specification-based testing, the user provides `s` and Δ`s`. To generate tests, we need a transformation τ that maps a specification `s` to its corresponding tests `t`. Also implementing test refinement Δ`t` enables alternative techniques for test generation. We use the Alloy Analyzer to implement τ. In addition, we use the analyzer to implement transformation τ' that automatically computes Δ`t`: τ' maps a test suite `t` and a specification refinement Δ`s` to a corresponding test refinement Δ`t`. Figure 2 shows the commuting diagram that corresponds to program $\mathtt{p}_0 = [\mathtt{s}_0, \mathtt{t}_0]$ composed with feature `search`.

4.1.1 Objects

An Alloy formula consists of a first-order logic constraint over primary variables (relations). An Alloy instance represents a valuation to these relations such that the formula evaluates to true. Mathematically, an

[1] Software product lines can be modeled in terms of functional and non-functional properties. Our approach focuses on functional properties as described in Section 3.

instance `i` is a function from a set of relations `R` to a power set of tuples $2^{\mathtt{T}}$ where each tuple consists of indivisible atoms, i.e., `i: R -> 2^T`, where `T` is the set of all tuples in the bounded universe of discourse. Thus, for each Alloy relation, an instance gives a set of tuples that represents a value of the relation.

Recall that to solve a formula, the Alloy Analyzer uses a scope that bounds the universe of discourse. The Kodkod back-end of the Alloy Analyzer [22] allows a scope to be specified using two bounds: a lower bound and an upper bound on the set of tuples that any valuation of a relation may take. Any instance must satisfy the following property: for every relation, each tuple in the lower bound must be present in the instance and no tuple that is not in the upper bound may be present in the instance. Mathematically, a bound `b` is a pair of two functions: a lower bound `l` and an upper bound `u`, each of type `R -> 2^T`. An instance can equivalently be viewed as bound `b = [l, u]` where `l = u`.

Thus, in our model, a specification `s` is a pair of a formula `f` and a bound `b`, i.e., `s = [f, b]`; a test suite `t` is a set of instances.

The specification refinement arrow $\Delta\mathtt{s}$ for specification `s = [f, b]` may refine the formula `f` or the bound `b` or both, i.e., and $\Delta\mathtt{s}$ `= [`Δ`f, `Δ`b]`. AHEAD's Jakarta notation provides the keyword `refines` to denote refinement. We overload this keyword to represent refinement of specifications. Refinement of a formula `f` transforms it into formula `f` $\wedge$ $\Delta\mathtt{f}$, where $\Delta\mathtt{f}$ represents the additional constraint. Refinement of a bound further restricts the lower or the upper bound or both.

The transformation arrow τ represents test generation from the given specification. The test suite refinement arrow $\Delta\mathtt{t}$ enables an alternative test generation technique. The transformation arrow τ' is a function from a test suite and a specification refinement to a test suite refinement. Implementing τ' provides an implementation for $\Delta\mathtt{t}$.

4.1.2 Paths

In a commuting diagram, all paths that start at a desired specification and terminate at a desired test suite are equivalent, i.e., following any path gives the same test suite (up to isomorphism), in particular $\tau \bullet \Delta\mathtt{s} = \Delta\mathtt{t} \bullet \tau$. However, not all paths have the same associated cost, i.e., test generation along certain paths can be more efficient than others. Note that in the presence of feature interactions (Section 6), it may not be practical to traverse some $\Delta\mathtt{t}$ arrows.

4.2 Test generation

Implementations of transformations τ and τ' enable alternative techniques for test generation for products from a product line. The conventional use of the Alloy Analyzer allows a fully automatic implementation of τ: execute the analyzer on specification `s` and enumerate its instances. However, the conventional use of the analyzer restricts any path (in a commuting diagram) from a specification `s` to a test suite `t` to contain horizontal arrows that are labeled $\Delta\mathtt{s}$ only. This restriction requires performing transformation τ *after* all specification refinements have been performed, i.e., constraint solving is performed on the most complex of the specifications along any equivalent path.

```
TestSuite τ'(SpecificationRefinement Δs,
             TestSuite suite) {
    TestSuite suite' = ∅;
    Formula formula = Δs.formula();
    foreach (Test test:  suite) {
        Bound bound = Δs.bound().update(test);
        suite' = suite' + Alloy.solve(formula, bound);
    }
    return suite';
}
```

Figure 4. Test refinement algorithm.

