Kodkod: A Relational Model Finder

Emina Torlak and Daniel Jackson

MIT Computer Science and Artificial Intelligence Laboratory
`{emina,dnj}@mit.edu`

Abstract. The key design challenges in the construction of a SAT-based relational model finder are described, and novel techniques are proposed to address them. An efficient model finder 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.

1 Introduction

Many computational problems can be expressed declaratively as collections of constraints, and then solved using a 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 general purpose *relational engine*: that is, a model finder for a constraint language that combines first order logic with relational algebra and transitive closure.

A relational engine is well-suited to solving a wide range of problems. For example,

- Design analysis. A software design, modeled as a state machine over structured states (expressed as relations), can be checked, within finite bounds, for preservation of invariants 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.
- Scheduling and planning. For example, given the overall requirements and prerequisite dependences 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.

We have established the feasibility of using a relational engine for design analysis [1], code analysis [2,3] and test case generation [4] in earlier work. The prototype tool that we describe in this paper has been applied to design analysis, code analysis [5], and course scheduling [6]; it is also a mean Sudoku player.

Our earlier work involved the development of the Alloy modeling language [1] and its analyzer. The Alloy Analyzer was designed for the analysis of software models, and attempts to use it as a generic relational engine have been hampered by its lack of a mechanism for exploiting a priori knowledge about a problem's solution. The user provides only a constraint to be solved, and if a partial solution—or, a *partial instance*—is available which the obtained solution should extend, it can be provided only in the form of additional constraints. Because the solver must essentially rediscover the partial instance from the constraints, this strategy does not scale well.

Kodkod is a new tool that, unlike the Alloy Analyzer, is suitable as a generic relational engine. Kodkod outperforms the Analyzer dramatically on problems involving partial instances, and, due to improvements in the core technology that we describe, outperforms Alloy even on the problems for which Alloy was designed. It also outperforms other SAT-based logic engines (such as Paradox [7] and MACE [8]) on a variety of TPTP [9] benchmarks.

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 instances; the inability of Alloy's scheme to accommodate partial instances 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 [10].
- A new scheme for detecting opportunities for sharing in the constraint abstract syntax tree inspired by boolean expression diagrams [11] and reduced boolean circuits [12].

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 [13], 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 [7,8,14,15,16,17,18,19]. Several of these [16,17,18,19] implement specialized search algorithms for exploring the space of possible interpretations of a formula. The rest [7,8,14,15] 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 [14], MACE [8], FALCON [18], and SEM [19] have all been used to solve open problems in abstract algebra. Formulation of group-theoretic problems requires only a minimal logic. SEM and FINDER, for example, work on a quantifier-free many-sorted logic of uninterpreted functions. MACE and Paradox [7] support quantifiers, but none of these tools handle relational operators directly, which are indispensable for succinct description of systems whose state has a graph-like shape (such as networks or file systems) or for modeling programs with graph-like data structures (such as red-black trees or binomial heaps). Furthermore, lack of a closure operator, which cannot be encoded using first order constructs, makes it impossible to express common reachability constraints.

Nitpick [16] 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 [20,21]. The usefulness of Nitpick was, however, limited by its poor scalability and lack of support for quantifiers and higher-arity relations.

The Alloy language and its analyzer [15] addressed both the scalability and expressiveness limitations of Nitpick. The underlying logic supports first order quantifiers, connectives, arbitrary-arity relations, and transitive closure. Alloy has been applied to a wide variety of problems, including the design of an intentional naming scheme [22], the safety properties of the beam scheduler for a proton therapy machine [23], code analysis [2,3], test-case generation [4], and network configuration [24].

Alloy's main deficiency as a general-purpose problem description language is its lack of support for partial instances. Logic programming languages such as Prolog [25] and Oz [26] provide mechanisms for taking advantage of partial knowledge to speed up constraint solving, but they lack quantifiers, relational operators, and transitive closure. The logic presented in this paper is a superset of the Alloy language that provides a mechanism for specifying partial instances. Its accompanying model finder, Kodkod, takes advantage of known information, scaling much better than the Alloy Analyzer in the presence of partial instances. Kodkod outperforms the Alloy Analyzer even on the problems without partial solutions, due to the new translation to propositional satisfiability based on sparse matrices and a new data structure, Compact Boolean Circuits (CBCs).

Compact Boolean Circuits, described in Section 4.3, are a hybrid between Reduced Boolean Circuits (RBCs) [12] and Boolean Expression Diagrams (BEDs) [11]. Like RBCs, CBCs are a representational form for a quantifier-free logic, and they restrict variable vertices to the leaves of the graph. Like BEDs, CBCs use a more extensive set of operators and rules than RBCs to maximize subformula sharing. CBCs differ from both RBCs and BEDs in that their sharing detection algorithm is parameterized by a user-controlled variable. In particular, the user controls the trade-off between the speed of circuit construction and the size of the resulting circuit by determining the depth d to which syntactically distinct sub-circuits are checked for semantic equivalence. All three circuit representations can be straightforwardly converted one to another.

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. A relational constant is a set 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.

3.1 Abstract Syntax

A Kodkod *problem* (Fig. 1) consists of 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 relational 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 union of all relations' lower bounds forms a problem's *partial instance*. The upper bound holds the tuples which the variable's value *may* include in an instance. The elements of the tuples in a constant are drawn from the problem's universe.

To illustrate, consider the following formulation of the pigeonhole principle—n pigeons cannot be placed into $n-1$ holes with each pigeon having a hole to itself—for the case of 3 pigeons and 2 holes:

```
{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)
```

The first line declares 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 {⟨P1⟩⟨P2⟩⟨P3⟩} and {⟨H1⟩⟨H2⟩}, which represent the sets of all pigeons and holes respectively. The variable nest $\subseteq$ Pigeon $\times$ Hole encodes the placement of pigeons into holes. Its value is constrained to be an injection by the problem's formula.

