MIT-CSAIL-TR-2006-068 September 29, 2006

# The Design of a Relational Engine

Emina Torlak and Daniel Jackson

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

# The Design of a Relational Engine

Emina Torlak and Daniel Jackson
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA
{emina, dnj}@mit.edu

## ABSTRACT

The key design challenges in the construction of a SAT-based relational engine are described, and novel techniques are proposed to address them. An efficient engine must have a mechanism for specifying partial solutions, an effective symmetry detection and breaking scheme, and an economical translation from relational to boolean logic. These desiderata are addressed with three new techniques: a symmetry detection algorithm that works in the presence of partial solutions, a sparse-matrix representation of relations, and a compact representation of boolean formulas inspired by boolean expression diagrams and reduced boolean circuits. The presented techniques have been implemented and evaluated, with promising results.

## Keywords

First order logic, relational logic, partial instance, model finding, constraint solving, SAT solvers.

## 1. INTRODUCTION

Many computational problems can be expressed declaratively as collections of constraints, and then solved using a general purpose constraint-solving engine. A variety of such engines have been developed, each tailored for a particular language: resolution engines for Prolog, Simplex for linear inequalities, SAT solvers for boolean formulas, etc.

This paper concerns the design of a *relational engine*: that is, an engine for a constraint language that is a relational logic, consisting of the quantifiers and logical connectives of first-order logic along with the operators of the relational algebra. The inclusion of the transitive closure operator makes the language strictly more expressive than first-order logic; the presence of the other operators (especially join) is a convenience, often allowing constraints to be written in a more succinct and natural fashion. Constraints are interpreted over a finite universe of uninterpreted atoms, which may nevertheless be bound to richer objects whose properties the relational engine does not exploit.

Relational logic is well-suited to problems involving graph-like configurations, static or dynamic. In the static case, the constraints describe a particular configuration. In the dynamic case, they describe a pair of configurations (if a single transition is sought), or a sequence of configurations (for a trace), which can be expressed as a higher-arity relation. For example, the connectivity of a network over time can be modeled as a ternary relation containing a tuple $\langle t, h_1, h_2\rangle$ when host $h_1$ is connected to host $h_2$ at time $t$. Because relational logic includes both existential and universal quantifiers, and negation, it is more flexible than traditional logic programming languages such as Prolog.

Examples of problems that are suitable for solving with a relational engine include:

- Design analysis. A software design expressed as a state machine over structured states (expressed as collections of relations) can be checked, within finite bounds, for preservation of invariants, for refinement against a more abstract specification, and for linear temporal properties over traces of bounded length, by presenting the engine with a constraint of the form $S \wedge \neg P$, whose solutions are counterexamples satisfying the description of the system ($S$) but violating the expected property ($P$).

- Code analysis. A procedure can be checked against a declarative specification using the same method, by translating its code to a relational constraint.

- Test case generation. Unit tests for modules implementing intricate datatypes, such as red-black trees, with complex representation invariants, can be generated by a relational engine from the invariants.

- Course scheduling. Given the prerequisite dependences, overall requirements of a degree program, information about which terms particular courses are offered in, and a set of courses already taken, a relational engine can plan a student's course schedule.

- Network configuration. The problem of configuring a network from a collection of devices to satisfy global connectivity goals while meeting compatibility and architectural constraints is easily expressed relationally.

- Component assembly and installation. The imperative scripts typically used for installing and configuring software components are fragile, because the writer cannot anticipate the context in which the script executes. A declarative approach in which the system is assembled using a relational engine would allow scripts to be presented as goals, and would accommodate contextual variations automatically.

We have established the feasibility of this approach for design analysis [19], code analysis [36, 34] and test case generation [21] in earlier work. The prototype tool that we describe in this paper has been applied to design analysis, code analysis [7], course scheduling [38], and network configuration; it is also a mean Sudoku player.

Our earlier work involved the development of the Alloy modeling language [19] and its analyzer. Alloy was designed for the analysis of software models, and although applicable in these other contexts, it is not ideally suited to them. Its principal defect in this regard is that it has no notion of *partial solution*. The user provides only a constraint to be solved, and if a partial solution is available which the obtained solution should extend, it can be provided only in the form of an additional constraint. Because the solver must essentially rediscover the partial solution by solving this constraint, this strategy does not scale well.

In order to overcome this limitation, we are developing a new tool that, unlike Alloy, will be suitable as a generic relational engine for a wide range of applications. This paper describes the key challenges in its design, and proposes novel ways in which to address them. A prototype embodying these ideas has been constructed, and is evaluated on a variety of examples. It outperforms Alloy dramatically on problems involving partial solutions, and, due to improvements over Alloy's technology that we describe, also outperforms Alloy even on the problems for which Alloy was designed. It also performs as well as other logic engines (such as Paradox [5] and MACE [25]) on some of their own benchmark problems.

The underlying technology involves translation from relational to boolean logic, and the application of an off-the-shelf SAT solver on the resulting boolean formula. The contributions of this paper are:

- A new symmetry-breaking scheme that works in the presence of partial solutions; the inability of Alloy's scheme to accommodate partial solutions was a key reason for not supporting them.
- A new sparse-matrix representation of relations that is both simpler to implement and better performing than the 'atomization' used in Alloy [11].
- A new scheme for detecting opportunities for sharing in the constraint abstract syntax tree inspired by boolean expression diagrams [2] and reduced boolean circuits [1].

Another major difference between the new tool and Alloy is its implementation as an API rather than as a standalone application. Alloy can in fact be accessed as an API, but the interface is string-based and awkward to use. The new tool is designed to be a plugin component that can easily be incorporated as a backend of another tool. These considerations, however, while crucial motivations of the project, are not the topic of the present paper.

## 2. RELATED WORK

A variety of tools have been developed for finding finite models of first order logic (FOL) formulas [5, 13, 14, 18, 25, 33, 40, 39]. Several of these [18, 33, 40, 39] implement specialized search algorithms for exploring the space of possible interpretations of a formula. The rest [5, 13, 14, 25] are essentially compilers. Given a FOL formula and a finite universe of uninterpreted atoms, they construct an equivalent propositional satisfiability problem and delegate the task of solving it to a SAT solver.

Most research on model finding has focused on producing high-performance tools for group-theoretic investigations. LDPP [13], MACE [25], FALCON [39], and SEM [40] have all been used to solve open problems in abstract algebra. Formulation of group-theoretic problems requires only basic FOL constructs. SEM and FINDER, for example, work on a quantifier-free many-sorted logic of uninterpreted functions. MACE and Paradox [5] support quantifiers, but none of these tools handle relations or transitive closure, which are essential for analyzing properties of graph-like configurations (e.g. networks, call graphs, module dependencies).

Nitpick [18] was the first model finder to handle binary relations and transitive closure in addition to quantifier-free FOL. This made it an attractive choice for analyzing small problems that involve structured state, such as a high-level description of the paragraph-style mechanism in Microsoft Word [17] or the first version of the IPv6 mobile host protocol [28]. The usefulness of Nitpick was, however, limited by its poor scalability and lack of support for quantifiers and higher-arity relations.

The Alloy Analyzer [14] addressed both the scalability and expressiveness limitations of Nitpick. It has been applied to a wide variety of problems, including the design of an intentional naming scheme [20], the safety properties of the beam scheduler for a proton therapy machine [8], code analysis [36, 34], test-case generation [21], and network configuration [27]. While the Alloy language and its analyzer are good lightweight formal method [15] tools, they are not well suited to problems with partially known solutions, such as the network configuration problem described in [27]. Because the known aspects of a network (e.g. the exact number of routers, the protocols supported by particular routers, etc.) can only be expressed as constraints in Alloy, the Alloy Analyzer must re-discover the known information by satisfying these constraints, in addition to solving the actual configuration problem.

The variant of FOL presented in this paper is a superset of the Alloy language (Section 3). Unlike Alloy, our logic provides a mechanism for specifying partial solutions. Its accompanying model finder, Kodkod, takes advantage of known information, scaling much better than the Alloy Analyzer in the presence of partial solutions (Section 5). Kodkod outperforms the Alloy Analyzer even on the problems without partial solutions, due to the new translation to propositional satisfiability (Section 4.2) based on sparse matrices and Compact Boolean Circuits.

Compact Boolean Circuits, described in Section 4.2.3, are a hybrid between Reduced Boolean Circuits (RBCs) [1] and Boolean Expression Diagrams (BEDs) [2]. Like RBCs, CBCs satisfy Properties 1-2, use edge signs to encode negation, and restrict variable vertices to the leaves of the graph. Like BEDs, they use a more extensive set of rules than RBCs to maximize subformula sharing. All three circuit representations can be straightforwardly converted to one another.

problem := univDecl relDecl* formula

univDecl := { atom[, atom]* }
relDecl := relVar $:_{\text{arity}}$ [constant, constant]
varDecl := quantVar : expr

constant := {tuple*}
tuple := ⟨atom[, atom]*⟩

arity := 1 | 2 | 3 | 4 | . . .
atom := identifier
relVar := identifier
quantVar := identifier

formula :=

| | |
|---|---|
| expr in expr | subset |
| \| some expr | non-empty |
| \| one expr | singleton |
| \| no expr | empty |
| \| not formula | negation |
| \| formula and formula | conjunction |
| \| formula or formula | disjunction |
| \| all varDecl \| formula | universal |
| \| some varDecl \| formula | existential |

expr :=

| | |
|---|---|
| expr + expr | union |
| \| expr & expr | intersection |
| \| expr - expr | difference |
| \| expr . expr | join |
| \| expr –> expr | product |
| \|˜expr | transpose |
| \|ˆexpr | closure |
| \| {varDecl \| formula} | comprehension |
| \| relVar | relation |
| \| quantVar | quantified variable |