As specification formulas become more complex, execution of τ becomes more costly. For example, the analyzer takes one minute to generate an acyclic structure with 35 nodes. In contrast, the generation of an acyclic structure that also satisfies search constraints with only 16 nodes does not terminate in 1 hour.

4.2.1 Algorithm

We provide an algorithm (Figure 4), which enables a fully automatic implementation of the transformation τ'. The algorithm assumes the monotonicity of feature semantics: when feature f is composed with base b, the resulting product's properties are a conjunction of b's properties and f's properties (Section 7). The impact of feature interactions on incremental test generation is discussed in Section 6.

The algorithm takes as input a test suite `t` and a specification refinement $\Delta\mathtt{s}$, and computes a new test suite, which refines the tests in t with respect to the constraints in $\Delta\mathtt{s}$. The algorithm enables an incremental approach to test generation using successive applications of test refinement: to generate tests for a product that is composed of a base and a desired set of features, first generate a test suite for the base, and then iteratively refine the suite with respect to each of the features. In the specification-tests commuting diagram, we thus follow the path that starts with a vertical τ arrow and then consists solely of horizontal $\Delta\mathtt{t}$ arrows. Indeed, our algorithm also enables other paths

to be followed in the commuting diagram and hence it enables new approaches for test generation (Section 6).

The algorithm transforms each test from the given suite into a test for the new suite. Incorporating the old test into the bound for the analyzer's search guarantees the satisfaction of old constraints; in addition, the new solution includes valuations for the new relations introduced by the feature and satisfies the new constraints on these relations. Indeed, for features that constrain existing relations, the Alloy Analyzer may be unable to refine certain original tests, in which case the algorithm filters them out.

In general, our algorithm τ' implements an arbitrary *relation* from a given test suite (`suite`) and a specification refinement (Δs) to a desired test suite: (1) a particular test in `suite` may be refined into several new tests; and (2) certain tests in `suite` may not be refined and just ignored by the algorithm. A common case is when each test is refined to (at most) one test, i.e., τ' is a (partial) *function*. Note that τ' may not map two distinct tests onto the same new test (because the values of relations in original tests are not modified), i.e., τ' is *injective*.

Illustration. Consider the commuting diagram for binary search trees (Figure 2). The following valuation represents a test input `i` from test suite $\mathtt{t}_0$ for the base specification formula `acyclic`, as shown in Figure 5 (a):

```
BinaryTree = { BT0 }
Node = { N0, N1, N2 }
root = { <BT0, N0> }
left = { <N0, N1> }
right = { <N0, N2>}
```

Now consider transforming the test `i` into a test `i´` for the specification formula of $\mathtt{s}_3$, which represents `acyclic` $\wedge$ `search`. We run the analyzer on the formula `search` and set the lower and upper bounds for `BinaryTree`, `Node`, `root`, `left` and `right` to the values in input `i`. The analyzer generates `i´` by adding to the relations in `i` the new relations `element` and `Int` that models a set of integers:

```
Int = { 0, 1, 2 }
element = { <N0, 1> <N1, 0>, <N2, 2> }
```

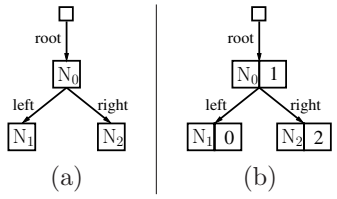

Figure 5. Test inputs. (a) An acyclic binary tree. (b) An acyclic binary search tree with elements 0, 1, and 2.

Figure 5 (b) graphically illustrates this tree, which is indeed a binary *search* tree.