The syntactic productions other than the universe and relation declarations define a standard relational logic with transitive closure, first order quantifiers, and connectives. The closure (ˆ) and transpose (˜) operators can only be applied to binary expressions. Mixed and zero arity expressions are not allowed. The arity of a relation variable and its declared bounds must match. The arity of the empty set constant, {}, is polymorphic, making it a valid bound in the context of any declaration.

```
problem := univDecl relDecl* formula

univDecl := { atom[, atom]* }
relDecl := rel :arity [constant, constant]
varDecl := var : expr

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

arity := 1 | 2 | 3 | 4 | ...
atom := identifier
rel := identifier
var := identifier
```

```
expr := rel | var | unary | binary | comprehension
unary := unop expr
unop := ~ | ^
binary := expr binop expr
binop := + | & | - | . | ->
comprehension := {varDecl | formula}

formula := elementary | composite | quantified
elementary := expr in expr | mult expr
mult := some | no | one
composite := not formula | formula logop formula
logop := and | or
quantified := quantifier varDecl | formula
quantifier := all | some
```

Fig. 1. Abstract syntax

$\mathrm{P} : \mathrm{problem} \to \mathrm{binding} \to \mathrm{boolean}$
$\mathrm{R} : \mathrm{relDecl} \to \mathrm{binding} \to \mathrm{boolean}$
$\mathrm{M} : \mathrm{formula} \to \mathrm{binding} \to \mathrm{boolean}$
$\mathrm{X} : \mathrm{expr} \to \mathrm{binding} \to \mathrm{constant}$
$\mathrm{binding} : (\mathrm{var} \cup \mathrm{rel}) \to \mathrm{constant}$

$\mathrm{P}[\![\mathcal{A}\ \mathrm{d}_1 \ldots \mathrm{d}_n\ \mathrm{F}]\!]\mathrm{b} =$
$\quad \mathrm{R}[\![\mathrm{d}_1]\!]\mathrm{b} \wedge \ldots \wedge \mathrm{R}[\![\mathrm{d}_n]\!]\mathrm{b} \wedge \mathrm{M}[\![\mathrm{F}]\!]\mathrm{b}$

$\mathrm{R}[\![\mathrm{r} : [\mathrm{c}_L, \mathrm{c}_U]]\!]\mathrm{b} = \mathrm{c}_L \subseteq \mathrm{b}(\mathrm{r}) \subseteq \mathrm{c}_U$

$\mathrm{M}[\![\mathrm{p\ in\ q}]\!]\mathrm{b} = \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b} \subseteq \mathrm{X}[\![\mathrm{q}]\!]\mathrm{b}$
$\mathrm{M}[\![\mathrm{some\ p}]\!]\mathrm{b} = \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b} \supset \emptyset$
$\mathrm{M}[\![\mathrm{one\ p}]\!]\mathrm{b} = |\mathrm{X}[\![\mathrm{p}]\!]\mathrm{b}| = 1$
$\mathrm{M}[\![\mathrm{no\ p}]\!]\mathrm{b} = \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b} \subseteq \emptyset$
$\mathrm{M}[\![\mathrm{not\ F}]\!]\mathrm{b} = \neg\, \mathrm{M}[\![\mathrm{F}]\!]\mathrm{b}$
$\mathrm{M}[\![\mathrm{F\ and\ G}]\!]\mathrm{b} = \mathrm{M}[\![\mathrm{F}]\!]\mathrm{b} \wedge \mathrm{M}[\![\mathrm{G}]\!]\mathrm{b}$
$\mathrm{M}[\![\mathrm{F\ or\ G}]\!]\mathrm{b} = \mathrm{M}[\![\mathrm{F}]\!]\mathrm{b} \vee \mathrm{M}[\![\mathrm{G}]\!]\mathrm{b}$

$\mathrm{M}[\![\mathrm{all\ v{:}\ p} \mid \mathrm{F}]\!]\mathrm{b} = \bigwedge(\mathrm{M}[\![\mathrm{F}]\!](\mathrm{b} \oplus \mathrm{v} \mapsto \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b}))$
$\mathrm{M}[\![\mathrm{some\ v{:}\ p} \mid \mathrm{F}]\!]\mathrm{b} = \bigvee(\mathrm{M}[\![\mathrm{F}]\!](\mathrm{b} \oplus \mathrm{v} \mapsto \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b}))$

$\mathrm{X}[\![\mathrm{p} + \mathrm{q}]\!]\mathrm{b} = \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b} \cup \mathrm{X}[\![\mathrm{q}]\!]\mathrm{b}$
$\mathrm{X}[\![\mathrm{p}\ \&\ \mathrm{q}]\!]\mathrm{b} = \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b} \cap \mathrm{X}[\![\mathrm{q}]\!]\mathrm{b}$
$\mathrm{X}[\![\mathrm{p} - \mathrm{q}]\!]\mathrm{b} = \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b} \setminus \mathrm{X}[\![\mathrm{q}]\!]\mathrm{b}$
$\mathrm{X}[\![\mathrm{p}\ .\ \mathrm{q}]\!]\mathrm{b} = \{\langle \mathrm{p}_1, \ldots, \mathrm{p}_{n-1}, \mathrm{q}_2, \ldots, \mathrm{q}_m \rangle \mid$
$\quad \langle \mathrm{p}_1, \ldots, \mathrm{p}_n \rangle \in \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b} \wedge \langle \mathrm{q}_1, \ldots, \mathrm{q}_m \rangle \in \mathrm{X}[\![\mathrm{q}]\!]\mathrm{b}$
$\quad \wedge\ \mathrm{p}_n = \mathrm{q}_1\}$
$\mathrm{X}[\![\mathrm{p} \mathbin{{-}{>}} \mathrm{q}]\!]\mathrm{b} = \{\langle \mathrm{p}_1, \ldots, \mathrm{p}_n, \mathrm{q}_1, \ldots, \mathrm{q}_m \rangle \mid$
$\quad \langle \mathrm{p}_1, \ldots, \mathrm{p}_n \rangle \in \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b} \wedge \langle \mathrm{q}_1, \ldots, \mathrm{q}_m \rangle \in \mathrm{X}[\![\mathrm{q}]\!]\mathrm{b}\}$
$\mathrm{X}[\![\tilde{}\,\mathrm{p}]\!]\mathrm{b} = \{\langle \mathrm{p}_2, \mathrm{p}_1 \rangle \mid \langle \mathrm{p}_1, \mathrm{p}_2 \rangle \in \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b}\}$
$\mathrm{X}[\![\hat{}\,\mathrm{p}]\!]\mathrm{b} = \{\langle \mathrm{x}, \mathrm{y} \rangle \mid \exists\ \mathrm{p}_1, \ldots, \mathrm{p}_n \mid$
$\quad \langle \mathrm{x}, \mathrm{p}_1 \rangle, \langle \mathrm{p}_1, \mathrm{p}_2 \rangle, \ldots, \langle \mathrm{p}_n, \mathrm{y} \rangle \in \mathrm{X}[\![\mathrm{p}]\!]\mathrm{b}\}$
$\mathrm{X}[\![\{\mathrm{v{:}\ p} \mid \mathrm{F}\}]\!]\mathrm{b} = \{\langle \mathrm{x} \rangle : (\mathrm{X}[\![\mathrm{p}]\!]\mathrm{b}) \mid$
$\quad \mathrm{M}[\![\mathrm{F}]\!](\mathrm{b} \oplus (\mathrm{v} \mapsto \mathrm{x}))\}$
$\mathrm{X}[\![\mathrm{r}]\!]\mathrm{b} = \mathrm{b}(\mathrm{r})$
$\mathrm{X}[\![\mathrm{v}]\!]\mathrm{b} = \mathrm{b}(\mathrm{v})$

