Faster Constraint Solving with Subtypes

Jonathan Edwards, Daniel Jackson,
Emina Torlak, Vincent Yeung

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139
{jedwards, dnj, emina, vshyeung}@mit.edu

ABSTRACT

Constraints in predicate or relational logic can be translated into boolean logic and solved with a SAT solver. For faster solving, it is common to exploit the typing of predicates or relations, in order to reduce the number of boolean variables needed to encode the constraint. Here we show how to extend this idea to constraints expressed in a language with subtyping. Our technique, called *atomization*, refactors the type hierarchy into a flat collection of disjoint atomic types. The constraints are then decomposed into equivalent constraints involving smaller relations or predicates over these new types, which can then be solved in the normal fashion. Experiments with an implementation of this technique within the Alloy Analyzer show improved performance on practical software checking problems.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Constraints;*
D.2.1 [**Software Engineering**]: Requirements/Specifications – *Languages, Tools*.

General Terms

Design, Languages, Verification.

Keywords

Relational logic, SAT, constraints, subtypes, verification, analysis.

1. INTRODUCTION

Constraints involving predicates or relations arise in many domains. An increasingly popular way to solve such a constraint, assuming the universe of atoms is finite, is to translate it to a boolean formula, and use an off-the-shelf SAT solver to find a solution. This strategy is used in planning [7] [29], in analyzing structural models of software [14], and in temporal logic checking of hardware [4].

The key insight underlying all these approaches is that a binary relation or two-place predicate can be represented as an adjacency matrix, with a one in the *i*-th row and *j*-th column when the *i*-th element of the universe is related to the *j*-th. To find the value of a collection of relations or predicates satisfying a logical constraint, these matrices are filled with boolean variables rather than constants, and the constraint is translated into a boolean formula in these variables. For a universe of k atoms, each such relation or predicate has k^2 boolean variables, each of which potentially doubles the solving time (although in practice, of course, SAT solvers rarely exhibit worst case behavior).

A dramatic reduction in the number of variables, and a concomitant improvement in performance, can be obtained by exploiting some rudimentary type information. The universe is partitioned into a collection of types, and each relation or predicate is assigned a type for each column. For a relation or predicate of type $\langle S, T \rangle$, $s \times t$ variables now suffice, where s and t are the number of atoms in the types S and T respectively. Many tools implement this optimization, and it has been refined and extended in various ways [13] [3].

In some domains, the universe of atoms can be given yet more structure. Rather than a simple partition of types, a hierarchy of subtypes can be constructed. The typing of each relation or predicate can thus convey more information about its value. For example, a model of a file system which, in a simple type system, distinguishes only file system objects from their names, may now classify objects into files and directories. The *contents* relation mapping a directory to its contents will now have the type $\langle Directory, Object \rangle$ rather than $\langle Object, Object \rangle$.

This paper presents a strategy for exploiting this subtyping. There are many unattractive candidates for such a strategy. One that we implemented in a prior version of our tool, for example, involves treating all relations as untyped, and using the type information to "zero out" entries determined to be outside the type of the relation. This wastes space during translation, and cannot accommodate union types.

The solution presented here involves a source-to-source rewriting called *atomization*. The subtype hierarchy is transformed into a flat partitioning of *atomic* types. Each relation is decomposed according to these atomic types, inducing a refactoring of formulas according to the types of the relations they mention. The result is an equivalent constraint which is correctly typed with respect to the atomic types. Having thus eliminated subtypes from the formulation, the conventional translation to a boolean formula can proceed. The values of the decomposed relations are determined by solving the refactored constraints, and the values of the original relations are reconstituted from them.

ISSTA'04, July 11-14, 2004, Boston, Massachusetts, USA.

Alice and Bob invite four other married couples to a party (for a total of 10 people). During the party some people shake hands. No one shakes hands with themselves or their spouse. At the end of the party Alice asks everyone how many hands they have shaken, and each one gives a true but different answer. How many hands did Bob shake?

```
abstract sig Person {shakes: set Person}
sig Man extends Person {wife: Woman}
sig Woman extends Person {husband: Man}
fact {
   husband = ~wife
   no ((husband + wife) & shakes)
   no p : Person | p in p.shakes
   shakes = ~shakes
}
assert AnswersDifferent {
   some Alice: Woman | {
      all disj p1, p2: Person-Alice |
         #p1.shakes != #p2.shakes
   }
}
check AnswersDifferent for
     exactly 5 Man, exactly 5 Woman
```

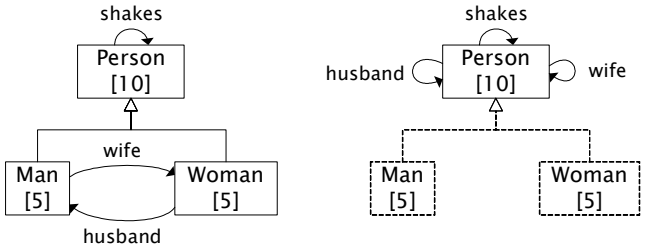
Figure 1. Example

This approach is simple and easy to implement. It has the great advantage that it preserves the straightforward translation between relational and boolean forms. In our case, it meant that we could continue to use our translation backend without modification. The rewriting can also exploit some simple algebraic rules, for example that the composition of two relations is empty when the range of the first and domain of the second are disjoint. This allows a further improvement in performance, by eliminating gratuitous clauses from the boolean form.

In addition to offering better performance, subtypes are more expressive than simple flat types, and thus allow more errors to be detected by type checking, prior to analysis. Union types complement subtypes nicely, offering the flexibility of introducing new relations without refactoring the type hierarchy. Fortuitously, they are handled easily by our scheme without any elaboration.

The contributions of the paper are:

- An identification of the opportunity offered by subtypes for performance improvement;
- A simple and effective algorithm that can be implemented within existing tools without disruption of interfaces;
- Some results of applying the algorithm, as embodied in a full-scale implementation, to a variety of software design analysis problems;
- A correctness argument for the algorithm, based on the manner in which the rewriting preserves semantics; and
- A discussion of how these ideas might be applied in other contexts, such as temporal logic checking and planning.



a) subtypes b) lifting

Figure 2. Subtypes vs lifting

2. OVERVIEW

Atomization will be illustrated with an example problem, due to Halmos [10]. A model to solve this problem, written in Alloy, our modeling language, is included in Figure 1, along with an informal statement of the problem.