Correctness. We next argue the soundness and completeness (with respect to the given input bounds) of our approach. We outline a simple induction argument. Consider generating tests for product $p_n = f_n \bullet \ldots \bullet f_1 \bullet f_0$, where f_0 is a base product and each f_i $(i > 0)$ is a feature. The induction base case holds trivially since the tests for the base are generated using a direct application of the Alloy Analyzer. For the induction step, consider generating test suite t_{k+1} for product p_{k+1} using test suite t_k for product p_k, where t_k consists of exactly all the valid tests for p_k.

The soundness follows from the fact that the invocation of the analyzer does not change any values of relations that appear in p_k. Thus, constraints for p_k continue to be satisfied. Moreover, since the analyzer directly solves the constraints in the specification refinement, any solution it generates satisfies the additional constraints of p_{k+1} by definition. Thus, if the invocation of the analyzer returns a solution, it satisfies all constraints for p_{k+1}. (Indeed, some tests for p_k may simply be filtered out.)

The completeness follows from the monotonicity of feature semantics: any valid test input for a product must satisfy properties of *all* its features. Let i_{k+1} be an arbitrary valid test input for p_{k+1}. Let i_k be an input that has the same values as i_{k+1} for all relations in p_k and contains no other values for any relation. Then by the monotonicity property, i_k is a valid input for p_k. Thus, by the induction hypothesis, $i_k \in t_k$. Therefore, the `foreach` loop performs an iteration that refines i_k. Since the analyzer enumerates all solutions, i_k can spawn several new inputs and the output of the solver includes all of them. Thus, one of the solutions returned by its invocation must be i_{k+1} (up to isomorphism). Hence, i_{k+1} is generated by the algorithm. Therefore, all valid inputs for p_{k+1} are generated.

5 Evaluation

The section presents an evaluation of our incremental approach to test generation using two subject product lines: binary trees and *intentional names* [1]. Section 2 introduced the binary tree product line. Section 5.1 describes the intentional naming product line. We tabulate and discuss the results for enumerating test inputs using the conventional approach and our incremental approach (Section 5.2).

All experiments were performed on a 1.8GHz Pentium M processor using 512MB of RAM. All SAT formulas were solved using MiniSat [11]. Our tool Kesit uses the Java API of the Kodkod back-end [22] of the Alloy Analyzer.

product	conventional			incremental					speed up
	vars	clause	total time	refine-ment	vars	clause	time ref	time total	
Binary Search Tree (scope=10)									
base	210	19618	19	n/a	n/a	n/a	n/a	n/a	n/a
size • *base*	242	20905	23	*size*	32	1092	21	40	0.58×
parent • *base*	310	21404	21	*parent*	100	442	12	29	0.72×
search • *base*	370	30139	5627	*search*	160	4773	170	189	29.77×
parent • *size* • *base*	342	22691	21	*size*	32	1092	21	51	0.41×
				parent	100	442	11		
search • *size* • *base*	562	38856	62059	*size*	32	1092	21	1125	55.16×
				search	320	11852	1085		
search • *parent* • *base*	470	31975	4280	*parent*	100	442	12	200	21.40×
				search	160	4773	169		
search • *parent* • *size* • *base*	662	40642	76809	*size*	32	1092	21	1156	66.44×
				parent	100	442	11		
				search	320	11852	1105		
INS (scope=16)									
base	288	74939	132	n/a	n/a	n/a	n/a	n/a	n/a
attr-val • *base*	832	97576	281	$attr - val$	544	24468	665	811	0.35×
label • *attr-val* • *base*	1952	178139	16625	$attr - val$	544	24468	665	1144	14.53×
				label	1120	17475	347		
record • *label* • *attr-val* • *base*	1969	179596	11224	*attr-val*	544	24468	665	1174	9.56×
				label	1120	17475	347		
				record	17	25	30		

Table 1. Performance results for the subject product lines. All times are in milliseconds.

5.1 Intentional naming

The Intentional Naming System (INS) [1] is a resource discovery architecture for dynamic networks. INS is implemented in Java; the core naming architecture is about 2000 lines of code. In previous work [15], we modeled INS in Alloy and discovered significant bugs in its design and implementation. Here, we show how incremental test generation gives a significant speed-up over the conventional approach.

