Alloy Annotations for Efficient Incremental Analysis via Domain Specific Solvers

Svetoslav Ganov
Electrical and Computer Engineering
University of Texas at Austin
svetoslavganov@mail.utexas.edu

Sarfraz Khurshid
Electrical and Computer Engineering
University of Texas at Austin
khurshid@ece.utexas.edu

Dewayne E. Perry
Electrical and Computer Engineering
University of Texas at Austin
perry@mail.utexas.edu

***Abstract*—Alloy is a declarative modelling language based on first-order logic with sets and relations. Alloy formulas are checked for satisfiability by the fully automatic Alloy Analyzer. The analyzer, given an Alloy formula and a scope, i.e. a bound on the universe of discourse, searches for an instance i.e. a valuation to the sets and relations in the formula, such that it evaluates to true. The analyzer translates the Alloy problem to a propositional formula for which it searches a satisfying assignment via an off-the-shelf propositional satisfiability (SAT) solver. The SAT solver performs an exhaustive search and increasing the scope leads to the combinatorial explosion problem.**

We envision annotations, a meta-data facility used in imperative languages, as a means of augmenting Alloy models to enable more efficient analysis by specifying the priority, i.e. order of solving, of a given constraint and the slover to be used. This additional information would enable using the solutions to a particular constraint as partial solutions to the next in case constraint priority is specified and using a specific solver for reasoning about a given constraint in case a constraint solver is specified.

I. Introduction

The world today is so tightly integrated with computers that life without them seems hard to imagine. We rely on computer software to fly planes, manage bank transactions, communicate etc. It has permeated our homes, offices, cars, etc. Software is so deeply weaved into the fabric of our lives that assuring its high quality is a task of paramount importance.

As processing power of computers increases [1], so does the complexity of software they run [2]. To manage this increasing complexity researchers have introduced various techniques for *verification*, i.e. to check if we are building the system right, and *validation*, i.e. are we building the right system [3][4][5][6]. Some techniques rely on formal specifications to describe structural properties [4][7] or define runtime behaviour [8][9] of software systems. A benefit of using formal specifications is they are amenable to automated analysis which is faster, consistent, and less error prone.

Alloy [10] is a declarative modelling language based on first-order logic with sets and relations. Alloy formulas are checked for satisfiability by the fully automatic Alloy Analyzer. The analyzer, given an Alloy formula and a *scope*, i.e. a bound on the universe of discourse, searches for an *instance*, i.e. a valuation to the relations in the formula, such that it evaluates to true. The analyzer translates the solved problem to a propositional formula for which it searches a satisfying assignment via an off-the-shelf SAT solver. The SAT solver performs exhaustive search and increasing the scope leads to a combinatorial explosion.

One key observation 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 structure is not exploited. Domain specific solvers are designed for tackling special classes of problems using special representations, and algorithms [11][12]. For example, representing a string variable as an automaton is more compact than explicit enumeration of all possible values and finding whether two strings variables can be equal is faster by getting the intersection of two automatons than exploring the cross product all possible values for the two variables.

Another key observation is that the author of an Alloy model is familiar with the problem domain and problem structure. Such knowledge can be utilized for more efficient solving by dividing the problem into sub-problems and solving them in order of dependence. For example, generating a sorted linked list can be partitioned in two sub-problems, generating the structure, and generating the data. Since one needs a structure to generate the data, the structure sub-problem can be solved first and its solution used for reasoning about the data sub-problem.

However, to take advantage of the problem domain and its structure, a mechanism for capturing that data in the model is required. We envision *annotations* as an easy-to-use and unobtrusive facility to perform this task. For example, initially the author may add no annotations to the model and do that incrementally as his or her knowledge of the problem domain and structure grows.

We propose an annotation mechanism for Alloy. The annotations allow the user to annotate a constraint and state which solver to use for that constraint as well as to state a priority for solving it. Prioritizing constraints allows leveraging an incremental technique [13] for solving Alloy formulas, where solution to a formula provides a partial solution to another formula, which can then be solved more efficiently.

Embedding annotations in an Alloy model would allow taking advantage of *1)* incremental analysis that limits the search space explored by the solver; *2)* use of domain specific solvers which are efficient for problems in their target domain.

This paper makes the following contributions:

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

Fig. 1. Alloy model of a binary search tree

- **Annotations for Alloy** We envision annotations to incorporate meta-data into an Alloy model to guide the solving process. Annotations are a commonly used mechanism in imperative languages; we introduce them for a declarative language.
- **Dedicated solver support for Alloy** We envision support for a dedicated constrint solvers in the Alloy analyzer, thus allowing a constrain to be annotated with the solver to be used for its analysis.

