A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications

by

Emina Torlak

M.Eng., Massachusetts Institute of Technology (2004)

B.Sc., Massachusetts Institute of Technology (2003)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2009

Author .....
Department of Electrical Engineering and Computer Science
December 16, 2008

Certified by .....
Daniel Jackson
Professor
Thesis Supervisor

Accepted by .....
Professor Terry P. Orlando
Chairman, Department Committee on Graduate Students

A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications

by

Emina Torlak

Submitted to the Department of Electrical Engineering and Computer Science
on December 16, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Relational logic is an attractive candidate for a software description language, because both the design and implementation of software often involve reasoning about relational structures: organizational hierarchies in the problem domain, architectural configurations in the high level design, or graphs and linked lists in low level code. Until recently, however, frameworks for solving relational constraints have had limited applicability. Designed to analyze small, hand-crafted models of software systems, current frameworks perform poorly on specifications that are large or that have partially known solutions.

This thesis presents an efficient constraint solver for relational logic, with recent applications to design analysis, code checking, test-case generation, and declarative configuration. The solver provides analyses for both satisfiable and unsatisfiable specifications—a *finite model finder* for the former and a *minimal unsatisfiable core extractor* for the latter. It works by translating a relational problem to a boolean satisfiability problem; applying an off-the-shelf SAT solver to the resulting formula; and converting the SAT solver's output back to the relational domain.

The idea of solving relational problems by reduction to SAT is not new. The core contributions of this work, instead, are new techniques for expanding the capacity and applicability of SAT-based engines. They include: a new interface to SAT that extends relational logic with a mechanism for specifying partial solutions; a new translation algorithm based on sparse matrices and auto-compacting circuits; a new symmetry detection technique that works in the presence of partial solutions; and a new core extraction algorithm that recycles inferences made at the boolean level to speed up core minimization at the specification level.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgments

Working on this thesis has been a challenging, rewarding and, above all, wonderful experience. I am deeply grateful to the people who have shared it with me:

To my advisor, Daniel Jackson, for his guidance, support, enthusiasm, and a great sense of humor. He has helped me become not only a better researcher, but a better writer and a better advocate for my ideas.

To my thesis readers, David Karger and Sharad Malik, for their insights and excellent comments.

To my friends and colleagues in the Software Design Group—Felix Chang, Greg Dennis, Jonathan Edwards, Eunsuk Kang, Sarfraz Khurshid, Carlos Pacheco, Derek Rayside, Robert Seater, Ilya Shlyakhter, Mana Taghdiri and Mandana Vaziri—for their companionship and for many lively discussions of research ideas, big and small. Ilya's work on Alloy3 paved the way for this dissertation; Greg, Mana and Felix's early adoption of the solver described here was instrumental to its design and development.

To my husband Aled for his love, patience and encouragement; to my sister Alma for her boundless warmth and kindness; and to my mother Edina for her unrelenting support and for giving me life more than once.

I dedicate this thesis to my mother and to the memory of my father.

The first challenge for computing science is to discover how to maintain order in a finite, but very large, discrete universe that is intricately intertwined.

E. W. Dijkstra, 1979

Contents

1 Introduction **13**

1.1 Bounded relational logic 15

1.2 Finite model finding 17

1.3 Minimal unsatisfiable core extraction 23

1.4 Summary of contributions 27

2 From Relational to Boolean Logic **31**

2.1 Bounded relational logic 32

2.2 Translating bounded relational logic to SAT 35

2.2.1 Translation algorithm 35

2.2.2 Sparse-matrix representation of relations 37

2.2.3 Sharing detection at the boolean level 40

2.3 Related work 44

2.3.1 Type-based representation of relations 44

2.3.2 Sharing detection at the problem level 46

2.3.3 Multidimensional sparse matrices 47

2.3.4 Auto-compacting circuits 50

2.4 Experimental results 51

3 Detecting Symmetries **55**

3.1 Symmetries in model extension 56

3.2 Complete and greedy symmetry detection 60

3.2.1 Symmetries via graph automorphism detection 61

3.2.2 Symmetries via greedy base partitioning 63
3.3 Experimental results 68
3.4 Related work 70
3.4.1 Symmetries in traditional model finding 71
3.4.2 Symmetries in constraint programming 72