**(a)**

$$\frac{E \vdash t = \langle a_1, \ldots, a_k \rangle}{E \vdash \text{arity}(t) = k}$$

$$\frac{E \vdash \text{arity}(t_1) = k, \quad \cdots \quad E \vdash \text{arity}(t_n) = k}{E \vdash \text{arity}(\{t_1 \ldots, t_n\}) = k}$$

$$\frac{k > 0}{E \vdash \text{arity}(\{\}) = k}$$

$$\frac{E \vdash \text{arity}(c_L) = k, \quad E \vdash \text{arity}(c_U) = k, \quad E \vdash r :_k [c_L, c_U]}{E \vdash \text{arity}(r) = k}$$

$$\frac{E \vdash \text{arity}(p) = 1, \quad E \vdash v : p}{E \vdash \text{arity}(v) = 1}$$

$$\frac{E \vdash \text{arity}(p) = a, \quad E \vdash \text{arity}(q) = a}{E \vdash p \text{ in } q, \quad E \vdash \text{arity}(p + q) = a, \quad E \vdash \text{arity}(p - q) = a, \quad E \vdash \text{arity}(p \,\&\, q) = a}$$

$$\frac{E \vdash \text{arity}(p) = a, \quad E \vdash \text{arity}(q) = b, \quad a + b \rangle 2}{E \vdash \text{arity}(p \,.\, q) = a + b - 2}$$

$$\frac{E \vdash \text{arity}(p) = a, \quad E \vdash \text{arity}(q) = b}{E \vdash \text{arity}(p \text{ –> } q) = a + b}$$

$$\frac{E \vdash \text{arity}(p) = 2}{E \vdash \text{arity}(\tilde{}\,\text{expr}) = 2, \quad E \vdash \text{arity}(\hat{}\,\text{expr}) = 2}$$

**(b)**

P : problem → binding → boolean
R : relDecl → binding → boolean
M : formula → binding → boolean
X : expr → binding → constant
binding : (quantVar ∪ relVar) → constant

$P \llbracket \mathcal{A}\ d_1 \ldots d_n\ F \rrbracket\, b =$
$\quad R\llbracket d_1 \rrbracket b \wedge \ldots \wedge R\llbracket d_n \rrbracket b \wedge M\llbracket F \rrbracket b$

$R \llbracket r : [c_L, c_U] \rrbracket\, b = c_L \subseteq b(r) \subseteq c_U$

$M \llbracket p \text{ in } q \rrbracket\, b = X\llbracket p \rrbracket b \subseteq X\llbracket q \rrbracket b$
$M \llbracket \text{some } p \rrbracket\, b = X\llbracket p \rrbracket b \supset \emptyset$
$M \llbracket \text{one } p \rrbracket\, b = |X\llbracket p \rrbracket b| = 1$
$M \llbracket \text{no } p \rrbracket\, b = X\llbracket p \rrbracket b \subseteq \emptyset$
$M \llbracket \text{not } F \rrbracket\, b = \neg\, M\llbracket F \rrbracket b$
$M \llbracket F \text{ and } G \rrbracket\, b = M\llbracket F \rrbracket b \wedge M\llbracket G \rrbracket b$
$M \llbracket F \text{ or } G \rrbracket\, b = M\llbracket F \rrbracket b \vee M\llbracket G \rrbracket b$
$M \llbracket \text{all } v\text{: } p \mid F \rrbracket\, b = \bigwedge(M\llbracket F \rrbracket(b \oplus v \mapsto X\llbracket p \rrbracket b))$
$M \llbracket \text{some } v\text{: } p \mid F \rrbracket\, b = \bigvee(M\llbracket F \rrbracket(b \oplus v \mapsto X\llbracket p \rrbracket b))$

$X \llbracket p + q \rrbracket\, b = X\llbracket p \rrbracket b \cup X\llbracket q \rrbracket b$
$X \llbracket p \,\&\, q \rrbracket\, b = X\llbracket p \rrbracket b \cap X\llbracket q \rrbracket b$
$X \llbracket p - q \rrbracket\, b = X\llbracket p \rrbracket b \setminus X\llbracket q \rrbracket b$
$X \llbracket p \,.\, q \rrbracket\, b = \{\langle p_1,\ldots, p_{n-1}, q_2,\ldots, q_m \rangle \mid$
$\quad \langle p_1,\ldots, p_n \rangle \in X\llbracket p \rrbracket b \wedge \langle q_1,\ldots, q_m \rangle \in X\llbracket q \rrbracket b$
$\quad \wedge\ p_n = q_1\}$
$X \llbracket p \text{ –> } q \rrbracket\, b = \{\langle p_1,\ldots, p_n, q_1,\ldots, q_m \rangle \mid$
$\quad \langle p_1,\ldots, p_n \rangle \in X\llbracket p \rrbracket b \wedge \langle q_1,\ldots, q_m \rangle \in X\llbracket q \rrbracket b\}$
$X \llbracket \tilde{}\,p \rrbracket\, b = \{\langle p_2, p_1 \rangle \mid \langle p_1, p_2 \rangle \in X\llbracket p \rrbracket b\}$
$X \llbracket \hat{}\,p \rrbracket\, b = \{\langle x, y \rangle \mid \exists\ p_1,\ldots, p_n \mid$
$\quad \langle x, p_1 \rangle, \langle p_1, p_2 \rangle,\ldots, \langle p_n, y \rangle \in X\llbracket p \rrbracket b\}$
$X \llbracket \{v\text{: } p \mid F\} \rrbracket\, b = \{\langle x \rangle : (X\llbracket p \rrbracket b) \mid$
$\quad M\llbracket F \rrbracket(b \oplus (v \mapsto x))\}$
$X \llbracket r \rrbracket\, b = b(r)$
$X \llbracket v \rrbracket\, b = b(v)$

**(c)**

**Figure 1: Abstract syntax, well-formedness rules, and semantics of the logic**

## 3. MODEL FINDING BASICS

A *formula* in relational logic is a sentence over an alphabet of relational variables. A *model*, or an *instance*, of a formula is a binding of the formula's free variables to *relational constants*, which makes the formula true. Relational constants are sets of *tuples* drawn from a universe of uninterpreted atoms. An engine that searches for models of a formula in a finite universe is called a *finite model finder* or, simply, a model finder.

Figure 1 defines the abstract syntax, well-formedness rules, and semantics of the relational logic used in the rest of this paper. The definition closely corresponds to that of the Alloy language [10, 11], with its support for relations of arbitrary arity, treatment of sets as unary relations, and representation of scalars as singleton sets. Unlike Alloy, however, the logic in Figure 1a provides a handle to the universe of discourse for a given formula. This is the crux of the mechanism for specifying partial solutions.

### 3.1 Abstract Syntax

A *problem* in our logic (Figure 1a) is a *universe declaration*, a set of *relation declarations*, and a *formula* in which the declared relations appear as free variables. Each relation declaration specifies the arity of a relation variable and bounds on its value. The lower bound contains the tuples which the variable's value *must* include in an instance of the formula. The upper bound holds the tuples which the variable's value *may* contain in an instance. The tuples of all constants are drawn from the problem's universe.

For example, the pigeonhole principle says that $n$ pigeons cannot be placed into $n-1$ holes so that each pigeons has a hole to itself. Taking $n$ to be three, we can state the principle as a model finding problem using the following formulation:[1]

{P1, P2, P3, H1, H2}

Pigeon $:_1$ [{⟨P1⟩⟨P2⟩⟨P3⟩}, {⟨P1⟩⟨P2⟩⟨P3⟩}]
Hole $:_1$ [{⟨H1⟩⟨H2⟩}, {⟨H1⟩⟨H2⟩}]
nest $:_2$ [{}, {⟨P1, H1⟩⟨P1, H2⟩⟨P2, H1⟩⟨P2, H2⟩⟨P3, H1⟩⟨P3, H2⟩}]

(all p : Pigeon | one p.nest) and
(all h : Hole | one nest.h or no nest.h)

[1] The user of our tool actually constructs problems through an API, which provides convenience methods for declaring bounds on relational variables.

The first line describes a universe of five uninterpreted atoms. We arbitrarily chose the first three of them to represent pigeons and the last two to represent holes. Because formulas cannot contain constants, a relational variable $v :_k [C, C]$ with the same upper and lower bound is declared for each $k$-arity constant $C$ that needs to be accessed in a problem's formula. The variables Pigeon and Hole, for example, serve as handles to the unary constants $\{\langle\texttt{P1}\rangle\langle\texttt{P2}\rangle\langle\texttt{P3}\rangle\}$ and $\{\langle\texttt{H1}\rangle\langle\texttt{H2}\rangle\}$, which represent the sets of all pigeons and holes respectively. The variable nest ⊆ Pigeon × Hole encodes the placement of pigeons into holes. Its value is constrained to be an injection by the formulas (all p : Pigeon | one p.nest) and (all h : Hole | one nest.h or no nest.h).

The syntactic productions (Figure 1a) other than the universe and relation declarations define a standard relational logic with transitive closure, first order quantifiers, and connectives. Expressions, formulas, and declarations are considered well-formed if they adhere to the arity rules in Figure 1b. The arity of the empty set constant, {}, is polymorphic, making it a valid bound in the context of any declaration.

## 3.2 Semantics

The meaning of a problem is determined by recursive application of four meaning functions: $P$, $R$, $M$ and $X$. The functions $R$ and $M$ evaluate relation declarations and formulas with respect to a *binding* of variables to constants. The function $P$ deems a problem true with respect to a given binding if and only if its declarations and formula are true under that binding. The function $X$ interprets expressions as sets of tuples. We take the meanings of atoms, tuples, and constants to be their standard set-theoretic interpretations. That is, the meaning of an atom is its name, the meaning of a tuple is a sequence of atoms, and the meaning of a constant is a set of tuples.

# 4. ANALYSIS

The abstract syntax presented in the previous section should be thought of as describing a directed acyclic graph rather than a tree. The user of our model finder constructs the syntactic graph of a problem's formula via API calls instead of creating a string that is parsed into an abstract syntax tree. The analysis of a well-formed problem graph involves five steps.

