Annotations for Alloy: Automated Incremental Analysis Using Domain Specific Solvers

Svetoslav Ganov, Sarfraz Khurshid, and Dewayne E. Perry

Electrical and Computer Engineering
University of Texas at Austin, Austin TX 78712, USA
`svetoslavganov@utexas.edu | {khurshid,perry}@ece.utexas.edu`

Abstract. Alloy is a declarative modeling language based on first-order logic with sets and relations. Alloy problems are analyzed fully automatically by the Alloy Analyzer. The analyzer translates a problem for given bounds to a propositional formula for which it searches a satisfying assignment via an off-the-shelf propositional satisfiability (SAT) solver. Hence, the performed analysis is a bounded exhaustive search and increasing the bounds leads to a combinatorial explosion.
We increase the efficiency of the Alloy Analyzer by performing incremental analysis via domain specific solvers. We introduce annotations that define data types, operations on these data types, and bindings from data types to domain specific solvers. This meta-data is utilized to automatically partition a problem into sub-problems and opportunistically solve independent sub-problems in parallel using dedicated constraint solvers. We integrate dedicated Integer and String constraint solvers into Alloy's SAT based backend. Experimental results show that using dedicated solvers and exploiting independent sub-problems provide better efficiency and scalability; for the chosen subjects, our technique enables up to an order of magnitude speed-up.

1 Introduction

Alloy [1] is a declarative modeling language based on first-order logic with sets and relations. It has been successfully used for identifying problems in semantic models and algorithms [11], detecting anomalous scenarios in security-critical systems [19], automated test generation [14], modeling software architecture [5], etc. Alloy problems are analyzed fully automatically by the Alloy Analyzer. The analyzer translates a problem for given bounds to a propositional formula for which it searches a satisfying assignment via an off-the-shelf propositional satisfiability (SAT) solver. Hence, the performed analysis is a bounded exhaustive search and increasing the bounds leads to a combinatorial explosion.

However, performing analysis within given bounds only guarantees that the obtained results are valid within these bounds. Therefore, increasing the bounds of the analysis would strengthen the confidence in the obtained results. To enable reasoning for increased bounds we focus on improving the speed of Alloy's SAT based backend by exploiting two key ideas: (1) a problem can be decomposed into sub-problems which can be solved incrementally and potentially in parallel; and (2) domain specific solvers enable faster evaluation of problems in their target domain.

The first key idea is that when an Alloy model is translated to SAT, an opportunity to perform a more efficient incremental analysis is not exploited. Incremental reasoning enables reducing the space searched by the solver [17] and enables tackling independent sub-problems in parallel, thus improving performance due to better utilization of contemporary multi-core architectures. For example, generating a binary tree may be performed by generating the structure and using this partial solution to solve in parallel the independent sub-problems of generating the keys, the size, and the parent relationship. To address this we employ an incremental technique for solving Alloy formulas [17], where one solution to a formula provides a partial solution to another formula, which can then be solved more efficiently. We improve this technique by recursively decomposing the problem into the sub-problems, as opposed to decomposing it into only two sub-problems, and opportunistically solve independent sub-problems in parallel.

The second key idea is that when an Alloy formula is translated for the SAT solver, domain specific knowledge is lost, thus an opportunity to take advantage of the problem domain is not exploited. Domain specific solvers are designed for tackling special classes of problems using special representations, and algorithms [6][12]. For example, finding whether two String variables can be equal is faster by getting the intersection of two automatons than by exploring the cross product of all possible values for the two variables. To address this we introduce *annotations*, an easy-to-use and unobtrusive facility to embed meta-data for mapping Alloy signatures to data types, Alloy predicates to operations on these data types, and bind data types to domain specific solvers. This enables us to opportunistically solve a predicate that depends only on variables of a single data type via the dedicated constraint solver mapped to this data type.

The benefit of our approach is three-fold: (1) incremental analysis limits the search space explored by the solver; (2) opportunistically solving independent sub-problems in parallel improves utilization of system resources; and (3) domain specific solvers are more efficient for problems in their target domain. Experiments show our technique enables better performance and scalability than a SAT-based approach.

This paper makes the following contributions:

– **Incremental analysis with parallel reasoning about independent sub-problems**. We perform incremental analysis of an Alloy problem by recursively decomposing it into sub-problems. We identify likely independent sub-problems and solve them in parallel if they are indeed independent.
– **Domain specific solver integration via annotations**. We introduce annotations that define data types, operations on these data types, and bindings from data types to domain specific solvers. This enables reasoning about when to use a domain specific solver and how to translate Alloy formulas to the language of that solver.

- **Dedicated solvers for Alloy**. We support a dedicated Integer and a dedicated String constraint solver integrated into Alloy's SAT based backend.
- **Implementation**. We implement our approach into a custom build of the Alloy Analyzer.
- **Evaluation**. We evaluate our approach using small but complex Alloy models, including a model from the standard Alloy distribution. Empirical results show our approach provides up to an order of magnitude speed-up over Alloy's purely SAT-based analysis.

2 Background

In this section we provide background knowledge about Alloy [1] and declarative slicing [17] for incrementally analyzing Alloy models in the context of a binary search tree example.

