Exploiting Subformula Sharing in Automatic Analysis of Quantified Formulas

Ilya Shlyakhter[1], Manu Sridharan[2], Robert Seater[1], and Daniel Jackson[1]

[1] Laboratory for Computer Science
Massachusetts Institute of Technology
{ilya_shl,dnj,rseater}@mit.edu
[2] Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
manu_s@cs.berkeley.edu

http://ilya.cc/sharing

Abstract. Recent advances in SAT solvers have made it attractive to translate a variety of problems to SAT. In many cases, the source language is a logic that includes quantifiers. Grounding out these quantifiers can be a bottleneck. One way to mitigate the problem is to join isomorphic subtrees of the ground formula, so that the syntax tree becomes a DAG. Because the ground tree is large, detecting isomorphism after grounding out is impractical. However, detecting isomorphism prior to grounding out will miss many isomorphisms in the ground form.

A method is presented that detects potential isomorphism in advance of grounding out, but which identifies the isomorphisms of the ground form. It partitions nodes of the quantified formula into classes whose expansion may yield identical ground subformulas. This allows isomorphisms to be discovered during grounding out that are not evident in the original formula.

Experience using this technique in the context of a model checker for relational specifications is presented. It gives significant reductions of both analysis time and memory usage, and has enabled analyses that were previously intractable. We speculate that the intermediate information generated by the technique can also aid QBF solvers.

1 Introduction

Quantified formulas – statements such as $\forall x P(x)$ – are frequently used in formal specifications. They allow concise and natural formalization of system properties and, for this reason, are present in many constraint languages. Languages that permit some form of quantifiers include first-order logic, Alloy [7], and Murphi [2]. The recently developed Bounded Model Checking techniques express Linear Temporal Logic formulas as quantified formulas [1].

Constraints with quantifiers can be analyzed in one of two ways: They can be converted to a Quantified Boolean Formula and solved using a QBF solver [5], or the quantifiers can be ground out and the resulting ground form converted to CNF and solved with a SAT solver [6,9]. Since the ground form can be much larger than the original quantified constraints, grounding out may not be practical in some cases. For the cases for which it is practical, grounding out and applying a SAT solver usually takes less time than converting to QBF and using a QBF solver [4].

In this paper we present a technique that extends the range of problems for which the "ground out and convert to CNF" approach is practical. The technique speeds up grounding out, and results in smaller CNFs that are solved faster. The resulting CNFs encode subformula sharing information not otherwise available to the SAT solver. The intermediate information we compute about the unground constraints may be of use to QBF solvers, and the speedup seen in CNF solvers suggests there might be similar benefits to QBF solvers.

The technique takes advantage of the large numbers of identical subformulas often present in ground constraints. Representing the ground form as a DAG allows identical subformulas to be shared. However, since the ground constraints are not explicitly represented in the original (quantified) constraints, identifying opportunities for sharing is nontrivial. Once a ground form is obtained, we could identify identical subtrees, but doing so would require first obtaining the (unshared) ground form, which can be infeasible. On the other hand, identifying isomorphisms in the unground form (and then grounding out) will miss many opportunities for sharing in the ground form. In this paper, we describe a technique for directly producing a DAG in which sharing is already present. We identify the structural isomorphisms of the ground form, but perform our analysis on the unground form.

The remainder of the paper is organized as follows. Section 2 presents an abstract constraint syntax (Subsection 2.1), describes how grounding out is performed and how sharing information can be used (Subsection 2.2), introduces the notion of a template and describes how templates can be used to detect sharing (Subsection 2.3), and elaborates on how templates are detected (Subsection 2.4). Section 3 gives empirical measurements of improvements obtained by detecting sharing, including an example of a previously intractable problem which our technique makes analyzable. Section 4 concludes the paper and indicates directions of future work.

2 Detecting and Using Sharing

2.1 Abstract Constraint Syntax

Rather than using a specific constraint language, we define an abstract syntax that serves as a schema for constraint languages with quantifiers. The only restriction we place on the constraint languages is that quantifiers range over finite domains (so that grounding-out is possible). This abstract syntax separates our techniques from the semantics and properties of any particular language.