At first sight, it is surprising that this problem is sufficiently constrained to have a unique solution. The fact that such simple constraints imposed on such a large space have only a single solution makes the problem more challenging for an analyzer than its size suggests.

2.1 Exploiting Typing

The challenge posed by subtyping is illustrated, for our example, in Figure 2. Both diagrams are UML-like object models, in which boxes represent declared sets and arrows represent relations. The diagram on the left hand side (2a) shows the relations with their declared types; note how the domain and range of each relation captures the conventional gender constraints (eg, that `wife` maps `Man` to `Woman`). In contrast, the diagram on the right (2b) shows the relations with their types as implied by a conventional analysis, with all the relations lifted over a single type, necessitating explicit constraints to capture the domain and range information.

The question this paper addresses is how to exploit the natural typing (of Figure 2a) in the boolean encoding of the relational model. The difficulty of solving a boolean constraint is strongly correlated to the number of boolean variables it mentions, and less strongly to its size (the number of clauses, assuming conjunctive normal form). Each additional boolean variable doubles the state space. Although a combination of carefully tuned heuristics and learning mechanisms allow the worst case often to be avoided, SAT solvers are still very sensitive to the number of variables. The scheme we describe can reduce this number by a third or more.

The naïve approach is to ignore types completely, treating all sets and relations as ranging over the entire universe. This is depicted in Figure 2b, in which the dotted lines indicate that the former subtypes are now just treated as subsets, and not as true types. The problem with this approach is that it causes many more boolean variables and clauses to be created than are necessary. For example, in a universe of 5 men and 5 women, the `wife` relation now requires a 10×10 matrix of 100 boolean variables, even though we know from its type that a 5×5 matrix of 25 variables would suffice, corresponding to that part of the larger matrix that maps men to women. Constraints that were implicit in the subtype

hierarchy, for example that `Man` and `Woman` are disjoint, must now be made explicit, and added to the model during translation, further increasing the number of clauses.

A more sophisticated approach, commonly employed, is to exploit the top-level types alone, and ignore the subtypes. All relations on subtypes are lifted to top-level types. In this case, this strategy actually adds nothing, since there is only one top-level type anyway. Suppose, however, we were to have an additional relation `name` that maps a person to his or her name, along with a constraint, say, that no two persons have the same name.

Now if we pick a universe of 5 men, 5 women and 5 names, the relations `wife`, `husband` and `shakes` are typed as `Person` to `Person`, with 10 × 10 matrices of 100 variables each, but the relation name, typed as `Person` to `Name`, will have a 10 × 5 matrix of only 50 variables. Without the distinction between the top-level types, the universe would have 15 atoms, and each relation would have a 15 × 15 matrix of 225 variables. The total number of boolean variables is thus reduced from 900 to 350. This is the approach taken by most tools, including, until now, the Alloy Analyzer [12].

Suppose we were able to exploit the subtyping. Then, `Man` and `Woman`, each with only 5 atoms, would form the domains and ranges of the relations `wife` and `husband`, reducing their representations to 5 × 5 matrices of 25 variables each, eliminating a further 150 variables. Accomplishing this is not trivial, as we shall see.

2.2 Translating with Sparse Matrices

To understand the nature of this problem, it is necessary to appreciate how the translation to SAT is performed. Each relational expression (that is, in the first-order relational logic) is translated into a matrix of boolean formulas. Each relational formula is translated into a boolean formula.

The translation is compositional. To translate the union of two relational expressions, for example, we compute a matrix for each subexpression, and then form a new matrix whose entries are the element-wise disjunction of the matching elements in the two subexpression matrices. The translation rules are easy to justify from the semantics of the relational operators. In this case, for example, the union of two relations relates the *i*-th element to the *j*-th element if either relation does. The intersection relates the *i*-th element to the *j*-th element if both relations do, and therefore conjoins the boolean formulas.

This scheme is explained in detail elsewhere [12]. For our purposes here, it suffices to note the key role that the column and row indices play. If the two matrices being composed have incompatible dimensions, they cannot be combined in this simple manner.

A simple and relatively effective solution to this dilemma is to make the dimensions of all matrices correspond to the universe of atoms; in this case, making all matrices 10 × 10. The subtyping information is then used to set the elements in certain regions of a matrix to zero. For example, if we assign the first 5 atoms of the universe to `Man`, and the second 5 to `Woman`, we can zero out the upper right hand corner of the `husband` relation, consisting of those entries in rows 0 to 4 and columns 5 to 9.

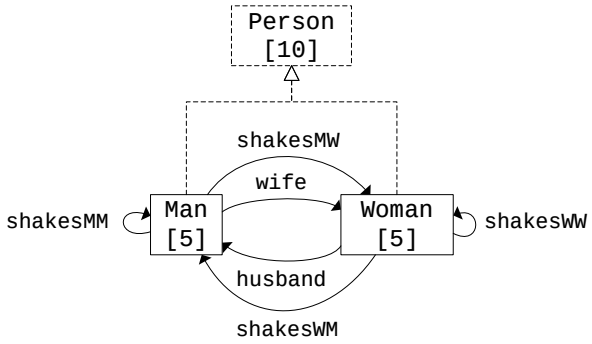


Figure 3. Atomization

This scheme was implemented as part of the Alloy Analyzer by Manu Sridharan in 2002. It works well on small problems, but it has a major drawback that prevents it from scaling. The zeros waste space, to an extent that grows quadratically with the scope and exponentially with the arity of the relations. A relation of arity 4, for example (which is not uncommon), with 5 elements in its domain and range, over a total scope of 20, will require for its representation 160,000 entries, of which only 625 are used.

To overcome this drawback, we began to devise a sparse matrix representation, in which the zero entries would not appear explicitly. This would eliminate the space problem, but we decided to abandon it because of the complexity of manipulating multidimensional sparse matrices. The scheme described in this paper achieves the same reduction in space, but requires no change to the matrix representation at all, and, because it is accomplished by an additional source-to-source translation step in the relational phase, requires no change to the translation itself whatsoever.