1. Two simple analyses are performed on the problem's formula: detection of syntactically shared subgraphs and detection of syntactic patterns that indicate a relational variable represents an acyclic graph or a total order.
2. The problem's relation declarations are analyzed to find *atom symmetry classes* (Section 4.1).[2]
3. Using the information from Step 1, the problem is translated into a Compact Boolean Circuit (Section 4.2). The circuit is constructed in such a way that it is satisfiable if and only if the problem itself is satisfiable.
4. The circuit is conjoined with a lex-leader symmetry breaking predicate [6], computed using the information from Step 2, and translated to conjunctive normal form (Section 4.3).
5. The CNF is handed to a SAT solver. If it is satisfiable, the CNF model is translated into a model of the original problem, using the mapping from relational to boolean variables generated in Step 3.

Step 1 of the analysis performs a depth-first search for the shared subgraphs. This information is used to ensure that the shared components are visited only once in the later phases of the analysis. The second analysis done in Step 1 is a simple depth-first pattern recognizer. Steps 2 through 4 are the focus of the remainder of this section. Step 5 is trivial.

## 4.1 Symmetry Detection Without Types

Many problems exhibit symmetries. For example, all Mars bars in a vending machine are interchangeable, as are all the white pawns in a game of chess. Such symmetries among a problem's atoms give rise to isomorphisms among its models [30]. Two models are considered isomorphic if one can be obtained from the other by a relabeling of atoms. More formally, we define the symmetries of a problem and the isomorphisms they induce as follows.

*Definition 1.* Let $\mathcal{A}$ be a universe, $D$ a set of declarations over $\mathcal{A}$, and $F$ a formula over $D$. A permutation $l : \mathcal{A} \to \mathcal{A}$ is a *symmetry* of the problem $P = (\mathcal{A}, D, F)$ if and only if $l(M)$ is a model of $P$, written $l(M) \models P$, for all $M \models P$. The models $M$ and $l(M)$ are said to be *isomorphic.*

The set of symmetries of $P$, denoted by $\text{Sym}(P)$, is an equivalence relation on the bindings $B$ that map the variables declared in $D$ to sets of tuples drawn from $\mathcal{A}$. Two bindings $b_x, b_y \in B$ are in the same equivalence class if $b_x = l(b_y)$ for some $l \in \text{Sym}(P)$. Each $l \in \text{Sym}(P)$ maps bindings that are models of $P$ to other models of $P$ and bindings that do not satisfy $P$ to other non-models. It is therefore sufficient to test one binding in each equivalence class induced by $\text{Sym}(P)$ to find a model of $P$. This insight is key to efficient model finding. In most cases, elimination of isomorphic models, or *symmetry breaking*, speeds up the search by orders of magnitude [40, 39, 31, 29, 5]. Many problems, such as the pigeonhole problem, are intractable without symmetry breaking [6].

There are two ways in which a model finder can exploit symmetry information. One is to use a specialized search algorithm that examines as few members of the same equivalence class as possible. The Nitpick [18], FALCON [39], and SEM [40] model finders use this approach. Another is to translate the model finding problem $P$ to conjunctive normal form $\text{CNF}(P)$, conjoin the translation with a *symmetry breaking predicate* $\text{SBP}(P)$ that is true of at least one binding per symmetry class [6], and solve the resulting propositional formula with an off-the-shelf SAT solver. Although larger than $\text{CNF}(P)$, the formula $\text{CNF}(P) \wedge \text{SBP}(P)$ is solved orders of magnitude faster because it is more constrained [30]. The Alloy Analyzer [31] and the Paradox [5] model finder use the predicate approach to symmetry breaking.

To use either approach, however, a model finder must first detect the symmetries of a problem. In the case of a standard typed logic such as the Alloy language or SEM's logic, symmetry detection in a universe of uninterpreted atoms is straightforward: $\text{Sym}(P)$ is the set of all permutations that map an atom of $\mathcal{A}$ to itself or to another atom of the same type. Atoms of the same type are interchangeable

[2] If the bounds on a total order or an acyclic relation found in Step 1 range over a symmetry class, they are minimized according to the rules in [30] before proceeding to Step 3.

because neither logic provides a means of referring to individual atoms. Our logic does (Figure 3.1), so even if it were typed, atoms of the same type would not necessarily be interchangeable. Below is a toy specification of a traffic lights system showing a case where the natural typing of atoms does not partition $\mathcal{A}$ into equivalence classes.

```
{N, E, G, Y, R}
Green   :1 [{⟨G⟩}, {⟨G⟩}]
Light   :1 [{⟨N⟩⟨E⟩}, {⟨N⟩⟨E⟩}]
display :2 [{}, {⟨N, G⟩⟨N, Y⟩⟨N, R⟩⟨E, G⟩⟨E, Y⟩⟨E, R⟩}]

(all light: Light | one light.display) and
(lone Light.display & Green)
```

The traffic-system universe consists of five atoms that are conceptually partitioned into two 'types': the atoms representing the stop lights at an intersection (north-south and east-west) and the atoms representing the colors red, green, and yellow. The formula constrains each light to display a color and requires that at most one of the displayed colors be Green. The stop-light atoms form an equivalence class, but the color atoms do not. In particular, only Y and R are interchangeable. To see why, consider the following model of the problem:

$$b = \{\mathsf{Green}\mapsto\{\langle \mathtt{G}\rangle\},\ \mathsf{Light}\mapsto\{\langle \mathtt{N}\rangle\langle \mathtt{E}\rangle\},\ \mathsf{display}\mapsto\{\langle \mathtt{N},\ \mathtt{Y}\rangle\langle \mathtt{E},\ \mathtt{G}\rangle\}\}.$$

Applying the permutations $l_1 = (\mathtt{N}\ \mathtt{E})(\mathtt{Y}\ \mathtt{R})$ and $l_2 = (\mathtt{G}\ \mathtt{Y}\ \mathtt{R})$[3] to $b$, we get

$$l_1(b) = \{\mathsf{Green}\mapsto\{\langle \mathtt{G}\rangle\},\ \mathsf{Light}\mapsto\{\langle \mathtt{E}\rangle\langle \mathtt{N}\rangle\},\ \mathsf{display}\mapsto\{\langle \mathtt{E},\ \mathtt{R}\rangle\langle \mathtt{N},\ \mathtt{G}\rangle\}\},$$
$$l_2(b) = \{\mathsf{Green}\mapsto\{\langle \mathtt{Y}\rangle\},\ \mathsf{Light}\mapsto\{\langle \mathtt{N}\rangle\langle \mathtt{E}\rangle\},\ \mathsf{display}\mapsto\{\langle \mathtt{N},\ \mathtt{R}\rangle\langle \mathtt{E},\ \mathtt{Y}\rangle\}\}.$$

The binding $l_1(b)$ is a model of the problem, but $l_2(b)$ is not because it violates the constraint $\{\langle \mathtt{G}\rangle\} \subseteq \mathsf{Green} \subseteq \{\langle \mathtt{G}\rangle\}$ imposed by the declaration of Green.

The traffic lights example reveals two important properties of declarations and formulas. First, a permutation is a symmetry of a set of declarations $D$ if it maps the constants occurring in $D$ to themselves (Lemma 1). The permutation $l_1$, for example, is a symmetry of the traffic-lights declarations (i.e. $l_1(b) \models D$ for all $b \models D$). Second, *any* permutation is a symmetry of a formula (Lemma 2). The permutation $l_2(b)$ is a model of the traffic-lights formula even though it is not a model of the entire problem.

*Definition 2.* A permutation $l : \mathcal{A} \to \mathcal{A}$ *fixes* the atom $a \in \mathcal{A}$ if $l(a) = a$. Similarly, $l$ fixes a subset $s$ of $\mathcal{A}$ if $s = l(s) = \{l(a) \mid a \in s\}$, and $l$ fixes the set of $k$-tuples $c$ drawn from $\mathcal{A}$ if $c = l(c) = \{\langle l(e_1), \ldots, l(e_k)\rangle \mid \langle e_1, \ldots, e_k\rangle \in c\}$.

LEMMA 1. *Let $D = \{r_1 :_{k_1} [c_1, c_2], \ldots, r_m :_{k_m} [c_{2m-1}, c_{2m}]\}$ be a set of declarations over $\mathcal{A}$ and $l : \mathcal{A} \to \mathcal{A}$ a permutation. If $l$ fixes $c_1, \ldots, c_{2m}$ and $M \models D$, then $l(M) \models D$.*

PROOF (BY CONTRADICTION). Suppose that there is some model $M$ such that $M \models D$ but $l(M) \not\models D$. Then, there must be some $r_j \in r_1, \ldots, r_m$ such that $c_{2j-1} \not\subseteq l(M)(r_j)$ or $l(M)(r_j) \not\subseteq c_{2j}$. Since $M \models D$, $M$ must map $r_j$ to $C_j$ such that $c_{2j-1} \subseteq C_j \subseteq c_{2j}$. Hence, $C_j = c_{2j-1} \cup x$ where $x = C_j - c_{2j-1}$. This gives us $l(M)(r_j) = l(C_j) = l(c_{2j-1} \cup x) = l(c_{2j-1}) \cup l(x) = c_{2j-1} \cup l(x)$, which implies that $c_{2j-1} \subseteq l(M)(r_j)$. But, since $l$ maps subsets of $c_{2j}$ to subsets of $c_{2j}$, we also have $(C_j \subseteq c_{2j}) \Rightarrow (l(C_j) \subseteq c_{2j}) \Rightarrow (l(M)(r_j) \subseteq c_{2j})$, a contradiction. □

LEMMA 2. *If $l : \mathcal{A} \to \mathcal{A}$ is a permutation of $\mathcal{A}$ and $M \models F$, then $l(M) \models F$.*

PROOF. By definition (Figure 3.1), $F$ contains no direct references to individual atoms, tuples, or sets of tuples. Therefore, the meaning of $F$ depends *solely* on the meanings of the relational variables mapped by $M$. Since the application of $l$ to the bindings in $M$ gives an isomorphic set of bindings, $l(M) \models F$. □

With a bit of extra work, we can turn the observations about the symmetries of declarations and formulas into a simple criterion for deciding whether a permutation $l$ is a symmetry of a problem. Namely, $l \in \mathrm{Sym}(P)$ for all $P = (\mathcal{A}, D, F)$ if and only if $l$ maps each constant that occurs in $D$ to itself (Thm. 1).

