Incremental Test Generation for Software Product Lines

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

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

Abstract—Recent advances in mechanical techniques for systematic testing have increased our ability to automatically find subtle bugs, and hence to deploy more dependable software. This paper builds on one such systematic technique, *scope-bounded testing*, to develop a novel specification-based approach for efficiently generating tests for products in a software product line. Given properties of features as first-order logic formulas in Alloy, 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 an incremental approach can provide an order of magnitude speed-up over conventional techniques. We also present a further optimization using dedicated integer constraint solvers for feature properties that introduce integer constraints, and show how to use a combination of solvers in tandem for solving Alloy formulas.

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.

Software testing, the most commonly used methodology for validating the quality of software, plays a vital role in our ability to deploy more dependable software by enabling us to find bugs before they manifest as failures. *Specification-based testing* [13], [27], [28] is a powerful technique that enables systematic testing of code using rich behavioral specifications. The importance of using specifications in testing was realized over three decades ago [27], 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 [33], [50].

For programs written in object-oriented languages, a suitable specification language is Alloy [30]—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 [23]—a fully automatic tool based on propositional satisfiability solvers—enables both test generation and correctness checking [33]. 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 representing the expected relation between inputs and outputs. The Alloy tool-set has been used to check designs of various applications such as Intentional Naming System for resource discovery in dynamic networks [33], static program analysis for checking structural properties of code [53], and formal analysis of security APIs [39].

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 [15].

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

An earlier version of this paper appeared at the 19th IEEE International Symposium on Software Reliability Engineering (ISSRE 2008).
The first author now works at Microsoft, Seattle WA.

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, 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, whereas the *conventional* use of the analyzer solves a complete specification of a program to generate tests. 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 defines incremental refinement of test suites. We present experimental results on a set of data structure product lines showing that incremental test generation can provide an order of magnitude speed-up over the conventional use.

To illustrate, consider composing a feature f with a base product b, which have specification formulas s_f and s_b respectively. Assume we want to generate a test input for the resulting product. Then the input specification is $\phi = s_f \wedge s_b$—any solution to this formula represents a test input. Instead of solving the entire formula ϕ at once (as is done conventionally), we first run the analyzer to solve s_b to generate an instance i_b, which is an assignment of sets of tuples to relations in s_b. Next, we run the analyzer on s_f while using i_b as a *lower bound* for the new instance, i.e., a new instance must contain tuples in i_b and may contain additional tuples, for example, for relations in s_f that are not in s_b. Note that even though we execute the analyzer twice, each execution is on a formula simpler than ϕ. Moreover, the second execution explores a much smaller state space since i_b, the lower bound, already prunes a significant part of the space.

Our incremental approach enables a novel re-use of tests: tests that are generated for one product are directly used to generate tests for another product. Considering the large number of possible products in a product line, such re-use is of great value and enables highly optimized test generation.

We developed a prototype, Kesit, that implements our approach based on the AHEAD theory [10] and uses the recently developed Kodkod [54] model finding engine for Alloy. We have used Kesit to generate tests for a variety of data structure product lines and evaluated the performance of incremental test generation. Experimental results show that Kesit can provide an order of magnitude speed-up over the conventional approach. We believe approaches like Kesit, which increase the feasibility of systematic testing will likely improve our ability to deploy more dependable software.

This paper builds on our previous work on Kesit [57] and makes the following 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;
- **Integer constraint solving for Alloy.** We show how a decision procedure for integer constraints can be used in conjunction with SAT for solving Alloy formulas;
- **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 show significant speed-ups over conventional techniques.

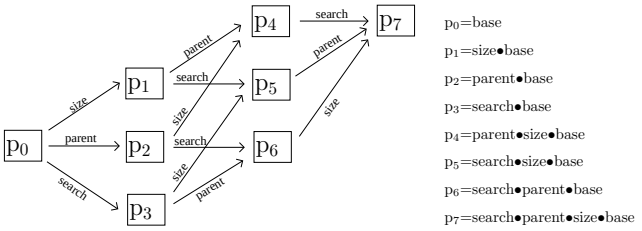

Fig. 1. Family of binary trees. Nodes represent products. Arrows represent feature inclusion.

2 EXAMPLE

This section illustrates a simple product line of data structures. We use AHEAD [10] and Alloy [30] 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 [19]. 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 [19], 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, 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)
      @   lone n.~(left + right)
      @   no n.left & n.right }
      @*/
    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 [40].

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 [7]. 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 sub-tree larger.