2.1 Alloy

An Alloy model s can be represented as a triple $< r, s, b >$, where r is the set of relations in s, and b is the bound on the universe of discourse. An *instance*, i.e. a solution, i satisfying an Alloy model is a function from the set of relations r to a power set of tuples 2^T where each tuple consists of indivisible atoms, i.e., $i : r \rightarrow 2^T$. Hence, an instance gives the set of tuples that valuate every relation. The *canonical form* of an Alloy model is $\wedge p_i$, for $i = 1, ..., n$, where each p_i is an arbitrary formula.

2.2 Alloy model - binary search tree example

A binary search tree is a node-based data structure where: (1) each node has at most two children–left and right–whose parent is the given node; (2) the left sub-tree rooted at a given node contains keys less than the key of that node; (3) the right sub-tree rooted at a given node contains keys greater than the key of that node; and (4) the left and right sub-trees are also binary search trees; In Fig. 1 is depicted the Alloy model for a binary search tree.

First, we declare the entities contained in the model. A node (line 3) has: (1) at most one left child (line 4); (2) at most one right child (line 5); (3) at most one parent (line 6); and (4) a key (line 7); A binary tree (line 9) has: (1) at most one node as its root (line 10); and (2) a size (line 11);

Next, we specify the relationships between the model entities to reflect the properties of the data structure. A binary search tree is *Acyclic* (line 13), which is for every node reachable from the root performing zero or more traversals (line 14): (1) at most one node is visited following the left and right relations in reverse direction (line 15); (2) a node cannot be reached by following one or more times the left and right relations beginning from that node (line 16); and (3) the left and right nodes are disjoint (line 17);

```
1   module BinarySearchTree
2
3   sig Node {
4     left:lone Node,
5     right:lone Node,
6     parent:lone Node,
7     key:Int
8   }
9   sig BinarySearchTree {
10    root:lone Node,
11    size:Int
12  }
13  pred Acyclic(t:BinarySearchTree) {
14    all n:t.root.*(left+right) {
15      lone n.~(left+right)
16      n !in n.^(left+right)
17      no n.left & n.right
18    }
19  }
20  pred Parent(t:BinarySearchTree) {
21    all n,n':t.root.*(left+right) | n in n'.(left+right) => n' = n.parent
22    no t.root.parent
23  }
24  pred Search(t:BinarySearchTree) {
25    all n:t.root.*(left+right) {
26      all n':n.left.*(left+right) | int n'.key < int n.key
27      all n':n.right.*(left+right) | int n.key < int n'.key
28    }
29  }
30  pred Size(t:BinarySearchTree) {
31    int t.size = #(t.root.*(left+right))
32  }
33  pred BinarySearchTree(t:BinarySearchTree) {
34    Acyclic[t] && Parent[t] && Search[t] && Size[t]
35  }
36  run BinarySearchTree exactly 1 BinarySearchTree, exactly 3 Node
```

Fig. 1. Alloy model of a binary search tree.

The nodes in the binary search tree have a *Parent* property (line 20), which is: (1) every node reachable from the root performing zero or more traversals is the parent of its left and right children (line 21); and (2) the root has no parent (line 22);

A binary search tree contains data satisfying the *Search* (line 24) property, which is for every node reachable from the root performing zero or more traversals (line 25): (1) every descendant reached by following the left and right relations of its left child zero or more times has a lesser key (line 26); and (2) every descendant reached by following the left and right relations of its right child zero or more times has a greater key (line 27);

A binary search tree has a *Size* property (line 30), which is the cardinality of the nodes reached from its root by performing zero or more traversals of the left and right relations (line 31).

In order for a data structure to be a *BinarySearchTree* (line 33) it has to satisfy the *Acyclic*, *Parent*, *Search*, and *Size* predicates (line 34).

Finally, we request from the analyzer to find an instance by specifying bounds on the cardinality of atoms (line 36). Upon running this command the analyzer

```
// free variables: {root, left, right}
all n:t.root.*(left+right) | lone n.~(left+right)
// free variables: {root, left, right}
all n:t.root.*(left+right) | n !in n.^(left+right)
// free variables: {root, left, right}
all n:t.root.*(left+right) | no n.left & n.right
// free variables: {root, left, right, parent}
all n,n':t.root.*(left+right) | n in n'.(left+right) => n' = n.parent
// free variables: {root, parent}
no t.root.parent
// free variables: {root, left, right, key}
all n:t.root.*(left+right) {
  all n':n.left.*(left+right) | int n'.key < int n.key }
// free variables: {root, left, right, key}
all n:t.root.*(left+right) {
  all n':n.right.*(left+right) | int n.key < int n'.key }
// free variables: {root, left, right, size}
int t.size = #(t.root.*(left+right))
```

Fig. 2. Normalized form of the constraints in the binary search tree model.

tries to find valuations to the relations such that the predicate declaring a binary search tree evaluates to true, which is an instance that satisfies the model exists.