THEOREM 1 (DETECTION CRITERION). *Let $\mathcal{A}$ be the universe of discourse and $D = \{r_1 :_{k_1} [c_1, c_2],\ r_2 :_{k_2} [c_3, c_4], \ldots, r_m :_{k_m} [c_{2m-1}, c_{2m}]\}$ a set of declarations over $\mathcal{A}$. The permutation $l : \mathcal{A} \to \mathcal{A}$ is a symmetry for all problems $P$ and formulas $F$ such that $P = (\mathcal{A},\ D,\ F)$ if and only if $l$ fixes $c_1, c_2, \ldots, c_{2m}$.*

PROOF. According to the semantics of $P$, a binding $M$ is a model of $P$ iff $M \models D$ and $M \models F$. By Def. 1, $l \in \mathrm{Sym}(P)$ iff $l(M) \models P$. Lemmas 1 and 2 therefore prove one half of the theorem: if $l$ fixes $c_1, \ldots, c_{2m}$, then $l(M) \models D$ and $l(M) \models F$ for all models $M$ of $P$.

We prove by contradiction that $l \in \mathrm{Sym}(P)$ for all $P = (\mathcal{A}, D, F)$ implies that $l$ fixes $c_1, \ldots, c_{2m}$. Suppose $l$ is a symmetry for all $P$ defined in terms of $D$, and there is a $c \in \{c_1, \ldots, c_{2m}\}$ such that $l(c) \neq c$. Clearly, $c$ is non-empty and there is an $r_i$ such that $r_i :_{k_i} [c, c_{2i}] \in D$ or $r_i :_{k_i} [c_{2i-1}, c] \in D$ or $r_i :_{k_i} [c, c] \in D$. Let $F^*$ be the formula "some $r_i$", $P^* \in P$ the problem $(\mathcal{A}, D, F^*)$, and $M$ the binding that maps $r_i$ to $c$ and all the other variables to the empty set. Since $M(r_i) = c$, $l(M(r_i)) = l(c)$. Given that $l$ is a permutation, $l(c) \neq c \Rightarrow c \not\subseteq l(M(r_i)) \wedge l(M(r_i)) \not\subseteq c$. Hence, $M \models P^*$ and $l(M) \not\models P^*$, a contradiction. □

The problem of symmetry detection as stated in Thm. 1 is equivalent to that of testing graph isomorphism. In particular, testing if $l \in \mathrm{Sym}(P)$ is the same as testing if $l$ is a symmetry, or automorphism, of the graphs $G = (V, E)$ that correspond to the constants used in $D$. The procedure in Figure 2 demonstrates that any constant can be turned into a graph. The illustration shows the result of applying CONSTANT-TO-GRAPH to the ternary constant $\{\langle \mathtt{G}, \mathtt{Y}, \mathtt{R}\rangle\langle \mathtt{R}, \mathtt{Y}, \mathtt{G}\rangle\}$.

There is no known polynomial time algorithm for deciding if two graphs are isomorphic. The best known bound on testing if an $n$-vertex graph $G$ is isomorphic to $l(G)$ (i.e. if $l$ is an automorphism of $G$) is $\exp(O(\sqrt{n \log n}))$ [4]. In practice, however, systems like Nauty [26] can find graph automorphisms very efficiently. Still, the cost of converting constants to graphs and then running Nauty on them can easily become prohibitive. For example, the graph representation of a 4-arity relation over a domain of size 10 (which we have encountered in practice) consists of about 40,000 nodes and 70,000 edges.

[3] Recall that cycle notation for permutations [3] indicates that each element in a pair of parenthesis is mapped to the one following it, with the last element being mapped to the first. The elements that are fixed under a permutation are not mentioned.

```
CONSTANT-TO-GRAPH($\mathcal{A}$ : universe, $C$ : set of tuples)
$V \leftarrow \mathcal{A}$
$E \leftarrow \{\}$
$k \leftarrow \text{arity}(C)$
for all $\langle e_1, e_2, \ldots, e_k \rangle \in C$ do
    $v \leftarrow \text{vertex}[1..k]$ ▷ an array of $k$ new vertices
    for all $1 \le i \le k$ do
        $V \leftarrow V \cup \{v[i]\}$
        $E \leftarrow E \cup \{\text{edge}(v[i], e_i)\}$
    for all $1 \le i < k$ do
        $E \leftarrow E \cup \{\text{edge}(v[i], v[i+1])\}$
return $(V, E)$
```

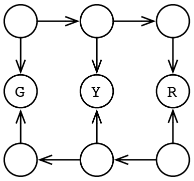


**Figure 2: Converting a Set of Tuples to a Graph**

Instead of using graph algorithms to partition a problem's universe into the optimal (coarsest) set of equivalence classes, we use the algorithm in Figure 3 to efficiently produce a sound, locally optimal partitioning. Because the constants in most declarations are not arbitrary graphs, our local optimality condition yields the coarsest partitioning for many problems (e.g. all problems in Section 5). In particular, most constants are expressible as unions of products of 'types' that have zero or more 'distinguished', non-symmetric atoms. For example, the bounds on the variables in the traffic lights problem can be expressed as Green $= T_{\{\mathtt{G}\}}$, Light $= T_{\text{light}}$, and display $\subseteq T_{\text{light}} \times T_{\{\mathtt{R},\mathtt{Y}\}} \cup T_{\text{light}} \times T_{\{\mathtt{G}\}}$, where the 'types' are $T_{\text{light}} = \{\mathtt{N}, \mathtt{E}\}$ with no distinguished atoms and $T_{\text{color}} = T_{\{\mathtt{R},\mathtt{Y}\}} \cup T_{\{\mathtt{G}\}} = \{\mathtt{G}, \mathtt{Y}, \mathtt{R}\}$ with the distinguished atom G. Our algorithm essentially infers a problem's conceptual types while isolating the non-symmetric atoms. Hence, the partitions inferred for the traffic lights problem are {N, E}, {R, Y}, and {G}.

The procedure DETECT-SYMMETRIES works on the compact representation of constants described in Section 4.2.1. It takes as arguments a universe $\mathcal{A}$ and a set of declarations $D$ over that universe, and returns the coarsest *base partitioning* (Def. 3) of $\mathcal{A}$ with respect to the constants in $D$. The correctness of DETECT-SYMMETRIES algorithm follows easily by induction from Theorems 2 and 3. The latter tells us that each call to REFINE-PARTITIONS generates the coarsest base partitioning of $\mathcal{A}$ with respect to the visited constants, and the former that a base partitioning is sound. It is also not difficult to see that the worst case running time of the algorithm is polynomial in the total number of nodes required to represent the constants in $D$ as graphs.

*Definition 3.* Let $\mathcal{A}$ be the universe of discourse, $C$ a constant (set of tuples) over $\mathcal{A}$, and $S = \{S_1, S_2, \ldots, S_n\}$ a set of sets that partition $\mathcal{A}$. We say that $S$ is a *base partitioning* of $\mathcal{A}$ with respect to $C$ if $C$ can be expressed as a union of products of elements in $S \cup \{\emptyset\}$, i.e.:

$$\textbf{let } k = \text{arity}(C) \textbf{ in } \exists x \geq 1 \mid \exists s_1, \ldots, s_{xk} \in S \cup \{\emptyset\} \mid C = \bigcup_{j=0}^{x-1} (s_{jk+1} \times \ldots \times s_{jk+k}).$$

THEOREM 2 (SOUNDNESS). *Let $D = \{r_1 :_{k_1} [c_1, c_2], \ldots, r_m :_{k_m} [c_{2m-1}, c_{2m}]\}$ be a set of declarations over $\mathcal{A}$ and $S = \{S_1, S_2, \ldots, S_n\}$ a base partitioning of $\mathcal{A}$ with respect to the constants $c_1, \ldots, c_{2m}$. If a permutation $l : \mathcal{A} \rightarrow \mathcal{A}$ fixes all $S_i \in S$, then $l \in \text{Sym}(P)$ for all problems $P$ and formulas $F$ such that $P = (\mathcal{A}, D, F)$.*

```
DETECT-SYMMETRIES($\mathcal{A}$ : universe, $D$ : declarations)
 1  $S \leftarrow \{\mathcal{A}\}$   ▷ $S$ is the set of set $\mathcal{A}$
 2  for all $r :_k [c_L, c_U] \in D$ do
 3      $S \leftarrow$ REFINE-PARTITIONS($c_L, S$)
 4      $S \leftarrow$ REFINE-PARTITIONS($c_U, S$)
 5  return $S$

REFINE-PARTITIONS($C$ : set of tuples, $S$ : set of sets)
 6  $S' \leftarrow \{\}$
 7  $k \leftarrow \text{arity}(C)$
 8  $C_{first} \leftarrow \pi_1(C)$   ▷ projects over the 1st column of $C$
 9  for all $s \in S$ do
10      if $s \subseteq C_{first}$ or $s \cap C_{first} = \emptyset$
11      then  $S' \leftarrow S' \cup \{s\}$
12      else  $S' \leftarrow S' \cup \{s \cap C_{first}\} \cup \{s - C_{first}\}$ ▷ splits $s$
13  if $k > 1$ then
14      $C_{rest} \leftarrow \pi_{2,\ldots,k}(C)$
15      $P_{first} \leftarrow \{s : S' \mid s \cap C_{first} \neq \emptyset\}$  ▷ gets partitions of $C_{first}$
16      $P_{rest} \leftarrow \{\}$
17      for all $p \in P_{first}$ do
18          $p_{part} \leftarrow \{\}$
19          $S' \leftarrow S' - p$
20          while $p \neq \emptyset$ do
21              $a \leftarrow \text{choose}(p)$   ▷ selects an atom of $p$
22              $a_{rest} \leftarrow \pi_{2,\ldots,k}((\{\langle a \rangle\} \times C_{rest}) \cap C)$
23              $a_{set} \leftarrow \{a' : p \mid a_{rest} = \pi_{2,\ldots,k}((\{\langle a' \rangle\} \times C_{rest}) \cap C)\}$
24              $p_{part} \leftarrow p_{part} \cup \{a_{set}\}$
25              $P_{rest} \leftarrow P_{rest} \cup \{a_{rest}\}$
26              $p \leftarrow p - a_{set}$
27          $S' \leftarrow S' \cup p_{part}$
28      for all $p \in P_{rest}$ do
29          $S' \leftarrow$ REFINE-PARTITIONS($p, S'$)
30  return $S'$
```