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 $\mathtt{p}_5$ = `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 $\mathtt{p}_5$'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 $\mathtt{p}_5$'s invariant, the analyzer takes 62 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 $\mathtt{s}_i$ represent specifications for test generation for the corresponding products, e.g., $\mathtt{s}_0$ represents the base specification—the acyclicity constraint. The nodes $\mathtt{t}_i$ represent the corresponding sets of test inputs. The horizontal arrow Δ_s represents a refinement of

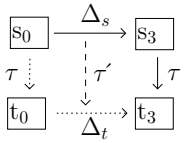

Fig. 2. BST commuting diagram.

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 even larger. 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.

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 [8], AHEAD [9], and FOMDD [5], [55].

3.1 GenVoca

GenVoca is a metaprogramming model of product-lines: 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 (zero-ary functions) 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. The expression a•b represents the composition of features a and b.

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 [6], [49].[1]

1. 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 [4].

3.2 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:

$$\begin{aligned} p_1 &= j \bullet f && \text{// GenVoca expression} \\ &= [\Delta c_j, \Delta s_j, \Delta b_j] \bullet [c_f, s_f, b_f] \\ &= [\Delta c_j \bullet c_f,\ \Delta s_j \bullet s_f,\ \Delta b_j \bullet b_f] \end{aligned}$$

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 ($\Delta s_j \bullet s_f$), and the bytecode of p_1's base with its refinement ($\Delta b_j \bullet b_f$).

3.3 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 [45]. Figure 3 shows the commuting diagram for program $p2 = i \bullet j \bullet h = [c_2, s_2, b_2]$.

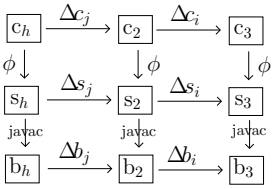

Fig. 3. Commuting diagram.

A fundamental property of a commuting diagram is that all paths between two objects represent equivalent results, i.e., products. For example, one way to derive the bytecode b_2 of program p_2 (lower right in Figure 3) 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:

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

In general, 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 each arrow of a commuting diagram has a cost. The shortest path between two objects in a commuting diagram is a *geodesic*. A geodesic represents the most efficient metaprogram that produces the target object [5].

4 Our Approach

This section describes our specification-based approach for test generation for systematic testing of implementations synthesized from an SPL. We developed a FOMDD model of our approach; specifications and tests are objects in the model, and transformations among tests and specifications are arrows (Section 4.1). We developed two key transformations that automate test generation using the Alloy Analyzer; we concretize the instances generated by the analyzer into Java object graphs that form test suites (Section 4.2).

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 (Δs_f) and its test suite (Δt_f).

In specification-based testing, the user provides a specification s and its refinement Δs, i.e., additional properties. To generate tests, we need a transformation τ that maps a specification s to its corresponding tests t. Also implementing test refinement Δt, i.e., a mapping from old tests to new tests, 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 Δt: τ' maps a test suite t and a specification refinement Δs to a corresponding test refinement Δt. Figure 2 shows the commuting diagram that corresponds to program p_0 = [s_0, 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 instance i is a function from a set of relations R to a power set of tuples 2^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 [54] 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 Δs for specification `s = [f, b]` may refine the formula f or the bound b or both, i.e., and Δ`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 $\wedge$ Δf, where Δf 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 Δt 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 Δt.

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 $\tau \bullet \Delta s = \Delta t \bullet \tau$. 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 7), it may not be practical to traverse some Δt 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

```
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´;
}
```

Fig. 4. Test refinement algorithm. The algorithm takes as input a specification refinement and a test suite, and outputs a new test suite subject to the given refinement.

any path (in a commuting diagram) from a specification `s` to a test suite `t` to contain horizontal arrows that are labeled Δ`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.

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 8). The impact of feature interactions on incremental test generation is discussed in Section 7.

The algorithm takes as input a test suite `t` and a specification refinement Δ`s`, and computes a new test suite, which refines the tests in t with respect to the constraints in Δ`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 Δ`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 7).

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

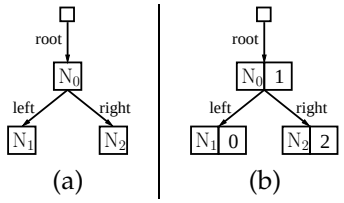

Fig. 5. Test inputs. (a) An acyclic binary tree. (b) An acyclic binary search tree with elements 0, 1, and 2.

