Constraint Prioritization for Efficient Analysis of Declarative Models

Engin Uzuncaova and Sarfraz Khurshid

The University of Texas at Austin
Austin, TX 78712
{uzuncaov,khurshid}@ece.utexas.edu

Abstract. The declarative modeling language Alloy and its automatic analyzer provide an effective tool-set for building designs of systems and checking their properties. The Alloy Analyzer performs bounded exhaustive analysis using off-the-shelf SAT solvers. The analyzer's performance hinges on the complexity of the models and so far, its feasibility has been shown only within limited bounds. We present a novel optimization technique that defines program slicing for declarative models and enables efficient analyses exploiting partial solutions. We present an algorithm that computes transient slices for Alloy models by partitioning them into a base and a derived slice. A satisfying solution to the base slice is systematically extended to generate a solution for the entire model, while unsatisfiability of the base implies unsatisfiability of the entire model.

By generating slices, our approach enables constraint prioritization, where the base slice assumes higher priority than the derived slice. Compared to the complete model, base and derived slices represent smaller and, ideally, simpler sub-problems, which, in turn, enables efficient analyses for the underlying SAT solvers. Our approach analyzes the structure of a given model and constructs a set of candidate slicing criteria. Our prototype tool, Kato, performs a small-scope analysis for each criterion to determine whether declarative slicing optimization provides any performance gain and, if so, to select a criterion that is likely to provide an optimal performance enhancement. The experimental results show that, with declarative slicing, it is possible to achieve significant improvements compared to the Alloy Analyzer.

1 Introduction

Testing and verification become more challenging as software systems grow in complexity. Automated techniques are even more critical today to achieve a certain level of confidence in software quality. Alloy [5] is a declarative modeling language that can be used for building designs of systems. Together with its fully automatic analyzer [6], Alloy provides an effective tool-set for checking system 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 Boolean satisfiability (SAT) solvers to generate a satisfying instance for and determine the consistency and feasibility of the Boolean formulas. Since Alloy's analysis is valid with respect to a

J. Cuellar and T. Maibaum (Eds.): FM 2008, LNCS 5014, pp. 310–325, 2008.

given bound, the analyzer's failure to generate a satisfying instance does not amount to a proof of non-existence of satisfying instances. It is therefore natural for Alloy users to (iteratively) check the formulas using a bound as large as is feasible within the amount of time they have.

The Alloy Analyzer already incorporates a variety of optimizations to improve the solving time by generating optimized Boolean formulas such as symmetry-breaking and type-based reduction of variables [10]. 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. However, the main limitation with the analyzer is that it generates one (typically large) SAT problem, which can choke the underlying solver.

In this paper, we present a new class of optimizations, *declarative slicing*, which are inspired by program slicing for imperative languages [14] but are applicable to analyzable declarative languages, in general, and Alloy, in particular. We present a novel algorithm for slicing declarative models. Given an Alloy model, our prototype tool, Kato, uses a *slicing criterion* to partition the model into a *base* and a *derived slice*. A base slice consists of a subset of the model constraints that constrains only the relations specified by the slicing criterion. A satisfying instance for the base is systematically extended into a satisfying instance for the entire model using the derived slice, while unsatisfiability of the base implies unsatisfiability of the entire model.

We use the partial solution support in the KodKod relational engine [11] for extending a satisfying instance of a base slice to a satisfying instance for the original model. We first generate a partial instance solving the base slice, and then we conjoin that instance with the constraints defined on the derived slice and re-execute the analyzer. Since slices typically consist of only a strict subset of the original model, the slices translate to smaller Boolean formulas with fewer variables. Even though this method executes SAT twice, each time it executes on, ideally, a simpler Boolean formula than the one that represents the whole model, which enables more efficient analysis for the underlying SAT solvers.

Slicing enables *constraint prioritization* [15] since a base slice assumes a higher priority than the derived slice. However, the potential performance improvement of using declarative slicing depends on how a given model is partitioned into slices. Kato performs a static analysis of the model to identify a set of *candidate slicing criteria*, where each criterion defines a valid base slice for the model. The candidate set is formed using the *free variables* that appear in each constraint. To select a slicing criterion that would provide a *likely* optimal performance gain, we perform a heuristic evaluation. The evaluation performs declarative slicing for each slicing criterion for a small scope and considers the overall solving time as the basis for selection.

While slicing in the context of Alloy is not a new idea because, for example, the Alloy Analyzer only translates the formulas that are relevant to the command being analyzed (akin to dead code elimination), we present a fundamentally different approach to slicing. Our slicing algorithm introduces a *transient* notion of slicing for Alloy models. Even though we partition the given model into a base and a derived slice, the partitioning is to optimize the underlying analysis, and the final result pertains to the complete

model. Our technique thus, applies even when the entire model is *necessary* to compute the result of the analysis. This contrasts even with traditional program slicing for imperative languages, where only the computed slice is of interest and the rest of the code is considered irrelevant and hence ignored [14].