INS allows describing services using their properties. This enables client applications to state *what* service they want without having to specify *where* in the network topology it resides. Service properties in INS are described using *intentional names*, which are implemented using *name-specifiers*—hierarchical arrangements of alternating levels of *attributes* and *values*. Attributes classify objects. Each attribute has a value that further classifies the object. A *wildcard* may be used if any value is acceptable. An attribute together with its value form an *av-pair*; each av-pair has a set of child av-pairs. The av-pairs form a tree structure. Services advertise themselves to *name resolvers* that maintain a database to store mappings between name-specifiers and *name records*, which include information about the current service locations. To test the correctness of key INS algorithms, we must generate advertisements and queries as test inputs.

We differentiate each product in the intentional name product line based on whether there are attribute and value nodes, or whether attributes and values have labels satisfying the constraints for a name-specifier, or whether the trees have pointers from their leaf value-nodes to name-records. The following AHEAD model describes this family: `INS = {base, attr-val, label, record}`. Actual Alloy model for INS is not included due to space considerations.

5.2 Results

Table 1 presents the experimental results for the two subject product lines. Conventional approach is test generation with the latest Alloy tool-set, whereas incremental refers to our approach Kesit. For each product, we tabulate the number of primary variables, the number of CNF clauses and the total time for the conventional approach. We also tabulate the number of additional Boolean variables, the number of additional CNF clauses, the additional time taken to refine previously generated tests and the total time for our incremental approach. The last column shows the speed-up.

We generated 100 test inputs for each product and the tabulated times represent the average time to generate a single test for the product. We tabulate results for binary trees for 10 nodes and for intentional names for 16 nodes; these scopes are representative of the general characteristics we have observed during the experiments. As mentioned earlier, a product can be generated following different paths in the corresponding commuting diagrams: For each product, we show the results for which Kesit most significantly outperformed the traditional approach.

Experiments show that Kesit can provide a speed-up of over 66×. However, it does not always provide a speed-up and for some products, such as `size•base` and `parent•base`, we observe a slow down in compari-

son with the conventional approach. While we expect SAT problems with fewer primary variables to be easier to solve, we observe that applying our algorithm to refinements that involve simple constraints introduces an excessive overhead. Therefore, the conventional approach seems to be more efficient for simple refinements. However, for more complex constraints, such as `search`, our incremental approach performs significantly better. Parallel to that, as the scope increases, the performance improvement Kesit provides becomes more significant not only for complex constraints but also for the simple ones. The experiment results pertaining larger scopes are not presented in this paper due to space considerations.

We obtain the highest speed-up for the `search` refinement in the Binary Tree subject. With the conventional approach, going beyond the scope of 12 seems infeasible. Our incremental approach enables SAT solvers to handle significantly larger scopes because the resulting SAT problems are much simpler. For example, generating test cases for binary tree with the `search` constraints involves 30139 clauses in the conventional approach, but Kesit works with only 19618 and 4773 clauses for the `base` and `search` features respectively. We observe this effect with the INS model too. The number of primary variables and clauses are greater (i.e., 1128 and 80645 respectively) for the conventional approach due to the complexity and size of the complete model. However, incremental generation reduces the problem to two smaller refinements, `attr-val` and `label`, which involve smaller numbers of variables and clauses.

To summarize, a key strength of Kesit is to solve more complex problems and reach larger scopes. There are two key findings that we have observed during our experiments: (1) for simple refinements, Kesit's performance is comparable to the conventional approach, and (2) for complex refinements, Kesit significantly outperforms the conventional approach.

	path	time	#filtered
conv	ρ_1: τ•search•balance•base	4.87	0
	ρ_2: τ•balance•search•base		
incr (basic)	ρ_3: search•balance•τ•base	1.34	315
	ρ_4: balance•search•τ•base	43.04	315
incr (mixed)	ρ_5: search•τ•balance•base	0.178	0
	ρ_6: balance•τ•search•base	18.39	860

results are averaged over 50 inputs

Table 2. Comparison of different paths. The path ρ_5 is optimal.

6 Future Work

6.1 Further Optimizations

FOMDD suggests that the conventional and incremental approaches are only two of many other approaches for generating tests, and that a combination of conventional and incremental may in fact be more efficient.