II. Background - Binary Search Tree Example

According to its definition 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 Figure 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 10) has: *1)* at most one node as its root (line 11); and *2)* a size (line 12);

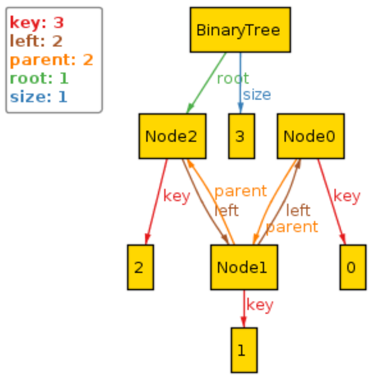

Fig. 2. Binary search tree instance

Once we have declared the elements of our model we specify the relationships between them to reflect the key properties of the modelled data structure. A binary search tree is *Acyclic* (line 15), which is for every node reachable from the root performing zero or more traversals (line 16): *1)* at most one node is visited following the left and right relations in reverse direction (line 17); *2)* a node cannot be reached by following one or more times the left and right relations beginning from that node (line 18); and *3)* the left and right nodes are disjoint (line 19);

The nodes in the binary search tree have a *Parent* property (line 23), which is: *1)* every node reachable from the root performing zero or more traversals is the parent of its left and right children (line 24-25); and *2)* the root has no parent (line 26);

A binary search tree contains data satisfying the *Search* (line 29) property, which is for every node reachable from the root performing zero or more traversals (line 30): *1)* every descendant reached by following the left and right relations of its left child zero or more times has a lesser key (line 31-32); 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 33-34);

A binary search tree has a *Size* property (line 38) which is the cardinality of the nodes reached from its root by performing zero or more traversals of the left and right relations (line 39).

In order for a data structure to be a *BinarySearchTree* (line 42) it has to satisfy the *Acyclic*, *Parent*, *Search*, and *Size* predicates (line 43).

Once we have created a declarative model of a binary search tree we request from the Alloy Analyzer to create one for us by specifying the bounds on the atoms, i.e. elements for each signature, we have defined (line 46). Upon running this command the analyzer tries to find valuations to the relations such that the predicate declaring a binary search tree evaluates to true which is there exists an instance that satisfies the

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

Fig. 3. Annotated alloy model of a binary search tree

declared model. If the analysis finds an instance that satisfies all constraints it is visualized.

In Figure 2 is shown an instance (there may be more that one such) which satisfies all constrains in the model on Figure 1 and more precisely the *BinarySearchTree* predicate. The atoms, i.e. instances of the *Node* and *BinarySearchTree* signatures, are represented as nodes and relations between them as arrows. The top left side shows the cardinality of the relations.

III. Our approach

The Alloy Analyzer performs an exhaustive search within a given scope and not finding a solution only guarantees that no such exists in that scope but one may be found for larger scopes. However, the combinatorial nature of the propositional formula to which an Alloy model is translated limits the scope for which an analysis can be performed within a reasonable amount of time. Hence, increasing analysis speed would enable reaching larger scopes, thus increasing confidence in the obtained results. Also reaching larger scopes is beneficial for tools that utilize Alloy models for test generation [14].

We believe that the analysis performed by the Alloy Analyzer does not exploit an opportunity to take advantage of the problem domain and structure. We propose annotation support in Alloy.

Using annotations the author can embed knowledge in terms of domain and structure into an Alloy model. This information can be used for both partitioning the problem and choosing a solver in each step of the performed incremental analysis.

Annotations are an excellent facility to embed domain and structural knowledge because: *1)* they can be placed on the statements to which they apply; *2)* they do not alter the model and can be optionally ignored by the analysis engine; *3)* they are a popular means of embedding meta-data; and *4)* they can be added incrementally as the domain and problem knowledge grow; In Figure 3 is presented an annotated version of the binary search tree model we have presented in Section II.

We propose one annotation–*@predicate* (line 15)–with two attributes–*priority* and *solver*. The *@predicate* annotation can be applied only at the predicate level and defines how this predicate relates to the entire problem. In particular, the priority attribute is a clue to the analysis engine about the order of solving the predicates in the model. The higher the priority the earlier the predicate is to be solved. For example, the priority of the *Acyclic* (line 16) predicate is three and is higher than the priority of the *Search* (line 32) predicate which is two, therefore the *Acyclic* predicate will be solved first and its solution will be used as a partial solution for reasoning about the *Search* one.

The *solver* attribute specifies which solver to be used for reasoning about the annotated predicate. Potential values for that attribute could be *SAT*, *INTEGER*, *STRING*, etc. *SAT* instructs the analysis engine to use a SAT solver, *INTEGER* implies integer constraint solver, *STRING* request a dedicated string constraint solver, etc.

Having defined the annotation facilities, let us see how they would apply to the binary search tree model on Figure 3. By running the command (line 50) we are requesting from the analysis engine to find an instance which satisfies the *BinaryTree* predicate (line 46). Note that the latter is a conjunction of four predicates, namely *Acyclic* (line 16), *Parent* (line 25), *Search* (line 32), and *Size* (line 42). Examining the annotations on each of these predicates suggests that the order of solving them is: *Acyclic* $\rightarrow$ *Parent* $\rightarrow$ *Search* $\rightarrow$ *Size*. Hence, the model will be analyzed in four steps where the solution of each step will be used as a partial solution for the next one. Further, a specific solver will be used for the analysis of each sequential step.

IV. Related work

This paper proposes annotations for Alloy models. These annotation can be used to guide solving of Alloy constraints using a variety of dedicated constraint solvers. To our knowledge, this is the first paper to present annotations for Alloy.