2.3 Atomizing Relations

Atomization is a scheme that allows subtypes to be exploited. Rather than lifting relations to supertypes, with the penalty of inflating their size, atomization pushes relations down to the lowest subtypes, minimizing their size. The relations of the constraint are broken down into subrelations according to the smallest subtypes; the constraint is rewritten in terms of these subrelations, obtaining a new constraint whose form is not fundamentally different from the original constraint, but which involves a different set of variables. These variables have domain and range types that can be regarded as 'top-level' types for the analysis of the new constraint, despite the fact that they originated from the bottom of the type hierarchy!

These new relations are shown in Figure 3. The relation `shakes` is *atomized* into four subrelations: `shakesMM`, `shakesMW`, `shakesWM`, and `shakesWW`. Each of these covers only the case of one specific gender shaking hands with one other, indicated by the `M/W` suffixes. The type `Person` is no longer necessary, leaving `Man` and `Woman` alone partitioning the universe, as if they were top-level types.

In a universe of 5 men and 5 women, each of the four subrelations of `shakes` now becomes a matrix of 25 variables. Together, these use no less space than the original shakes matrix. The value of this transformation is that the type constraints on `wife` and `husband` can now be exploited, by assigning only 25 variables to each. In total, the 6 relations (4 `shakes` subrelations, `wife` and `husband`)

require 150 boolean variables, in contrast to the 300 variables that would be required by the naïve typing.

It is important to note that the transformation of the constraint into the atomized form is not seen by the user. When a solution is found, giving values to the atomized relations, the values of the original relations are reassembled from them, and displayed to the user as if no transformation had ever occurred.

2.4 Atomizing Constraints

The subrelations that result from atomizing a relation decompose it; the original relation is simply the union of the subrelations. So the constraint as a whole may be transformed by replacing each occurrence of a relation name with a union expression. This preserves the semantics, but results in a constraint that is no longer well-typed, according to a simple flat typing, since it involves the union of relations of different types. Without such typing, translation to booleans is awkward, since it requires combining matrices of incompatible dimensions.

Our solution is to atomize the constraint itself, splitting it into a conjunction of constraints, each corresponding to a product of simple types. This accomplishes a larger goal than the particular goal of atomization: it allows a constraint that is well-typed only in a more liberal type system to be converted into one that is well-typed in a simple, flat type system. A fortuitous byproduct of atomization is therefore the ability to handle a more flexible constraint language that provides subtypes, and in fact union types also. It permits, for example, the perfectly reasonable constraint

```
no p: Person | p in p.(husband+wife)
```

(saying that nobody is his or her own husband or wife) which a simple type system might reject, since husband and wife are relations with different types. We have described the type system itself elsewhere [6], and concentrate here on atomization.

To see how atomization of constraints works, consider the constraint

```
no ((husband + wife) & shakes)
```

which says that no one shakes hands with his or her spouse – that there are no tuples belonging to the intersection (`&`) of the `shakes` relation and the union (`+`) of the `husband` and `wife` relations. We replace the `shakes` relation with its subrelations:

```
no ((husband + wife) & (shakesMM + shakesMW +
                        shakesWM + shakesWW))
```

Note that `husband` and `wife` remain, and are not atomized, since they permit no further decomposition (their domains and ranges already being atomic types). We now distribute intersection through union, applying the law of relational logic

```
(A + B) & (C + D) =
(A & B) + (A & C) + (B & C) + (B & D)
```

to obtain:

```
no ((husband & shakesMM) + (wife & shakesMM) +
    (husband & shakesMW) + (wife & shakesMW) +
    (husband & shakesWM) + (wife & shakesWM) +
    (husband & shakesWW) + (wife & shakesWW))
```

From the types of the relations alone, it is easy to determine that most of the intersection expressions are empty, giving

```
no ((husband & shakesWM) + (wife & shakesMW))
```

(This simplification step, incidentally, not only helps obtain a simply-typed constraint, but also eliminates tautological clauses that might otherwise appear in the boolean formula.) Applying another law of relational logic, namely that a union is empty only if its constituents are empty

```
no (A + B) = no A and no B
```

gives our final result:

```
no (husband & shakesWM) and
     no (wife & shakesMW)
```

which is simply typed (over the atomic types `Man` and `Woman`), and is semantically equivalent to the original constraint, under the condition that the shakes relation is equal to the union of its atomized subrelations.

In effect, what we have done is to 'bubble up' the atomization through expressions and then formulas, 'popping' some of the bubbles along the way when expressions are found to be empty. The full algorithm, described more formally below, has a rewrite rule for each kind of expression or formula; it is because the other relational operators distribute through union that this is possible. In comparison to the standard translation without subtyping, atomization does increase the size of the relational constraint. But more importantly, it reduces the size of the final boolean formula, essentially by applying reasoning at the level of types that would otherwise have to be performed in the solver.

2.5 Atomizing Supertypes

Until this point, our treatment of supertypes has been oversimplified. Atomization has simply ignored supertypes and created atomic types from the leaves of the type hierarchy. In the example model this worked fine, because the sole supertype `Person` was declared `abstract`, meaning that it contains no instances not also contained in a subtype. In general, however, a supertype may contain 'direct' instances of its own. To accommodate them, we introduce a new atomic type, called a *remainder type*, containing just the direct instances. The atomic types thus consist of both the leaf subtypes and a remainder type for every non-abstract supertype, which together partition the entire universe of instances.

2.6 Scope

The key to analysis of relational logic with a SAT-solver is to limit the size of the universe. In the Alloy language, this limit is expressed by the user as a *scope*, which limits the cardinalities of the basic types. Unlike many other analyzers and model checkers, Alloy segregates scopes within commands, decoupling the model from the pragmatics of its analysis.

In the simple scheme, without types, the scope would be just an integer giving the number of elements in the universe. In the scheme with flat types, the scope is a mapping assigning a bound to each type; with subtypes, it assigns a bound to each subtype. With more precise typing comes a scope of finer granularity, and as we have seen, a smaller boolean formula.

formula ::=		*formula*
	`not` *formula*	*negation*
	formula `and` *formula*	*conjunction*
	formula `or` *formula*	*disjunction*
	expr `in` *expr*	*subset*
	`all` *var*: *expr* `|` *formula*	*universal*
	`some` *var*: *expr* `|` *formula*	*existential*
expr ::=		expression
	var	variable
	`none`	empty set
	~ expr	transpose
	expr + expr	union
	expr – expr	difference
	expr & expr	intersection
	expr → expr	product
	expr . expr	composition

Figure 4. Language syntax

Finer scoping, it should be noted, constrains the problem being solved. If we specify a scope of 5 men and 5 women, for example, we are ruling out a solution involving 7 men and 3 women, which a scope of 10 persons (and no constraints on the subtypes) would admit. In this particular model, however, the constraints already imply balanced genders, so no solution is ruled out by the finer scoping.