We define an abstract constraint syntax for expressing predicates on a collection of variables. A predicate is expressed as an abstract syntax tree (AST),

```
U: all values      V: variables         Q: quantified variables
N: nodes           F: node functions    M: templates

is_var(n: N): Bool     // is n a leaf AST node representing a variable?
node_var(n: N): V      // if isvar(n), return the variable at n
node_func(n: N):F      // what function of its children does n compute?
node_chldrn(n:N): N*   // return the children of n
node_templ(n:N): M     // the template matched by node n
node_args(n:N): N*     // the argument list with which n matches node_templ(n)
is_quant(n:N): Bool    // is n a quantifier node?
quant_var(n:N): Q      // the quantified variable declared at n
quant_body(n:N): N     // the sole child node of a quantifier node
quant_range(n:N): U*   // the range of the quantified variable of n
```

Fig. 1. Definition of notation.

where each inner node computes a predefined function of its children (the root computing a Boolean which becomes the value of the predicate). Leaf nodes of the tree include the variables the predicate is constraining, quantified variables, and constants. Inner nodes include quantifier nodes. A quantifier node has one child, and defines a range of values for a free quantified variable in its subtree. The quantifier node's value is computed by applying its node function to the result of evaluating the quantifier body, as the quantified variable runs over the range. This abstraction can express the standard quantifiers, $\forall$ and $\exists$, but is general enough to express quantified constructs such as set comprehension and integer summation.

This abstract constraint syntax can be instantiated as a number of different languages. For example, to represent first-order logic, the node functions would include `AND`, `OR`, and `NOT`. Set theory can be represented by including as node functions the standard set operators (UNION, INTERSECT, etc). Our work was motivated by the Alloy language [7], but applies to other constraint languages with finite-domain quantifiers.

Figure 1 gives the formal notation used in this paper. Figure 2 shows an unground AST expressing a constraint on a single variable v_1, and the ground form of the constraint.

2.2 Grounding Out

An AST can be converted into a quantifier-free (ground) form by *grounding out* the quantifier nodes.

```
groundout(n: N, qvarvals: Q->U): N {
  return new_node(node_func(n), !is_quant(n) ?
    map(lambda c . groundout(c,qvarvals), node_chldrn(n)) :
    map(lambda u . groundout(quant_body(n),
        qvarvals[quant_var(n)->u]), quant_range(n))) }
```

qvarvals gives the values of free quantified variables used in n's subtree. qvarvals[quant_var(n)->u] is qvarvals with the quantified variable defined at node n set to value u. new_node constructs a new ground node with the specified node function and children.

We would like an "oracle" that keeps track of the ground forms already produced, and tells whether a particular invocation of groundout will generate an already-produced ground form. The difficulty lies in determining whether the ground form *about to be* generated matches an existing ground form. In the following sections, we describe how to construct such an oracle by adding *template annotations* to the unground tree.

2.3 Using Templates to Detect Sharing

Here we describe the format of the template information, and how it is used during grounding-out. Later, in Section 2.4, we describe how templates are detected.

Before grounding out, we compute a *template annotation* for each node of the AST. In effect, we represent every node as an instantiation of a parameterized template. During grounding out, for each template we keep track of ground forms of all nodes that match the template. When we visit a node, we look in its template's cache of ground forms to see whether the ground form we're about to generate is already available.

More specifically, the template annotation of a node n comprises a template name node_templ(n) and a list of template arguments node_args(n). Each template argument is a constant-valued node in the subtree rooted at n. (A node is constant-valued if its subtree contains no non-quantified variables. The ground form of a constant-valued node simplifies to a single value.)

The template information lets us quickly determine whether two given invocations of groundout will produce the same ground form. Formally, template information satisfies the following *template invariant*:

```
node_templ(n1) = node_templ(n2)
 && argsMatch(node_args(n1),node_args(n2),A1,A2)
=> groundout(n1,A1)=groundout(n2,A2)

argsMatch(args1, args2: N*, A1, A2: Q->U): Bool
   forall(lambda a1 a2 . eval(a1,A1)=eval(a2,A2), args1, args2)

eval(n: N, a: Q -> U): U { let f=node_func(n) in if(!is_quant(n))
  then { f(map(lambda c . eval(c,a), node_chldrn(n))) }
  else { f(map(lambda u . eval(c,a[quant_var(n)->u]), quant_range(n))) }
```

During grounding out, for each template we keep a cache of ground forms keyed on the value of the template argument list. When groundout is called to produce the ground form of node n under the quantified variable settings qvarvals, we evaluate the template arguments of n under qvarvals, and use