The most closely related work to this paper is a recent paper [15], co-authored by the second author. It introduces *mixed constraints*, which are written using a combination of a declarative language, namely Alloy, and an imperative language, namely Java. Further, it supports annotating *def-use* sets of variables to factilitate 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. A key design goal of mixed constraints is to facilitate writing of constraints using a combination of declarative and imperative programming paradigms and solving them, whereas this paper proposes an approach for solving of models written purely in Alloy and hence does not require the user to learn a new notation to benefit from efficient solving.

Incremental solving for Alloy models, where a solution to one formula is fed as a partial solution to efficiently solve another formula, was introduced in Uzuncaova's doctoral work [13], [16], [17], which also applied it in the context of test input generation for product lines. Their work did not support annotations, rather used a heuristic def-use analysis to prioritize constraints written in Alloy. For product lines, the structure of a product line was leveraged for incremental solving.

An example of extending the Alloy syntax to describe dynamic properties of systems via actions is presented in [18]. The actions enable specifying dynamic properties of execution traces as dynamic logic specifications. Our technique is similar to this work with respect to extending the Alloy syntax with new semantic features. While this work focuses on adding constructs for specifying dynamic behavior, our approach focuses on embedding meta-data in a standard Alloy model.

An approach of scope-bounded checking is explored in [19]. The key idea is to reason about a program correctness by dividing the problem into sub-problems for different code paths. Each sub-problem is analyzed via the Alloy Analyzer. We, on the other hand, propose an annotation facility that enables incremental analysis into the Alloy Analyzer and use multiple domain specific solvers.

V. Conclusion

We have proposed annotations for Alloy to embed meta-data that can be used for specifying the order of analyzing the constraints in a model and specifying a domain specific solver to be used for a given constraint enabling more efficient incremental analysis.

Annotations are a commonly used facility in imperative languages and adopting such in a declarative language, namely Alloy, would not impose a steep learning curve to model authors while giving them finer grained control over the analysis process.

VI. Acknowledgements

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

References

[1] K. Robert W., "The impact of moore's law," vol. 20-3, pp. 25–27, 2006.

[2] D. L. Dvorak, "Nasa study on flight software complexity," in *Technical Report, NASA Office of Chief Engineer Technical Excellence Program*, 2009.

[3] P. Gluck and G. Holzmann, "Using spin model checking for flight software verification," in *Aerospace Conference Proceedings, 2002. IEEE*, vol. 1, 2002, pp. 1–105 – 1–113 vol.1.

[4] M. Popovic, V. Kovacevic, and I. Velikic, "A formal software verification concept based on automated theorem proving and reverse engineering," in *Engineering of Computer-Based Systems, 2002. Proceedings. Ninth Annual IEEE International Conference and Workshop on the*, 2002, pp. 59 –66.

[5] D. Lettnin, M. Winterholer, A. Braun, J. Gerlach, J. Ruf, T. Kropf, and W. Rosenstiel, "Coverage driven verification applied to embedded software," in *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, 2007, pp. 159 –164.

[6] T. T. Chen and W. H. Hsieh, "Uncovering the main research themes of software validation," in *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, 2010, pp. 1 –6.

[7] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 7, pp. 1165 –1178, 2008.

[8] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid, "Korat: A tool for generating structurally complex test inputs," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, May 2007, pp. 771 –774.

[9] L. Clarke, "A system to generate test data and symbolically execute programs," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 3, pp. 215 – 222, 1976.

[10] (2011, Mar.) Alloy analyzer 4. [Online]. Available: http://alloy.mit.edu/alloy4/

[11] A. Christensen, A. Mller, and M. Schwartzbach, "Precise analysis of string expressions," in *Static Analysis*, ser. Lecture Notes in Computer Science, R. Cousot, Ed. Springer Berlin / Heidelberg, 2003, vol. 2694, pp. 1076–1076.

[12] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 105–116.

[13] E. Uzuncaova, "Ecient specication-based testing using incremental techniques," in *Dissertation*, 2008.

[14] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," *Automated Software Engineering, International Conference on*, vol. 0, p. 22, 2001.

[15] S. A. Khalek, V. Priyadarshini, and S. Khurshid, "Mixed constraints for test input generation," (Submitted for publication at ASE 2011).

[16] E. Uzuncaova and S. Khurshid, "Constraint prioritization for efficient analysis of declarative models," in *FM*, 2008, pp. 310–325.

[17] E. Uzuncaova, S. Khurshid, and D. S. Batory, "Incremental test generation for software product lines," *IEEE Trans. Software Eng.*, vol. 36, no. 3, pp. 309–322, 2010.

[18] M. F. Frias, J. P. Galeotti, C. G. López Pombo, and N. M. Aguirre, "Dynalloy: upgrading alloy with actions," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 442–451. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062535

[19] D. Shao, S. Khurshid, and D. Perry, "An incremental approach to scope-bounded checking using a lightweight formal method," in *FM 2009: Formal Methods*, ser. Lecture Notes in Computer Science, A. Cavalcanti and D. Dams, Eds. Springer Berlin / Heidelberg, 2009, vol. 5850, pp. 757–772.