2.3 Declarative slicing - binary search tree example

Declarative slicing [17] in the context of an Alloy model in a canonical form - $\wedge p_i$, for $i = 1, ..., n$, where each p_i is an arbitrary formula - is to partition an Alloy model s into a *base* slice s_b and a *derived* slice s_d. The base and the derived slices consist of disjoint subsets of the model constraints. The base slice is derived via a *slicing criterion* c which is a subset of the model relations r, i.e. $c \subseteq r$. The base slice contains the constraints that involve only relations in the slicing criterion, i.e., $s_b : \wedge q_i$ for $i = 1, \ldots, m$, where each $q_i \in \{p_1, \ldots, p_n\}$ and $free\_variables \cap predicate\_variables(q_i) \subseteq c$. The rest of the model constraints belong to the derived slice, i.e. $s_d : \wedge d_i$ for $i = 1, \ldots, t$, where each $d_i \in \{p_1, \ldots, p_n\}$ and $free\_variables \cap predicate\_variables(d_i) \nsubseteq c$. Once the model is partitioned into a base and a derived slice, a solution for the base slice is extended into a solution for the entire model.

The first step in the declarative slicing technique is model normalization during which composite constraints (e.g. nested quantified formulas) are partitioned, i.e. the model is translated to a canonical form. The normalized form of the binary search tree model from Section 2.2 is presented in Fig. 2.

The second step is choosing an optimal slicing criterion, since more than one such may exist, by using each possible slicing criterion to analyze the model for a smaller bounds and based on some metrics selecting the one that is likely to provide the most significant speed up. The possible slicing criteria are ordered under set containment where each slicing criterion represents a free variable combination from some model constraints. The possible slicing criteria for the binary search tree are presented in Fig. 3.

The third step is solving the problem for the desired bounds using the optimal slicing criterion from the previous step. It is possible that a solution for the base

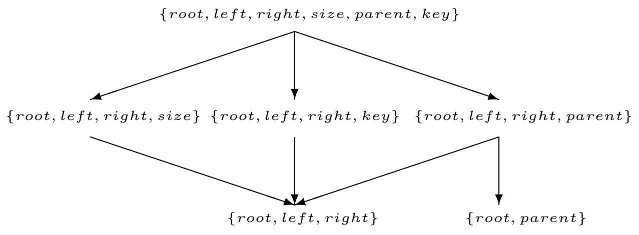

{root, left, right, size, parent, key}
{root, left, right, size}
{root, left, right, key}
{root, left, right, parent}
{root, left, right}
{root, parent}

Fig. 3. Partially ordered set of slicing criteria.

slice cannot be extended to a solution for the entire problem since the derived slice may additionally constrain the relations in the slicing criterion. Therefore, all possible solutions for the base slice are attempted to be extended to a complete solution. If a solution for the entire problem is found, the problem is reported consistent, otherwise the problem is declared inconsistent for the given bounds.

3 Our approach

In this section we present our approach for incremental analysis of Alloy problems via domain specific solvers. We exploit two key observations: (1) a problem can be decomposed into sub-problems which can be solved incrementally and potentially in parallel; and (2) domain specific solvers enable faster evaluation of problems in their target domain.

3.1 Incremental analysis with parallel reasoning

When an Alloy model is translated to SAT, an opportunity for a more efficient incremental analysis is not exploited. Previous work [17] introduced an approach to decompose the problem into a base and a derived slice, and extend a solution for the base to one for the entire problem. This approach improves traditional analysis but there are two areas for enhancement: (1) the problem is partitioned only into two sub-problems; and (2) the two sub-problems are solved sequentially.

Instead of partitioning the problem into two sub-problems, as the declarative slicing technique presented in Section 2.3, we partition the problem into multiple sub-problems. We apply the partitioning procedure for declarative slicing recursively. We select the smallest slicing criterion, i.e. the criterion with the minimum number of free variables, and partition the problem into a base and a derived slice. Then we select the next smallest slicing criterion and partition the derived slice from the previous step into a base and a derived slice. We repeat the procedure until the slicing criterion includes all free variables.

Recall that the possible slicing criteria for declarative slicing can be envisioned as a partially ordered set under set containment which is represented as a join-semi-lattice. We use that partially ordered set of slicing criteria to construct a dependency graph $G = (V, E)$ with a set of vertices V and a set of edges E as follows: (1) add a vertex $v_i \in V$ for every slicing criterion c_i where $i \in (1, \dots, n)$;

$$\begin{tikzcd}
& c_6 & \\
c_3 \arrow[ur] & c_4 \arrow[u] & c_5 \arrow[ul] \\
& c_1 \arrow[ul] \arrow[u] \arrow[ur] & c_2 \arrow[u]
\end{tikzcd}$$

$c_1 = \{root, left, right\}$
$c_2 = \{root, parent\}$
$c_3 = \{root, left, right, size\}$
$c_4 = \{root, left, right, key\}$
$c_5 = \{root, left, right, parent\}$
$c_6 = \{root, left, right, size, parent, key\}$

Fig. 4. Slicing criteria dependency graph.

and (2) add a directed edge $e_{ij} \in E$ from the vertex representing criterion c_i to the vertex representing criterion c_j if the slicing criterion c_j is the smallest slicing criterion that contains c_i, i.e. $c_i < c_j \wedge \nexists c_k : c_i < c_k < c_j \rightarrow e_{ij} = (v_i, v_j)$ where $i \neq j \neq k$ and $i, j, k \in (1, \ldots, n)$. The dependency graph for the binary search tree from Section 2.2 is shown in Fig. 4.