Figure 6 illustrates a three-dimensional commuting diagram for *balanced* binary search trees as described earlier. Our incremental approach is represented by a pair of paths in this cube, starting from base specification s_0 to test t_3 that first descends and then walks the bottom of the cube:

search•balance•τ•base

balance•search•τ•base

The conventional approach, in contrast, follows a different set of paths that walks the top of the cube before descending:

τ•search•balance•base

τ•balance•search•base

Clearly, there are other paths, and among them is a more efficient generation strategy. The path with bold arrows reflects this alternative strategy, where we first solve for `base` and `balance` constraints *together* and then incrementally solve for `search` constraints:

search•τ•balance•base

The reason why this alternative path is more efficient is that many solutions of the base program are discarded when additional constraints (e.g., search or balance) are added. It is actually cheaper to start with a slightly more complex specification, generate and extend its solutions, than starting from the base.

We discovered this optimal path by examining all paths (see Table 2) [23]. Note that the number of paths that are filtered (meaning that the number of solutions that are subsequently discarded as they do not extend to solutions of more complex programs) is an important indicator of a path's performance. We

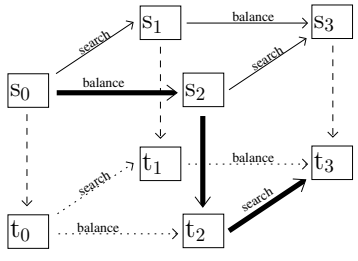

Figure 6. Specification-tests commuting for BST. The path with bold arrows is ρ_5: search•τ•balance•base.

recently developed a constraint prioritization approach that can assist in identifying an optimal path for test generation; details are described elsewhere [25].

6.2 Feature Interactions

While features often represent additional program functionality as we assumed in incremental test generation, one of the key issues in feature-based development is accounting for feature interactions [17], where features may *replace* existing functionality. Although it has been the subject of a large body of research [5], much about feature interactions is still not well understood.

FOMDD allows features to have a more profound impact on properties than can be expressed by conjunction. An interaction occurs when a feature replaces (not just extends) existing constraints. In general, features can *transform* a property of a program (such as replacing an existing constraint with another, thus disrupting the monotonic increase that we assumed earlier), in which case our incremental approach may not apply directly. Certainly, an incremental approach could apply to feature compositions from the point of the last non-monotonic (i.e., last property-replacing) feature. And it might apply if replaced properties are simply removed from earlier features in order to emulate monotonic compositions. In any case, this will be an interesting subject for future research.

7 Related Work

Nebut et al. [7] states that software product line processes still lack support for testing end-products using methods and techniques that are based on specific features of a product line, i.e., commonality and variability. While classical testing approaches can be applied in the product line domain, the very nature of feature composition and the large number of possible product configurations introduce a serious challenge for scalability. Much of the literature in testing software product lines focuses on planning and assessment of software testing [10, 18]. Our approach introduces an opportunity for tailoring the practices from the classical testing domain with respect to the specific requirements of software product lines.

Approaches for regression testing [19] bear similarities to our work. A key problem addressed by these approaches is of *test selection*: a code-based selection technique attempts to identify a subset of existing tests that are likely to reveal faults in the modified program. Our incremental generation contrasts with test selection since we do not select tests from a given suite but instead we refine given tests using constraint solving.

Barrett et al. [2] present an incremental approach for translating first-order logic formulas into SAT problems. Instead of translating the entire formula up front, they translate it incrementally as the search is conducted by the SAT solver. This approach deals with the SAT solver semantics and interacts directly with the solver. In contrast, our approach works at a higher level and manipulates Alloy formulas. The two approaches are thus complementary and can be used in conjunction to further optimize test generation.

Recently Cohen et al. [9] investigated the use of incremental satisfiability solvers for generating interaction test suites. Their algorithm uses the incremental solver MiniSAT [11] to optimize the AETG [8] test generation algorithm. Incremental SAT solvers have a direct application for incremental test generation and we plan to explore their use in testing product lines.

