Kato: A Program Slicing Tool for Declarative Specifications

Engin Uzuncaova and Sarfraz Khurshid
The University of Texas at Austin
Austin, TX 78712
{uzuncaov,khurshid}@ece.utexas.edu

Abstract

This paper presents Kato, a tool that implements a novel class of optimizations that are inspired by program slicing for imperative languages but are applicable to analyzable declarative languages, such as Alloy. Kato implements a novel algorithm for slicing declarative models written in Alloy and leverages its relational engine KodKod for analysis. Given an Alloy model, Kato identifies a slice representing the model's core: a satisfying instance for the core can systematically be extended into a satisfying instance for the entire model, while unsatisfiability of the core implies unsatisfiability of the entire model. The experimental results show that for a variety of subject models Kato's slicing algorithm enables an order of magnitude speed-up over Alloy's default translation to SAT.

1 Introduction

As software systems grow in complexity, the need for efficient automated techniques for design, testing and verification becomes even more critical. The declarative modeling language Alloy [5] and its fully automatic analyzer [6] provide an effective tool-set for building designs of systems and checking their properties.

Alloy is a first-order relational logic with transitive closure, which allows expressing rich structural properties using succinct and intuitive path expressions. The Alloy Analyzer translates Alloy models into boolean formulas using a *scope*—bound on the universe of discourse—provided by the user, and uses off-the-shelf SAT technology to solve the resulting boolean formulas.

For practical examples, Alloy's analysis is often limited to small scopes (e.g., 10 atoms in a set), which sometimes are impractically small for modeling realistic systems. The Alloy Analyzer already incorporates a variety of optimizations, such as symmetry-breaking, partial functions, type-based reduction of variables, and guidelines for manual rewriting of Alloy formulas [9] to optimize the solving time. In past work [7], we presented a suite of optimizations inspired by traditional compiler optimizations, such as common subexpression elimination and loop unrolling, to perform source-to-source translations on Alloy models to enable the SAT solvers to perform more efficiently.

KodKod [10] is a new relational engine for Alloy models. It provides a better support for partial solutions and a clean Java API to work with Alloy models. Combined with an improved scheme for further sharing opportunities, KodKod achieves significant performance improvement over total solving time for Alloy models. Kato uses the partial instance support in KodKod and introduces a new class of optimizations that are inspired by program slicing for imperative languages but are applicable to analyzable declarative languages, in general, and Alloy, in particular.

Program slicing has widely been used in the context of imperative programming languages. Weiser [13] defines slicing as a "source code transformation with respect to a certain behavioral aspect". The behavior of interest is specified in a particular form, which is called the slicing criterion. Weiser also points out two desirable properties in a slice. One, the slice should be obtained from the original program by statement deletion. Two, behavior of the slice should be the same as observed through the original program.

We have evaluated the potential speed-up in solving time that Kato can provide using a suite of subject examples that model structurally complex data. The results show the existence of opportunities for significant performance gains. For the binary search tree example, we observe a speed-up of 19.13X and for the linked-list example, a speed-up of 6.16X.

2 Kato: program slicing for Alloy

Kato [11, 12] performs a static analysis on a given Alloy model to identify its core based on a set of heuristics. The static analysis traverses the abstract syntax tree of the given Alloy formula, which is a conjunction of several subformulas, and builds a use-set of relations that appear in

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

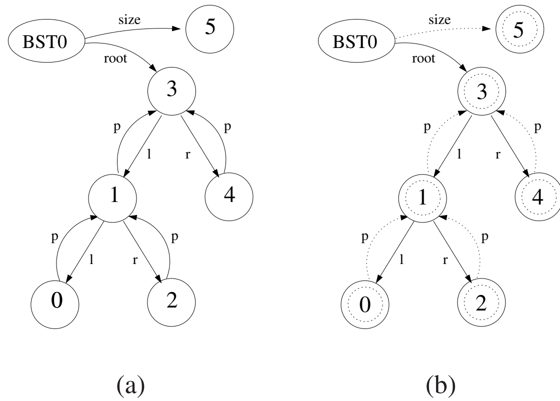

Figure 1. The binary search tree instance generated by the tool. The tree in (a) presents the complete solution. In (b), derived properties are indicated by dotted lines. (p: parent, l: left, r: right

each sub-formula. Kato uses these use-sets to compute the set of core relations based on various heuristics. We take the model slice that consists of the core relations and the constraints that apply to only these relations, and use KodKod to find a satisfying instance for the model's core. Since the model slice typically consists of only a strict subset of the original model, the slice translates to smaller boolean formulas with fewer variables, which results in a simpler, thus shorter, analysis for the underlying SAT solver. Once a solution is found for the core slice, Kato combines the derived relations and associated constraints with this partial instance and finds a solution satisfying the complete model.