In general, then, our scheme allows the user to express cardinality constraints directly in the scope, so that they can be exploited in translation. Our implementation actually selects the atomic types according to the specified scope, allowing the user to trade performance for coverage. For example, the command

```
check AnswersDifferent for
     exactly 5 Man, exactly 5 Woman
```

would result in an atomization on `Man` and `Woman`, Alternatively, the command

```
check AnswersDifferent for exactly 10 Person
```

would erase the type distinction between `Man` and `Woman`, and treat them simply as disjoint subsets of `Person`, as if the model had been specified without recourse to subtypes.

3. FORMAL DETAILS

We formalize atomization on a subset of Alloy (syntax in Figure 4; semantics in the appendix) – a simple first-order relational logic that captures the language's essential features.

As we have seen, an Alloy model declares a set of base types, a subtype relationship between them, and a set of relations over those types using signatures and their fields. The syntax is immaterial, of course; to emphasize this, and simplify the presentation, we shall assume that the semantic structures implied by such declarations are given. Accordingly, there is a set *Base* of base types, a set *Relation* of relation names, and a function *Decl* that maps each relation name to its type. An indication of which base types are abstract is also required in general, but this is a minor complication that we shall ignore.

Each column of a relation has a declared base type. The type of a relation will be represented as the tuple of the base types of its columns. The possible relation types of arity n are $Type_n = Base^n$, the n-tuples over *Base*. Under this interpretation, a binary relation from type A to type B will have type $\langle A, B\rangle \in Type_2$. For consistency, sets are interpreted as unary relations (containing one-tuples), and will take a one-tuple as their type ($Base^1$ is interpreted to be the set of one-tuples of *Base*). The set of all relation types is $Type = \{Type_n \mid n \geq 1\}$. The type declarations of the relations will be given as a function $Decl$: $Relation \rightarrow Type$.

The subtype relation between base types is given by ≤, which must be a transitive, reflexive, anti-symmetric binary relation on the base types. This is extended in the obvious way to relation types: $\langle A_1, \ldots A_n \rangle \leq \langle B_1, \ldots B_n \rangle$ iff $\forall i.\ A_i \leq B_i$.

To summarize, the declarations in an Alloy model give rise to *Base*, *Relation*, *Decl*, and ≤. The values for the example model are as follows:

$$\begin{aligned} Base &= \{Man, Woman, Person\} \\ Relation &= \{shakes, wife, husband\} \\ Decl(shakes) &= \langle Person, Person \rangle \\ Decl(wife) &= \langle Man, Woman \rangle \\ Decl(husband) &= \langle Woman, Man \rangle \\ Man &\leq Person \\ Woman &\leq Person \end{aligned}$$

3.1 Atomization of base types and relations

Atomization converts a subtype hierarchy into an equivalent type partitioning. For each base type, a new atomic type is defined that contains only those elements in the type which are *not* elements of any of its subtypes. These atomic types are disjoint and together partition the same universe of objects included in the base types. Technically, a fresh set of type names should be introduced for the atomic types, with a function mapping the normal types into them. Instead, to minimize notation, the atomic types will use the same names as their base types, relying on context to disambiguate.

In the example model, the `Person` type is declared `abstract`. This means that it contains no instances not also contained in a subtype. Thus the atomic type *Person* must be empty and so would actually be eliminated from the set of atomic types.

The relations defined in an Alloy model must also be atomized to reflect the new type partitioning. A set of fresh variables is introduced for this, denoted by using relation types as subscripts on the original relation names. For each relation $r \in Relation$ there will be a set of fresh variables $\{r_t \mid t \leq Decl(r)\}$. Each of these atomic relations represents a slice of the relation: r_t contains the tuples in r which are a member of relation type t but not any of its subtypes. The atomic relations in the example model are:

$$\{shakes_{\langle Man,Man\rangle}, shakes_{\langle Man,Woman\rangle}, shakes_{\langle Woman,Man\rangle}, shakes_{\langle Woman,Man\rangle}, wife_{\langle Man,Woman\rangle}, husband_{\langle Woman,Man\rangle}\}$$

$$\frac{\Gamma \vdash f \rhd g}{\Gamma \vdash \texttt{not}\ f \rhd \texttt{not}\ g}\ \textit{negation} \qquad \frac{\Gamma \vdash f \rhd f' \quad \Gamma \vdash g \rhd g'}{\Gamma \vdash f\ \texttt{and}\ g \rhd f'\ \texttt{and}\ g'}\ \textit{conjunction} \qquad \frac{\Gamma \vdash f \rhd f' \quad \Gamma \vdash g \rhd g'}{\Gamma \vdash f\ \texttt{or}\ g \rhd f'\ \texttt{or}\ g'}\ \textit{disjunction}$$

$$\frac{\Gamma \vdash a \rhd \{a_i : A_i\} \quad \Gamma \vdash b \rhd \{b_j : B_j\}}{\Gamma \vdash a\ \texttt{in}\ b \ \rhd\ \texttt{and}\left(\{a_i\ \texttt{in}\ b_j \mid A_i = B_j\} \cup \{a_i\ \texttt{in none} \mid \forall j.\ A_i \neq B_j\}\right)}\ \textit{subset}$$

$$\frac{\Gamma \vdash a \rhd \{a_i : A_i\} \quad \text{for each } A_i: \ \Gamma[v : A_i] \vdash f \rhd f_i}{\Gamma \vdash \texttt{all}\ v : a \mid f \ \rhd\ \texttt{and}\{\texttt{all}\ v : a_i \mid f_i\}}\ \textit{universal} \qquad \frac{\Gamma \vdash a \rhd \{a_i : A_i\} \quad \text{for each } i: \ \Gamma[v : A_i] \vdash f \rhd f_i}{\Gamma \vdash \texttt{some}\ v : a \mid f \ \rhd\ \texttt{or}\{\texttt{some}\ v : a_i \mid f_i\}}\ \textit{existential}$$