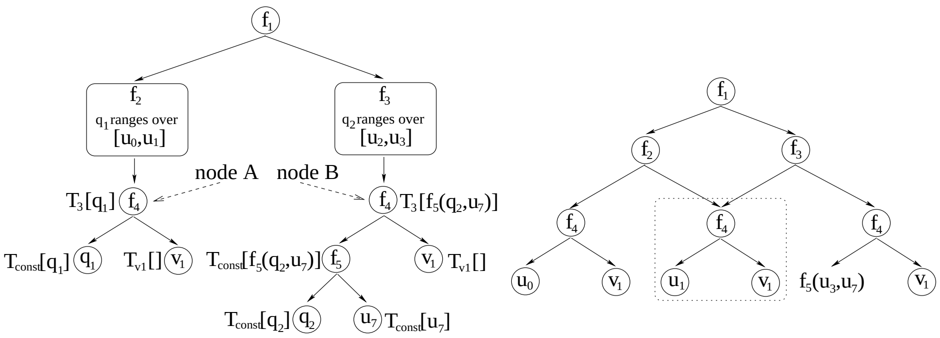

Fig. 2. Using templates to effect sharing during grounding-out. The DAG on the right is the grounding-out of the AST on the left. Rounded rectangles indicate quantifier nodes. Nodes A and B match the same template T_3. During grounding-out, node A for $q_1 = u_1$ has the same ground form (dotted rectangle) as node B for $q_2 = u_2$, if $f_5(u_2, u_7) = u_1$.

the resulting list of values as a key into the cache of ground forms kept for `n`'s template.

The use of templates to produce sharing of common subformulas is illustrated in Figure 2. Nodes `A` and `B` match the same template T_3. During grounding-out we maintain a cache for T_3, mapping argument list values to ground forms. Initially the map is empty. When `groundout` visits node `A` with `q1=u1`, it computes the key into T_3's cache to be `[u1]` (evaluating node `A`'s template argument list `[q1]`). This gives a cache miss; `groundout` computes a ground form (dotted rectangle in Figure 2) and stores it in T_3's cache with the key of `[u1]`. When `groundout` subsequently visits node `B` with `q2=u2`, it computes the key into T_3's cache as `[f5(u2,u7)]`. Suppose `f5(u2,u7)=u1`. Then `groundout` will get a cache hit, retrieve the previously computed ground form from T_3's cache and return it immediately. The cached ground form will therefore be shared among two nodes, as shown in the rightmost DAG in Figure 2.

2.4 Detecting Templates

We describe the template detection algorithm and illustrate it on the running example in Figure 2. Full details are available in a technical report [10].

Template detection is done by a single depth-first traversal of the unground AST. For each node, we determine the template name and template arguments satisfying the template invariant defined in section 2.3. When we visit a node, we first recursively determine the template information for the node's children, then use that information to determine the correct template annotation for the node itself.

First, consider two base cases. All leaf nodes referencing a given non-quantified variable match the same template, with no arguments. Any two such nodes have the same ground form, so the template invariant is trivially satisfied. In the running example, the two v_1 nodes both match the template T_{v1}, with empty argument list [ ].

Another base case involves constant-valued nodes. A node is constant-valued if its subtree references no non-quantified variables; all leaves are either constants or quantified variables. All such nodes match a single template, T_{const}, with the node itself as the sole argument. Since the ground form of a constant-valued node simplifies to a single value from `U`, the template invariant is trivially satisfied[3]. In the running example, the constant-valued nodes q_1 and $f_5(q_2, u_7)$ match T_{const} with argument lists $[q_1]$ and $[f_5(q_2, u_7)]$ respectively. For any quantified variable setting A_1 that sets q_1 and A_2 that sets q_2, the template invariant asserts that whenever q_1 under A_1 evaluates to the same ground form (i.e. to the same element of U) as $f_5(q_2, u_7)$ under A_2, the two constant-valued nodes have the same ground form.

Now we consider template detection for a non-leaf, non-constant node that does not define a quantified variable. There are three such nodes in the running example: node `A`, node `B` and the root; here we will focus on the first two. We need to determine if the node we're visiting matches a previously seen template, or a new template. (Recall that we're traversing the AST in depth-first order; for the running example, assume that we always visit the left subtree before visiting the right.)

Scanning all existing templates would make the algorithm quadratic in AST size. A simple optimization leads to linear running time in practice. We first compute a (much smaller) subset of *candidate templates*, containing all previously seen templates that the node could match. For this purpose we keep a set of *parent templates* for each template. `m'` is a parent of `m` if there exist nodes `n` and `n'` such that `n` matches `m`, `n'` matches `m'` and `n'` is the parent of `n`. When we have determined the template matched by a node, we add that template as a parent template to each child's template.

