Chronological Test Generation for Software Project SLoC

[1]P Ratna Kumar, [2]Raghu D, [3]Udayasree. D, [4]Amanullah Mohd.
[1]Dept. of CSE, Sir C R Reddy College of Engineering, Eluru, Andhra Pradesh, India
[2]Dr. Paul Raj Engineering College, Bhadrachalam, Khammam, India
[3]Gurunanak Engineering College, Ibrahimpatnam, Hyderabad, India
[4]WISE Engineering College, Tadepalligude, India

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 speedup 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.

Keywords
Software/program verification, testing and debugging, software engineering.

I. 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 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, 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.

For programs written in object-oriented languages, a suitable specification language is Alloy —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 —a fully automatic tool based on propositional satisfiability solvers— enables both test generation and correctness checking. 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. The 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, static program analysis for checking structural properties of code, and formal analysis of security APIs.

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 10 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.

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 twofold. 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, 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 speedup over the conventional use.

To illustrate, consider composing a feature f with a base product b, which have specification formulas sf and sb, respectively. Assume that we want to generate a test input for the resulting product. Then the input specification is ø = sf Λ sb—any solution to this formula represents a test input. Instead of solving the entire formula ø at once (as is done conventionally), we first run the analyzer to solve sb to generate an instance ib, which is an assignment of sets of tuples to relations in sb. Next, we run the analyzer on sf while using ib as a lower bound for the new instance, i.e., a new instance must contain tuples in ib and may contain
additional tuples, for example, for relations in sf that are not in

sb. Note that, even though we execute the analyzer twice, each execution is on a formula simpler than ø. Moreover, the second execution explores a much smaller state space since ib, the lower bound, already prunes a significant part of the space.
Our incremental approach enables a novel reuse 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 reuse is of great value and enables highly optimized test generation. We developed a prototype, Kesit, which implements our approach based on the AHEAD theory and uses the recently developed Kodkod 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 speedup over the conventional approach. We believe that approaches like Kesit, which increases the feasibility of systematic testing, will likely improve our ability to deploy more dependable software.

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

II. Example

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

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, 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). Fig. 1 characterizes the eight distinct products of the BT family.

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 prestate of a method execution. 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 the 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 the cardinality of a set. When this refinement is applied to our original definition of BinaryTree, the size field is added to BinaryTree and 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 Binary
Tree 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 }
@ }
```

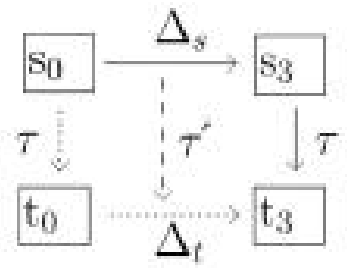


FIg. 2: BST commuting diagram.

```
@*/ }
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 subtree of a node are smaller than its element and those in the right subtree are larger.

III. Test Generation

In this section, we 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 p5 ' 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 p5's class invariant, i.e., acyclicity, size, and binary search constraints (from Fig. 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 p5'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. The commuting diagram in fig. 2 illustrates how our approach differs from the conventional approach. The nodes si represent specifications for test generation for the corresponding products, e.g., s0 represents the base specification—the acyclicity constraint. The nodes ti represent the corresponding sets of test inputs. The horizontal arrow Δs represents a refinement of the class invariant, i.e., the addition of search constraints. The vertical arrows Δ represent test generation using Alloy Analyzer. Δt represents a transformation of tests for the base product into tests for search●base; Δt is computed from Δs and t0 using the analyzer (Section IV). To generate tests t3, the conventional approach follows the path ↑ ●Δs. Our approach follows the alternative but equivalent path Δt● ↑(dotted arrows). Given p5'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 times speedup. Since our approach reuses tests already generated for another product, when testing each product in a product line, the overall speedup can be even larger.

IV. Expermential

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, AHEAD, and FOMDD.

A. 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 the 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.

B. 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 (Δsf) and its test suite (Δtf). 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 T 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 T. In addition, we use the analyzer to implement transformation T' that automatically computes Δt: T'maps a test suite t and a specification refinement Δs to a corresponding test refinement Δt. Fig. 2 shows the commuting diagram that corresponds to program p0 = [s0, t0] composed with feature search.