Once we have constructed the dependency graph, we perform a topological sort of the nodes in that graph to obtain an ordering of the slicing criteria used for incremental analysis. For example, a topological ordering of the nodes in the dependency graph from Fig. 4 is the sequence $< c_1, c_2, c_3, c_4, c_5, c_6 >$. This slicing criteria sequence is used to incrementally analyze the model.

We slice the model based on the first slicing criterion from the sequence, i.e. select constraints that involve only relations from the slicing criterion. Then we solve the constraints in the slicing criterion. If a solution is found, we slice the model based on the second slicing criterion. Before solving the constraints for current base we set the relations in these constraints to valuations we found in the previous step (e.g. we set a partial solution). We now solve the constraints. This procedure is repeated until a solution for the entire problem is found or we determine that the problem is inconsistent.

Since the constraints in the current iteration may constrain relations whose valuations were found in a previous one (e.g. such relations were under constrained in the previous iteration), it is possible that a solution for these constraints cannot be found. In such a case we backtrack to the sub-problem we solved in the previous iteration for which we try to find another solution which is then propagated to the current sub-problem and a new attempt to solve the current sub-problem is made. In case no solution for the previous sub-problem can be extended to a solution for the current one, we declare the previous sub-problem inconsistent and try to backtrack to the sub-problem preceding it. If no solution for the first sub-problem can be extended to a solution for the entire problem, we declare the problem inconsistent.

Note that a subsequence of slicing criteria in the topologically sorted sequence may share no common relations. In such a case we solve the sub-problems for each slicing criterion in parallel. It is also possible that a subsequence of slicing

```
Solution solve(Problem problem) {
  // Try solving potentially parallel problems.
  List bases = getParallelBases(problem);
  Solution solution = solveParallel(bases);
  if (solution.isValid()) {
    Problem slice = problem - bases;
    if (slice == null) {
      return solution;
    }
    slice.setPartialSolution(solution);
    solution = solve(slice);
    if (solution.isValid()) {
      return solution;
    }
  }
  // Try extending a base solution to a full one.
  Problem base = getBase(problem);
  Problem slice =  problem - base;
  for (Solution solution in solveAll(base)) {
    if (slice == null) {
      return solution;
    }
    slice.setPartialSolution(solution);
    solution = solve(slice);
    if (solution.isValid()) {
      return solution;
    }
  }
  return Solution.INVALID;
}
```

Fig. 5. Incremental analysis with parallel reasoning algorithm.

```
module Integer

. . .

@datatype(solver="IntegerSolver")
sig integer {
    value: int
}

@operation(name="lessThan")
pred integerLessThan(lhs, rhs: int) {
    lhs.value < rhs.value
}

. . .

```

Fig. 6. Sample of the Annotated Integer module.

```
for (BinaryTree t:eval(BinaryTree)) {
  for (Node n:eval(t.root.*(left+right))) {
    for (Node n':eval(n.root.*(left+right))) {
      IntVar lhs = Solver.newVar(
          t.name() + n'.name());
      IntVar rhs = Solver.newVar(
          t.name() + n.name());
      Solver.lessThan(lhs, rhs);
    }
  }
}
```

Fig. 7. Constraint translation for a dedicated Integer solver.

criteria share common relations but none of the predicates that belong to their corresponding slices imposes constraints on the common relations. Hence, sub-problems resulting from slicing the model based on a subsequence of independent, i.e. with no edge in the dependency graph, slicing criteria *may* be independent. In such a case we try to solve such sub-problems in parallel for small bounds and, in case we succeed, we solve the sub-problems in parallel for the current bounds. The described algorithm is presented in Fig. 5.

In particular, for the binary search tree model our algorithm performs an incremental analysis based on the following slicing criteria
$c_1 \rightarrow c_2 \rightarrow parallel(c_3, c_4, c_5) \rightarrow c_6$, where the predicates corresponding to slicing criteria c_3, c_4, and c_5 are solved in parallel. Note that our algorithm on Fig. 5 is doing a best effort for solving as many sub-problems as possible in parallel. In case the solutions for the parallel sub-problems are conflicting (due to additional constraints on the common relations), we perform systematic exploration of each slicing criterion one at a time.

3.2 Dedicated solver integration via annotations

When an Alloy problem is translated to propositional logic, domain specific knowledge is lost. However, knowledge about the problem domain creates an opportunity of using domain specific solvers–solvers designed for tackling special classes of problems via special representations and algorithms. Performing an incremental analysis of an Alloy model creates an opportunity for employing domain specific solvers for relevant sub-problems, i.e. slices. This opportunity has been recognized by previous work on declarative slicing [17]. However, there are three limitation of this work that we address: (1) there is no generic mechanism for integrating domain specific solvers into the Alloy engine; (2) a mechanism for determining when a domain specific solver can be used is lacking; and (3) analysis only with an Integer domain specific solver has been presented.