In order for `n` to match the template `t'` already matched by some (previously visited) node `n'`, the corresponding children of `n` and `n'` must have matching templates. Thus, if a template `t` is not a parent template of the template of each child of `n`, then `n` cannot match `t`. We therefore let the candidate template set be the intersection of the template parent sets of the templates matched by the children of `n`.

In our example, consider our visit to node `A`. At that point neither T_1 (matched by `A`'s first child) nor T_{v1} (matched by `A`'s second child) have any parent templates. Consequently the set of candidate templates for `A` is empty, and `A` matches a previously unseen template; we name the new template T_3. We record T_3 as a parent template of both T_1 and T_{v1}. Later we visit node `B`, its left child matches T_1 and its right child matches T_{v1}. At that point the candidate template set is computed as the intersection of $\{T_3\}$ and $\{T_3\}$, yielding the sole

[3] In our actual implementation the AST nodes have types, and there is one template for constant-valued nodes of each type. This prevents AST nodes that will never ground out to the same ground form from matching a common template.

template T_3; if node B matches a previously seen template, that template must be T_3.

Now that we have narrowed down the templates we need to consider, we examine each in turn to see if it is matched by the node n we're visiting. From each candidate template, we take a representative node n' which we visited earlier and which matches the candidate template. We perform three tests to see whether n and n' match the same template: whether they have the same node functions, same numbers of children, and whether their corresponding children match the same templates. If the three tests pass, we declare that n matches n''s template. Regardless of whether n matches a previously seen template or a new template, the template argument list is obtained by concatenating the template argument lists of n's children.

For example, when we visit node A, it matches a previously unseen template; we compute the template argument list by concatenating $[q_1]$ with $[\,]$ to obtain $[q_1]$. When we subsequently visit node B, we need to test whether it matches the previously seen template T_3. We take a previously seen node that matches T_3, node A. We observe that nodes A and B compute the same node function (f_4), have the same number of children (2), the left children of both match T_1, and the right children match T_{v1}. We therefore determine that node B matches template T_3, with argument list $[f_5(q_2, u_7)]$.

We will now show that the template annotations computed as described above satisfy the template invariant. Suppose nodes n1 and n2 both match template t and satisfy the three tests; and quantified variable assignments A1,A2: Q->U meet the condition

```
argsMatch(node_args(n1),node_args(n2),A1,A2)
```

Since the template arguments of n1 and n2 were obtained by concatenating the template arguments of their children, we have

```
forall(lambda c1 c2 . argsMatch(node_args(c1),node_args(c2),A1,A2),
       node_chldrn(n1), node_chldrn(n2))
```

Assuming child template information is correct, the corresponding children of n1 and n2 ground out to the same ground forms (under A1 and A2 respectively). Since n1 and n2 combine their children using the same functions, the ground forms of n1 and n2 are the same.

Computing template arguments for a quantifier node has an added complication; template arguments for the child may include the quantified variable introduced at the node, however, the template arguments for this node itself cannot include that variable. This impacts how we compute the template arguments for the node; we cannot simply take the template arguments of the child (as we would do for a one-child non-quantifier node). The full algorithm for template detection, including the handling of quantifier nodes, is described in [10].

3 Results

We present preliminary performance results for a benchmark suite of six Alloy [7] models.

dijkstra: a model of Dijkstra's algorithm for mutex ordering to prevent deadlocks. We check that the algorithm works correctly for 10 processes and 10 mutexes, for traces up to length 10.

stable_mutex_ring: a model of Dijkstra's self-stabilizing K-state mutual exclusion algorithm for rings [3]. We run a function which finds a non-repeating trace of the system with 3 nodes and 17 steps.

ins: a model of an intentional naming system [8]. We check a structural correctness condition for 4 nodes and 2 name records.

chord: a partial model of the Chord distributed hashtable lookup algorithm for rings [11]. We check a structural correctness condition for 3 nodes and 5 Chord identifiers.

shakehands: a model of a logic puzzle by Paul Halmos involving handshakes between pairs of people. We run a function which solves the puzzle for 10 people.

life: a model of Conway's Game of Life. We run a function which finds an execution of 3 time steps on a 12 point grid.