Fig. 2. Semantics

3.2 Semantics

The meaning of a problem (Fig. 2) 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. Atoms, tuples, and constants have 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 analysis of a Kodkod problem P involves four steps:

1. Detecting P's symmetries.
2. Translating P into a Compact Boolean Circuit, CBC(P).
3. Computing SBP(P), a symmetry breaking predicate [27,28] for P.
4. Transforming CBC(P) $\wedge$ SBP(P) into conjunctive normal form, CNF(P).
5. Applying a SAT solver to CNF(P), and, if CNF(P) is satisfiable, interpreting its model as an instance of P.

The first two steps are the focus of this section. The third step is done in a standard way (e.g. [28]), by computing a simple lex-leader symmetry breaking predicate for the symmetry classes detected in the first step. The fourth step is performed using the standard translation from boolean logic to conjunctive normal form (see, for example, [29]). The last step is delegated to an off-the-shelf SAT solver, such as zchaff [30] or MiniSat [31].

4.1 Symmetry Detection

Many problems exhibit symmetries. For example, the pigeons in the pigeonhole problem are symmetric, as are the pigeonholes; if there were a solution with a particular assignment of pigeons to holes, exchanging two pigeons or two holes would yield another solution. More formally, we define the symmetries of a problem as follows.

Definition 1. *Let $\mathcal{A}$=$\{a_0,\ldots,a_n\}$ be a universe, D a set of declarations over $\mathcal{A}$, and F a formula over D. Let $l : \mathcal{A} \rightarrow \mathcal{A}$ be a permutation, and define $l(t)$ to be $\langle l(a_{i_0}),\ldots,l(a_{i_k})\rangle$ for all tuples t=$\langle a_{i_0},\ldots,a_{i_k}\rangle$, $l(c)$ to be $\{l(t)|t \in c\}$ for all constants $c \subseteq \mathcal{A}^k$, etc. The permutation l is a* symmetry *of the problem $P = (\mathcal{A}, D, F)$ if and only if, for all bindings B, the binding $l(B) : \text{rel} \rightarrow \text{constant}$ is a model of P, written $l(B) \models P$, whenever $B \models P$, and $l(B) \not\models P$ whenever $B \not\models P$. The bindings B and $l(B)$ are said to be* isomorphic.

The set of symmetries of P, denoted by Sym(P), induces an equivalence relation on the bindings that map the variables declared in D to sets of tuples drawn from $\mathcal{A}$. Two bindings B and B' are equivalent if $B' = l(B)$ for some $l \in$ Sym(P). Because each $l \in$ 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 sufficient to test one binding in each equivalence class induced by Sym(P) to find a model of P. Isomorph elimination (a.k.a. *symmetry breaking*), using either a symmetry-aware model finder on P [18,19,16] or a SAT solver on CNF($P \wedge$SBP(P)) [7,32], typically speeds up the model search by orders of magnitude. Many interesting problems are intractable without symmetry breaking [27,33].

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: 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 because neither logic provides a means of referring to individual atoms. The Kodkod logic does, however, so even if it were typed, atoms of the same type would not necessarily be interchangeable.

Here, for example, is a toy specification of a traffic lights system showing a case where the conceptual 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
(one Light.display & Green or no 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 green, yellow, and red. 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 $l_2 = (\mathtt{G}\ \mathtt{Y}\ \mathtt{R})$[1] 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.[2] First, a permutation l is a symmetry of a set of declarations D if it *fixes* the constants in D, i.e. if $l(c) = c$ for each c occurring in D. The permutation l_1, for example, is a symmetry of the traffic-lights declarations. Second, *any* permutation is a symmetry of a formula. The binding $l_2(B)$ is a model of the traffic-lights formula even though it is not a model of the problem.

These observations lead to a simple criterion for deciding whether a permutation l is a symmetry of a problem: $l \in \mathrm{Sym}(P)$ for all $P = (\mathcal{A}, D, F)$ if and only if l maps each constant that occurs in D to itself.

Theorem 1 (Symmetry 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 $c_1, c_2, \ldots, c_{2m}$.*

[1] Recall that cycle notation for permutations [34] 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, i.e. $(\mathtt{N}\ \mathtt{E})(\mathtt{Y}\ \mathtt{R}) = (\mathtt{N}\ \mathtt{E})(\mathtt{Y}\ \mathtt{R})(\mathtt{G})$.

[2] The proofs of all assertions and theorems stated in this section can be found in the technical report on Kodkod [35], available at http://hdl.handle.net/1721.1/34218.