In essence, our algorithm prioritizes constraints for efficient solving [14]. While SAT solvers already use constraint prioritization, the translation from Alloy to SAT reduces the efficiency of the prioritization techniques. The body of semantic information in Alloy specifications are lost at the SAT solving level since these techniques use boolean representation of the specifications. We have developed our heuristics based on the semantic properties of Alloy language observed in models in general.

2.1 Core and Derived Relations

Kato partitions the set of relations declared in an Alloy model into two sets: *core* and *derived*. We use core relations to define a slice of the given Alloy model, which is analyzed first using SAT. Next, we use the constraints on the derived relations to extend satisfying instances of core relations into satisfying instances of the complete model.

Definition. Let R be the set of all relations. Let C and D partition R. Let f_C be the formulas in f that only involve relations in C. Let I be the set of all instances of f. Let I_C be the set of all valuations to relations in C and I_D be the set of all valuations to relations in D. C is a *core set* if and only if:

$$\forall i_C \in I_C \mid f_C(i_C) \Rightarrow \exists i_D \in I_D \mid f(i_C + +i_D)$$

Definition. A relation r is *core* if and only if there exists a core set such that r belongs to that set. Similarly, a relation r is *derived* if and only if a core set exists and r is not in the core set.

3 Example: Binary Search Tree

This section presents an example use of Kato to slice a model of binary search trees [3]. Figure 2 shows the model that we analyze with Kato. The predicate `Acyclic` represents a parameterized formula that can be invoked elsewhere, or executed using a `run` command. `Acyclic` defines the acyclicity constraints of a binary tree. The fact `Parent` represents the constraint on the `parent` relation; `Size` specifies the constraint on the `size` relation. `Search` specifies the ordering property of a general binary search tree regarding the values in the nodes. The rest of the model defines acyclicity and connectivity. The `run` command checks if the model is satisfiable for a scope of 12.

For the boolean formula corresponding to the complete model, the KodKod analyzer solver takes 25.44 seconds (on average) and produces a valid instance for the binary search tree with 12 nodes. Figure 1(a) illustrates this instance with five nodes.

As the initial step, Kato constructs the use-sets for each formula. Each use-set contains the relations referred to by that formula. Our implementation uses this information to evaluate the core and derived slices in the model. For example, one of the heuristics we experiment is the union of all the relations referred by the declarations in all the quantified formulas in the model. We obtain the core relations from the model with respect to the command being executed. In order to do this, the first pass of the static analysis builds the use-sets for each formula. The core relations computed by Kato for the binary search tree model are `left`, `right` and `root`.

To intuitively see what we mean by the core of the model, notice that `parent` relation can be computed, once the left and right relations are determined. Similarly `size` and in fact even `key` can also be computed given the values for the other relations. Unsatisfiable constrainsts over these relations would lead to unsatisfiability of the entire model. Our main benefit is finding the solution in shorter time. Figure 1(b) shows a partial instance corresponding to the core relations using solid lines; the dotted lines depict the derived relations. The core relations computed by Kato are `left`, `right` and `root`. The generated core slice re-

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

```
 1 module binarytree
 2
 3 sig Tree {
 4   root: lone Node,
 5   size: Int
 6 }
 7
 8 sig Node {
 9   left : lone Node,
10   right : lone Node,
11   parent: lone Node,
12   key : Int
13 }
14
15 pred Acyclic(t: Tree) {
16   all n: t.root.*(left + right) {
17     n !in n.(left + right)
18     no n.left & n.right
19     lone n.(left + right)
20   }
21 }
22
23 fact Parent {
24   all t: Tree {
25     all n, n': t.root.*(left+right) |
26       n in n'.(left+right) => n' = n.parent
27     no t.root.parent
28   }
29 }
30
31 fact Size {
32   all t: Tree |
33     int t.size = #t.root.*(left + right)
34 }
35
36 fact Search {
37   all n: Tree.root.*(left+right) {
38     all n': n.left.*(left+right) |
39       int n'.key < int n.key
40     all n': n.right.*(left+right) |
41       int n.key < int n'.key
42   }
43 }
44
45 fact Connected {
46   Tree.root.*(left + right) = Node
47 }
48
49 run Acyclic for 12
50   but 1 Tree, 5 int, 13 Int
```

Figure 2. Alloy model for binary search tree

moves the relations `key`, `size` and `parent` and the constraints defined on these relations from the complete model.

Running the tool for the slice computed by Kato takes only 1.33 seconds (on average) to generate a satisfying instance for the slice, a 19.13X improvement in total solving time. In addition to improved performance, Kato also enables testing for larger scopes. For example, we are able to run analyses upto a scope of 30 for the binary tree model. Without slicing we could test only upto the scope of 14.