We introduce annotations that define data types, operations on these data types, and bindings from data types to domain specific solvers. These annotations allow us to determine when to use a given domain specific solver as well as facilitates the translation from Alloy to the language of the specialized solver.

The *@datatype* annotation can be placed only on a signature definition and specifies a mapping from an Alloy signature to a domain specific data type. This annotation has one attribute, *solver*, that specifies which solver to use for reasoning about constraints over the annotated signature (the data type is inferred by the solver). We also define an annotation *@operation* that can be placed only on a predicate and specifies to which domain specific operation to map the predicate (the solver is inferred from the type of the arguments). An annotated model for an Integer type which serves as a wrapper around the built-in *int* to enable annotation is presented in Fig. 6.

We use the meta-data from the annotations and *evaluation* of an expression against a given instance (solution) supported by Alloy's backend Kodkod [16] to determine whether a domain specific solver may be used for solving the constraints in the currently analyzed slice and how to translate these constraints to the language of the dedicated solver. We illustrate this with an example.

Assume we already have a solution of a binary search tree with three nodes for the previous slicing criterion $\{root, left, right\}$ and the current slicing criterion is $\{root, left, right, key\}$ for which we have set the partial solution from the previous slice. We traverse the constraints in the current slice, and for each of them: first check via an evaluation whether it is already satisfied and, if not, what free variables it constrains. If all unsatisfied constraints constrain free variables of the same type, we can use the dedicated solver specified in the *@datatype* annotation, otherwise we fall back to SAT. In the example, all constraints for slicing criterion $\{root, left, right, key\}$ constrain only the free variable *key* which is of type Integer annotated to use the *IntegerSolver*. Hence, we can use the dedicated Integer constraint solver.

Now we have to translate the constraints in the slice to the language of the solver. Without loss of generality consider the constraint $one\ t : BinaryTree\ \&\&$ $all\ n : t.root.*(left + right)\{all\ n' : n.left.*(left + right)\ |$ $integerLessThan(n'.key, n.key)\}$ (added declaration of t for clarity and changed key to our custom Integer type). We evaluate the variable t from declaration $one\ t : BinaryTree$ which returns $\{BinaryTree\$0\}$. Hence, for some binary tree in this set the constraint must hold. We next evaluate the variable n from the next declaration $n : t.root.*(left + right)$ for every t which returns $\{Node\$0, Node\$1, Node\$2\}$. Follows evaluation of the variable n' from the next declaration $n' : n.left.*(left + right)$ for every n which returns $\{Node\$1, Node\$2\}$, $\{Node\$2\}$, $\{\}$. Now we have identified all nodes whose keys are constrained and we can use them to construct a problem in the domain of the Integer solver. We add a variable for the key of each node and a constraint for every pair on nodes constrained by $integerLessThan(n'.key, n.key)$. Note that the predicate $integerLessThan$ is mapped to the $lessThan$ domain specific operation. A translation of the constraint to the language of the Integer constraint solver is presented in Fig. 7. For solving integer constraints we use Choco [4].

In addition to the dedicated Integer constraint solver we provide a specialized solver for String constraints. Similarly, we define a custom module for the String data type to which we apply the corresponding annotations. Since existing off-the-shelf String solvers [6][12] do not support multi-variable problems, we have implemented a String constraint solver that supports multi-variable problems with binary constraints. We use recursive backtracking search with constraint propagation, via the AC-3 [13] algorithm, applying the *minimal remaining values* and *degree* heuristics to guide the search. We use finite state automata [3] to represent variable domains and performing operations on them.

Adding other solvers can be done similarly to the Integer and String ones. An Alloy module, i.e. a model file, with operation mappings has to be written and a JAR file with the solver implementation has to be deployed. No changes to the Alloy source are required. Hence, we provide a general mechanism for adding domain specific solvers to the Alloy's SAT based backend.

4 Evaluation

We have evaluated our approach on several data structure models with Integer and String data and a P2P protocol model. Each model was analyzed with the conventional Alloy Analyzer (4.1.10) and a version that incorporates our technique. We report analysis results in terms of solving time for given bounds and maximal bounds reached within reasonable time.

The evaluation system was a laptop with an Intel i7 M620 2.67GHz processor, 4GB of RAM running Ubuntu 10.04 Lucid. For all experiments the analyzer was set to use 4096MB of memory and 65536K maximal stack size. All tests were run on a cold VM to avoid just-in-time compilation or cache skewing the results.

An analysis command takes as arguments a bund for each signature, recall the command for the binary search tree model in Fig 1 (line 46). We have set these parameters as follows: (1) the bound for the signature representing the modeled data structure or state for the protocol was always one; and (2) the bound for the signature representing nodes of the data structure or the protocol was incrementally increased.