V. Result

(i) Test Generation

Implementations of transformations T and T' 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 T: Execute the analyzer on specification s

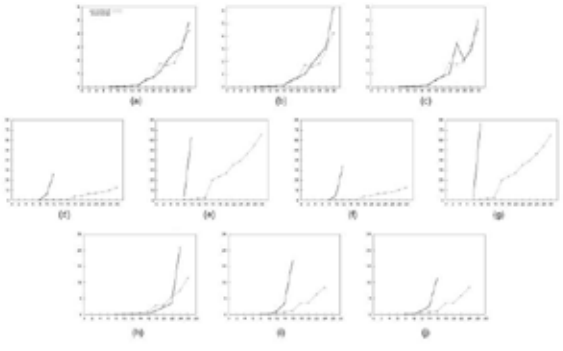
and enumerate its instances. However, the conventional use of the analyzer restricts any path (in a commuting diagram) from a specification s to a test suite t to contain horizontal arrows that are labeled Δs only. This restriction requires performing transformation T after all specification refinements 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 T 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.

VI. Evaluation

This section presents an evaluation of our incremental approach to test generation using two subject product lines: binary trees and intentional names introduced the binary tree product line. We tabulate and discuss the results for enumerating test inputs using the conventional approach and our incremental approach. 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. All experiments were performed on a 1.8 GHz PentiumM processor using 512 MB of RAM. All SAT formulas were solved using MiniSat. Our tool Kesit uses the Java API of the Kodkod back end of the Alloy Analyzer.

VII. 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 to 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 speedup.



Flg. 3: Performance chart for the subject product lines: (a) (g) binary three and (h) (j) INS. in each graph the x-axis shows the scope and y-axis show the time measurements (in seconds). Also, in each graph, solid line plots the results using the conventional approach and dashed line plots the results usings the incrementel approach. (a) size base, (b) parent base, (c) parent size base, (d) search base, (e) search size base, (f) search size base, (g) search size parent base, (h) attr-val base, (i) label attr-val base, and (j) record lavel attr-val base.

VIII. Conclusions

Testing software product lines is an important and difficult problem. We presented a novel technique that incrementally generates tests for product lines rAHEAD 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 that 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 that 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.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, J. Lilley, "The Design and Implementation of an Intentional Naming System", Proc. 17th ACM Symp. Operating Systems Principles, Dec. 1999.

[2] T. Asikainen, T. Soininen, T. Ma¨nnisto, "A Koala Based Approach for Modelling and Deploying Configurable Software Product Families", Proc. Fifth Int'l Workshop SoftwareProduct-Family Eng., 2003.

[3] C.W. Barrett, D.L. Dill, A. Stump, "Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT", Proc. 14th Int'l Conf. Computer Aided Verification, July 2002.

[4] D. Batory, "Feature Models, Grammars, and Propositional Formulas", Proc. Ninth Int'l Software Product Line Conf., 2005.

[5] D. Batory, "From Implementation to Theory in Program Synthesis", Proc. 34th Ann. ACM Symp. Principles of Programming Languages, 2007.

[6] D. Batory, G. Chen, E. Robertson, T. Wang, "Design Wizards and Visual Programming Environments for Genvoca Generators", IEEE Trans. Software Eng., vol. 26, no. 5, pp. 441-452, May 2000.

[7] D. Batory, B. Lofaso, Y. Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages," Proc. Int'l Conf. Software Reuse, 1998.

[8] D.Batory, S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", Comm. ACM, vol. 1, no. 4, pp. 355-398, 1992.

[9] D. Batory, J.N. Sarvela, A. Rauschmayer, "Scaling Step-Wise Refinement," Proc. 25th Int'l Conf. Software Eng., 2003.

[10] D. Batory, J.N. Sarvela, A. Rauschmayer, "Scaling Step-Wise Refinement", IEEE Trans. Software Eng., vol. 30, no. 6, pp. 355-371, June 2004.

[11] A. Bertolino, S. Gnesi, "PLUTO: A Test Methodology for Product Families", Proc. Fifth Int'l Workshop Software Product- Family Eng., 2003.

[12] M. Calder, M. Kolberg, E.H. Magill, S. Reiff-Marganiec,

"Feature Interaction: A Critical Review and Considered Forecast," Computer Networks, vol. 41, no. 1, pp. 115-141, 2003.

P Ratna Kumar, Associate Professor Computer Science and Engineering Department, Sir C R Reddy College of Engineering, Eluru, West Godavari District, Andhra Pradesh. He is presently pursuing Ph. D from Acharaya Nagarjuna University, Guntur. He completed his M .tech from REC, Warangal. He is having total 13 years of experience.
Raghu D, Associate Professor, Computer Science and Engineeing Department, Dr. Paul Raj Engineering College, Bhadrachalam, Khammam District. He completed his M. Tech from Kakathiya University, Warangal. He is having total 9 years experience.
Udayasree D, Assistant Professor from Computer Science Engineering Department, She is having total 5 years of teaching Experience.