In previous work, we developed the TestEra [15] framework for specification-based testing of Java programs. TestEra generates inputs and checks program correctness using Alloy specifications. TestEra performs bounded exhaustive testing. TestEra's generation enabled achieving high code coverage for unit testing of library code. TestEra discovered subtle bugs in stand-alone applications including a fault-tree analyzer [21]. This paper shows how we build on TestEra to provide a significantly more efficient test generator, Kesit, for testing software product lines using an incremental approach. Kesit not only generates tests more efficiently than TestEra, but also scales to generation of larger inputs, which enables novel strategies for software testing. To illustrate, bounded exhaustive testing can be complemented by testing on selected larger inputs that are generated using the same constraints.

While this paper focuses on the use of Alloy for test generation, Alloy has various other applications, including modeling and checking of designs of software artifacts [14], and static analysis of code [26]. We believe our approach will enable new efficient analyses to support checking of not only the implementations but also the designs of product lines.

8 Conclusions

Testing software product lines is an important and difficult problem. We presented a novel technique that incrementally generates tests for product lines, in general, and FOMDD models of product lines in particular. Our key insight to test generation comes from the definition of a feature: an *increment* in program functionality. We introduced an automatic technique for mapping a formula that specifies a feature into a transformation that defines incremental refinement of

test suites. Our approach performs test generation *incrementally*. The experimental results with our prototype Kesit shows that incremental test generation provides significant performance improvements over conventional means of test input generation.

We believe incremental approaches hold much promise, not just in the context of product lines but also in the more general software testing context, e.g., for refining tests for regression testing.

Acknowledgments

This work was funded in part by NSF's Science of Design Awards #CCF-0702680, #CCF-0724979 and #IIS-0438967.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. of the 17th Int'l Conference on Verification, Model Checking and Abstract Interpretation*, Kiawah Island, SC, December 1999.

[2] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to sat. In *Proc. of the 14th Int'l Conference on Computer Aided Verification*, July 2002.

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

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

[5] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[6] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proc. of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 1999.

[7] Y. L. T. Clémentine Nebut and J.-M. Jézéquel. System testing of product lines: From requirements to test cases. In *Software Product Lines - Research Issues in Engineering and Management*, pages 447–478. Springer, 2006.

[8] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi. Exploiting constraint solving history to construct interaction test suites. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 121–132, 10-14 Sept. 2007.

[10] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *Proc. of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, NY, July 2006.

[11] N. Een and N. Sorensson. An extensible sat-solver. In *Proc. of the 6th Int'l Conference on Theory and Applications of Satisfiability Testing*, Italy, May 2003.

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

[13] D. Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2), April 2002.

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

[15] S. Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, December 2003.

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

[17] J. Liu, D. Batory, and S. Nedunuri. Modeling interactions in feature oriented software designs. In *Proc. of the the 8th Int'l Conference on Feature Interactions in Telecommunications and Software Systems*, 2005.

[18] H. Muccini and A. van der Hoek. Towards testing product line architectures. In *Proc. of the Int'l Workshop on Test and Analysis of Component-Based Systems*, Warsaw, Poland, April 2003.

[19] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[20] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, September 2005.

[21] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *Proc. of the Int'l Symposium on Software Testing and Analysis*, 2004.

[22] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Proc. of the 13th Int'l Conference on Tools and Algorithms for Construction and Analysis of Systems*, Braga, Portugal, March 2007.

[23] S. Trujillo, D. Batory, and O. Diaz. Feature oriented model driven development: A case study for portlets. In *Proc. of the 29th Int'l Conference on Software Engineering*, Minneapolis, MN, May 2007.

[24] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. A specification-based approach to testing software product lines. In *Proc. of the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Croatia, September 2007. Poster Paper.

[25] E. Uzuncaova and S. Khurshid. Constraint prioritization for efficient analysis of declarative models. In *Proc. of the 15th Int'l Symposium on Formal Methods*, Turku, Finland, May 2008.

[26] M. Vaziri. *Finding Bugs Using a Constraint Solver*. PhD thesis, Computer Science and Artificial Intelligence Lab, MIT, 2003.