Table 1. Sorted linked list with Integer data

Node count	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**20**	**30**
Standard Analyzer (ms)	111	335	896	2401	15695	58307	124335	N/A	N/A	N/A
Incr. SAT solver (ms)	202	372	822	1251	1743	2460	4197	4131	52256	252572
Incr. multi-solver (ms)	327	412	781	1244	1251	1760	3179	2971	46817	234658

Table 2. Sorted linked list with String data

Node count	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**20**	**24**
Standard Analyzer (ms)	1763	3276	7293	69851	147406	294445	N/A	N/A	N/A	N/A
Incr. SAT solver (ms)	2158	2702	6168	4688	4738	19822	81546	58312	160286	N/A
Incr. multi-solver (ms)	673	984	1358	1515	2280	2516	3841	6602	92722	183440

Table 3. Binary search tree with Integer data

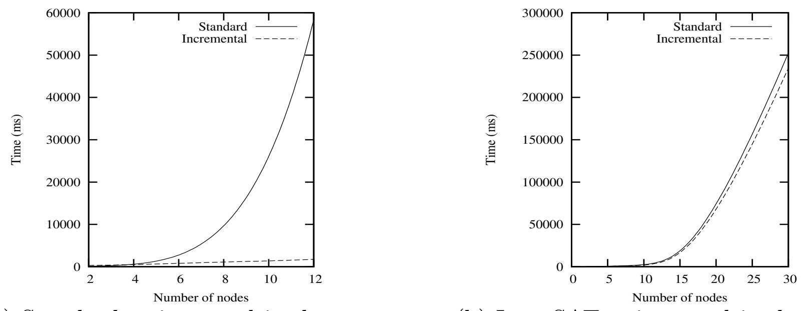
Node count	**2**	**4**	**6**	**8**	**10**	**20**	**30**	**40**
Standard Analyzer (ms)	116	322	1174	8724	N/A	N/A	N/A	N/A
Incr. SAT solver (ms)	260	414	1076	1522	2546	9400	59527	N/A
Incr. multi-solver (ms)	368	422	990	1317	1534	4935	29297	166369

Table 4. Binary search tree with String data

Node count	**2**	**4**	**6**	**8**	**10**	**20**	**30**	**40**
Standard Analyzer (ms)	2307	5571	14206	127185	1241175	N/A	N/A	N/A
Incr. SAT solver (ms)	2464	3422	5181	6422	24150	94048	N/A	N/A
Incr. multi-solver (ms)	1069	1424	1752	2090	2815	13389	40847	1020314

Table 5. Chord P2P protocol

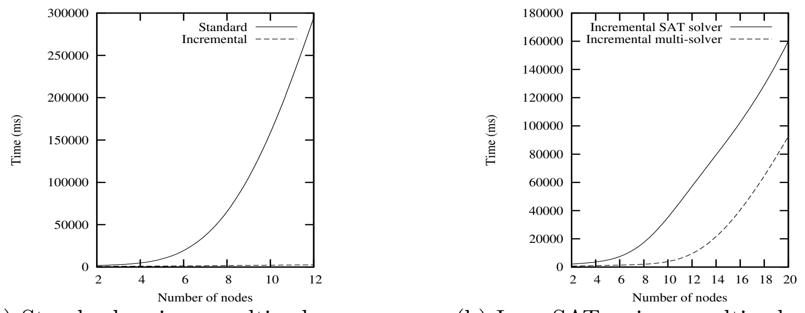
Node count	**2**	**4**	**6**	**8**	**10**
Standard Analyzer (ms)	15	89	792	7739	127987
Incr.SAT solver (ms)	52	122	708	5736	60139



(a) Standard vs incr. multi-solver (b) Incr. SAT vs incr. multi-solver

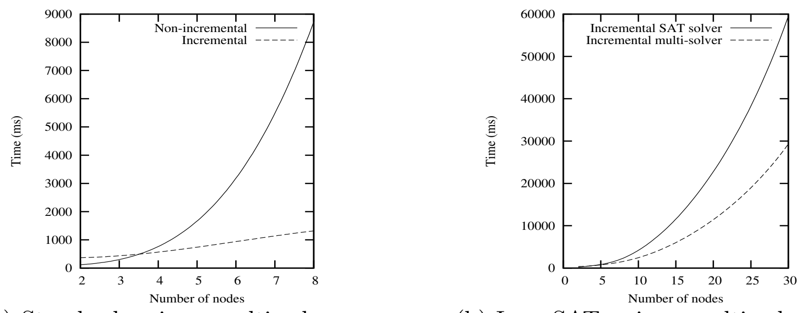
Fig. 8. Sorted linked list with Integer data

In Table 1 are presented the results for a sorted linked list with Integer data. The first row is the number of nodes, the second row is solving time via the standard Alloy Analyzer, the third row is solving time via incremental analysis



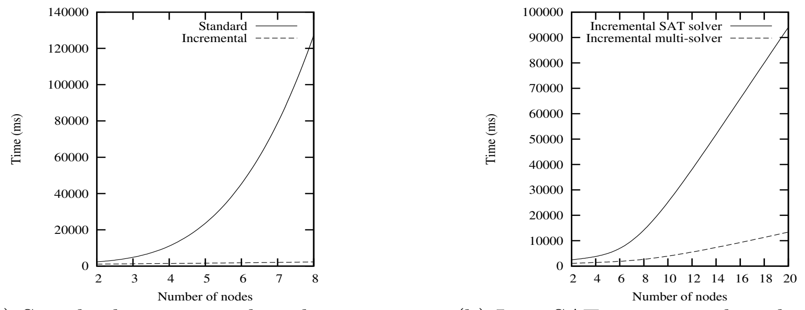
(a) Standard vs incr. multi-solver (b) Incr. SAT vs incr. multi-solver