We have evaluated the potential speedup in solving time that Kato can provide using a suite of benchmark examples that model structurally complex data. The results evince the existence of opportunities for significant performance gains. This paper builds on our FSE 2006 poster presentation [12] and ICSE 2007 tool demonstration [13] and makes the following contributions:

- We introduce the notion of transient program slicing for declarative specifications;
- We present an algorithm that computes transient slices for Alloy models by partitioning them into base and derived components;
- We present an algorithm for solving declarative models using slices; and,
- We present experimental results that show significant opportunities for optimizing analyses of Alloy models using slicing.

2 Example: Binary Search Tree

This section illustrates our optimization technique using a binary search tree example. We describe the example using Alloy notations [5]. Section 4 presents a more detailed discussion of our approach.

2.1 Alloy Model: Binary Search Tree

Consider a binary search tree [2], which is acyclic, satisfies the search constraints, has parent pointers and caches the number of nodes in the tree. The Alloy model for this data structure is shown in Figure 1.

The keyword `sig` is a declaration and introduces a set of (indivisible) atoms; the signatures `BinaryTree` and `Node` respectively declare a binary tree atom and a set of node atoms. The *fields* of a signature declare *relations*. The field `root` introduces a relation of type `(BinaryTree x Node)` and `left`, `right` and `parent` relations of type `(Node x Node)`. `lone` indicates that these relations are partial functions. The field `key` declares a partial relation of type `(Node x Int)` and specifies that the tree stores integer values at each node, where `Int` is a built-in Alloy type that represents the domain of integers. `size` relation has a type `(Tree x Int)` and caches the number of nodes in the tree.

The `Acyclic` *predicate* constrains the structure to be acyclic. The predicate is a universally quantified (`all`) formula which represents an implicit conjunction of three sub-formulas. The expression `t.root.*(left+right)` defines all the nodes reachable from the root node following zero or more traversals along the `left` and `right` children. The operator '`*`' denotes reflexive-transitive closure. The first sub-formula uses the quantifier `lone` to say that all nodes are either left or right child of another node or none. The second sub-formula says that a node's itself is not reachable by following its `left` and `right` children. The operator '`+`' denotes set union and '`~`' denotes transpose of a relation.The expression `n.^(left+right)` represents the set of

```
sig BinaryTree { root : lone Node }

sig Node {
  left, right, parent : lone Node,
  key : Int }

pred Acyclic(t:Tree) {
  all n: t.root.*(left+right) {
    lone n.~(left+right)
    n !in n.^(left+right)
    no n.left & n.right }}

pred Search(t:Tree) {
  all n: t.root.*(left+right) {
    all n': n.left.*(left+right) | int n'.key < int n.key
    all n': n.right.*(left+right) | int n.key < int n'.key }}

pred Parent(t:Tree) {
  all n, n': t.root.*(left+right) | n in n'.(left+right) ⇔ n' = n.parent
  no t.root.parent }

pred Size(t:Tree) {
  int t.size = #(t.root.*(left+right)) }

pred Generate[t:Tree] {
  Acyclic[t] && Search[t] && Parent[t] && Size[t]}

run Generate exactly 1 Tree, exactly 8 Node
```

Fig. 1. Alloy model for binary search tree. The model captures `acyclicity`, `search`, `size` and `parent` constraints.

all nodes reachable from `n` following one or more traversals along the `left` and `right` fields. The operator '`^`' denotes transitive closure. The third sub-formula says that left and right children are distinct using the set intersection operator '`&`'.

The `Search` predicate defines the ordering over the integer values stored at each node using a nested quantification. The keyword `int` represents the actual integer value denoted by the expression it precedes, for instance, '`n.key`'. The operator '`<`' is used for integer comparison.

The `Parent` predicate defines the parent pointer for each node and that `root` node has no parent. The operator '⇔' represents bi-implication.

The `Size` predicate constrains the `size` field to represent the number of nodes in the tree. The '`#`' operator denotes the set cardinality.

To instruct the analyzer, we formulate the `Generate` predicate specifying the constraints and write a `run` command stating that we want to generate a binary search tree with exactly 8 nodes.

Performance of the Alloy Analyzer: Invoked on the `Generate` predicate, the analyzer takes 4.05 seconds on average to generate a binary search tree with exactly 8 nodes. An instance satisfying all the model constraints is illustrated in Figure 7(b). As the scope increases the solving time increases significantly; for example, the analyzer takes 128.78 seconds for scope 12 and 538.54 seconds for scope 16 on average. Due to the increasing size of the generated Boolean problem, the Alloy Analyzer (the underlying SAT solver) fails to generate a binary search tree instance beyond scope 16 within a reasonable time (1 hour).