Figure 5. Formula atomization rules

$$\frac{B \in Base}{\Gamma \vdash B \rhd \{T : \langle T \rangle \mid T \leq B\}}\ \textit{base type} \qquad \frac{r \in Relation}{\Gamma \vdash r \rhd \{r_t : t \mid t \leq Decl(r)\}}\ \textit{relation} \qquad \frac{v : t \in \Gamma}{\Gamma \vdash v \rhd \{v : t\}}\ \textit{local variable}$$

$$\Gamma \vdash \texttt{none} \rhd \varnothing\ \textit{empty set} \qquad \frac{\Gamma \vdash a \rhd \{a_i : \langle A_{i,1}, \ldots A_{i,n} \rangle\}}{\Gamma \vdash {\sim} a \rhd \{{\sim} a_i : \langle A_{i,n}, \ldots A_{i,1} \rangle\}}\ \textit{transpose}$$

$$\frac{\Gamma \vdash a \rhd \{a_i : A_i\} \quad \Gamma \vdash b \rhd \{b_j : B_j\}}{\Gamma \vdash a + b \ \rhd\ \{a_i + b_j : A_i \mid A_i = B_j\} \cup \{a_i : A_i \mid \forall j. A_i \neq B_j\} \cup \{b_j : B_j \mid \forall i. A_i \neq B_j\}}\ \textit{union}$$

$$\frac{\Gamma \vdash a \rhd \{a_i : A_i\} \quad \Gamma \vdash b \rhd \{b_j : B_j\}}{\Gamma \vdash a - b \ \rhd\ \{a_i - b_j : A_i \mid A_i = B_j\} \cup \{a_i : A_i \mid \forall j. A_i \neq B_j\}}\ \textit{difference}$$

$$\frac{\Gamma \vdash a \rhd \{a_i : A_i\} \quad \Gamma \vdash b \rhd \{b_j : B_j\}}{\Gamma \vdash a \,\&\, b \ \rhd\ \{a_i \,\&\, b_j : A_i \mid A_i = B_j\}}\ \textit{intersection} \qquad \frac{\Gamma \vdash a \rhd \{a_i : \langle A_{i,1}, \ldots A_{i,n} \rangle\} \quad \Gamma \vdash b \rhd \{b_j : \langle B_{j,1}, \ldots B_{j,m} \rangle\}}{\Gamma \vdash a \to b \ \rhd\ \{a_i \to b_j : \langle A_{i,1}, \ldots A_{i,n}, B_{j,1}, \ldots B_{j,m} \rangle\}}\ \textit{product}$$

$$\frac{\Gamma \vdash a \rhd \{a_i : \langle A_{i,1}, \ldots A_{i,n} \rangle\} \quad \Gamma \vdash b \rhd \{b_j : \langle B_{j,1}, \ldots B_{j,m} \rangle\} \quad n + m \geq 3}{\Gamma \vdash a\,.\,b \ \rhd\ \{a_i\,.\,b_j : \langle A_{i,1}, \ldots A_{i,n-1}, B_{j,2}, \ldots B_{j,m} \rangle \mid A_{i,n} = B_{j,1}\}}\ \textit{composition}$$

Figure 6. Expression atomization rules

3.2 Atomization of formulas and expressions

Armed with the set of atomic types and atomic relations, it is now possible to translate any Alloy formula into an equivalent one using only those atomic forms. The translated formula will be strictly typed: different atomic types will never be unioned, differenced, intersected, or composed. The result is that subtyping is removed from the model: the atomic types form a simple partitioning of the universe. This facilitates the efficient translation into boolean formulas for SAT-solving.

We formalize atomization with a set of inference rules that make atomization judgments. For formulas, an atomization judgment takes the form $\Gamma \vdash f \triangleright g$, saying that formula f atomizes to formula g in the binding environment Γ. Binding environments are used for quantified variables, and are sets of binding pairs of the form $\{v_i : T_i\}$, where $v_i \in Variable$ are local variables and $T_i \in Type$ are relation types. A variable can be bound only once in an environment. The bound type is interpreted as an atomic relation type. Extending an environment Γ with a binding pair $v{:}T$ results in the new environment $\Gamma[v{:}T]$, defined as:

$$\Gamma[v:T] = \{v_i : T_i \mid \text{if } v_i = v \text{ then } T_i = T \text{ else } v_i : T_i \in \Gamma\}$$

The top-level formulas of a model are atomized with an empty environment: $\varnothing \vdash f \triangleright g$. The rules for formula atomization are presented in Figure 5.

An expression is atomized into a set of expressions along with their types; correctness relies on the fact that the union of the values of these new expressions will be equal to the value of the original expression. Expression atomization judgments take the form $\Gamma \vdash a \triangleright \{a_i : A_i\}$ saying that, in the environment Γ, expression a atomizes to the set of expressions $\{a_i\}$ with corresponding atomic types $A_i \in Type$. The rules for expression atomization are shown in Figure 6.

3.3 Sample Atomization

An example atomization judgment will be derived for the same formula discussed informally in section 2.4:

```
no ((husband + wife) & shakes)
```

The (*relation*) rule gives us the judgments for the relations:

$$\varnothing \vdash shakes \triangleright \begin{Bmatrix} shakes_{\langle Man,Man\rangle} : \langle Man, Man\rangle, \\ shakes_{\langle Man,Woman\rangle} : \langle Man, Woman\rangle, \\ shakes_{\langle Woman,Man\rangle} : \langle Woman, Man\rangle, \\ shakes_{\langle Woman,Man\rangle} : \langle Woman, Man\rangle \end{Bmatrix}$$

$$\varnothing \vdash wife \triangleright \{wife_{\langle Man,Woman\rangle} : \langle Man, Woman\rangle\}$$

$$\varnothing \vdash husband \triangleright \{husband_{\langle Woman,Man\rangle} : \langle Woman, Man\rangle\}$$

The (*union*) rule gives

$$\varnothing \vdash husband + wife \triangleright \begin{Bmatrix} husband_{\langle Woman,Man\rangle} : \langle Woman, Man\rangle, \\ wife_{\langle Man,Woman\rangle} : \langle Man, Woman\rangle \end{Bmatrix}$$