Fig. 9. Sorted linked list with String data



(a) Standard vs incr. multi-solver (b) Incr. SAT vs incr. multi-solver

Fig. 10. Binary search tree with Integer data



(a) Standard vs incr. multi-solver (b) Incr. SAT vs incr. multi-solver

Fig. 11. Binary search tree with String data

using a SAT solver, and the fourth row is solving time via incremental analysis using multiple solvers. Note that for certain scopes no results are reported because either the analysis took more than thirty minutes or the solver ran out of memory. In Fig. 8(a) is presented the improvement in terms of solving time for

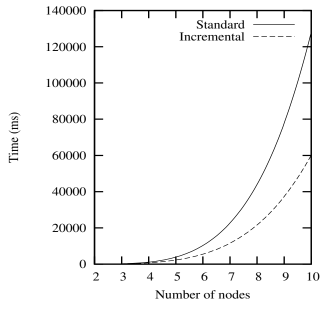


Fig. 12. Chord P2P protocol model - standard vs incremental SAT.

incremental multi-solver analysis as opposed to the standard Alloy Analyzer and in Fig. 8(b) is depicted the improvement in terms of solving time for incremental multi-solver as opposed to incremental SAT analysis. We are using Bezier curves in all figures to depict smoothened trends rather than local fluctuations.

We present results for a sorted linked list with String data (Table 2, Fig. 9(a), Fig. 9(b)), a binary search tree with Integer data (Table 3, Fig. 10(a), Fig. 10(b)), and a binary search tree with String data (Table 4, Fig. 11(a), and Fig. 11(b)) similarly to the ones for a sorted linked list with Integer data.

We have also evaluated our technique on a model of the Chord P2P protocol [2] which is one of the sample models posted on the Alloy home page. Note that the model does not incorporate Integer or String constraints which precludes the use of multiple solvers. However, the model can be analyzed incrementally. The results of our analysis are presented in Table 5 and Fig. 12. The data in the table and the figure are arranged similarly to the ones for already presented data structures. Note that our technique is twice as fast for ten nodes as opposed to the standard analysis. This example demonstrates that, even if employing multiple solvers is not feasible due to the model nature, performing incremental reasoning leads to an improvement in terms of analysis speed.

Our results indicate that for small bounds our incremental multi-solver analysis is as fast as the standard Alloy Analyzer but, as the bounds grow, the gains of using our technique become significant. This can be explained with the linearly growing costs of problem translation and multi-solver initialization which is amortized over an exponentially growing search space. Our approach is almost two orders of magnitude faster (except for the binary search tree with Integer data and the Chord P2P model) for the bounds reachable by the standard analyzer. Further, even if a model is not suitable for employing multiple solvers, it can be solved incrementally, increasing analysis speed.

5 Related work

This paper introduces annotations for Alloy models that define data types, operations on these data types, and bindings from data types to domain specific solvers. We utilize this information to automatically partition the problem into multiple sub-problems. We also employ this meta-data to determine when to use a dedicated solver as well as to facilitate translation of the solved sub-problem into the language of the domain specific solver. Additionally, we integrate an Integer constraint solver and a String constraint solver into Alloy's backend.

In a recent publication [8] we have proposed the idea of using annotations in Alloy for guiding problem partitioning. In particular, explicitly specifying the priority of a predicate as well as the solver to be used for its analysis. In the current work we present an approach to automatically partition the problem into multiple sub-problems some of which are solved opportunistically with domain specific solvers. In this paper we introduce annotations that define data types, mapping operations on such types to their domain specific counterparts, and mappings from data types to dedicated constraint solvers.

Incremental solving for Alloy models, where a solution to one formula is fed as a partial solution to solve another formula, was introduced by Uzuncaova et al. [17, 18] in the context of test input generation for software product lines. This work partitioned the problem into two sub-problems while our technique aims to maximize the number of partitions. Further, this work does not specify an algorithm for determining when it is possible to use a dedicated solver and we provide such a technique. This work solves the two sub-problems in sequence but we try to opportunistically parallelize analysis of independent sub-problems. We also have introduced a dedicated String constraint solver for Alloy.

The second author co-authored a recent paper [10] that introduces *mixed constraints*, which are written using a combination of a declarative language (Alloy), and an imperative language (Java). It supports annotating *def-use* sets of variables to facilitate solving of mixed constraints using different solvers, where each solver is designed for constraints written using one particular paradigm. Mixed constraints offer a complementary approach to this paper. They facilitate writing of constraints using a combination of declarative and imperative paradigms, whereas this paper focuses on efficient solving of models written purely in Alloy, hence does not require learning a new notation.