Declarative Slicing: We next illustrate how using declarative slicing can improve the solving time. We compare our approach against the *conventional* use of the analyzer, where models are solved in a single execution of the underlying SAT solver. Given the same Alloy model for binary search tree, Kato identifies a set of candidate slicing criteria and selects a likely optimal one by evaluating each criterion for a small scope. For the binary search tree example, Kato identifies the following set of relations as the slicing criterion:

c = {root, left, right, parent}

The base and the derived slices are generated based on the given criterion. With declarative slicing, we solve the base slice first and then conjoin that instance with the constraints defined on the derived slice and re-execute the analyzer with this new SAT problem. Our approach improves the solving time significantly, where it takes 0.77 second on average to generate a binary search tree with exactly 8 nodes. An illustration of declarative slicing is shown in Figure 7. This amounts to a performance gain of 5.24 times. As opposed to the behavior observed with the conventional approach, the analysis time does not increase drastically as the scope increases. It takes 1.81 seconds for scope 12 and 2.90 seconds for scope 16. In addition, we were able to generate a binary search tree with 32 nodes in 23.84 seconds on average using the same slicing criterion, which is far beyond the largest scope, 16, we were able to reach with the conventional approach.

3 Background: Alloy and Program Slicing

3.1 Alloy

Alloy [5] is a first-order declarative language based on sets and relations. The Alloy Analyzer [6] is a tool for automatically analyzing models written in Alloy. The analyzer translates Alloy models into boolean formulas and uses off-the-shelf SAT technology to solve the formulas. The analyzer consists of the following: a front-end that parses Alloy models into an intermediate representation (IR), a set of optimizations on this IR, and a back-end that translates IR into boolean formulas.

Each Alloy model consists of data (i.e., sets and relations), formulas that constrain data, and commands that represent invocations of the analyzer. The formulas can be structured using predicates (i.e., parameterized formulas that can be invoked elsewhere), which the analyzer inlines. Additionally, each analysis specifies a *scope* (i.e., a bound on the size of basic sets within which to check the formulas). The analyzer translates

a conjunction of all formulas relevant to the command being executed into a boolean formula—the boolean formula has a solution iff there are some sets and relations that satisfy all the constraints represented by the relevant Alloy formulas (thus providing a satisfying instance). Alloy is a relational language; every expression in Alloy denotes a relation (or a set in the case of a relation of arity one). Even the scalars are represented as singleton sets. More details of the Alloy language are available elsewhere [5].

3.2 Constraint Partitioning and Prioritization

Yuan et. al. introduces *constraint prioritization* to address some of the challenges in constraint-based verification of electronic designs [15]. Instead of solving a set of constraints as a whole, some constraints may assume higher priority than others. In such a case, the variables constrained by the lower-priority constraints can be decided using the valuations to the higher-priority constraints. For example, in our binary search tree example, one can generate a binary tree based on the constraints pertaining to the tree properties and then assign integer values to the nodes in this tree to satisfy the search constraints. However, they also note that the necessary information for how to partition the constraints is usually only known by the user. They propose that the user can either provide priorities or specify the constrained variables for each constraint.

Given a set of constraints, *constraint prioritization* enables identifying independent subsets of those constraints that can be analyzed independently and/or incrementally. Yuan et.al. [15] presents a framework for constrained functional verification, where prioritization is used as a technique to minimize representation of constraints and complexity of constraint solving as well. Constraint prioritization provides for enhanced expressiveness of constraints, but simplification of complicated constraint system into manageable blocks which make constraint solving more efficient.

Yuan et al. also uses *constraint partitioning* to partition constraints by identifying disjoint input variable support within the constraint, which enables solving each partition separately. While this approach is based on the assumption that disjoint sets of variables exist in a constraint, our approach follows a partitioning approach based on the semantic information observed in Alloy models. In either way, the main benefit of partitioning is reduction in problem size and, therefore, more efficient SAT solving.

4 Our Approach

4.1 Alloy Language Structure

An Alloy model consists of first-order logic formulas over free variables (relations). Figure 2 shows the language syntax for a subset of Alloy along with how free variables are computed for each formula type. An Alloy instance represents a valuation to all the declared relations such that the formulas evaluate to true; in other words, all the model constraints are satisfied. 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`$^{\mathrm{T}}$. Thus, for each Alloy relation, an instance gives the set of tuples that valuate the relation.

Language Syntax	Free Variables
$model$ ::= $relDecl^*$ $formula$	
$formula$ 'f ::=	
$formula\ binOp\ formula$	$f_v(f.left) \cup f_v(f.right)$
$mult\ expr$	$f_v(f.expr)$
not $formula$	$f_v(f.formula)$
$expr$ [**not**]$compOp\ expr$	$f_v(f.left) \cup f_v(f.right)$
$quantifier\ varDecl^+\ formula$	$f_v(f.formula) - f_v(f.varDecl)$
$expr$ e ::=	
$expr\ binExprOp\ expr$	$f_v(e.left) \cup f_v(e.right)$
$unaryOp\ expr$	$f_v(e.expr)$
rel	$e.identifier$
var	$\emptyset$
$relDecl$ r ::= $rel : arity$	$\cup f_v(r.rel)$
$varDecl$ v ::= $var : expr$	$f_v(v.expr) - f_v(v.var)$
rel ::= identifier	
var ::= identifier	
$arity$ ::= 1 | 2 | ...	
$binOp$ ::= **and** | **or** | **implies** | **iff**	
$mult$ ::= **one** | **lone** | **some** | **no**	
$compOp$ ::= **in** | **=**	
$quantifier$ ::= **all** | **some**	
$binExprOp$ ::= **+** | **&** | − | **.** | → | **++**	
$unaryOp$ ::= ˜ | ^ | *	

Fig. 2. On the left column, a subset of the Alloy language syntax is shown. For each language construct, right column shows how corresponding set of free variables are computed.