**Figure 3: Symmetry Detection Algorithm**

PROOF. Because $l$ fixes all $S_i \in S$, it maps any cross product of the sets in $S$ or a union of such cross products to itself. Together with the fact that $S$ is a base partitioning with respect to the constants in $D$, this gives us $l(c_i) = c_i$ for all $1 \leq i \leq 2m$. Hence, by Thm. 1, $l \in \text{Sym}(P)$. □

THEOREM 3 (LOCAL OPTIMALITY). *Let $\mathcal{A}$ be the universe of discourse, $C$ a constant (set of tuples) over $\mathcal{A}$, and $S = \{S_1, S_2, \ldots S_n\}$ a set of sets that partition $\mathcal{A}$. Applying* REFINE-PARTITIONS *to $C$ and $S$ will subdivide $S$ into the coarsest $S' = \{S'_1, S'_2, \ldots, S'_m\}$ that is a base partitioning of $\mathcal{A}$ with respect to $C$.*

PROOF. We proceed by induction on the arity of $C$.

BASE CASE. Let $\text{arity}(C) = 1$. If $C = \emptyset$ or $C$ is partitioned by a subset of $S$, then the condition on line 10 is always true, so $S' = S$ is returned. Suppose that $C$ is neither empty nor partitioned by a subset of $S$. Then, all $s \in S$ such that $s \cap C_{first} \neq \emptyset$ and $s \not\subseteq C_{first}$ are split into two pieces, $s \cap C_{first}$ and $s - C_{first}$ (line 12). The remaining elements of $S$ are passed into $S'$ unchanged. When the loop exists, $C$ is partitioned by a subset $s'$ of $S'$, and merging any of the split pieces back together would remove a non-empty element from $s'$. Hence, $S'$ is the coarsest base partitioning of $\mathcal{A}$ with respect to $C$ that can be derived from $S$.

INDUCTIVE CASE. Suppose that the theorem holds for all constants of arity $k$. Let $\text{arity}(C) = k + 1$. If $C = \emptyset$, we are done. So, assume that $C \neq \emptyset$. By the same reasoning used in the proof of the base case, we conclude that lines 9-12 subdivide $S$ into the minimal base partitioning with respect to the first column of $C$. On line 15, we obtain $P_{\mathit{first}}$, the subset of $S'$ that partitions $C_{\mathit{first}}$. Lines 17-27 process each $p \in P_{\mathit{first}}$ as follows. If all atoms in $p$ are mapped to the same set of tuples of arity $k-1$, $a_{rest}$, by $C$, then $a_{set} = p$, and $p$ makes it to $S'$ unchanged. If not, $p$ is split into $q$ partitions such that all atoms in a new partition have the same image under $C$.

LEMMA 3. *Let $p$ be an element of $P_{\mathit{first}}$ as defined on line 15. Let $p_1, \ldots, p_q$ be partitions of $p$ such that the atoms in each $p_i$ have the same image under $C$. No partitioning of $p$ coarser than $p_1, \ldots, p_q$ can exist in a base partitioning of $\mathcal{A}$ with respect to $C$ that is derived from $S$.*

PROOF (BY CONTRADICTION). Suppose that there are two pieces of $p$, $p_v$ and $p_w$, that can be merged into $p_z$ to form a coarser partitioning of $p$. Then, there are two cases to consider: (1) $p_z$ is partitioned during one of the recursive calls to REFINE-PARTITIONS (line 29), and (2) $p_z$ is not partitioned again. The first case trivially leads to a contradiction. The second case implies that there must be a set $X$ of arity $k-1$ such that $p_z \times X$ is a subset of $C$. But, $p_z \times X = (p_v \cup p_w) \times X = p_v \times X \cup p_v \times X$, contradicting the assumption that $p_v$ and $p_q$ have different images under $C$. □

By the inductive hypothesis, the recursive call on line 29 will subdivide $S'$ into the minimal number of partitions necessary to turn it into a base partitioning with respect to the images ($P_{rest}$) of the new partitions of $C_{\mathit{first}}$ under $C$. This, together with Lemma 3, proves the theorem. □

## 4.2 Translation to Boolean Logic

As shown in [14], a relational formula can be translated to a boolean formula that has a model exactly when the original formula has a model in a given universe of discourse. Using a similar approach, we translate a problem $P = (\mathcal{A}, D, F)$ to a Compact Boolean Circuit (Section 4.2.3) that has a model exactly when $P$ does. Here is how. Given a declaration $r :_k [c_L, c_U]$ over a universe $\mathcal{A} = \{a_0, \ldots, a_{n-1}\}$, we represent $r$ as a $k$-dimensional boolean matrix $m$ such that

$$m[i_1, \ldots, i_k] = \begin{cases} \mathsf{true} & \Leftrightarrow \langle a_{i_1}, \ldots, a_{i_k} \rangle \in c_L \\ \text{freshVar}() & \Leftrightarrow \langle a_{i_1}, \ldots, a_{i_k} \rangle \in c_U - c_L \\ \mathsf{false} & \phantom{\Leftrightarrow} \text{otherwise,} \end{cases}$$

where $i_1, \ldots, i_k \in [0 \mathinner{..} n)$ and freshVar() returns a fresh boolean variable. Expressions are then translated using matrix operations, and formulas become constraints over matrix entries.

Translation rules are given in Figure 4. The rules use *flat indices* to access matrix entries. The flat index $x$ for a vector index $[i_1, \ldots, i_k]$ is given by $\sum_{j=1}^{k} (i_j |\mathcal{A}|^{k-j})$. Both $x$ and $[i_1, \ldots, i_k]$ encode the structure of the $k$-tuple $t = \langle a_{i_1}, \ldots, a_{i_k} \rangle$ drawn from the universe $\mathcal{A} = \{a_0, \ldots, a_{n-1}\}$ that indexes its atoms starting at 0. The presence of $t$ in a relation $r$ is therefore designated by the boolean formula at the index $x$ (or $[i_1, \ldots, i_k]$) of the matrix that represents $r$. In fact, $x$ is exactly the integer representation of $t$ described below.

### *4.2.1 Representation of Constants*

A direct encoding of constants as sets of sequences of atoms would be expensive. For example, a direct representation of a $k$-arity constant over a domain of size $d$ would take up $kd^k$ space. We avoid this cost in practice by encoding tuples as integers and, having done that, representing constants as balanced interval trees [12].

The encoding reduces the space needed to represent a $k$-arity constant by a factor of $k$, if the universe of discourse is given as a random sequence of atoms or if the constant is an arbitrary graph. As noted in Section 4.1, however, universe declarations usually group atoms according to their conceptual types and constants are rarely arbitrary graphs. Under these conditions, our encoding actually reduces the space requirement from $kd^k$ to $d$ in most cases.

The key idea behind the encoding is the following: a $k$-tuple drawn from a universe of size $n$ can be uniquely represented as a $k$-digit number in base $n$ by concatenating the indices of the tuple's atoms within the universe. For example, consider the universe from the traffic-lights problem, {N, E, G, Y, R}. If we index the atoms N through R starting at 0, the tuple ⟨E, G⟩ becomes the number 12 in base 5. Similarly, the tuple ⟨E, Y⟩ becomes 13, and so on.

If the universe of discourse groups its atoms according to their conceptual types, consecutive tuples such as ⟨E, G⟩ and ⟨E, Y⟩ yield consecutive numbers in base $n$. Since base conversion preserves sequencing, we can convert all tuple representations to base 10 and encode constants as balanced interval trees of decimal numbers. The $2 \times 2 \times 3$-sized constant bounding the display variable in the traffic-lights problem thus becomes a tree with two nodes: [0, 4], representing the tuples {⟨N, G⟩⟨N, Y⟩⟨N, R⟩}, and [10, 14] representing {⟨E, G⟩⟨E, Y⟩⟨E, R⟩}.

### *4.2.2 Representation of Matrices*

A key difference between our translation and the one presented in [14] is that the latter is based on types. The Alloy Analyzer (AA), which implements the translation rules from [14], encodes a $k$-arity relation $r$ of type $T_1 \rightarrow \ldots \rightarrow T_k$ as a boolean matrix with dimensions $|T_1| \times \ldots \times |T_k|$. Since operands of many matrix operators must have particular dimensions, the operands of their corresponding relational operators are forced to have specific types. For example, in a world with three women and three men, AA's translator would reject the perfectly reasonable attempt to form the grandmother relation by joining the relation mother: Person → Woman with itself, because a $6 \times 3$ matrix cannot be multiplied by itself. There are two ways to remedy this problem: (1) force the type of mother up to Person → Person, doubling the size of its boolean representation, or (2) *atomize* mother into two pieces, $\mathsf{mother_w}$ : Woman → Woman and $\mathsf{mother_m}$ : Man → Woman, and split the expression mother.mother into $\mathsf{mother_w.mother_w} + \mathsf{mother_m.mother_w}$ before handing it to the translator [11]. AA takes the latter approach which, while elegant and efficient in theory, has not worked well in practice.