Parallel analysis of Alloy models was first proposed in [15]. This work explores providing a parallel SAT solver with Alloy specific enhancements by partitioning the propositional formula among parallel SAT solvers. In contrast, we partition the problem into sub-problems before it has been translated to propositional logic enabling the use of domain specific solvers. We also identify sub-problems that may be solved in parallel, some via a SAT solver.

An example of extending Alloy's syntax to describe dynamic properties of systems via actions is presented in [7]. The actions enable specifying dynamic properties of execution traces as dynamic logic specifications. Our technique is similar with respect to extending the Alloy syntax with new semantic features and implementing automated analysis of the latter. While this work focuses on adding constructs for specifying dynamic behavior, we embed meta-data in a standard Alloy model to achieve scalable and efficient analysis.

An approach of using SAT Modulo Theories solver (SMT) for analyzing Alloy specifications is presented in [9]. The SMT solver does not replace the Alloy SAT-based back-end, rather complements it by potentially detecting if a formula is a tautology, a capability the Alloy Analyzer is lacking. This does not require finitizing the values for each relation. Similarly, we introduce a set of dedicated solvers for augmenting Alloy's analysis backend. However, we are using domain specific solvers and perform an incremental analysis. Our technique is to partition the problem and solve it in finitized bounds with specialized solvers.

6 Conclusion

We increase the efficiency of the Alloy Analyzer by performing an incremental analysis via domain specific solvers. We introduce annotations that define data types, operations on these data types, and bindings from data types to domain specific solvers. This meta-data is utilized to opportunistically solve a sub-problem using a dedicated constraint solver. Our technique automatically partitions the problem into sub-problems and opportunistically solves independent sub-problems in parallel. We integrate a dedicated Integer constraint solver and a String constraint solver in Alloy's SAT based backend.

We have evaluated our approach on selected data structure models with both Integer and String data and one P2P protocol. Our technique achieves a substantial increase in analysis speed, thus enabling us to reach greater bounds. We believe that annotations have an important role to play in analysis of declarative programs as well as in the context of other analyzers such as model checkers and theorem provers.

7 Acknowledgement

This work was funded in part by the National Science Foundation under Grant Nos. IIS-0438967, CCF-0828251, CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

1. Alloy Analyzer 4 (Mar 2011), http://alloy.mit.edu/alloy4/
2. Alloy model of Chord (May 2011), http://alloy.mit.edu/community/files/chord.pdf
3. Automaton library (Mar 2011), http://www.brics.dk/automaton/
4. Choco constraint solver (Mar 2011), http://www.emn.fr/z-info/choco-solver/
5. Auguston, M.: Software architecture built from behavior models. SIGSOFT Softw. Eng. Notes 34(5), 1–15 (Oct 2009)
6. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Proc. 10th International Static Analysis Symposium (SAS). LNCS, vol. 2694, pp. 1–18. Springer-Verlag (June 2003)
7. Frias, M.F., Galeotti, J.P., López Pombo, C.G., Aguirre, N.M.: DynAlloy: upgrading Alloy with actions. In: Proceedings of the 27th international conference on Software engineering. pp. 442–451. ICSE '05, ACM, New York, NY, USA (2005)

8. Ganov, S., Khurshid, S., Perry, D.: A case for Alloy annotations for efficient incremental analysis via domain specific solvers. In: 26th IEEE/ACM International Conference on Software Engineering, Lawrence, Kan, USA (2011)
9. Ghazi, A.A.E., Taghdiri, M.: Relational reasoning via SMT solving. In: 17th International Symposium on Formal Methods (FM) (June 2011)
10. Khalek, S., Narayanan, V., Khurshid, S.: Mixed constraints for test input generation - an initial exploration. In: Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on. pp. 548 –551 (nov 2011)
11. Khurshid, S., Jackson, D.: Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In: Proceedings of the 15th IEEE international conference on Automated software engineering. ASE '00 (2000)
12. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: Proceedings of the eighteenth international symposium on Software testing and analysis. pp. 105–116. ISSTA '09, ACM, New York, NY, USA (2009)
13. Mackworth, A.K., Freuder, E.C.: The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artif. Intell. 25(1), 65–74 (1985)
14. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: Proceedings of the 16th IEEE international conference on Automated software engineering. ASE '01 (2001)
15. Rosner, N., Galeotti, J., Lopez Pombo, C., Frias, M.: ParAlloy: Towards a framework for efficient parallel analysis of Alloy models. In: Frappier, M., Glsser, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) Abstract State Machines, Alloy, B and Z, Lecture Notes in Computer Science, vol. 5977, pp. 396–397. Springer Berlin / Heidelberg (2010)
16. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4424, pp. 632–647. Springer Berlin / Heidelberg (2007)
17. Uzuncaova, E., Khurshid, S.: Constraint prioritization for efficient analysis of declarative models. In: In Proc. of the 15th Intl Symposium on Formal Methods (2008)
18. Uzuncaova, E., Khurshid, S., Batory, D.S.: Incremental test generation for software product lines. IEEE Trans. Software Eng. 36(3), 309–322 (2010)
19. Woodcock, J., Aydal, E.G., Chapman, R.: The Tokeneer experiments. In: Roscoe, A., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C.A.R. Hoare, pp. 405–430. History of Computing, Springer London (2010)