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 (Δ`s`) 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 are 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). The small unlabeled square represents the `BinaryTree` atom `BT0`; nodes $\mathtt{N}_0$, $\mathtt{N}_1$, $\mathtt{N}_2$ are `Node` atoms. Edges represent valuations of binary relations:

```
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 (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.

4.2.2 Concretization

To use an instance as a test input to a Java program, we need to *concretize* the instance, i.e., translate it into a Java object graph. The translations are automatic when the only primary variables in Alloy formulas are relations that correspond to declared object fields. The TestEra tool [33] (Section 8) implements these translations. A user may choose to define formulas using an abstraction over the concrete object fields; then the user defines a specialized translation from an abstract instance to a concrete object graph [42], [52].

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).

The basis of our evaluation is a performance comparison for test generation between the traditional approach and our incremental approach. Specifically, we measure and compare the time taken by these two approaches for generating test inputs. Section 7.1 discusses how our approach enables more effective testing.

All experiments were performed on a 1.8GHz Pentium M processor using 512MB of RAM. All SAT formulas were solved using MiniSat [25]. Our tool Kesit uses the Java API of the Kodkod back-end [54] of the Alloy Analyzer.

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 [33], 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.

We present the Alloy models that represent test inputs. Note that the models do not represent the data structures at the concrete representation level because INS's Java implementation uses container classes that are not directly supported in Alloy. We model the data structures at an abstract level using Alloy's sets and relations. Doing so necessitates writing specialized translations for concretizing Alloy instances into Java objects; we developed these translations in previous work [42].

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}`.

The `base` product for INS is a rooted tree of *nodes*:

```
sig LabelTree {
   Node root;
   Set<Node> nodes;
   children: nodes one -> (nodes - root)
} {
   nodes = root.*children
   some root.children
   no root.~children
}
sig Node {}
```

The Alloy keyword `sig` declares a basic set. `LabelTree` is a set of atoms that model trees. The field `root` introduces a relation of type `LabelTree x Node`; this relation is a total function. The field `nodes` introduces a relation of type `LabelTree x Node`; the keyword `set` declares `nodes` to be an arbitrary relation. The field `children`

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

TABLE 1
Performance results for the subject product lines. Times are in milliseconds.

introduces a ternary relation of type `LabelTree x Node x Node`. For a `LabelTree l`, `l.children` represents the edge-set of the tree. The keyword `one` ensures each node except `root` has exactly one parent.

Next, we add the `attr-val` feature to `base`:

```
refines sig LabelTree {}
{
    all n,m: nodes {
        disj[n,m] => n.attr!=m.attr
        disj[n,m] => n.val!=m.val
    }
}
refines sig Node {
    attr: Attribute,
    val : Value
}
sig Attribute {}
sig Value {}
```

We use the Jakarta keyword `refines` to denote refinement of Alloy specifications. Note that Alloy does not support refinement, however, we show how Alloy models can be built using refinement. Each node in the tree now represents an `av-pair` and has an attribute and a value. This refinement transforms the simple rooted tree in to an *AVTree*.

Next, we add the `label` feature to `attr-val•base`. In INS, attributes and values are defined as free-form strings that are defined by applications for classifying objects. For example, to classify the services provided by a certain provider, 'service' can be used as the class (`attribute`) and 'printer' and 'camera' as the the classifications (`values`) under the 'service' class. We use `label` to allow re-use of attributes and values in a tree to represent a labeled AVTree, i.e., a query:

```
refines sig LabelTree {}
{
    root.attr.label = Null
    root.val.label  = Null

    Null !in
        ((nodes-root).val.label +
         (nodes-root).attr.label)

    no Wildcard.~label.~attr

    no Wildcard.~label.~val.children

    no (nodes-root).attr.label &
       (nodes-root).val.label

    all n: nodes { all i, j: n.children {
        disj[i,j] => i.attr.label != j.attr.label}}
}
refines sig Attribute { label: Label }
refines sig Value { label: Label }
sig Label {}
one sig Null, Wildcard extends Label {}
```

Next, we add the `record` feature to `label•attr-val•base` to represent an advertisement. Each name-specifier has a pointer from each of its leaf value-nodes to a name-record:

```
refines sig LabelTree {
    name_record: Record
} {
    all n: nodes |
        no n.children <=> n.val in nameRecord.values
}
sig Record { values: set Value }
```

5.2 Results

Table 1 presents the experimental results for the two subject product lines. The conventional approach is test generation with the latest Alloy tool-set, whereas incremental refers our Kesit approach. 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

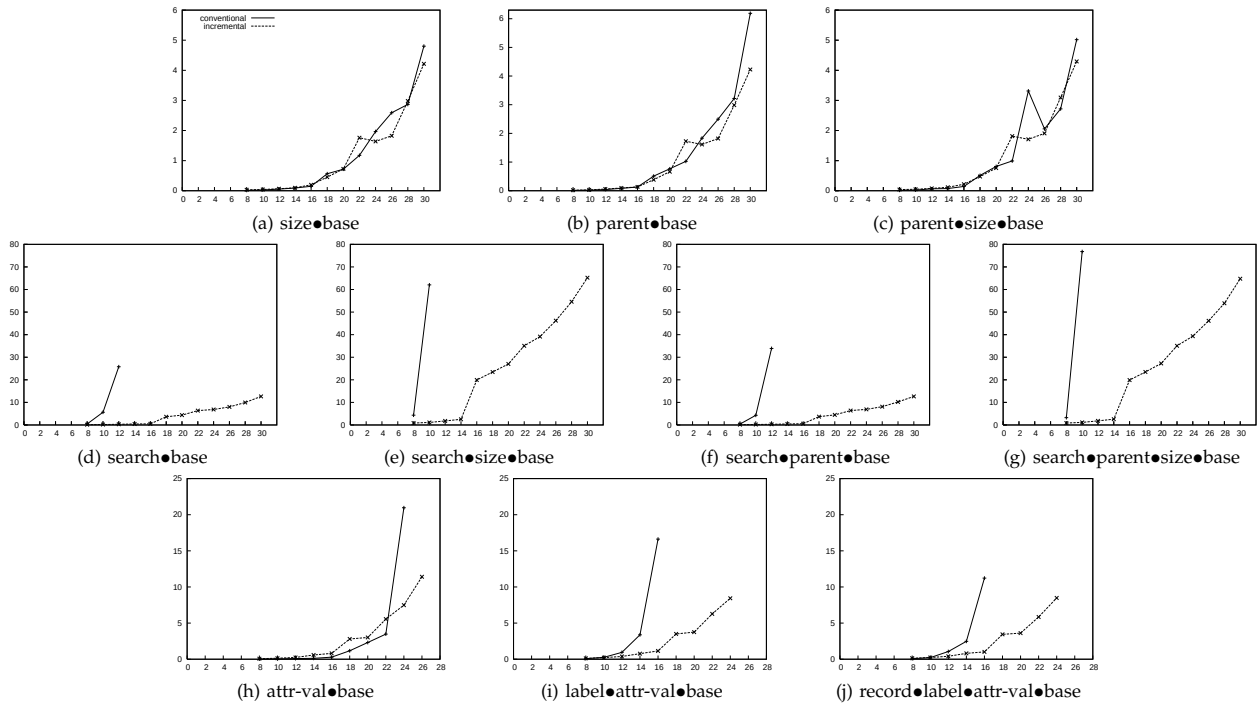

Fig. 6. Performance charts for the subject product lines: Binary Tree (a-g) and INS (h-j). In each graph, the x-axis shows the scope and the y-axis shows the time measurements (in seconds). Also, in each graph, solid line plots the results using the conventional approach and dashed line plots the results using the incremental approach.

for binary trees with 10 nodes and intentional names with 16 nodes; these scopes are representative of the general characteristics we have observed during the experiments. Figure 6 graphically illustrates the results for various other sizes. 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, we observe a slow down in comparison with the conventional approach such as `size•base` and `parent•base`. 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 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. The experiment results for INS spanning a range of scopes is shown in Figure 6.

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. Moreover, since Kesit allows solving the complete problem using sub-problems that have significantly fewer variables and clauses, Kesit provides an approach that promises better scalability and allows more effective testing strategies (Section 7.1).

6 Optimization

Alloy has a basic support for integers. Integer expressions have primitive integer values and arithmetic operators allow addition, subtraction and comparison. Similar to non-integer relations, there is a scope defined on the

scope	conventional	incremental		
		base	search	
			AA	Z3
8	2.164	0.023	0.056	0.030
16	$>$ 1 hr	1.926	0.160	0.069
24	$>$ 1 hr	9.785	1.712	0.115
32	$>$ 1 hr	52.216	2.753	0.282
36	$>$ 1 hr	140.640	45.099	0.292

TABLE 2
Comparison of the Alloy Analyzer(AA) with Z3 for p_3=search•base. Times are in seconds.

integer values as well. A bound of k for integer atoms limits integer values to be between -2^{k-1} and 2^{k-1}-1. For example, a scope of 4 on integer values generates a range of integer atoms from -8 to 7.

As previously discussed, we observed the most significant performance improvements during our experiments for the search feature of the binary tree product line (Figure 6 (d-g)). This is mainly because of the additional integer atoms introduced by this feature and the relative impact of this feature on the size of the boolean formula (in terms of the number of variables and clauses). Our incremental approach benefits from working on smaller and ideally simpler problems. However, as the scope increases, larger instances are generated, and both conventional and incremental approaches face a scalability problem. Column 4 in Table 2 shows the growth of the analysis time for the search feature for incremental approach.

This section presents an optimization to our approach to provide more efficient analysis for integer constraints. Instead of the Alloy Analyzer, we use a specialized integer constraint solver. We illustrate our approach using the search feature from the binary tree product line.

6.1 Z3: SMT Solver

An alternative approach to solve integer constraints is to use a dedicated integer constraint solver such as Z3 [21]. Instead of relying on the analyzer's encoding and the underlying SAT solver, we implemented a translator from Alloy to Z3. The translator takes an Alloy formula as input and uses the Z3 ANSI C API to generate the input formulas for Z3. The result of Z3's analysis is translated back to Alloy.

Z3 is an efficient *satisfiability modulo theories* (SMT) solver. SMT is a generalized form of boolean satisfiability, where input formulas are evaluated with respect to combinations of theories such as arithmetic, bit-vectors, arrays, and uninterpreted functions. While such decision problems can also be solved by general SAT solver, the main advantage of SMT solvers is the tight integration between satisfiability analysis with theory-specific solvers.

Our overall approach has two keys steps:

1) Use the previously generated (partial) instance to partially evaluate the additional constraints; and
2) Translate the resulting constraints to the input language of Z3.

Illustration. Consider the binary tree instance shown in Figure 5 (a). The translator performs a partial evaluation of the following search constraint with respect to the tree instance:

```
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
}
```

The translator traverses the abstract syntax tree of this nested quantified formula and generates the following input formula for Z3:

```
N0 > N1
N2 > N0
-8 <= N0 <= 7
-8 <= N1 <= 7
-8 <= N2 <= 7
```

Each of {N0, N1, N2} represents an integer variable bounded by the scope defined on the integers, in this case 4.

6.2 Evaluation

Table 2 tabulates the results pertaining to the comparison between the Alloy Analyzer and Z3 for product p_3=search•base. The table also includes the results for the conventional use of the analyzer. The tabulated times are in seconds and averaged over 50 instances. For the incremental approach, base product is generated using the Alloy Analyzer and the search feature is solved using the Analyzer and Z3. Columns 2 and 3 shows the impact of increasing scope on the analyzer's performance. Whereas, the impact on Z3's performance is minimal thus Z3 scales better compared to both conventional and the incremental approach using the Alloy Analyzer only.

7 Discussion and Future Work

7.1 How our approach helps test better

The conventional use of Alloy for specification-based (black-box) testing enables *scope-bounded testing*—where a program is tested against all small inputs (e.g., all binary search trees with up to 5 nodes)—which has effectively been used for achieving high code coverage in library code and for finding deep semantic bugs in applications [34], [52]. Our work shows how to scale test generation to larger inputs. For these inputs, however, scope-bounded testing may be impractical, since there may be too many inputs to test against even if they can be generated more efficiently. The ability to generate larger inputs, nonetheless, enables novel testing techniques that are likely to be more effective at finding bugs, e.g., (1) it enables a new strategy for black-box testing—test exhaustively on all small inputs and selectively on larger inputs—that is likely more effective than scope-bounded testing, which only uses small inputs; (2) it enables efficient specification-based, *white-box*

techniques, e.g., where constraints from pre-conditions are conjoined with path conditions (built using symbolic execution [26], [35], [36]) to direct generation to specific code paths that require larger inputs [51].

7.2 Testing larger product lines

We demonstrated how our approach provides efficient test generation for two small product lines using specifications. To apply a specification-based approach to test real product lines, possibly with hundreds of features, requires addressing two key challenges: (1) the availability of specifications; and (2) the scalability of test generation.

Similar to many other real applications, software product lines are rarely accompanied by detailed specifications. However, recent advances in specification languages, specifically their tighter integration with programming languages in the form of annotations [38], hold promise for a wider use of specifications in product lines. In a new project on infrastructure development, we are initiating an effort to write specifications for a suite of product-lines from AHEAD, which we expect will provide a platform for more rigorous evaluation and comparison of techniques for testing product-lines as well as provide insights into how specifications for product lines may be structured to reduce the burden of writing specifications.

Moreover, the recent advances in constraint solving technology are likely to make systematic testing of product lines viable. Witness, for example, the recent resurgence of symbolic execution due to advances in our ability to solve constraints from path conditions. Our approach to incremental generation provides a key step at more efficiently utilizing off-the-shelf solvers for product line testing, in particular, and constraint solving for checking of code, in general.

7.3 Solving Alloy formulas using multiple solvers

Our approach to incremental test generation naturally lends itself to an application of a combination of specialized decision procedures. Section 6 showed how to use an off-the-shelf SMT solver in conjunction with SAT to more efficiently solve binary search tree constraints. Given a pure integer constraint, it is natural to expect that a dedicated integer constraint solver will perform better than a SAT solver on simple integer constraints, such as the "less-than" constraints that represent binary search. A key contribution our approach makes is to demonstrate how to use a combination of solvers even when integer constraints are interleaved with non-integer constraints [56], e.g., consider the following constraint for a sorted, doubly-linked, circular list, say l:

```
all n: l.header.*next |
  n.next != l.header => n.key < n.next.key
```

Note the integer part of the constraint applies only subject to a certain constraint on non-integers. Hence we cannot directly separate the constraints into a conjunction of integer constraints and non-integer constraints. Our insight is to use partial evaluation to boil down such mixed constraints into integer constraints by unrolling the quantifiers and using concretized values for non-integer variables.

Since SMT solvers already support a variety of decision procedures using sophisticated partitioning strategies, an alternative technique [32] for utilizing multiple solvers for Alloy is to solve Alloy formulas by translating Alloy to the input language of an SMT solver, such as Z3, e.g., by modeling relations in Alloy as arrays in SMT. An advantage of this approach is that it immediately enables the use of multiple solvers. A disadvantage is that since it involves an encoding from one data model (relational) to another (array-based), it may create a (partial) loss of semantic structure that can be exploited at the level of the problem domain but not at the level of the solving domain. We plan to further investigate SMT for Alloy.

Our incremental generation approach follows the *partial assignments and backtracking* approaches in SAT and SMT solvers. However, a key advantage is that our approach applies at the level of Alloy formulas, i.e., the level of the problem domain, and can utilize the high level problem structure. The advantage of having (some) high level structure during analysis is demonstrated by the SERA framework [60], which encodes Alloy to *sequential circuits* [24], which are imperative and stateful, and therefore permit succinct encodings, e.g., by using loops to represent quantifiers, and enable the use of techniques for transformation-based verification (a popular technique for hardware verification) [37]. Experimental results show that sequential circuits enable significantly faster solving than CNF formulas, primarily because the re-writing techniques for TBV, such as compositional minimization and abstraction, can utilize the problem structure.

We believe it is possible to communicate some high level structure to SAT solvers in an explicit fashion by modifying the interface and implementation of a solver, e.g., to prioritize the partial assignments using heuristics. We plan to investigate this further.

In ongoing work, we are generalizing our incremental approach for Alloy to show how Alloy formulas can be solved efficiently using a suite of different solvers, including string solvers, e.g., the Java String Analyzer [14], and set constraint solvers, in addition to SAT and SMT.

7.4 Memory usage and Alloy/SAT

There are two basic metrics for evaluating systematic analyzers, such as the Alloy Analyzer: time taken and (peak) memory used. Our evaluation in this paper focuses on time taken, and shows how the incremental approach, Kesit, is more efficient. Since Kesit solves formulas with fewer variables and clauses, the number of *conflict clauses* maintained by SAT during its search is expected to be smaller, and hence the amount of memory required is also expected to be smaller than for the conventional approach.

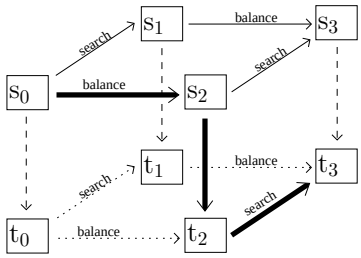

Fig. 7. Specification-tests commuting for BST. The path with bold arrows is ρ_5: search•τ•balance•base.

	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 3
Comparison of different paths. Path ρ_5 is optimal. Times are in seconds.

Monitoring the memory usage confirms that Kesit outperforms the conventional approach. Specifically, for the binary search tree product with base and search features, Kesit has a peak memory usage of 4.5MB whereas the conventional approach uses 25MB (to generate 50 inputs with 8 nodes). Similarly, for the Intentional Naming System product with base, attr-val, and label features, Kesit has a peak memory usage of 7.9MB whereas the conventional approach uses 36MB (to generate 50 names with 10 nodes).

While it is important to prevent memory usage from becoming too high, SAT solvers can avoid running out of memory by discarding new conflict clauses when the memory usage approaches a pre-defined upper bound. However, the presence of conflicts allow the solvers to do more efficient pruning. Thus, Kesit allows SAT to more effectively utilize the pruning algorithms than the conventional approach, and to scale better.

7.5 Evaluating Alternative Paths

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 7 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 3) [55]. Note that the number of instances 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 recently developed a constraint prioritization approach that can assist in identifying a geodesic (i.e., an optimal path) for test generation; details are described elsewhere [58].

7.6 Feature Interactions and other characteristics

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 [41], where features may *replace* existing functionality. Although it has been the subject of a large body of research [12], 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.

In addition to feature interactions, product lines in general may have other characteristics that limit a direct application of our incremental approach. For example, if the products in a product line have minimal sharing, incremental generation may not provide significant speedups. Similarly, for more general composition operators, new test refinement operators need to be defined. We plan to build on our work to develop new techniques that allow handling a wider class of product lines.

8 Related Work

This paper presents an incremental test generation approach for specification-based testing of software product lines developed using the FOMDD (AHEAD) methodology [9]. This section discusses related work on modeling and testing of product lines, incremental testing techniques in more broader context, and specification-based testing techniques that use the Alloy tool-set.

8.1 Modeling and testing of product lines