To solve a formula, the Alloy Analyzer uses a scope that bounds the universe of discourse. The Kodkod back-end of the Alloy Analyzer allows specifying a scope 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. Hence, a lower bound represents a partial solution for a given model. Kodkod's support for partial solution is one of the key advantages for efficient analysis. Mathematically, a bound `b` is a pair of two functions: a lower bound `l` and an upper bound `u`, each of type `R -> 2`$^{\mathtt{T}}$. An instance can equivalently be viewed as a bound `b = [l, u]`, where `l = u`.

Thus, an Alloy model can be seen as a triple $\langle d,\ s,\ b\rangle$, where d represents the data elements, i.e., declared relations, s represents the formula that constrains the data elements (i.e., constraints), and b represents the bound on the universe discourse. We assume (without loss of generality) that s is a conjunction of several sub-formulas, i.e., $s = \bigwedge s_i$.

4.2 Declarative Slicing

Declarative slicing is an optimization technique that exploits partial solutions for efficient analysis of declarative specifications. It is inspired by program slicing for imperative languages [14], and is applicable to analyzable declarative languages, in general,

```
1  all n: t.root.*(left+right) | lone n.~(left+right)          //{root, left, right}
2  all n: t.root.*(left+right) | n !in n.^(left+right)         //{root, left, right}
3  all n: t.root.*(left+right) | no n.left & n.right           //{root, left, right}
4  all n: t.root.*(left+right) {                               //{root, left, right, key}
     all n': n.left.*(left+right) | int n'.key < int n.key }
5  all n: t.root.*(left+right) {                               //{root, left, right, key}
     all n': n.right.*(left+right) | int n.key < int n'.key }
6  all n, n': t.root.*(left+right) |                           //{root, left, right, parent}
     n in n'.(left+right) ⇔ n' = n.parent
7  no t.root.parent                                            //{root, parent}
8  int t.size = #(t.root.*(left+right))                        //{root, left, right, size}
```

Fig. 3. *Free variables* for each sub-formula is shown next to it in parenthesis. This form is equivalent to the original model.

and Alloy, in particular. Our technique introduces an alternative way to analyze Alloy models and provides significant performance enhancement compared to the conventional use of the Alloy Analyzer.

With declarative slicing, an Alloy model is partitioned into a *base* and a *derived* slice, where each slice is a disjoint subset of the model constraints. A base slice is defined by a *slicing criterion*, c, which is a set of relations to be constrained by the constraints in the base slice. Given a model and a slicing criterion, our algorithm (Figure 4) generates the base and the derived slices. The base slice is analyzed first and then the constraints in the derived slice are used to extend a satisfying instance for the base into a satisfying instance for the entire model (Figure 5).

Considering all the relations declared in a given Alloy model, there can be many different slicing criteria, hence many different base slices. We refer to the set of all possible criteria for a model as the set of candidate slicing criteria. Note that a model itself can also be considered as a slice with a slicing criterion consisting all the declared relations. Recall that slicing optimization aims at improving the performance of the Alloy analyzer by generating smaller and, ideally, simpler sub-problems from the complete model: The performance gain that can be achieved varies based on the partitioning defined by a particular slicing criterion. From the set of all candidate criteria, we select a criterion that would provide a *likely* optimal improvement in solving time using a heuristic evaluation (Section 4.3). Next, we describe the notion of $slice$, the algorithm for constraint partitioning and how we solve Alloy models using slices.

Definition 1. Let R be the set of all relations. Let R_b and R_d partition R. Let f_b be the formulas in f that only involve relations in R_b. Let I be the set of all instances of f. Let I_b be the set of all valuations to relations in R_b and I_d be the set of all valuations to relations in R_d. R_b defines a *base slice* if and only if:

$$\forall i_b \in I_b \mid f_b(i_b) \Rightarrow \exists i_d \in I_d \mid f(i_b +\!\!+ i_d)$$

Essentially, a base instance represents a partial solution for the complete model and the key idea behind declarative slicing is to extend this solution to a complete solution using the derived slice. However, it is also possible for a particular partial instance to

a base slice, there is no way to extend it to a complete instance using the given scope. Such cases can be handled by using an enumerating SAT solver, such as mchaff [8]. Enumerating partial instances continues until there is one that can be extended, which we refer to as *filtering*. Indeed, if no such partial solution is found it means that the complete model is unsatisfiable for the given scope. Therefore, it is imperative to identify a base slice properly (if there exists any) to prevent inefficient applications of declarative slicing.

Constraint Partitioning. To generate slices, we use a *constraint partitioning* algorithm as described in Figure 4. The algorithm traverses the abstract syntax tree (AST) of a given model and evaluates each formula based on the provided slicing criterion. For this, a preprocessing is performed on the AST before the algorithm starts. First, the composite constraints (e.g., nested quantified formulas) are broken into sub-formulas by a mechanical transformation. This form represents the model as a conjunction of sub-formulas. Figure 3 shows this form for the binary search tree model (Section 2).