The (*intersection*) rule considers all 8 combinations of atomic expressions from *(husband+wife)* and *shakes*, eliminating the ones which are empty to yield:

$$\varnothing \vdash (husband + wife) \,\&\, shakes \triangleright \begin{Bmatrix} husband_{\langle Woman,Man\rangle} \,\&\, shakes_{\langle Woman,Man\rangle} : \langle Woman, Man\rangle, \\ wife_{\langle Man,Woman\rangle} \,\&\, shakes_{\langle Man,Woman\rangle} : \langle Man, Woman\rangle \end{Bmatrix}$$

The formula "`no` *expr*" is desugared into "*expr* `in none`". The (*empty set*) rule gives an empty set as the atomization of `none`. The (*subset*) rule is thus forced to test the left-hand atomic subexpressions for emptiness and to conjunct the results:

$$\varnothing \vdash ((husband + wife) \,\&\, shakes) \texttt{ in none} \triangleright \begin{pmatrix} husband_{\langle Woman,Man\rangle} \,\&\, shakes_{\langle Woman,Man\rangle} \texttt{ in none} \\ \texttt{and} \\ wife_{\langle Man,Woman\rangle} \,\&\, shakes_{\langle Man,Woman\rangle} \texttt{ in none} \end{pmatrix}$$

This is equal to the result of the informal atomization:

```
no (husband & shakesWM) and
   no (wife & shakesMW)
```

4. RESULTS

We evaluated the effect of atomization on three models. All are realistic, having been developed in earlier case studies, not as pedagogical examples, and involve subtyping that in the original analyses was not exploited. We chose analyses that yielded no counterexamples, since the performance of a SAT solver on satisfiable formulas is highly unstable (as the solver can get lucky after examining only a tiny fraction of the space).

The first example, *PatientIdentity*, is a model of a scheme for ensuring that patients irradiated in a radiotherapy facility are correctly identified. It involved explicit representation of the behavior of nurses and patients, the status of open and closed doors, and the relationship between a patient's identity and physical identifiers. Subtyping is present in the classification of persons into nurses and patients, and identifiers into room and patient identifiers.

The second example, *Javatypes*, is a model developed to check the type soundness of Java. Subtypes arise in the structure of the abstract syntax tree, and in the partitioning of values into objects and null. The analysis takes a traditional inductive form, checking that the relationship between compile-time types and run-time types is preserved when a statement is executed.

Model	Variables	Clauses	Time(sec)
PatientIdentity	**74125/44010 (-41%)**	**208486/123210 (-41%)**	**138/82 (1.68)**
Javatypes	**105625/63669 (-40%)**	**303308/179886 (-41%)**	**220/109 (2.03)**
Naming	**62473/60448 (-3%)**	**178346/172335 (-3%)**	**74/35 (2.13)**

Table 1. Some performance results.
Each entry takes the form: unatomized/atomized (% change for counts; speedup-factors for times)

The third example, *Naming*, is a model of a scheme for serving names for network resources [17], in which both names and the server's database take the structure of trees with alternating attributes and values along each path. Subtypes arise because it is convenient to write part of the model over generic trees with nodes and labels, and specialize according to the attribute/value distinction only for some properties.

We picked scopes that are representative of those typically used in practice. For some types, a small scope clearly suffices. In the Javatypes example, it is pointless to consider more than two program states or one statement, for example, since a counterexample cannot require more. The problem domain naturally suggests constraints on the scope; for example, that there be at least as many identifiers as persons in the PatientIdentity example. Otherwise, we simply pick a bound on a type that produces a reasonable search space, giving some confidence if no counterexample is found, but which terminates in a reasonable time. In the Javatypes analysis, for example, we considered statements involving at most 8 subexpressions. The search space is typically hundreds of bits wide, although the translation to boolean typically involves thousands of variables (most introduced to avoid blowups due to disjunction).

Our experimental method was as follows. Rather than comparing to our previous implementation, which might give an unfair advantage to the new implementation because of unrelated improvements, we compared, for each analysis, a scope that assigned bounds to subtypes to one that did not (thereby essentially turning off atomization). To ensure that the unatomized case was not solving a harder problem, we added explicit cardinality constraints. For example, in Javatypes, the atomized version was given 5 interfaces and 5 classes; in the unatomized version, the scope specified 10 types, and included constraints that there be at most 5 interfaces and 5 classes.

To mitigate variance due to system conditions and random seeds, we ran each analysis ten times and averaged the results. All analyses were performed on a Intel Pentium 4 2.2GHz with 512KB L2 cache and 1GB RAM, using the Berkmin solver [9]. We did some informal tests using ZChaff [23] and found very similar results, suggesting that the improvements are genuine and not solver-dependent.

The results (in Table 1) show that atomization can reduce the number of variables and clauses in the boolean formula significantly. The running time is roughly halved (even for the Naming analysis, discussed below, for which atomization eliminated fewer variables).

We measured, in addition to solver time, the time taken to generate the boolean formula, being concerned that the larger atomized formula might take more time. It does not. The time taken to perform the atomization itself is negligible.

The implementation of the atomization scheme described here has been incorporated into the latest version of the Alloy Analyzer, and is enabled by default. User experience to date confirms its benefits.

5. HANDLING TRANSITIVE CLOSURE

Our discussion to this point has not mentioned the handling of the transitive closure operator, which turns out to be problematic. The obvious approach is to expand a closure expression into a union of joins, determining the number of joins according to the scope. Unfortunately, this results in an unacceptable blowup: for non-trivial models, the translation step itself becomes a bottleneck.

The problem is that atomization of a join essentially multiplies out the subexpressions. Closure of an expression that atomizes to 3 subexpressions in a scope of 10 requires 10 nested joins, resulting in an expression with 3^{10} terms! One might think that iterated squaring should help, since then only a logarithmic number of joins would be required. But, in contrast to the standard applications of iterative squaring in model checking, here the squaring is applied to an AST, and although the number of joins drops, the size of the expression being squared rises, so that the computation is still intractable.