FOMDD is one of several methodologies for product line development [2], [20], [29], [46]. While our incremental approach uses FOMDD's AHEAD as an enabling technology, we believe our core ideas can serve as a basis for new analysis techniques for other methodologies too—for testing end products by defining test refinement for different composition operators, and for analyzing *feature models* (which define constraints among features) using incremental algorithms.

Testing software product lines is a relatively young area of research. Nebut et al. [15] 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 [11], [18], [43]. Our approach introduces an opportunity for tailoring the practices from the classical testing domain with respect to the specific requirements of software product lines.

Kahsai et al. [31] recently developed a specification-based approach that uses the CSP-CASL [47] specification language as a basis of generating event sequences as test inputs. The approach is illustrated using a remote control product line. Our incremental generation technique has a different focus: it addresses product line implementations that manipulate dynamic data structures that require structural constraint solving.

Denger et al. [22] conducted an empirical study to compare code inspections and functional testing in the context of product lines using defect finding potential. Their study provides a basis for comparing new analytical techniques for testing software product lines using a metric based on the ability to find faults.

8.2 Incremental testing techniques

Pap et al. [44] introduce a bounded incremental algorithm to automatically re-generate tests cases for deterministic finite state machine models. They assume a changing specification for an existing model and utilize an existing test case of the previous version to generate a complete test with the same fault detection capability. Their approach, although applied in a different domain, carries the same fundamental principles as far as using the changes in the specifications to refine the generated test cases. The key difference, however, is that we incrementally solve constraints that are richer than those represented by state machines.

Approaches for regression testing [48] 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. [3] 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. [17] investigated the use of incremental satisfiability solvers for generating interaction test suites. Their algorithm uses the incremental solver MiniSAT [25] to optimize the AETG [16] 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.

8.3 Alloy for specification-based testing

The TestEra framework [33], [42] introduced the use of Alloy for scope-bounded testing. TestEra generates inputs using preconditions and checks program correctness using postconditions written in Alloy. Scope-bounded testing has been used to generate high quality test suites, which provided high code coverage for library code and found subtle bugs in stand-alone applications, including a fault-tree analyzer [52].

The use of Alloy for test generation requires a two step process: (1) generation of Alloy *instances* that represent desired inputs at an abstract level; and (2) translation of abstract instances into concrete tests, say as Java object graphs. Initial work on using Alloy for test generation modeled concrete inputs *up to isomorphism*, i.e., an abstract input represented essentially all elements of a concrete input. The fault-tree analyzer study [52] used a novel approach of generating an input fault tree by first generating an abstract structure that represented only a part of the input using the Alloy Analyzer, and then translating the abstract structure into *several* concrete fault trees as inputs, where additional input elements were added through a combinatorial assignment.

Kesit, the incremental approach presented in this paper for testing product lines, shows how to leverage a

product-line setting for separation of constraints, and to use multiple invocations of the same solver or use different solvers, in synergy. Kesit provides significantly more efficient generation than TestEra. 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 (as discussed in Section 7). To our knowledge, Kesit is the first framework for specification-based testing of software product lines using Alloy.

Alloy's traditional use for specification-based testing has been in a *black-box* setting, where the internal structure of code under test is not used to guide test generation. A more recently developed framework Whispec [51] shows how to use Alloy in a *white-box* setting, similar to generalized symbolic execution [35] or concolic execution [50]. We expect Kesit's incremental generation to optimize Whispec's performance.

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 [30], and static analysis of code [59]. We believe our approach will enable new efficient analyses to support checking of not only the implementations but also the designs of product lines.

9 Conclusions

Testing software product lines is an important and difficult problem. We presented a novel technique that incrementally generates tests for product lines represented using the AHEAD methodology. 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 show that incremental test generation provides significant performance improvements over the conventional use of the Alloy tool-set for test generation.

This paper focused on the basic underpinnings of a specification-based approach for testing product lines. We hope our work will provide a catalyst for a wider use of specifications in product-line development and allow creation of new approaches that scale systematic testing to real product-lines. 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 material is based upon work partially supported by the NSF under Grant Nos. CCF-0438786, CCF-0724979, IIS-0438967, CCF-0702680, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

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 ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.