Because every relational constant is isomorphic to a graph, Thm. 1 equates the task of finding Sym(P) to that of computing the automorphisms of the graphs that correspond to the constants in D—a problem with no known polynomial time solution [36]. So, we use the algorithm in Fig. 3 to find a polynomially computable subset of Sym(P) that is equal to Sym(P) for many problems, including the pigeonhole, traffic lights, and all problems in Section 5.

```
BASE($\mathcal{A}$: univDecl, $D$: relDecl*)
  $S \leftarrow \{\mathcal{A}\}$
  for all $r :_k [c_L, c_U] \in D$ do
    $S \leftarrow$ PART$(c_L, S)$
    $S \leftarrow$ PART$(c_U, S)$
  return $S$

PART($c$: constant, $S$: set of sets)
  $S' \leftarrow \{\}$
  $k \leftarrow$ arity$(c)$
  $C \leftarrow \{a_1 \mid \langle a_1, \ldots, a_k\rangle \in c\}$
  for all $s \in S$ do
    if $s \subseteq C$ or $s \cap C = \emptyset$
    then $S' \leftarrow S' \cup \{s\}$
    else $S' \leftarrow S' \cup \{s \cap C\} \cup \{s \setminus C\}$
  if $k > 1$ then
    $\overline{C} \leftarrow \{\langle a_2, \ldots, a_k\rangle \mid \langle a_1, a_2, \ldots, a_k\rangle \in c\}$
    $P \leftarrow \{s \mid s \in S' \wedge s \cap C \neq \emptyset\}$
    $\overline{P} \leftarrow \{\}$
    for all $p \in P$ do
      $S' \leftarrow S' \setminus p$
      while $p \neq \emptyset$ do
        $x \leftarrow$ choose$(p)$ ▷ pick an atom from $p$
        $\overline{X} \leftarrow \{\langle a_2, \ldots, a_k\rangle \mid \langle x, a_2, \ldots, a_k\rangle \in c\}$
        $X \leftarrow \{a \mid a \in p \wedge (\langle a\rangle \times \overline{C}) \cap c = \overline{X}\}$
        $p \leftarrow p \setminus X$
        $S' \leftarrow S' \cup \{X\}$
        $\overline{P} \leftarrow \overline{P} \cup \{\overline{X}\}$
    for all $\bar{p} \in \overline{P}$ do
      $S' \leftarrow$ PART$(\bar{p}, S')$
  return $S'$
```

BASE({b,c,d,e}, g:$_2$[{},{⟨b,c⟩⟨b,d⟩⟨e,e⟩}])

$S = \{\{$b, c, d, e$\}\}$

↻	$r =$ g:$_2$[{},{⟨b,c⟩⟨b,d⟩⟨e,e⟩}]
S	PART({}, {{b,c,d,e}}) = {{b,c,d,e}}
S	PART({⟨b,c⟩⟨b,d⟩⟨e,e⟩}, {{b,c,d,e}})

⋮

PART({⟨b,c⟩⟨b,d⟩⟨e,e⟩}, {{b, c, d, e}})

$S' = \{\}$

$k = 2$

$C = \{$b, e$\}$

↻	$s = \{$b, c, d, e$\}$
S'	{{b,e}, {c,d}}

$\overline{C} = \{$⟨c⟩⟨d⟩⟨e⟩$\}$

$P = \{\{$b, e$\}\}$

$\overline{P} = \{\}$

↻	$p = \{$b, e$\}$	
S'	{{c,d}}	
↻	1st	2nd
x	b	e
$\overline{X}$	{⟨c⟩⟨d⟩}	{⟨e⟩}
X	{b}	{e}
p	{e}	{}
S'	{{b},{c,d}}	{{b},{c,d},{e}}
$\overline{P}$	{{⟨c⟩⟨d⟩}}	{{⟨c⟩⟨d⟩},{⟨e⟩}}

↻	$\bar{p}$={⟨c⟩⟨d⟩}	$\bar{p}$={⟨e⟩}
S'	{{b},{c,d},{e}}	{{b},{c,d},{e}}

return {{b},{c,d},{e}}

Fig. 3. Symmetry detection algorithm and a sample trace. Trace events are horizontally aligned with the pseudocode. Loops are shown as tables, with a column per iteration.

The intuition behind the algorithm is the observation that constants in most problem declarations are expressible as unions of products of 'types' with zero or more 'distinguished' 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. We call the sets $\{\mathtt{R}, \mathtt{Y}\}$, $\{\mathtt{G}\}$ and $\{\mathtt{N}, \mathtt{E}\}$ a *base partitioning* of the traffic-lights universe with respect to the problem's declarations.

Definition 2. *Let $\mathcal{A}$ be a universe, c a constant over $\mathcal{A}$, and $S = \{S_1, \ldots, S_n\}$ a set of sets that partition $\mathcal{A}$. 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.: $\exists x \geq 1 \mid \exists s_1, \ldots, s_{xk} \in S \cup \{\emptyset\} \mid c = \bigcup_{j=0}^{x-1} (s_{jk+1} \times \ldots \times s_{jk+k})$, where k=*arity(c)*.*

The algorithm BASE finds the coarsest base partitioning for a given universe $\mathcal{A}$ and declarations D. It works by minimally refining the unpartitioned universe, $S = \{\mathcal{A}\}$, until each constant in D can be expressed as a union of products of the computed partitions (lines **??**-**??**). The correctness and local optimality of BASE follow by induction from Theorems 2 and 3:

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 $S = \{S_1, \ldots, S_n\}$ a base partitioning of $\mathcal{A}$ with respect to the constants $c_1, \ldots, c_{2m}$. If a permutation $l : \mathcal{A} \to \mathcal{A}$ fixes all $S_i \in S$, then it also fixes $c_1, \ldots, c_{2m}$.*

Theorem 3 (Local Optimality). *Let $\mathcal{A}$ be the universe of discourse, c a constant over $\mathcal{A}$, and $S = \{S_1, \ldots S_n\}$ a set of sets that partition $\mathcal{A}$. Applying* PART *to c and S will subdivide S into the coarsest $S' = \{S'_1, \ldots, S'_m\}$ that is a base partitioning of $\mathcal{A}$ with respect to c.*

The former tells us that the set of permutations induced by a base partitioning for D satisfies the symmetry criterion (Thm. 1), and the latter that each call to PART generates the coarsest base partitioning of $\mathcal{A}$ with respect to a given constant in D. It is also not difficult to see that the worst case running time of the algorithm is polynomial in the size of D, where $|D| = O(K|\mathcal{A}|^K)$ with $K = max(k_1, \ldots, k_m)$. In practice, the proportion of time spent on symmetry detection during analysis is negligible because K is usually small (< 5), and the algorithm works on a compact, interval tree representation of constants [35], which reduces the memory overhead exponentially for most problems.