The current implementation therefore adopts a simple-minded but effective solution. The domain and range type of an expression under closure is treated as an atomic type, and its subtypes are treated as subsets. In other words, the components of the closure are not atomized. Fortunately, even with this loss of opportunity, atomization is still effective. In the Naming example, closures prevented the treatment of attribute and value nodes as distinct, atomizable subtypes. This is why the variable reduction was less than for the other examples. The effect even of partial atomization is still noticeable, though.

We have a better solution to this problem, but are not convinced that the effort of coding it would be worthwhile. The idea is to take the types that appear as domain and range types in closures, and create a collection of fresh 'matching' types. The matching type becomes the parent of the original type's subtypes, and we introduce a bijection between each type and its matching type that acts as a typecast. These bijections are inserted explicitly in the source-to-source translation, and are atomized along with other relations. The original closure types are not atomized, so that closure expressions do not require atomization, but the matching types are atomized, so that other expressions can benefit from the presence of the subtype structure. By breaking symmetry on the

bijections (setting each, without loss of generality, to a constant) [26], we can ensure that their addition would not expand the state space.

6. RELATED WORK

Translation to SAT is becoming an increasingly popular way to solve constraints in a variety of higher level languages, as SAT solving technology advances, and SAT solvers become universal constraint solving engines. To our knowledge, the Alloy Analyzer is the only tool that solves constraints written in a relational logic by translation to SAT.

Other tools for languages based on relational logic use different underlying engines. The ProB tool [20] for analyzing specifications in the B language, and a tool for JML [27] use solvers for set constraints; animators for the Z language (such as PiZa [25]) use Prolog. These tend to perform much worse than SAT solvers on large state spaces, although they can be better suited to simulation problems that involve larger states but less search. It is not clear whether atomization can be applied to these tools.

Tools that work by translation to SAT include: Denali, an assembler that generates optimal instruction sequences by search [15]; bounded model checkers [5] [1] [21], which check state machines against temporal logic formulas by unrolling the transition relation and searching for a counterexample trace; and AI planners [7] [15], which find plans by unrolling actions and searching for a sequence of steps satisfying a goal.

These are all potential candidates for atomization. If the assembly language can be typed [22], atomization would allow the relation representing the machine's memory to be decomposed. Model checkers typically have only very low level datatypes; SMV, for example, offers only enumerations, integer subranges and arrays. But if a SAT-based checker were to be built for a language with richer datatypes, such as Promela [11] or Zing [1], atomization should be useful. It seems that the rich variety of datatypes in TLA+ [18] has been an impediment to the development of a symbolic checker. Atomization might be an important ingredient in such a checker, although it would be necessary to type the language at least implicitly. The developer of TLA+ has argued that specification languages should be untyped [19]. We believe that the opportunity to perform optimizations such as atomization militates in favor of typing, in addition to concerns we have addressed elsewhere [6], such as the ability to detect redundancies that signal specification errors.

AI planners already make use of decomposition by top-level types [7] [29]. For domains that involve classification of objects, atomization might bring further improvements. Even in the standard toy example of blocks on a table there is an opportunity for subtyping: the set of surfaces on which blocks are placed can be split into the faces of blocks and the top of the table. Most planners cannot currently handle plans that involve dynamic allocation of objects [29]; such an extension would make atomization more valuable.

The Alloy Analyzer itself is used as a backend by a tool that finds bugs in Java methods [27]. Even for object-oriented programs with little explicit inheritance, atomization is likely to bring significant benefit, since the presence of any field of type `Object` prevents any decomposition of the universe unless subtypes are available.

It should be noted that although atomization relies on typing, the types might be discovered automatically, rather than being provided by the user. Type inference has been used for both specifications (e.g. [13]) and code (e.g. [24], [7]) to obtain finer-grained types than those given explicitly.

7. CONCLUSIONS

Atomization is a simple and effective scheme for improving the translation of relational logic to SAT. It can be applied whenever constraints are expressed in a first order logic with typed relations or predicates. The advantage it brings is proportional to the extent of the subtyping present. Because atomization involves a source-to-source transformation, it can be introduced into existing tools with minimal disruption. Opportunities to exploit atomization should grow as SAT solvers become more popular, and more tool builders recognize the feasibility of analyzing first order logic.

ACKNOWLEDGEMENTS

Manu Sridharan designed and implemented the 'zeroing out' method in an earlier version of Alloy, with help from Ilya Shlyakhter, recognizing the opportunity for an optimization based on subtype structure. Mandana Vaziri contributed crucial insights at the start of this project; it was her intuition that an analysis might split the universe into atomic types.

This research was supported by grant 0086154 ('Design Conformant Software') from the ITR program of the National Science Foundation, by grant 6895566 ('Safety Mechanisms for Medical Software') from the ITR program of the National Science Foundation, and by the the High Dependability Computing Program from NASA Ames (Cooperative Agreement NCC-2-1298).

APPENDIX: SEMANTICS

A denotational semantics of the language of Figure 4 is presented here. In the standard style, it employs an environment η mapping variables to their denotations. Base types in *Base* are mapped to sets, while relations in *Relation* and local variables in *Variable* are mapped to relations over those sets. An environment must respect the subtype relation:

$$\forall A, B \in Base.\quad A \le B \quad \Rightarrow \quad \eta(A) \subseteq \eta(B)$$
$$\forall A, B \in Base.\quad \neg(A \le B) \wedge \neg(B \le A) \quad \Rightarrow \quad \eta(A) \cap \eta(B) = \varnothing$$

An environment is extended with the notation

$$\eta[v \mapsto x](u) = \begin{cases} x & \text{if } u = v \\ \eta(u) & \text{otherwise} \end{cases}$$

The denotation function $[\![\ ]\!]\eta$ maps formulas to boolean values and expressions to relations. A solution of a formula is an environment in which the formula's denotation is true. Note that the denotation of an expression is always a relation: a set of tuples. The denotation of a base type is the set of one-tuples over the set that the environment maps the type to. This allows a

uniform treatment of the variables naming the base types and the variables naming relations over those types.