Next, the algorithm computes the free variables for each sub-formula. Figure 2 shows how free variables are computed for a subset of the Alloy language constructs. Free variables for a formula states which variables (relations in Alloy models) are constrained by that formula. Free variables for the sub-formulas of the binary search tree model are also shown in Figure 3.

After the initial preprocessing, each sub-formula is evaluated based on the given slicing criterion: If a formula contains a free variable that is not part of the slicing criterion it is deleted from the model; hence, it is not included the base slice and added to the derived slice. As a result, the algorithm generates two disjoint subsets from the model's sub-formulas, where the conjunction of the base slice s_b and the derived slice s_d is equivalent to the original model, s: $s \Leftrightarrow s_b \wedge s_d$. Figure 6 show an example for how base and derived slices are formed.

Approach for Slicing-based Constraint Solving. We provide an algorithm (Figure 5), which enables a fully automatic solver for declarative slicing. The algorithm takes as

```
⟨Formula, Formula⟩ Slicer(Model s, Set⟨Relation⟩ c){
   Formula base = s.formulas();
   Formula drv = true;
   Set⟨Relation⟩ R = s.relations();
   for(Formula f: s.formulas()){
      if (f_v(f) ⊈ c){
         base = base.remove(f);
         drv = drv.and(f);
      }
   }
   return ⟨base, drv⟩;
}
```

Fig. 4. Algorithm for slicing an Alloy model

```
Instance SlicingSolver(Model s, Set⟨Relation⟩ c){
   Slice s_b, s_d = Slicer(s, c);
   Bound bound = s_b.bound().update(null);
   Iterator⟨Instance⟩ itr = Alloy.solveAll(s_b, bound);

   Instance I, I_b = itr.getNext();

   while(I_b.outcome == SAT){
      bound = s_d.bound().update(I_b);
      I = solve(s_d, bound);
      if(I.outcome() == SAT)
         break;
      else
         I_b = itr.getNext();
   }
   return I;
}
```

Fig. 5. Algorithm for analyzing an Alloy model using base and derived slices

c = {root, left, right}

s_b = all n: t.root.*(left+right) | lone n.∼(left+right)
all n: t.root.*(left+right) | n !in n.^(left+right)
all n: t.root.*(left+right) | no n.left & n.right

s_d: all n: t.root.*(left+right) {all n': n.left.*(left+right) | int n'.key < int n.key }
all n: t.root.*(left+right) { all n': n.right.*(left+right) | int n.key < int n'.key }
all n, n': t.root.*(left+right) | n in n'.(left+right) ⇔ n' = n.parent
no t.root.parent
int t.size = #(t.root.*(left+right))

Fig. 6. For the given slicing criterion, the base and the derived slices represent a disjoint subset of the sub-formulas shown in Figure 3

.

input an Alloy model, s, and a slicing criterion for the base slice, c, and generates a satisfying instance for the model by analyzing the base and the derived slices and combining their solutions. Successive calls are made to the analyzer to generate a complete instance satisfying all the model constraints. We first generate an instance for the base, and then extend the instance with respect to the derived slice. In case a base instance cannot be extended to a complete solution, we continue enumerating the base instances until either a satisfying instance is found or all the base instances are exhausted. The later case implies the unsatisfiability of the entire model.

The algorithm extends a base instance into an instance for the complete model by conjoining the base instance with the derived slice. This is achieved by tightening the bound for the derived slice, which forces the lower and upper bounds to be equal to the

valuations in the base instance for the base relations (`s`$_d$`.bound().update(I`$_b$`)`). This guarantees the satisfaction of base constraints in the final solution. In addition, the final solution includes valuations for the relations constrained by the derived slice and satisfies the derived constraints on all relations.

Illustration. Consider the base and the derived slices in Figure 6. Invoking the Alloy Analyzer on the base slice results in the following valuation, I_b, for scope 3, which is graphically illustrated in Figure 7(a):

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

We run the analyzer on the derived slice and set the lower and upper bounds for the free variables that are solved for base (i.e., `BinaryTree`, `Node`, `root`, `left` and `right`) to the values in the instance I_b. The analyzer generates a satisfying solution for the entire model, I, by combining the valuations in I_b with the new relations `Int`, `size`, `key` and `parent`:

```
Int = { 0, 1, 2 }       key = { <N0, 1> <N1, 0>, <N2, 2> }
size = { <BT0, 3> }     parent = { <N1, N0>, <N2, N0> }
```

Figure 7 (b) graphically illustrates this tree, which is indeed a binary *search* tree.

4.3 Selecting a *Likely* Optimal Criterion

Previous section described the declarative slicing optimization in detail. It is important to note that the efficiency of the optimization heavily depends on the selected slicing criterion. We introduce a heuristic evaluation approach as an enabling technique for the declarative slicing optimization. Using sub-formulas and free variables generated during initial preprocessing (Figure 3), we identify a set of *candidate* slicing criteria. Each criterion in this set represents a valid partitioning. We evaluate each candidate for a small scope to; (1) determine whether the optimization is likely to produce a significant performance gain, and (2) if so, select a likely optimal criterion.