4.2 Sparse-Matrix Translation to Boolean Logic

We translate a Kodkod problem $P = (\mathcal{A}, D, F)$ to an equisatisfiable boolean formula using the same basic idea employed by the Alloy Analyzer—that a relational expression can be represented as a matrix of boolean values [15]. Given a relation declaration $r :_k [c_L, c_U]$ over a universe $\mathcal{A} = \{a_0, \ldots, a_{n-1}\}$, we encode r as a k-dimensional boolean matrix m with

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

where $i_1, \ldots, i_k \in [0 \,..\, n)$ and freshVar() returns a fresh boolean variable. Expressions are then translated using matrix operations, and formulas become constraints over matrix entries (Fig. 4). For example, the join of two expressions, p.q, is translated as the matrix product of the translations of p and q, and the non-emptiness formula, some p, is translated as the disjunction of the entries in the matrix translation of p.

$\mathrm{T_P}$: problem $\rightarrow$ bool
$\mathrm{T_R}$: relDecl $\rightarrow$ univDecl $\rightarrow$ matrix
$\mathrm{T_M}$: formula $\rightarrow$ env $\rightarrow$ bool
$\mathrm{T_X}$: expr $\rightarrow$ env $\rightarrow$ matrix
env: (quantVar $\cup$ relVar) $\rightarrow$ matrix
freshVar: boolVar
bool := true | false | boolVar |
 $\neg$ bool | bool $\wedge$ bool | bool $\vee$ bool
boolVar := identifier

$\overrightarrow{x}, \overrightarrow{y}, \langle i_1, ..., i_k \rangle$ $\triangleright$ vectors
$\lfloor\,\rfloor$: matrix$\rightarrow\langle$int$\rangle$ $\triangleright$ minimum index
$\lceil\,\rceil$: matrix$\rightarrow\langle$int$\rangle$ $\triangleright$ maximum index
$||$: matrix$\rightarrow$dim $\triangleright$ dimensions

$\mathcal{M}$: dim$\rightarrow$($\langle$int$\rangle\rightarrow$bool)$\rightarrow$matrix $\triangleright$ constructor
$\mathcal{M}(s^d, f) = \{m \mid |m| = s^d \wedge$
 $\forall \overrightarrow{x} \in \{0, ..., s-1\}^d, m[\overrightarrow{x}] = f(\overrightarrow{x})\}$

$\mathcal{M}$: dim$\rightarrow\langle$int$\rangle\rightarrow$matrix $\triangleright$ constructor
$\mathcal{M}(s^d, \overrightarrow{x}) = \mathcal{M}(s^d,$
 $\lambda \overrightarrow{y}.$ if $\overrightarrow{y} = \overrightarrow{x}$ then true else false)

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

$\mathrm{T_R}[\mathrm{r} :_k [\mathrm{c}_L, \mathrm{c}_U]]\mathcal{A} = \mathcal{M}(|\mathcal{A}|^k, \lambda[i_1, ..., i_k].$
 if $\langle a_{i_1}, ..., a_{i_k}\rangle \in \mathrm{c}_L$ then true
 else if $\langle a_{i_1}, ..., a_{i_k}\rangle \in \mathrm{c}_u - \mathrm{c}_L$ then freshVar()
 else false)

$\mathrm{T_M}[\text{p in q}]e = \bigwedge(\neg \mathrm{T_X}[\mathrm{p}]e \vee \mathrm{T_X}[\mathrm{q}]e)$
$\mathrm{T_M}[\text{some p}]e = \bigvee(\mathrm{T_X}[\mathrm{p}]e)$
$\mathrm{T_M}[\text{one p}]e = \text{let } (\mathrm{m} = \mathrm{T_X}[\mathrm{p}]e) \text{ in}$
 $\bigvee_{\overrightarrow{x}=\lfloor m \rfloor}^{\lceil m \rceil} (\bigwedge(\neg \mathcal{M}(|\mathrm{m}|, \overrightarrow{x}) \oplus \mathrm{m}))$
$\mathrm{T_M}[\text{no p}]e = \bigwedge(\neg \mathrm{T_X}[\mathrm{p}]e)$
$\mathrm{T_M}[!\mathrm{F}]e = \neg\ \mathrm{T_M}[\mathrm{F}]e$
$\mathrm{T_M}[\mathrm{F}\ \&\&\ \mathrm{G}]e = \mathrm{T_M}[\mathrm{F}]e \wedge \mathrm{T_M}[\mathrm{G}]e$
$\mathrm{T_M}[\mathrm{F}\ ||\ \mathrm{G}]e = \mathrm{T_M}[\mathrm{F}]e \vee \mathrm{T_M}[\mathrm{G}]e$
$\mathrm{T_M}[\text{all v: p} \mid \mathrm{F}]e = \text{let } (\mathrm{m} = \mathrm{T_X}[\mathrm{p}]e) \text{ in}$
 $\bigwedge_{\overrightarrow{x}=\lfloor m \rfloor}^{\lceil m \rceil} (\neg \mathrm{m}[\overrightarrow{x}] \vee \mathrm{T_M}[\mathrm{F}](e{:}\mathrm{v} \mapsto \mathcal{M}(|\mathrm{m}|, \overrightarrow{x})))$
$\mathrm{T_M}[\text{some v: p} \mid \mathrm{F}]e = \text{let } (\mathrm{m} = \mathrm{T_X}[\mathrm{p}]e) \text{ in}$
 $\bigvee_{\overrightarrow{x}=\lfloor m \rfloor}^{\lceil m \rceil} (\mathrm{m}[\overrightarrow{x}] \wedge \mathrm{T_M}[\mathrm{F}](e{:}\mathrm{v} \mapsto \mathcal{M}(|\mathrm{m}|, \overrightarrow{x})))$