$$[\![v]\!]\eta = \eta(v)^1 \quad v \in Base$$
$$[\![v]\!]\eta = \eta(v) \quad v \in Relation \cup Variable$$
$$[\![\text{none}]\!]\eta = \varnothing$$
$$[\![\sim a]\!]\eta = \{\langle a_n, \ldots a_1\rangle | \langle a_1, \ldots a_n\rangle \in [\![a]\!]\eta\}$$
$$[\![a + b]\!]\eta = [\![a]\!]\eta \cup [\![b]\!]\eta$$
$$[\![a - b]\!]\eta = [\![a]\!]\eta - [\![b]\!]\eta$$
$$[\![a \,\&\, b]\!]\eta = [\![a]\!]\eta \cap [\![b]\!]\eta$$
$$[\![a \to b]\!]\eta = \{\langle a_1, \ldots a_n, b_1, \ldots b_m\rangle | \langle a_1, \ldots a_n\rangle \in [\![a]\!]\eta \wedge \langle b_1, \ldots b_m\rangle \in [\![b]\!]\eta\}$$
$$[\![a \,.\, b]\!]\eta = \{\langle a_1, \ldots a_{n-1}, b_2, \ldots b_m\rangle | \langle a_1, \ldots a_n\rangle \in [\![a]\!]\eta \wedge \langle b_1, \ldots b_m\rangle \in [\![b]\!]\eta \wedge a_n = b_1\}$$
$$[\![v]\!]\eta = \eta(v)^1 \quad v \in Base$$
$$[\![v]\!]\eta = \eta(v) \quad v \in Relation \cup Variable$$
$$[\![\text{none}]\!]\eta = \varnothing$$
$$[\![\sim a]\!]\eta = \{\langle a_n, \ldots a_1\rangle | \langle a_1, \ldots a_n\rangle \in [\![a]\!]\eta\}$$
$$[\![a + b]\!]\eta = [\![a]\!]\eta \cup [\![b]\!]\eta$$
$$[\![a - b]\!]\eta = [\![a]\!]\eta - [\![b]\!]\eta$$
$$[\![a \,\&\, b]\!]\eta = [\![a]\!]\eta \cap [\![b]\!]\eta$$
$$[\![a \to b]\!]\eta = \{\langle a_1, \ldots a_n, b_1, \ldots b_m\rangle | \langle a_1, \ldots a_n\rangle \in [\![a]\!]\eta \wedge \langle b_1, \ldots b_m\rangle \in [\![b]\!]\eta\}$$
$$[\![a \,.\, b]\!]\eta = \{\langle a_1, \ldots a_{n-1}, b_2, \ldots b_m\rangle | \langle a_1, \ldots a_n\rangle \in [\![a]\!]\eta \wedge \langle b_1, \ldots b_m\rangle \in [\![b]\!]\eta \wedge a_n = b_1\}$$

REFERENCES

[1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. *Zing: A Model Checker for Concurrent Software.* Microsoft Research Technical Report MSR-TR-2004-10. January, 2004.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, March, 1999.

[3] Avrim Blum. *Fast Planning Through Planning Graph Analysis.* Artificial Intelligence 90:281-300, 1997.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, 1999.

[5] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2000.

[6] Jonathan Edwards, Daniel Jackson, and Emina Torlak. *A Type System for Object Models*. Submitted for publication. http://sdg.lcs.mit.edu/pubs/TR/typesforobjectmodels.pdf

[7] Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine Soerensen, and Mads Tofte. AnnoDomini: From Type Theory to Year 2000 Conversion Tool. In *26th ACM Symposium on the Principles of Programming Languages*, January, 1999.

[8] Michael Ernst, Todd Millstein and Daniel Weld. Automatic SAT compilation of planning problems. *Proceedings of the International Joint Conference on Artificial Intelligence* (IJCAI), 1997.

[9] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT Solver. *Design Automation and Test in Europe (DATE)* 2002.

[10] Paul Halmos. *Problems for Mathematicians, Young and Old.* The Mathematical Association of America, 1991.

[11] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2004.

[12] Daniel Jackson. Automating First-Order Relational Logic. *ACM SIGSOFT Conference on Foundations of Software Engineering*, San Diego, CA, November, 2000.

[13] Daniel Jackson, Somesh Jha and Craig A. Damon. Isomorph-free Model Enumeration: A New Method for Checking Relational Specifications. *ACM Transactions on Programming Languages and Systems* 20(2):302-343, 1998.

[14] Daniel Jackson, Ilya Shlyakhter and Manu Sridharan. A Micromodularity Mechanism. *ACM SIGSOFT Conference on Foundations of Software Engineering / European Software Engineering Conference*, Vienna, Austria, September, 2001.

[15] Rajeev Joshi, Greg Nelson, and Keith Randall. *Denali: a goal-directed superoptimizer.* http://citeseer.nj.nec.com/joshi01denali.html

[16] H. Kautz, and B. Selman. Planning as satisfiability. *Proc. European Conference on Artificial Intelligence,* Vienna, Austria, 1992.

[17] Sarfraz Khurshid and Daniel Jackson. Exploring the Design of an Intentional Naming Scheme with an Automatic Constraint Analyzer. *Proc. 15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, September, 2000.

[18] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Professional, 2002.

[19] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems* 21(3):502-526, 1999.

[20] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. *Proceedings Formal Methods Europe*, 2003.

[21] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[22] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *25th ACM Symposium on the Principles of Programming Languages*, 1998.

[23] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. *39th Design Automation Conference*, Las Vegas, June, 2001.

[24] Robert O'Callahan and Daniel Jackson. Lackwit: A Program Understanding Tool Based on Type Inference. *Proc. International Conference on Software Engineering*, Boston, MA, May, 1997, pp. 338-348.

[25] *Piza, The Prolog Z Animator.* http://www.noodles.demon.co.uk/PiZA/PiZAHome.html

[26] Ilya Shlyakter. *Generating effective symmetry-breaking predicates for search problems.* Electronic Notes in Discrete Mathematics, Vol. 9 (2001).

[27] Mandana Vaziri and Daniel Jackson. Checking Properties of Heap-Manipulating Procedures using a Constraint Solver. *Ninth International Conference on Tools and Algorithms for the Construction and Analysis of System*s, Warsaw, Poland, April, 2003.

[28] Tim Wahls, Gary T. Leavens, and Albert L. Baker. *Executing formal specifications with constraint programming.* Technical Report 97-12a, Department of Computer Science, Iowa State University, August, 1998.

[29] D. S. Weld. *Recent advances in AI planning.* AI Magazine, 20(2):93-123, 1999.