Intuitively, slicing criteria could be selected from the power set of declared relations for a given model. However, such an approach would cause a substantial amount of analysis overhead to find an optimal criterion; for example, the binary search tree model

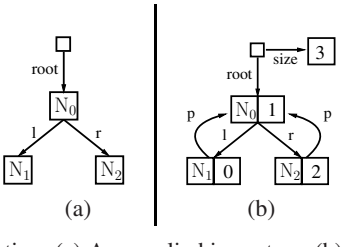

Fig. 7. Base and complete solution. (a) An acyclic binary tree. (b) An acyclic binary search tree with size field and parent pointers. The small unlabeled square represents the `BinaryTree` atom `BT0`; nodes `N`$_0$, `N`$_1$, `N`$_2$ are `Node` atoms. Edges represent valuations of binary relations.

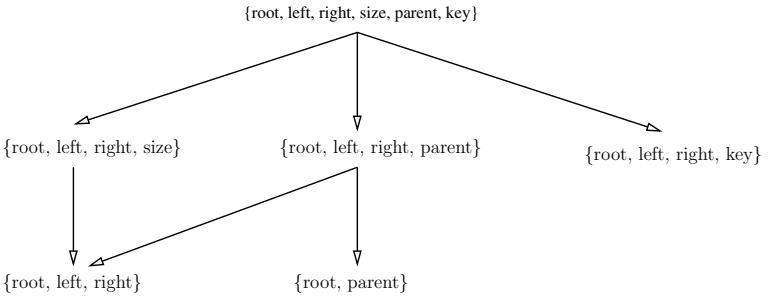

Fig. 8. Partially ordered set of free variables for the binary search tree model

we used earlier declares 6 relations, which results in 64 distinct set of relations that could be use as slicing criteria.

Slicing (constraint partitioning), ρ can be thought of as a function from a slicing criterion, c, to a base slice, s_b,: $\rho : \ c \rightarrow \ s_b$. Given a random criterion, it may map to an empty slice, i.e., there is no constraint in the model that constrains only the relations in that criterion. For example, using $\{\texttt{root},\texttt{left}\}$ as the slicing criterion for the binary search tree example would map to an empty slice: $\rho(\{\texttt{root},\texttt{left}\}) \rightarrow \emptyset$. This indicates that a slicing criterion, c, is valid candidate if and only if there is at least one sub-formula s_i that the set of its free variables is a subset of c: $f_v(s_i) \ \subseteq \ c$.

Using this as the basis of our approach, we use free variables of the sub-formulas to prune the candidate set, where each distinct set of free variables is a candidate slicing criterion. Using the binary search tree example again, we identify that there are only 5 different slicing criteria that would map to a base slice (numbers next to each criterion indicate the sub-formulas from Figure 3 to be included in the base slice):

$\rho(\{\texttt{root, left, right}\}) \rightarrow s_b = 1 \ \wedge \ 2 \ \wedge \ 3$
$\rho(\{\texttt{root, left, right, key}\}) \rightarrow s_b = 1 \ \wedge \ 2 \ \wedge \ 3 \ \wedge \ 4 \ \wedge \ 5$
$\rho(\{\texttt{root, left, right, parent}\}) \rightarrow s_b = 1 \ \wedge \ 2 \ \wedge \ 3 \ \wedge \ 6$
$\rho(\{\texttt{root, left, right, size}\}) \rightarrow s_b = 1 \ \wedge \ 2 \ \wedge \ 3 \ \wedge \ 8$
$\rho(\{\texttt{root, parent}\}) \rightarrow s_b = 7$

The set of candidate slicing criteria can also be shown as a partially ordered set using subset inclusion as in Figure 8. The partially ordered set of free variables reflects a hierarchy of criteria, as in this case, including only 5 criteria. The topmost criterion includes all the declared relations in the binary search tree model, which defines a base slice as the complete model. Following the downward arrows, each criterion defines a smaller base slice, where the criteria at the bottom defines the smallest possible base slices that can be generated for the model. This approach helps us work with a small candidate set; however, we still have to try each criterion to select an optimal one.

Heuristic Evaluation. Once the candidate set is generated, we perform a *small-scope analysis* on each slicing criterion. The analysis executes the declarative slicing optimization with each candidate criterion and evaluates the results with respect to performance. The assumption is that the performance observed during small-scope analysis

indicates the performance for larger scopes as well. As a result of the evaluation, we expect to find a criterion that is likely to provide an optimal performance gain. In case there is none, declarative slicing is not effective for the given model due to the fact that we cannot find an appropriate partitioning.

For each candidate criterion, we collect and evaluate overall analysis time and relative complexity of the Boolean formulas generated by the Alloy Analyzer. If an optimal base slice exists, we expect to be able to observe it through these variables. Note that, we compare the performance against the conventional use of the Alloy analyzer. Next, we describe each variable and why they are important to measure performance of the analyzer.