$\mathrm{T_X}[\mathrm{p} + \mathrm{q}]e = \mathrm{T_X}[\mathrm{p}]e \vee \mathrm{T_X}[\mathrm{q}]e$
$\mathrm{T_X}[\mathrm{p}\ \&\ \mathrm{q}]e = \mathrm{T_X}[\mathrm{p}]e \wedge \mathrm{T_X}[\mathrm{q}]e$
$\mathrm{T_X}[\mathrm{p} - \mathrm{q}]e = \mathrm{T_X}[\mathrm{p}]e \wedge \neg \mathrm{T_X}[\mathrm{q}]e$
$\mathrm{T_X}[\mathrm{p}\ .\ \mathrm{q}]e = \mathrm{T_X}[\mathrm{p}]e \cdot \mathrm{T_X}[\mathrm{q}]e$
$\mathrm{T_X}[\mathrm{p}{-}{>}\mathrm{q}]e = \mathrm{T_X}[\mathrm{p}]e \times \mathrm{T_X}[\mathrm{q}]e$
$\mathrm{T_X}[\tilde{}\,\mathrm{p}\ ]e = (\mathrm{T_X}[\mathrm{p}]e)^{\mathsf{T}}$
$\mathrm{T_X}[\hat{}\,\mathrm{p}\ ]e = \text{iterative-square}(\mathrm{T_X}[\mathrm{p}]e)$
$\mathrm{T_X}[\{\text{v: p} \mid \mathrm{F}\}]e = \text{let } (\mathrm{m} = \mathrm{T_X}[\mathrm{p}]e) \text{ in}$
 $\mathcal{M}(|\mathrm{m}|, \lambda \overrightarrow{x}.\ \mathrm{m}[x] \wedge \mathrm{T_M}[\mathrm{F}](e{:}\mathrm{v} \mapsto \mathcal{M}(|\mathrm{m}|, \overrightarrow{x})))\}$

Fig. 4. Translation rules

A key difference between the Kodkod and Alloy [15] translation algorithms is that the latter is based on types. The Alloy Analyzer 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, the Alloy Analyzer would reject the perfectly reasonable attempt to form the maternalGrandmother relation by joining the relation mother: Person $\rightarrow$ Woman with itself, because a 6×3 matrix cannot be multiplied by itself. There are two ways to remedy this problem: (1) force the type of mother up to Person $\rightarrow$ Person, doubling the size of its boolean representation, or (2) *atomize* mother into two pieces, mother$_w$: Woman $\rightarrow$ Woman and mother$_m$: Man $\rightarrow$ Woman, and split the expression mother.mother into mother$_w$.mother$_w$ + mother$_m$.mother$_w$ before handing it to the translator [10]. AA takes the latter approach which has not worked well in practice because of its awkward handling of transitive closure expressions [10].

We avoid the problems of a type-based translation by encoding all k-arity relations over $\mathcal{A}$ as k-dimensional sparse matrices $|\mathcal{A}| \times \ldots \times |\mathcal{A}|$. A sparse translation matrix is represented as a sorted map from *flat indices* [35] to boolean formulas. Each k-tuple, and its corresponding matrix index, is encoded as an integer in the range $[0 \ldots |\mathcal{A}|^k)$. A sparse matrix maps a tuple's integer representation only if it is non-false. For example, the sparse matrix representation of the display relation from the traffic-lights problem maps the flat indices of the upper bound tuples $\{\langle \texttt{N}, \texttt{G}\rangle\langle \texttt{N}, \texttt{Y}\rangle\langle \texttt{N}, \texttt{R}\rangle\langle \texttt{E}, \texttt{G}\rangle\langle \texttt{E}, \texttt{Y}\rangle\langle \texttt{E}, \texttt{R}\rangle\}$ to boolean variables, and leaves the indices of the remaining tuples in $\mathcal{A} \times \mathcal{A}$ unmapped (i.e. false). Consecutive indices that map to true are encoded using a run-length encoding, enabling a compact representation of lower bounds.

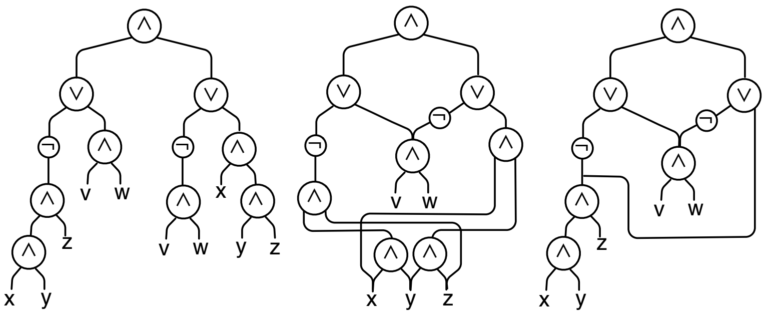
4.3 Sharing Detection with Compact Boolean Circuits

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 [37]. Our implementation uses Compact Boolean Circuits to detect sharing at the boolean level.

A Compact Boolean Circuit (CBC) is a partially canonical, directed, acyclic graph (V, E, d). The set V is partitioned into operator vertices $V_{\text{op}} = V_{\text{AND}} \cup V_{\text{OR}} \cup V_{\text{NOT}}$ and leaves $V_{\text{leaf}} = V_{\text{VAR}} \cup \{\mathsf{T}, \mathsf{F}\}$. An AND or an OR vertex has two children, and a NOT vertex has one child. The degree of canonicity is determined by an equivalence relation on vertices (which embodies standard properties of the logical operators, such as commutativity, associativity, etc.) and the circuit's *compaction depth* $d \geq 1$. In particular, no vertex $v \in V$ can be transformed into another vertex $w \in V$ by applying an equivalence transformation to the top $d \geq 1$ levels of the subgraph rooted at v.

An example of a non-compact boolean circuit and its compact equivalents is shown in Fig. 5. Fig. 5(a) contains the formula $(x \wedge y \wedge z) \Leftrightarrow (v \wedge w)$ encoded using the operators {AND, OR, NOT} as $(\neg((x \wedge y) \wedge z) \vee (v \wedge w)) \wedge (\neg(v \wedge w) \vee (x \wedge (y \wedge z)))$. Fig. 5(b) shows an equivalent CBC with the minimal compaction depth of $d = 1$, which enforces partial canonicity at the level of inner nodes' children. That is, the depth of $d = 1$ ensures that all nodes in the circuit are syntactically distinct, forcing the subformula $(v \wedge w)$ to be shared. Fig. 5(c) shows the original circuit represented as a CBC with the compaction depth of $d = 2$, which enforces partial canonicity at the level of nodes' grandchildren. The law of associativity applies to the subformulas $((x \wedge y) \wedge z)$ and $(x \wedge (y \wedge z))$, forcing $((x \wedge y) \wedge z)$ to be shared.

The partial canonicity of CBCs is 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. This ensures 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 by looking at the top $d \geq 1$ levels of their subgraphs. 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.