This suite reflects a variety of modelling idioms, including the BMC-style [1] models which motivated this work. It also balances checking conditions that are satisfiable (stable_mutex_ring, shakehands, and life) with those that are not (dijkstra, ins, chord). The benchmarks were run on a Pentium III 1GHz laptop with 256MB of RAM running Windows 2000.

model	num vars sharing	num vars no sharing	num clauses sharing	num clauses no sharing
dijkstra	40631	55463	95948	123758
stable_mutex_ring	16309	19897	38440	50237
ins	22742	timeout	110564	timeout
chord	22856	43102	58106	104117
shakehands	7706	16575	20539	54673
life	37322	92464	161289	383541

Table 1. Formula sizes for benchmarks with and without sharing detection.

Table 1 shows the effects of our sharing detection algorithm on the size of the generated CNF formula. We measure both the number of variables and the number of clauses. Sharing detection consistently reduces the size of the generated CNF by a large amount, more than a factor of 2 in some cases. For the *ins* model, no CNF was generated without sharing because the grounding

	ground-out	ground-out	mchaff	mchaff	bermkin	berkmin
model	sharing	no sharing	sharing	no sharing	sharing	no sharing
dijkstra	6.02	7.82	2.74	9.70	19.87	67.98
stable_mutex_ring	0.91	1.25	18.77	32.65	6.40	24.20
ins	4.01	timeout	2.17	timeout	2.58	timeout
chord	1.01	3.74	55.03	93.04	21.87	48.07
shakehands	0.54	0.89	295.58	timeout	2.14	57.20
life	4.87	13.17	2.04	44.11	3.28	20.67

Table 2. Runtimes for benchmarks with and without sharing detection. All times are in seconds.

out phase ran out of memory, illustrating how sharing detection has made some previously intractable models analyzable.

Runtime comparisons for our benchmarks are given in Table 2 (all times are in seconds). We present times for grounding out and solving with two different modern SAT solvers, mchaff [9] and BerkMin [6]. The "no sharing" columns give runtimes with sharing detection disabled. We see consistent and often dramatic improvements with sharing detection enabled for both grounding out and solving. The improvements are seen for both SAT solvers, indicating that the better performance with sharing is independent of differing solver techniques. The *ins* model is particularly interesting, as it is easily analyzable with sharing detection and intractable without. For the *shakehands* model with sharing detection disabled, mchaff was unable to find a solution after 15 minutes of runtime. We plan to implement more optimizations for the sharing detection in the near future, including handling of commutative operators, and we expect to have more results like the *ins* model, where sharing detection makes the difference in tractability.

Two possible factors contribute to the performance improvements. First, the CNF encodings of formulas with shared subtrees is more compact; only one batch of Boolean variables and clauses is needed to encode the shared subtree. As a result, SAT solver operations such as unit propagation execute faster. Second, the subformula sharing information implicitly encoded in the smaller CNF may prevent the solver from performing redundant computations. Understanding the relative importance of these factors will be one direction of future work.

4 Conclusion

We have described a new algorithm for exploiting structural redundancy in quantified formulas during grounding out. The algorithm reduces running time and memory usage of the groundout procedure, and produces easier-to-solve CNFs. The technique does not depend on the details of the constraint language, and applies to languages that include non-standard quantification constructs.

Results on a variety of software models suggest that the approach is practical. It never worsens performance; often it produces a significant improvement, and in one documented case it made a previously intractable model tractable.

The template annotations produced for nodes of the source tree have simple semantics; they implicitly encode information about the ground form. QBF solvers similarly attempt to derive information about the ground form, without explicitly grounding out. It would be interesting to see whether QBF solvers can use the sharing information to achieve the speedups seen with CNF solvers.

References

1. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Design Automation Conference*, 1999.
2. A. J. H. David L. Dill, Andreas J. Drexler and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.
3. E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
4. E. Giunchiglia, May 2002. Personal communication.
5. E. Giunchiglia, M. Narizzano, and A. Tacchella. Qube: A system for deciding quantified boolean formula satisfiability. In *In Proc. International Joint Conference on Automated Reasoning (IJCAR - 01)*, 2001.
6. E. Goldberg and Y. Novikov. Berkmin: a fast and robust SAT-solver. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, March 2002.
7. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, September 2001.
8. S. Khurshid and D. Jackson. 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)*, September 2000.
9. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference (DAC)*, June 2001.
10. I. Shlyakhter, M. Sridharan, R. Seater, and D. Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. http://ilya.cc/transl.ps, 2003.
11. I. Stoica, R. Morris, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001*, August 2001.