We avoid the problems of a type-based translation by encoding all $k$-arity relations over $\mathcal{A}$ as $|\mathcal{A}|_1 \times \ldots \times |\mathcal{A}|_k$ sparse matrices. The matrix $m$ created from the declaration $r :_k [c_L, c_U]$ is represented as a map from flat indices of the tuples in $c_U$ to their corresponding boolean formulas. The false entries representing the tuples outside of $c_U$ are not explicitly stored.

```
$\mathrm{T_P}$: problem $\rightarrow$ bool
$\mathrm{T_R}$: relDecl $\rightarrow$ univDecl $\rightarrow$ boolMatrix
$\mathrm{T_M}$: formula $\rightarrow$ env $\rightarrow$ bool
$\mathrm{T_X}$: expr $\rightarrow$ env $\rightarrow$ boolMatrix
env: (quantVar $\cup$ relVar) $\rightarrow$ boolMatrix
freshVar: boolVar

bool :=
  $\mathsf{true}$ | $\mathsf{false}$ | boolVar | $\neg$ bool
  bool $\wedge$ bool | bool $\vee$ bool
boolVar := identifier

$\mathrm{T_P}\ [\mathcal{A}\ \mathrm{d}_1\ \ldots\ \mathrm{d}_n\ \mathrm{F}] = \mathrm{T_M}\ [\mathrm{F}]\ (\bigcup_{i=1}^{m}\ (\mathrm{r}_i \mapsto \mathrm{T_R}\ [\mathrm{d}_i]\ \mathcal{A}))$

$\mathrm{T_R}\ [\mathrm{r}\ :_k\ [\mathrm{c}_L,\ \mathrm{c}_U]]\ \mathcal{A} =$
  $\mathrm{create}(|\mathcal{A}|^k,\ \lambda j.\ (\mathbf{let}\ t = \mathrm{index2tuple}(j, k, \mathcal{A})\ \mathbf{in}$
    if $t \in \mathrm{c}_L$ then $\mathsf{true}$
    else if $t \in \mathrm{c}_u - \mathrm{c}_L$ then freshVar()
    else $\mathsf{false}$))

$\mathrm{T_M}\ [\mathrm{p\ in\ q}]\ \mathrm{e} = \mathrm{fold}(\neg\mathrm{T_X}[\mathrm{p}]\mathrm{e} \vee \mathrm{T_X}[\mathrm{q}]\mathrm{e},\ \wedge)$
$\mathrm{T_M}\ [\mathrm{some\ p}]\ \mathrm{e} = \mathrm{fold}(\mathrm{T_X}[\mathrm{p}]\mathrm{e},\ \vee)$
$\mathrm{T_M}\ [\mathrm{one\ p}]\ \mathrm{e} = \mathbf{let}\ \mathrm{m} = \mathrm{T_X}[\mathrm{p}]\mathrm{e}\ \mathbf{in}$
  $\bigvee_{i=0}^{|m|-1}\ \mathrm{fold}(\neg\mathrm{mask}(\mathrm{m},\ i) \wedge \neg\mathrm{m} \vee \mathrm{mask}(\mathrm{m}, i) \wedge \mathrm{m},\ \wedge)$
$\mathrm{T_M}\ [\mathrm{no\ p}]\ \mathrm{e} = \mathrm{fold}(\neg\mathrm{T_X}[\mathrm{p}]\mathrm{e},\ \wedge)$
$\mathrm{T_M}\ [!\mathrm{F}]\ \mathrm{e} = \neg\ \mathrm{T_M}[\mathrm{F}]\mathrm{e}$
$\mathrm{T_M}\ [\mathrm{F}\ \&\&\ \mathrm{G}]\ \mathrm{e} = \mathrm{T_M}[\mathrm{F}]\mathrm{e} \wedge \mathrm{T_M}[\mathrm{G}]\mathrm{e}$
$\mathrm{T_M}\ [\mathrm{F}\ ||\ \mathrm{G}]\ \mathrm{e} = \mathrm{T_M}[\mathrm{F}]\mathrm{e} \vee \mathrm{T_M}[\mathrm{G}]\mathrm{e}$
$\mathrm{T_M}\ [\mathrm{all\ v\!:\ p}\ |\ \mathrm{F}]\ \mathrm{e} = \mathbf{let}\ \mathrm{m} = \mathrm{T_X}[\mathrm{p}]\mathrm{e}\ \mathbf{in}$
  $\bigwedge_{i=0}^{|m|-1}(\mathrm{m}[i] \Rightarrow \mathrm{T_M}[\mathrm{F}](\mathrm{e} \oplus \mathrm{v} \mapsto \mathrm{mask}(\mathrm{m},\ i)))$
$\mathrm{T_M}\ [\mathrm{some\ v\!:\ p}\ |\ \mathrm{F}]\ \mathrm{e} = \mathbf{let}\ \mathrm{m} = \mathrm{T_X}[\mathrm{p}]\mathrm{e}\ \mathbf{in}$
  $\bigvee_{i=0}^{|m|-1}(\mathrm{m}[i] \wedge \mathrm{T_M}[\mathrm{F}](\mathrm{e} \oplus \mathrm{v} \mapsto \mathrm{mask}(\mathrm{m},\ i)))$

$\mathrm{T_X}\ [\mathrm{p} + \mathrm{q}]\ \mathrm{e} = \mathrm{T_X}[\mathrm{p}]\mathrm{e} \vee \mathrm{T_X}[\mathrm{q}]\mathrm{e}$
$\mathrm{T_X}\ [\mathrm{p}\ \&\ \mathrm{q}]\ \mathrm{e} = \mathrm{T_X}[\mathrm{p}]\mathrm{e} \wedge \mathrm{T_X}[\mathrm{q}]\mathrm{e}$
$\mathrm{T_X}\ [\mathrm{p}\ \text{-}\ \mathrm{q}]\ \mathrm{e} = \mathrm{T_X}[\mathrm{p}]\mathrm{e} \wedge \neg\mathrm{T_X}[\mathrm{q}]\mathrm{e}$
$\mathrm{T_X}\ [\mathrm{p}\ .\ \mathrm{q}]\ \mathrm{e} = \mathrm{T_X}[\mathrm{p}]\mathrm{e} \cdot \mathrm{T_X}[\mathrm{q}]\mathrm{e}$
$\mathrm{T_X}\ [\mathrm{p}\ \text{->}\ \mathrm{q}]\ \mathrm{e} = \mathbf{let}\ \mathrm{m_p} = \mathrm{T_X}[\mathrm{p}]\mathrm{e},\ \mathrm{m_q} = \mathrm{T_X}[\mathrm{q}]\mathrm{e}\ \mathbf{in}$
  $\mathrm{create}(|\mathrm{m_p}| * |\mathrm{m_q}|,\ \lambda i.\ \mathrm{m_p}[\lfloor i/|\mathrm{m_q}|\rfloor] \wedge \mathrm{m_q}[i\%|\mathrm{m_q}|]\ \}$
$\mathrm{T_X}\ [\tilde{}\,\mathrm{p}\ ]\ \mathrm{e} = \mathbf{let}\ \mathrm{m} = \mathrm{T_X}[\mathrm{p}]\mathrm{e},\ s = \sqrt{|\mathrm{m}|}\ \mathbf{in}$
  $\mathrm{create}(|\mathrm{m}|,\ \lambda i.\ \mathrm{m}[s * (i\%s) + \lfloor i/s \rfloor])$
$\mathrm{T_X}\ [\hat{}\,\mathrm{p}\ ]\ \mathrm{e} = \mathbf{let}\ \mathrm{m_p} = \mathrm{T_X}[\mathrm{p}]\mathrm{e},$
  $f = (\lambda m.i.\ \text{if } i = 0 \text{ then } m \text{ else } f(m \vee m \cdot m), \lceil i/2 \rceil))\ \mathbf{in}$
   $f(\mathrm{m_p}, \lceil \log_2 \sqrt{|\mathrm{m_p}|} \rceil)$
$\mathrm{T_X}\ [\{\mathrm{v\!:\ p}\ |\ \mathrm{F}\}]\ \mathrm{e} = \mathbf{let}\ \mathrm{m_p} = \mathrm{T_X}[\mathrm{p}]\mathrm{e}\ \mathbf{in}$
  $\mathrm{create}(|\mathrm{m_p}|,\ \lambda i.\ \mathrm{m_p}[i] \wedge \mathrm{T_M}[\mathrm{F}](\mathrm{e} \oplus \mathrm{v} \mapsto \mathrm{mask}(\mathrm{m_p},\ \mathrm{i}))\}$

index2tuple: int $\rightarrow$ int $\rightarrow$ univDecl $\rightarrow$ tuple
$\mathrm{index2tuple}(x,\ k,\ \{\mathrm{a}_0,\ \ldots,\ \mathrm{a}_{n-1}\}) =$
  $(\langle \mathrm{a}_{i_1},\ \ldots,\ \mathrm{a}_{i_k} \rangle\!:\ \mathrm{tuple}\ |\ x = \sum_{j=1}^{k}(i_j * n^{k-j}))$

create: int $\rightarrow$ (int $\rightarrow$ bool) $\rightarrow$ boolMatrix
$\mathrm{create}(s,\ f) =$
  $(m : \mathrm{boolMatrix}\ |\ |m| = s \wedge \forall i \in [0\,..\,s)\ |\ m[i] = f(i))$

fold: boolMatrix $\rightarrow$ (bool $\rightarrow$ bool $\rightarrow$ bool) $\rightarrow$ bool
fold: boolMatrix $\rightarrow$ (bool $\rightarrow$ bool $\rightarrow$ bool) $\rightarrow$ int $\rightarrow$ bool
$\mathrm{fold}(m,\ f) = \mathrm{fold}(m,\ f,\ 0)$
$\mathrm{fold}(m,\ f,\ i = |m| - 1) = m[i]$
$\mathrm{fold}(m,\ f,\ i \in [0\,..\,|m| - 1)) = f(m[i],\ \mathrm{fold}(m,\ f,\ i + 1))$

mask: boolMatrix $\rightarrow$ int $\rightarrow$ boolMatrix
$\mathrm{mask}(m,\ i) = \mathrm{create}(|m|,\ \lambda j.\ \text{if } i = j \text{ then } \mathsf{true} \text{ else } \mathsf{false})$
```

**Figure 4: Translation Rules and Matrix Operations**

### 4.2.3 Representation of Boolean Formulas

Formal specifications make frequent use of quantified formulas whose ground form contains many identical subcomponents. Detection and exploitation of this and other kinds of structural redundancy can greatly reduce the size of a problem's boolean encoding, leading to a more scalable analysis. Equivalent subformulas can be detected either at the problem level or at the boolean level. The Alloy Analyzer takes the former approach [32]. Our implementation uses Compact Boolean Circuits (Def. 5) to ensure that all *syntactically* equivalent ground formulas and expressions are translated into the same circuit. *Semantically* equivalent nodes are encoded using the same circuit if their equivalence can be established in a given number of steps (Def. 5, Property 3). CBCs also end up catching structural redundancies in the boolean representation itself that could not be detected at the problem level. The net result is a tighter encoding than can be generated using a problem-level detection mechanism.