(a) Original circuit (b) CBC, $d = 1$ (c) CBC, $d = 2$

Fig. 5. A non-compact boolean circuit and its compact equivalents

5 Results

We have compared the performance of Kodkod to that of three other tools, the Alloy Analyzer (version 3), MACE (version 4), and Paradox (version 1.3)[3] on two sets of problems:

- Symmetric problems include the pigeonhole problem and the 'Ceilings and Floors' problem from the Alloy 3 distribution. Like the pigeonhole problem, 'Ceilings and Floors' is unsatisfiable and highly symmetric.
- TPTP problems consist of twelve TPTP [9] benchmarks from various problem domains. Because the TPTP problems had to be translated to our logic by hand (in the obvious way, by translating predicates and functions as relations, predicate application as membership testing, function application as join, etc.), the overriding criterion for benchmark selection was syntactic succinctness. Other selection criteria were a high difficulty rating (> 0.6), complex relationships between predicates and functions (geo, med, and set problems), prevalence of unit equalities (alg212 and num374), and presence of partial instances (alg195 and num378).

The results are given in Fig. 6. The table shows each problem's rating, if any, the size of its universe ($|\mathcal{A}|$), and the model finders' performance on it. For symmetric problems, we use two different universe sizes to demonstrate how changing search bounds impacts model finders' performance. For TPTP problems, the shown universe size is the largest universe for which at least three of the model finders produced a result in five minutes. The performance data for Alloy 3, Kodkod, and Paradox includes the analysis time, rounded to the nearest second, and the size of the generated CNF, given as the total number of variables and clauses. MACE4 does not report CNF statistics. All analyses were performed on a 3.6 GHz Pentium 4 with 3 GB RAM. Alloy 3, Kodkod and Paradox were configured with MiniSat [31] as their SAT engine; MACE4 uses its own internal SAT solver. Kodkod's sharing detection parameter d was set to 3. Analyses that did not

[3] Paradox 2.0b, the latest version, does not perform as well as 1.3 on our benchmarks.

				ALLOY ANALYZER 3			KODKOD			PARADOX 1.3			MACE4
	PROBLEM	RATING	$\|\mathcal{A}\|$	SEC	VARS	CLAUSES	SEC	VARS	CLAUSES	SEC	VARS	CLAUSES	SEC
SYMMETRIC PROBLEMS	ceil		12	1	2,723	11,704	0	1,749	3,289	0	299	1,373	4
			20	16	9,987	46,740	14	6,477	12,449	–	695	4,850	–
	pigeon		39	2	15,703	76,994	0	6,953	12,648	–	1,041	9,889	0
			99	33	92,191	576,554	5	82,613	156,843	–	5,901	143,344	0
TPTP PROBLEMS	alg195	0.89	14	31	195,408	834,508	3	77,240	254,239	22	30,771	982,467	–
	alg212	1.00	7	277	395,297	6,432,170	64	301,725	1,012,808	1	20,747	135,588	14
	com008	0.67	11	6	15,384	77,378	6	8,565	14,624	94	6,467	31,275	–
	geo091	1.00	8	72	81,267	587,728	16	33,463	73,292	84	19,146	145,373	–
	geo158	1.00	6	8	29,831	185,038	2	12,574	26,552	9	8,262	49,955	–
	med007	0.67	15	10	19,052	108,454	2	15,072	31,476	36	7,981	48,449	–
	med009	0.67	17	12	25,177	144,968	3	20,198	42,263	28	11,850	66,758	–
	num374	1.00	5	50	70,229	291,573	55	63,661	200,238	3	6,763	52,671	9
	num378	1.00	21	–	–	–	1	0	0	193	74,736	1,692,990	1
	set943	1.00	7	159	25,124	101,040	20	18,883	43,694	11	8,648	46,977	–
	set948	1.00	7	7	40,776	159,735	1	24,970	60,787	61	16,226	86,932	–
	top020	1.00	9	–	–	–	48	1,378,863	2,343,728	54	96,232	1,545,950	6

Fig. 6. Results for symmetric and TPTP problems. Gray shading indicates the fastest time(s) for each problem; dashes indicate timeouts.

complete within five minutes are indicated by dashes. The fastest analysis time for each problem is highlighted with gray shading.

The data on symmetric problems demonstrates the effectiveness of our symmetry detection algorithm compared to that of Alloy 3, which derives optimal symmetry information from Alloy's type system, and Paradox, which employs a sort inference algorithm to find symmetry classes. MACE4 performs no symmetry breaking, but, interestingly, its internal simplifications allow it to determine that the pigeonhole problem is unsatisfiable apparently without performing any search. The TPTP data shows that Kodkod's performance is competitive with Paradox's and MACE4's on a variety of classical logic problems. Kodkod outperforms MACE4 and Paradox on problems describing complex relationships between predicates and functions (e.g. geo091 or set948) and on problems with partial instances (alg195 and num378). MACE4 and Paradox, however, are superior on problems that contain many unit equalities or deeply nested universal quantifiers, such as alg212 and num374. These results are consistent with our overall experience using Alloy 3, Kodkod, MACE4 and Paradox.

The above problems were chosen to compare Kodkod to other SAT-based model finders, but they are in fact not representative of the class of problems for which Kodkod and Alloy were developed. Software design problems, in contrast to these mathematical problems, tend to have less regular structure, despite the grounding out of quantifiers. We compared Kodkod to Alloy 3 on three design problems: Dijkstra's mutual exclusion scheme [38], leader election in a ring [39], and the transfer protocol of the Mondex smart card [40].

The results are shown in Fig. 7. For the mutual exclusion and leader election problems, we use two different universe sizes; for the Mondex problem, we check

			ALLOY ANALYZER 3						KODKOD					
PROBLEM		$\lvert\mathcal{A}\rvert$	VARS	CLAUSES	FOL $\rightarrow$ CNF	BERK MIN	MINI SAT	ZCH AFF	VARS	CLAUSES	FOL $\rightarrow$ CNF	BERK MIN	MINI SAT	ZCH AFF
exclusion		30	74,818	722,236	20	7	1	10	20,080	120,097	3	0	0	1
		45	357,253	4,874,911	142	150	9	–	67,695	543,597	19	13	5	10
election		15	14,272	78,031	2	1	1	1	8,665	29,590	1	0	0	0
		24	91,594	662,188	16	143	109	–	45,136	183,484	3	62	76	–
mondex	A241	51	50,926	416,744	10	27	90	52	35,791	86,402	1	4	87	9
	OpTotal	51	43,256	381,458	7	3	2	1	0	0	0	0	0	0
	IgnoreInv	51	43,413	386,812	6	7	4	4	28,243	57,604	1	2	22	2
	TransferInv	51	50,902	419,094	7	174	–	173	35,761	83,172	1	46	–	53