Overall Analysis Time: *Overall analysis time* T consists of both the time required to translate an Alloy model into a Boolean formula and the time spent during SAT solving. Generally, translation takes much less time than SAT solving. A proper slicing criterion would be likely to yield a *significantly* better overall analysis time for declarative slicing, T_{ds}, compared to the conventional use, T_{conv}: $\lambda_T = T_{conv} \div T_{ds} > 1$.

Complexity of the Boolean Formula: *Complexity* C of a Boolean formula (generated by the analyzer) can be expressed with respect to the number of variables and clauses used in that formula: $\langle V, C \rangle$. With our slicing optimization, we generate two smaller sub-problems representing the same Boolean formula; i.e., base,$\langle V_b, C_b \rangle$, and derived, $\langle V_d, C_d \rangle$. While the total number of primary variables is equal to the original formula, $V = V_b + V_d$, the total number of clauses used for the base and the derived slices decreases: $C_b + C_d < C$. This is because each slice contains a subset of the model constraints defined on a subset of the relations. We expect a proper partitioning to yield a ratio greater than 1 for complexity: $\lambda_C = C_{conv} \div (C_b + C_d) > 1$.

5 Experiments

The sections first presents experimental results, and then presents a discussion of our technique and some future work.

5.1 Results

We present evaluation for three Alloy models: binary search tree, doubly-linked list, and red-black tree [2]. We run the experiments with our prototype tool, Kato, implemented as an extension to the KodKod relational engine and use MiniSat SAT solver [3]. The experiments were run on a Windows XP machine with 1.8 GHz processor using Java 2 SDK 1.5.

Table 9 summarizes the results we obtained from small scope analysis for each subject. Binary search tree and doubly linked list models are solved for scope 8 and red-black tree for scope 7 due to the model's complexity. While doubly linked list is a cyclic data structure, the tree subjects are acyclic. Red-black tree can be seen as a more constrained version of binary search tree, where nodes have color attributes. We generated 10 different instances for each criterion. The overall analysis times, T_{conv} and T_{ds}, represents the average time to generate one instance. Detailed times for base and derived slices are not tabulated due to space considerations. The number of clauses for

slicing criteria	solving time			clauses		
	T_{conv}	T_{ds}	λ_T	C_{conv}	$\langle C_b, C_d \rangle$	λ_C
Binary Search Tree (scope=8)						
$\{root, left, right\}$	4.12	0.82	5.01	23496	⟨8319,10826⟩	1.23
$\{root, left, right, size\}$	4.12	0.81	5.06	23496	⟨9543,9636⟩	1.23
$\{root, left, right, parent\}$	4.12	0.79	5.29	23496	⟨9324,10388⟩	1.19
$\{root, left, right, key\}$	4.12	7.24	0.57	23496	⟨21251,1365⟩	1.04
Doubly Linked List (scope=8)						
$\{header, next\}$	2.34	0.12	19.50	19177	⟨7076,4003⟩	1.73
$\{header, next, size\}$	2.34	0.12	19.50	19177	⟨7636,3600⟩	1.71
$\{header, next, prev\}$	2.34	0.09	26.00	19177	⟨8058,3731⟩	1.63
$\{header, next, key\}$	2.34	3.89	0.60	19177	⟨18125,676⟩	1.02
Red-Black Tree (scope=7)						
$\{root, left, right, color\}$	4.61	0.42	10.89	14379	⟨7621,3743⟩	1.27
$\{root, left, right, color, size\}$	4.61	0.48	9.65	14379	⟨8140,3366⟩	1.25
$\{root, left, right, color, size, parent\}$	4.61	0.61	7.52	14379	⟨8834,3110⟩	1.20

Fig. 9. The results obtained from small-scope analysis for subject models. For space considerations, criteria causing filtering are not included in the table. One important observation is the consistency between the λ_T and λ_C values.

each Boolean formula generated by the Alloy analyzer is also shown in the table. The analyzer finds satisfying solutions for all three subject models.

The speed-up in solving time, λ_T, and the reduction in the formula size, λ_C, are used for selecting a likely optimal slicing criterion as highlighted for each subject. Overall analysis times indicate significant performance gains for some of the slicing criteria, while some others provide either a smaller or negative gain. While we use the speed-up as the basis for evaluation, the performance gain and complexity reduction variables are consistent with each other for all cases, which we consider as an interesting point for further exploration. Our assumption is that the criterion with the highest positive gain represents a likely optimal base slice. Indeed, when we run declarative slicing optimization with the identified criteria on the binary tree and the linked list models, we achieve substantial performance improvements for larger scopes. This is mainly because as the scope increases SAT problems face a state explosion problem, where slicing addresses this issue by generating smaller sub-problems. While the conventional approach fails to generate instances for binary search tree beyond scope 16, Kato is able to test scope 32 only within 24 seconds on average. Similarly, Kato can generate doubly linked list instances with 32 nodes in 20.59 seconds on average using the slicing criterion identified by our approach.