[2] T. Asikainen, T. Soininen, and T. Männistö. A Koala-based approach for modelling and deploying configurable software product families. In *5th International Workshop Software Product-Family Engineering (PFE)*, 2003.

[3] 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.

[4] D. Batory. Feature models, grammars, and propositional formulas. In *Proc. of the 9th Int'l Software Product Line Conference (SPLC)*, 2005.

[5] D. Batory. From implementation to theory in program synthesis. In *Proc. of the 34th Annual ACM Symposium on the Principles of Programming Languages*, 2007. Keynote.

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

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

[8] 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.

[9] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *Proc. of the 25th Int'l Conference on Software Engineering*, 2003.

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

[11] A. Bertolino and S. Gnesi. PLUTO: A test methodology for product families. In *5th International Workshop Software Product-Family Engineering (PFE)*, 2003.

[12] 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.

[13] 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.

[14] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *10th International Static Analysis Symposium (SAS)*, 2003.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *5th International Conference Generative Programming and Component Engineering (GPCE)*, 2006.

[21] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th Int'l Conference on Tools and Algorithms for Construction and Analysis of Systems*, Budapest, Hungary, April 2008.

[22] C. Denger and R. Kolb. Testing and inspecting reusable product line components: first empirical results. In *ACM/IEEE International Symposium on Empirical Software Engineering (ISESE)*, 2006.

[23] D.Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[24] S. A. Edwards. The challenges of hardware synthesis from C-like languages. In *Design Automation and Test in Europe*, 2005.

[25] 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.

[26] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

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

[28] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. of the Int'l Symposium on Software Testing and Analysis*, July 2002.

[29] S. A. Hendrickson and A. van der Hoek. Modeling product line architectures through change sets and relationships. In *29th international conference on Software Engineering (ICSE)*, 2007.

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

[31] T. Kahsai, M. Roggenbach, and B.-H. Schlingloff. Specication-based testing for software product lines. In *Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, 2008.

[32] S. A. Khalek and S. Roychowdhury. Modeling linked data structures using SMT solvers. Class Project Report for Khurshid's Graduate Course on Verification and Validation of Software (EE382C-3), 2008.

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

[34] S. Khurshid and D. Marinov. Checking Java implementation of a naming architecture using TestEra. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 55. Elsevier Science Publishers, 2001.

[35] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of the 9th Int'l Symposium on Theoretical Aspects of Computer Software*, Warsaw, Poland, April 2003.

[36] J. C. King. Symbolic execution and program testing. *Comm. of the ACM*, 19(7):385–394, 1976.

[37] A. Kuehlmann and J. Baumgartner. Transformation-based verification using generalized retiming. In *Computer-Aided Verification*, 2001.

[38] 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.

[39] A. H. Lin. Automated analysis of security APIs. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2005.

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

[41] 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.

[42] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. of the 16th IEEE Int'l Conference on Automated Software Engineering*, San Diego, CA, November 2001.

[43] 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.

[44] Z. Pap, M. Subramaniam, G. Kovács, and G. Á. Németh. A bounded incremental test generation algorithm for finite state machines. In *Proc. of the 7th Int'l Workshop on Formal Approaches to Testing of Software*, 2007.

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

[46] K. Pohl and A. Metzger. Variability management in software product line engineering. In *28th International Conference on Software Engineering (ICSE)*, 2006.

[47] M. Roggenbach. CSP-CASL—a new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 2006.

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

[49] 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 *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, 1979.

[50] 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.

[51] D. Shao, S. Khurshid, and D. Perry. Whispec: White-box testing of libraries using declarative specifications. In *ACM SIGPLAN Symposium on Library-Centric Software Design*, Montreal, Canada, October 2007.

[52] 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.

[53] M. Taghdiri. Inferring specifications to detect errors in code. In *Proc. of the 19th IEEE Int'l Conference on Automated Software Engineering*, Washington, DC, 2004.

[54] 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.

[55] 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.

[56] E. Uzuncaova. *Efficient Specification-based Testing Using Incremental Techniques*. PhD thesis, University of Texas at Austin, 2008.

[57] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. Testing software product lines using incremental test generation. In *Proc. of the 19th International Symposium on Software Reliability Engineering*, Redmond, WA, Nov 2008.

[58] 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.

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

[60] F. Zaraket, A. Aziz, and S. Khurshid. Sequential circuits for relational analysis. In *Proc. of the 29th Int'l Conference on Software Engineering*, Minneapolis, MN, May 2007.