Fig. 7. Results for design problems. Gray shading indicates the fastest SAT solving time(s) for each problem; dashes indicate timeouts.

a variety of assertions in the same universe. The performance data includes the size of the generated CNF and the time, in seconds, taken to generate and solve it using various SAT engines [41,31,30]. In all cases, Kodkod produces smaller formulas, which are solved faster by BerkMin [41] and zChaff [30]. Interestingly, on the Mondex problem (and a few others we encountered), MiniSat actually performs worse on Kodkod's formulas than on Alloy 3's larger formulas. Note that translation time is dramatically lower in Kodkod than in Alloy 3; the translation scheme in Alloy 3 used a more complicated (and apparently less effective) template mechanism for detecting sharing.

References

1. Jackson, D., Shlyakhter, I., Sridharan, M.: A micromodularity mechanism. In: ESEC / SIGSOFT FSE. (2001) 62–73
2. Vaziri, M., Jackson, D.: Checking properties of heap-manipulating procedures with a constraint solver. In: TACAS. (2003) 505–520
3. Taghdiri, M.: Inferring specifications to detect errors in code. In: ASE. (2004) 144–153
4. Khurshid, S., Marinov, D.: TestEra: Specification-based testing of java programs using sat. ASE **11**(4) (2004) 403–434
5. Dennis, G., Chang, F., Jackson, D.: Modular verification of code. In: ISSTA, Portland, Maine (2006)
6. Yeung, V.: Declarative configuration applied to course scheduling. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA (2006)
7. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: CADE-19 Workshop on Model Computation, Miami, FL (2003)
8. McCune, W.: A Davis-Putnam program and its application to finite first-order model search: quasigroup existence problem. Technical report, ANL (1994)
9. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning **21**(2) (1998) 177–203
10. Edwards, J., Jackson, D., Torlak, E., Yeung, V.: Faster constraint solving with subtypes. In: ISSTA '04, New York, NY, USA, ACM Press (2004) 232–242
11. Andersen, H.R., Hulgaard, H.: Boolean expression diagrams. In: LICS, Warsaw, Poland (1997)

12. Abdulla, P.A., Bjesse, P., Eén, N.: Symbolic reachability analysis based on sat-solvers. In: TACAS '00, London, UK, Springer-Verlag (2000) 411–425
13. Torlak, E., Dennis, G.: Kodkod for Alloy users. In: First ACM Alloy Workshop, Portland, Oregon (2006)
14. Fujita, M., Slaney, J., Bennett, F.: Automating generation of some results in finite algebra. In: 13th IJCAI, Chambéry, France (1993)
15. Jackson, D.: Automating first order relational logic. In: FSE, San Diego, CA (2000)
16. Jackson, D., Jha, S., Damon, C.A.: Isomorph-free model enumeration: a new method for checking relational specifications. ACM TPLS **20**(2) (1998) 302–343
17. Slaney, J.K.: Finder: Finite domain enumerator - system description. In: CADE-12, London, UK, Springer-Verlag (1994) 798–801
18. Zhang, J.: The generation and application of finite models. PhD thesis, Institute of Software, Academia Sinica, Beijing (1994)
19. Zhang, J., Zhang, H.: SEM: a system for enumerating models. In: IJCAI95, Montreal (1995)
20. Jackson, D., Damon, C.A.: Elements of style: analyzing a software design feature with a counterexample detector. TOSEM (1996) 484–495
21. Ng, Y.C.: A Nitpick specification of IPv6. Senior Honors thesis, Computer Science Department, Carnegie Mellon University (1997)
22. Khurshid, S., Jackson, D.: Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In: ASE. (2000) 13–22
23. Dennis, G., Seater, R., Rayside, D., Jackson, D.: Automating commutativity analysis at the design level. In: ISSTA. (2004) 165–174
24. Narain, S.: Network configuration management via model finding. In: ACM Workshop On Self-Managed Systems, Newport Beach, CA (2004)
25. O'Keefe, R.: The Craft of Prolog. Logic Programming. MIT Press, Cambridge, MA (1990)
26. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
27. Crawford, J., Ginsberg, M.L., Luck, E., Roy, A.: Symmetry-breaking predicates for search problems. In: KR'96. Morgan Kaufmann, San Francisco (1996) 148–159
28. Shlyakhter, I.: Generating effective symmetry breaking predicates for search problems. Electronic Notes in Discrete Mathematics **9** (2001)
29. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. In: SBMC. Volume 2. (2006) 1–26
30. Mahajan, Y.S., Fu, Z., Malik, S.: zchaff2004: An efficient sat solver. In: SAT (Selected Papers). (2004) 360–375
31. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT'03. Volume LNCS 2919. (2004) 502–518
32. Shlyakhter, I.: Declarative Symbolic Pure Logic Model Checking. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (2005)
33. Sabharwal, A.: SymChaff: A structure-aware satisfiability solver. In: 20th National Conference on Artificial Intelligence (AAAI), Pittsburgh, PA (2005) 467–474
34. Armstrong, M.A.: Groups and Symmetry. Springer-Verlag, New York (1988)
35. Torlak, E., Jackson, D.: The design of a relational engine. Technical Report MIT-CSAIL-TR-2006-068, MIT (2006)
36. Babai, L., Kantor, W.M., Luks, E.M.: Computational complexity and the classification of finite simple groups. In: IEEE SFCS, IEEE CSP (1983) 162–171
37. Shlyakhter, I., Sridharan, M., Seater, R., Jackson, D.: Exploiting subformula sharing in automatic analysis of quantified formulas. In: SAT, Portofino, Italy (2003)

38. Dijkstra, E.W.: Cooperating sequential processes. In Genuys, F., ed.: Programming Languages. Academic Press, New York (1968) 43–112
39. Chang, E.J.H., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun. ACM **22**(5) (1979) 281–283
40. Ramananandro, T.: The Mondex case study with Alloy. http://www.eleves.ens.fr/home/ramanana/work/mondex/ (2006)
41. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT solver. In: Design Automation and Test in Europe. (2002) 142–149