While we were also able to achieve speed-ups for the red-black tree, the largest scope we were able to reach with declarative slicing was 14. For the same model, the conventional approach failed to generate instances for scope 8. In its simplest terms, this is because the red-black tree is a more complex data structure; therefore, the analysis cannot scale. In the declarative slicing case, note that every criteria (except the ones causing filtering) contains the `color` field, which causes most of the complex constraints to be included in each corresponding base slice. While declarative slicing helps the analyzer to scale, for this reason, it is not as significant as the other subject models.

In addition to the tabulated criteria, there some others that cause filtering, i.e., not all base instances can be extended for that particular base slice (not included due to space considerations). Main reason for this is that the generated base slice is underconstrained and satisfying instances for those slices are often invalid with respect to the complete model. These are identified during analysis and discarded from the candidate set, which is crucial for pruning the candidate set even further. For all subject models, the results indicate that declarative slicing optimization enables efficient analysis and our approach identifies a likely optimal slicing criterion.

5.2 Discussion

Our approach for 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 2). 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 [9] or CVC-lite [1], for integer constraints. Thus, Kato enables the use of a variety of solvers (and optimizations as well) in conjunction, and we plan to explore this further.

Currently, Kato makes successive invocations to the analyzer and the underlying SAT solver to identify a proper base slice. As the experiment results suggest, there is a direct correlation between the overall analysis time and the size of the Boolean formula generated by the analyzer. We strongly believe that more efficient techniques can be developed by introducing a tighter integration between the semantic and structural properties of declarative models and the internal decision procedures of SAT solvers. We are planning to explore ways to use such properties to directly influence the decision procedures, such as branching heuristics and learning procedures [16], within SAT solvers.

It is worth pointing that the problem of generating boolean formulas that optimize analysis of underlying SAT solvers is particularly challenging because the performance 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 (presumably because it reduces the search space that the SAT solver must explore) and increasing the number of constraints also tends to reduce the solving time (again because it reduces the search space). These are just heuristics and do not hold always [4]. In the context of Alloy the problem is even more interesting because of the semantic information associated with models and the optimizations done internally. We plan to systematically explore these issues.

6 Conclusion

We have presented declarative slicing, a novel optimization that defines program slicing for declarative models and enables efficient analyses exploiting partial solutions. As opposed to the conventional use of the analyzer, where models are solved in a single execution of the underlying SAT solver, our approach identifies two sub-problems for a model and solves each problem separately and combines their solution. This not only enables the Alloy Analyzer to scale to larger scopes, but it is also complementary to the

other optimizations addressing the scalability problem. Since the slices generated by the incremental analysis are valid SAT problems, our optimization can be used in conjunction with other optimizations to further improve the performance. 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.

References

1. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: Proc. of the 16th Int'l Conference on Computer Aided Verification(CAV) (July 2004)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press, Cambridge (1990)
3. Een, N., Sorensson, N.: An extensible sat-solver. In: Proc. of the 6th Int'l Conference on Theory and Applications of Satisfiability Testing (2003)
4. Ganai, M.K., Zhang, L., Ashar, P., Gupta, A., Malik, S.: Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In: Proc. of the 39th Conference on Design Automation (DAC), June 2002, pp. 747–750 (2002)
5. Jackson, D.: Software Abstractions: Logic, Language and Analysis. The MIT Press, Cambridge (2006)
6. Jackson, D., Schechter, I., Shlyakhter, I.: ALCOA: The Alloy constraint analyzer. In: Proc. of the 22nd Int'l Conference on Software Engineering (ICSE), Limerick, Ireland (June 2000)
7. Marinov, D., Khurshid, S., Bugrara, S., Zhang, L., Rinard, M.: Optimizations for Compiling Declarative Models into Boolean Formulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 187–202. Springer, Heidelberg (2005)
8. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of the 38th Conference on Design Automation (DAC) (2001)
9. Pugh, W.: The Omega test: A fast and practical integer programming algorithm for dependence analysis. Communications of the ACM 31(8) (August 1992)
10. Shlyakhter, I.: Declarative Symbolic Pure Logic Model Checking. PhD thesis, MIT (February 2005)
11. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Proc. of the 13th Int'l Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS) (2007)
12. Uzuncaova, E., Khurshid, S.: Program slicing for declarative models. In: Proc. of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), Portland, OR (November 2006) (poster paper)
13. Uzuncaova, E., Khurshid, S.: Kato: A program slicing tool for declarative specifications. In: Proc. of the 29th Int'l Conference on Software Engineering (ICSE) (May 2007) (Research Demo)
14. Weiser, M.: Program slicing. In: Proc. of the 5th Int'l Conference on Software Engineering (ICSE), March 1981, pp. 439–449. IEEE Computer Society Press, Los Alamitos (1981)
15. Yuan, J., Pixley, C., Aziz, A.: Constraint-Based Verification. Springer, Heidelberg (2006)
16. Zhang, L., Malik, S.: The quest for efficient boolean satisfiability solvers. In: Proc. of the 8th Conference on Automated Deduction (CADE) (July 2002)