*Definition 4.* A *Labeled Boolean Circuit* (LBC) is a directed, acyclic graph $(V, E)$. The set $V$ is partitioned into *operator* vertices $V_{\mathrm{op}}$, *variable* vertices $V_{\mathrm{var}}$, and the *constant* vertex $\top$. Each vertex $v \in V$ has an integer label $\mathit{label}(v)$. An operator vertex $v \in V_{\mathrm{op}}$ also has an operator attribute $\mathit{op}(v) \in \{\text{AND}, \text{OR}\}$ and two input edges, $\mathit{left}(v) \in E$ and $\mathit{right}(v) \in E$. An edge $e \in E$ has a source $\mathit{source}(e) \in V$, a target $\mathit{target}(e) \in V$, and a $\mathit{sign}(e) \in \{+, -\}$ designating the polarity of the source in the target formula. Vertex labels are constrained as follows:

· $\mathit{label}(\top) = 0$

· $\forall v \in V_{\mathrm{var}}\ |\ \mathit{label}(v) \geq 1$

· $\forall v \in V_{\mathrm{op}}\ |\ \mathit{label}(v) > \mathit{label}(\mathit{source}(\mathit{left}(v))) \wedge \mathit{label}(v) > \mathit{label}(\mathit{source}(\mathit{right}(v)))$

· $\forall v, w \in V |\ \mathit{label}(v) = \mathit{label}(w) \Rightarrow v = w$

*Definition 5.* A *Compact Boolean Circuit* (CBC) is an LBC with the following properties:

1. $\top \in V \Rightarrow V = \{\top\}$
2. $\forall v \in V_{\mathrm{op}}\ |\ \mathit{label}(\mathit{source}(\mathit{left}(v))) < \mathit{label}(\mathit{source}(\mathit{right}(v)))$
3. No vertex $v \in V$ can be simplified to a constant or to a vertex $w \in V - v$ by applying an equivalence law (i.e. absorption, idempotency, commutativity, associativity, complements, or distributivity) to the top $d \geq 1$ levels of the subgraph rooted at $v$.

An example of an LBC and its corresponding CBC for the formula $((((v_1 \wedge \neg v_2) \vee \neg v_3) \vee v_4) \wedge \neg(v_1 \wedge (\neg v_2 \wedge \mathsf{true})))$ is given in Figure 5. A square with a label $i$ corresponds to the variable $v_i$. The CBC (5b) simplifies the subcircuit $(v_1 \wedge (\neg v_2 \wedge \mathsf{true}))$ of the LBC (5a) to $(v_1 \wedge \neg v_2)$ and shares its output between the gates 6 and 8.

Properties 1-3 of CBCs are maintained in our implementation by a *factory* data structure, which synthesizes and caches CBCs. The factory creates a new circuit from given components only if it does not find an equivalent (up to depth $d$) one in its cache.

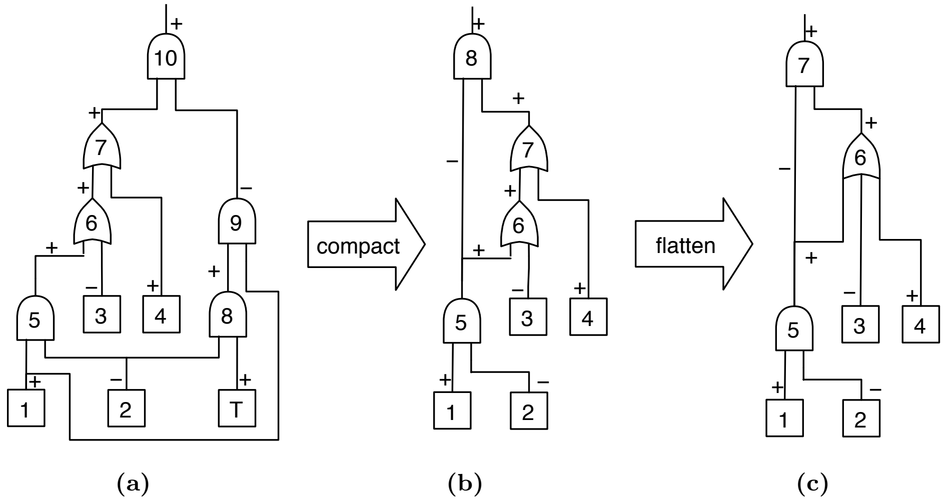
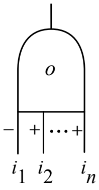
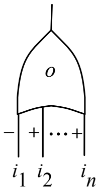


**Figure 5: A Labeled Boolean Circuit, Compacted and Flattened**

### 4.3 Translation to CNF

Before converting a CBC to CNF, we *flatten* it so that no unshared gate drives a gate of the same kind. For example, gate 6 of the CBC in Figure 5b is eliminated from the flattened circuit (5c), but gate 5 is not because it drives both gates 6 and 8. Flattening usually reduces the size of the final CNF by a couple of thousand variables and clauses. A flattened circuit is converted to conjunctive normal form using the standard transformation described in [37] and reproduced below.

| Sample Circuit | CNF clauses |
|---|---|
|  | $(\neg i_1 \vee \neg o) \wedge$<br>$(i_2 \vee \neg o) \wedge$<br>$\cdots$<br>$(i_n \vee \neg o) \wedge$<br>$(i_1 \vee \neg i_2 \vee \cdots \vee \neg i_n \vee o)$ |
|  | $(i_1 \vee o) \wedge$<br>$(\neg i_2 \vee o) \wedge$<br>$\cdots$<br>$(\neg i_n \vee o) \wedge$<br>$(\neg i_1 \vee i_2 \vee \cdots \vee i_n \vee \neg o)$ |

## 5. RESULTS

The analysis described in the previous section has been implemented in a model finder called Kodkod (KK). We have compared KK's performance to that of the Alloy Analyzer (AA) [14], MACE4 (M4) [25], and Paradox (PX) [5] on three sets of problems:[4]

- Constrained problems include a Sudoku puzzle from the M4 website, and the Tough Nut puzzle proposed by McCarthy [24] as a difficult problem for automated proof procedures. The Sudoku puzzle has exactly one solution. The Tough Nut puzzle is unsatisfiable. It proves that an 8 × 8 checkerboard with two opposite corners deleted cannot be tiled with dominos.
- Symmetric problems consist of two instances of the pigeonhole problem and two instances of the 'Ceilings and Floors' problem that comes with the AA distribution. 'Ceilings and Floors' is a toy problem inspired by a Paul Simon song. Like the pigeonhole problem, it is unsatisfiable and exhibits a high degree of symmetry.
- Design problems contain the formulations of Dijkstra's algorithm for mutex ordering [9] and the ring leader election algorithm described in [16]. We check that Dijkstra's algorithm prevents deadlocks, and that the leader election algorithm elects at most one leader. Both formulations require the use of transitive closure, which cannot be handled by M4 or PX.

The results are shown in Table 1. For each example, we show the size of the problem's CNF encoding and the total analysis time rounded to the nearest second. The Sudoku and Tough Nut problems have fixed universes of size nine and eight, respectively. Other problems have been analyzed in universes of varying sizes. All analyses were performed on a 1.33 GHz PowerPC with a 512 KB L2 cache and 1.25 GB RAM. AA and KK were instructed to use ZChaff [23] as their SAT engine. PX is packaged with Satnik [5], and M4 uses its own internal SAT solver. M4 does not report the size of the CNFs it generates, indicated by blank variable and clause entries. The analyses that did not complete within five minutes (> 300 seconds) were interrupted.

As indicated by the data in Table 1, KK significantly outperforms AA on both the problems with partial solutions (Sudoku and Pigeonhole) and the classic relational specifications for which Alloy was designed (Mutex Ordering for $|\mathcal{A}| \in \{30, 45\}$ and Ring Leader Election for $|\mathcal{A}| \in \{15, 30\}$). The first table shows that KK solves the Sudoku and Toughnut problems as fast as PX and M4, and generates much smaller CNFs for them than PX. In fact, KK's internal simplifications alone are sufficient to decide that Tough Nut is unsatisfiable (KK's variable and clause entries are 0 in the first table). The data in the second and third table demonstrate the effectiveness of our symmetry detection algorithm.