4 Tool Overview

We implemented an optimization module, Kato, to perform analysis and execution of program slicing for Alloy models. The tool is currently implemented as an extension to the publicly available KodKod API.

In order to identify core and derived slices for a given Alloy model, Kato takes various heuristics into consideration. Similar to the notion of 'data-flow analysis', we perform a static analysis and collect data about the use of relations with respect to each formula. The heuristics we use can be interpreted as the data-flow equations in the traditional compilers domain. We have experimented two main heuristics with Kato so far,

- union of the relations declared in quantified formulas approximates to a core set of relations.
- relations used in all of the formulas in a model approximates to a core set of relations.

While these heuristics hold to a great extent, this part of the tool, where we integrate and improve our heuristics, is being improved constantly as we observe their effects on our analysis.

The execution of Kato is sound such that the analysis of any given model does not produce a conflicting result with respect to the actual evaluation of the model. A satisfying solution to the core is used as a constraint for the derived slice; hence, the later execution of SAT solving finds valuations only for the derived properties.

Kato makes use of use-chains from traditional compiler optimization domain [1]. For a given Alloy model, Kato performs a static analysis and identifies the use-set for each formula. While the current state of the tool uses this approach for a limited number of heuristics, we are planning to extend this into a more general framework to support a wide range of optimizations.

4.1 Discussion

Kato's idea of slicing declarative models opens a new avenue for developing a range of novel optimizations for analyzing Alloy models. To illustrate, consider the `Search` constraints for binary search tree (Section 3). There is no reason why we must use a SAT solver or a Java program for that matter to computes the values of keys. We could instead use a dedicated solver, such as the Omega library [8] or CVC-lite [2], for integer constraints. Thus, Kato enables the use of a variety of solvers in conjunction, and we plan to explore this further.

It is worth pointing that the problem of generating boolean formulas that optimize analysis of underlying SAT solvers is particularly challenging because the performance

IEEE COMPUTER SOCIETY

of SAT solvers cannot be described in any simple terms. There are two guiding heuristics in the field: reducing the number of variables tends to reduce the solving time and increasing the number of constraints also tends to reduce the solving time(presumably because it reduces the search space that the SAT solver must explore). These are just heuristics and do not hold always [4]. Thus, even though our slicing optimizations aims at generating smaller boolean formulas with fewer variables, there is no guarantee that the formulas that represent a core slice would generate a solution faster than a formula that represents the entire model. In the context of Alloy the problem is even more interesting because of the optimizations that the Alloy Analyzer does internally. We plan to systematically explore these issues.

The problem of augmenting a partial solution to represent a complete solution is non-trivial to solve in general; in the most general case, it is as complex as SAT solving. However, even if the computation of derived relations is expensive, our approach still offers potential benefits. For example, analysis of just a core slice may reveal that the whole model is infeasible. In another scenario, it may turn out that enumerating and augmenting partial instances is actually faster than directly solving the complete model.

5 Conclusions

We have presented Kato, a novel technique that defines program slicing for declarative models and enables efficient analyses using existing analyzers. Given a declarative model, Kato identifies a slice, which represents the model's *core*: a satisfying solution to the slice can be systematically extended to generate a solution for the entire model, while unsatisfiability of the core implies unsatisfiability of the entire model. The experimental results show that it is possible to achieve a significant improvement in the solving time for Alloy models. We believe analyses based on program slicing hold a lot of promise for efficiently checking declarative specifications.

Acknowledgements

This work was funded in part by NSF ITR-SoD award #0438967.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Boston, MA, 1986.

[2] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference On Computer Aided Verification*, Boston, MA, July 2004.

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

[4] M. K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proceedings of the 39th Conference on Design Automation (DAC)*, pages 747–750, June 2002.

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

[6] D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[7] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, and M. Rinard. Optimizations for compiling declarative models into boolean formulas. In *8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, St.Andrews, Scotland, 2005.

[8] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 31(8), August 1992.

[9] I. Shlyakhter. *Declarative Symbolic Pure Logic Model Checking*. PhD thesis, MIT, February 2005.

[10] E. Torlak and G. Dennis. Kodkod for alloy users. In *First Alloy Workshop, co-located with the 14th ACM/SIGSOFT Symposium on Foundations of Software Engineering (FSE'06)*, Portland, OR, November 2006.

[11] E. Uzuncaova and S. Khurshid. Program slicing for declarative models. (Under submission).

[12] E. Uzuncaova and S. Khurshid. Program slicing for declarative models. In *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, November 2006. (Poster paper).

[13] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449, San Diego, California, March 1981. IEEE Computer Society Press.

[14] J. Yuan, C. Pixley, and A. Aziz. *Constraint-Based Verification*. Springer-Verlag, 2006.

IEEE COMPUTER SOCIETY