[4] Available at http://web.mit.edu/emina/www/problems/

| | Sudoku (9 × 9) | | | Tough Nut (8 × 8) | | |
|---|---|---|---|---|---|---|
| solver | time | vars | clauses | time | vars | clauses |
| AA | 38 | 53,037 | 545,070 | 82 | 0 | 0 |
| **KK** | **1** | **1,544** | **9,006** | **1** | **0** | **0** |
| M4 | $< 1$ | | | 12 | | |
| PX | 2 | 6,075 | 20,825 | $< 1$ | 749 | 1,212 |

| Ceilings and Floors | | | | | | |
|---|---|---|---|---|---|---|
| | 5 men, 5 platforms | | | 10 men, 10 platforms | | |
| solver | time | vars | clauses | time | vars | clauses |
| AA | 1 | 1,737 | 7,220 | 14 | 9,987 | 46,740 |
| **KK** | **1** | **1,082** | **3,990** | **14** | **6,317** | **25,095** |
| M4 | $> 300$ | | | $> 300$ | | |
| PX | $> 300$ | 1,363 | 8,775 | $> 300$ | 5,428 | 61,799 |

| Pigeonhole Problem | | | | | | |
|---|---|---|---|---|---|---|
| | 10 pigeons, 9 holes | | | 20 pigeons, 19 holes | | |
| solver | time | vars | clauses | time | vars | clauses |
| AA | 4 | 2,729 | 11,042 | 10 | 14,963 | 68,314 |
| **KK** | **1** | **1,127** | **3,855** | **2** | **6,937** | **24,855** |
| M4 | 50 | | | $> 300$ | | |
| PX | 9 | 1,962 | 12,698 | $> 300$ | 7,332 | 15,669 |

| Mutex Ordering (M) and Ring Leader Election (L) | | | | | | |
|---|---|---|---|---|---|---|
| | AA | | | KK | | |
| prob. | time | vars | clauses | time | vars | clauses |
| M30 | 112 | 74,818 | 722,236 | 16 | 20,973 | 154,540 |
| M45 | $> 300$ | - | - | 167 | 70,833 | 661,070 |
| L15 | 13 | 14,272 | 78,031 | 2 | 8,654 | 36,743 |
| L30 | $> 300$ | 223,582 | 1,866,996 | 23 | 101,399 | 502,953 |

**Table 1: Results**

M4 has no symmetry breaking mechanism. Although PX performs symmetry detection and breaking, its algorithm is limited by the fact that PX's logic offers no variable bounding scheme. All variables are assumed to range over the entire universe, making it difficult to detect equivalence between atoms.

While we are confident that the data in Table 1 presents a fair comparison between AA and KK, the comparison between KK and M4/PX should be taken with a grain of salt. First, the sample size is small (four problems). Because all four model finders have different input formats, four different formulations of each problem had to be constructed by hand, forcing us to choose a small set of representative examples to present in this paper. More importantly, the AA/KK logics, translation engines, design rationales, and intended uses are very different from those that characterize M4 and PX. It is not clear what would constitute a fair comparison between the AA/KK and M4/PX classes of tools. The data shown in Table 1 should therefore not be construed as a comparison between the AA/KK and M4/PX technologies, but rather as an additional indicator of the feasibility of the model finding approach embodied in KK.

We have used KK as a stand-alone model finder on these and many other problems. Our primary design goal, however, has been to create a relational engine that is scalable, lightweight, and easy to integrate into domain specific applications. Dennis et al [7] and Yeung [38] have used KK in this capacity with good results. Forge [7], is a static program analyzer that translates Java code into our logic and uses KK on the resulting problem to find execution traces that violate user-specified properties. It is the first static analyzer to automatically check several popular implementations of the `java.util.List` interface against its full JML specification [22]. Yeung's tool, a course scheduler [38], will soon be available as a web-based service for the MIT community. It takes as input courses taken by a student, a set of degree requirements, and a listing of offered subjects. These constraints are translated to our logic, and KK is used to construct the student's schedule for a given semester.

## 6. CONCLUSIONS

We have presented a collection of techniques that address the key design challenges in the construction of an efficient relational engine based on SAT technology. To be applicable to a wide range of problems, such an engine must have a mechanism for specifying partial solutions, an effective symmetry detection and breaking scheme, and an economical translation from the engine's input language to boolean logic. We have proposed a declaration construct that bounds the value of a relational variable by two constants as a simple mechanism for specifying partial solutions. A new symmetry detection and translation schemes that take advantage of partial solutions are described. Kodkod, a prototype implementation of these ideas, has been evaluated against other model finders based on SAT. The preliminary evaluation indicates that Kodkod significantly outperforms its closest relative, the Alloy Analyzer, and that it works as well as Paradox and MACE4.

Our future plans include further evaluation of the ideas presented here, and several improvements to our logic and analysis. The logic currently has no support for commonly used set operations that require integer manipulation, such as the computation of a set's cardinality. Our analysis represents all binary relations from a set $D$ to a set $R$ with $|D||R|$ bits. Functions, however, can be represented with $|D| \log |R|$ bits, a saving that has been demonstrated to significantly reduce model finding time for problems generated by static code checkers [35].

## 7. ACKNOWLEDGMENTS

We would like to thank Robert Seater, Greg Dennis, Mana Taghdiri, and Jonathan Edwards for their comments on various drafts of this paper.

## 8. REFERENCES

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *TACAS/ETAPS 2000, LNCS 1785*, pages 411–425, 2000.

[2] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *LICS: IEEE Symposium on Logic in Computer Science*, Warsaw, Poland, June 1997.

[3] M. A. Armstrong. *Groups and Symmetry.* Springer-Verlag, New York, 1988.

[4] L. Babai, W. M. Kantor, and E. M. Luks. Computational complexity and the classification of

finite simple groups. In *Proceedings of the IEEE Symposium Foundations of Computer Science*, pages 162–171. IEEE Computer Society Press, 1983.

[5] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *CADE-19 Workshop on Model Computation*, Miami, FL, July 2003.

[6] J. Crawford, M. L. Ginsberg, E. Luck, and A. Roy. Symmetry-breaking predicates for search problems. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, San Francisco, California, 1996.

[7] G. Dennis, F. Chang, and D. Jackson. Modular verification of code. In *International Symposium on Software Testing and Analysis*, Portland, Maine, July 2006.

[8] G. Dennis, R. Seater, D. Rayside, and D. Jackson. Automating commutativity analysis at the design level. In *ISSTA*, pages 165–174, 2004.

[9] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[10] J. Edwards, D. Jackson, and E. Torlak. A type system for object models. In *FSE*, Newport Beach, CA, November 2004.

[11] J. Edwards, D. Jackson, E. Torlak, and V. Yeung. Faster constraint solving with subtypes. In *ISSTA*, Boston, MA, July 2004.

[12] M. Erwig. Diets for fat sets. *Journal of Functional Programming*, 8(6):627–632, January 1998.

[13] M. Fujita, J. Slaney, and F. Bennett. Automating generation of some results in finite algebra. In *Thirteenth International Joint Conference on Artificial Intelligence*, Chambéry, France, 1993.

[14] D. Jackson. Automating first order relational logic. In *FSE*, San Diego, CA, November 2000.

[15] D. Jackson. Lightweight formal methods. In *FME*, page 1, 2001.

[16] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT Press, Cambridge, MA, 2006.

[17] D. Jackson and C. A. Damon. Elements of style: analyzing a software design feature with a counterexample detector. *IEEE TOSEM*, pages 484–495, July 1996.

[18] D. Jackson, S. Jha, and C. A. Damon. Isomorph-free model enumeration: a new method for checking relational specifications. *ACM Transactions on Programming Languages and Systems*, 20(2):302–343, March 1998.

[19] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *ESEC / SIGSOFT FSE*, pages 62–73, 2001.

[20] S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *ASE*, pages 13–22, 2000.

[21] S. Khurshid and D. Marinov. TestEra: Specification-based testing of java programs using sat. *ASE*, 11(4):403–434, 2004.

[22] G. T. Leavens. The java modeling language. Iowa State University, http://www.cs.iastate.edu/ leavens/JML/.

[23] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. In *SAT (Selected Papers)*, pages 360–375, 2004.

[24] J. McCarthy. A tough nut for proof procedures. Technical report, Stanford University, 1964.

[25] W. McCune. A Davis-Putnam program and its application to finite first-order model search: quasigroup existence problem. Technical report, Argonne National Laboratory, 1994.

[26] B. D. McKay. Practical graph isomorphism. *Congress Numerantium*, 30:45–87, 1981.

[27] S. Narain. Network configuration management via model finding. In *ACM Workshop On Self-Managed Systems*, Newport Beach, CA, October 2004.

[28] Y.-C. Ng. A Nitpick specification of IPv6. Senior Honors thesis, Computer Science Department, Carnegie Mellon University, 1997.

[29] A. Sabharwal. SymChaff: A structure-aware satisfiability solver. In *20th National Conference on Artificial Intelligence (AAAI)*, pages 467–474, Pittsburgh, PA, July 2005.

[30] I. Shlyakhter. Generating effective symmetry breaking predicates for search problems. *Electronic Notes in Discrete Mathematics*, 9, June 2001.

[31] I. Shlyakhter. *Declarative Symbolic Pure Logic Model Checking*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, February 2005.

[32] I. Shlyakhter, M. Sridharan, R. Seater, and D. Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Portofino, Italy, May 2003.

[33] J. K. Slaney. Finder: Finite domain enumerator - system description. In *CADE-12: Proceedings of the 12th International Conference on Automated Deduction*, pages 798–801, London, UK, 1994. Springer-Verlag.

[34] M. Taghdiri. Inferring specifications to detect errors in code. In *ASE*, pages 144–153, 2004.

[35] M. Vaziri. *Finding Bugs in Software with a Constraint Solver*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, February 2004.

[36] M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *TACAS*, pages 505–520, 2003.

[37] M. N. Velev. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, pages 310–315, January 2004.

[38] V. Yeung. Declarative configuration applied to course scheduling. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2006 (to appear).

[39] J. Zhang. *The generation and application of finite models*. PhD thesis, Institute of Software, Academia Sinica, Beijing, 1994.

[40] J. Zhang and H. Zhang. SEM: a system for enumerating models. In *IJCAI95*, Montreal, August 1995.