4 Finding Minimal Cores **75**
4.1 A small example 76
4.1.1 A toy list specification 76
4.1.2 Sample analyses 77
4.2 Core extraction with a resolution engine 81
4.2.1 Resolution-based analysis 82
4.2.2 Recycling core extraction 86
4.2.3 Correctness and minimality of RCE 88
4.3 Experimental results 89
4.4 Related work 93
4.4.1 Minimal core extraction 93
4.4.2 Clause recycling 94

5 Conclusion **97**
5.1 Discussion 98
5.2 Future work 101
5.2.1 Bitvector arithmetic and inductive definitions 101
5.2.2 Special-purpose translation for logic fragments 102
5.2.3 Heuristics for variable and constraint ordering 103

List of Figures

1-1 A hard Sudoku puzzle 13

1-2 Sudoku in bounded relational logic, Alloy, FOL, and FOL/ID 18

1-3 Solution for the sample Sudoku puzzle 19

1-4 Effect of partial models on the performance of SAT-based model finders . 21

1-5 Effect of partial models on the performance of a dedicated Sudoku solver . 22

1-6 An unsatisfiable Sudoku puzzle and its core 24

1-7 Comparison of SAT-based core extractors on 100 unsatisfiable Sudokus . . 26

1-8 Summary of contributions 28

2-1 Syntax and semantics of bounded relational logic 33

2-2 A toy filesystem 34

2-3 Translation rules for bounded relational logic 36

2-4 A sample translation 38

2-5 Sparse representation of translation matrices 39

2-6 Computing the d-reachable descendants of a CBC node 41

2-7 A non-compact boolean circuit and its compact equivalents 43

2-8 GCRS and ECRS representations for multidimensional sparse matrices . . 48

2-9 An example of a non-optimal two-level rewrite rule 52

3-1 Isomorphisms of the filesystem model 57

3-2 Isomorphisms of an invalid binding for the toy filesystem 57

3-3 Complete symmetry detection via graph automorphism 62

3-4 A toy filesystem with no partial model 64

3-5 Symmetry detection via greedy base partitioning 66

3-6 Microstructure of a CSP 74

4-1 A toy list specification 78

4-2 The resolution rule for propositional logic 84

4-3 Resolution-based analysis of $(a = b) \wedge (b = c) \wedge \neg(a \Rightarrow c)$ 84

4-4 Core extraction algorithms 87

List of Tables

1.1 Recent applications of Kodkod 29

2.1 Simplification rules for CBCs 42
2.2 Evaluation of Kodkod's translation optimizations 53

3.1 Evaluation of symmetry detection algorithms 69

4.1 Evaluation of minimal core extractors 90
4.2 Evaluation of minimal core extractors based on problem difficulty 91

5.1 Features of state-of-the-art model finders 99

Chapter 1

Introduction

Puzzles with simple rules can be surprisingly hard to solve, even when a part of the solution is already known. Take Sudoku, for example. It is a logic game played on a partially completed 9×9 grid, like the one in Fig. 1-1. The goal is simply to fill in the blanks so that the numbers 1 through 9 appear exactly once in every row, column, and heavily boxed region of the grid. Each puzzle has a unique solution, and many are easily solved. Yet some are 'very hard.' Target completion time for the puzzle in Fig. 1-1, for example, is 30 minutes [58].

6			2				5	
	1	8		6			2	
		3				4		
			6		7	8		
4		2		5				
			9		8			
5		4		9		3		
	2						1	4
3					5			7

Figure 1-1: A hard Sudoku puzzle [58].

Software engineering is full of problems like Sudoku—where the rules are easy to describe, parts of the solution are known, but the task of filling in the blanks is computationally intractable. Examples include, most notably, *declarative configuration* problems such as network configuration [99], installation management [133], and

scheduling [149]. The configuration task usually involves extending a valid configuration with one or more new components so that certain validity constraints are preserved. To install a new package on a Linux machine, for example, an installation manager needs to find a subset of packages in the Linux distribution, including the desired package, which can be added to the installation so that all package dependencies are met. Also related are the problems of *declarative analysis*: software design analysis [69], bounded code verification against rich structural specifications [31, 34, 126, 138], and declarative test-case generation [77, 114, 134].

Automatic solutions to problems like Sudoku and declarative configuration usually come in two flavors: a special-purpose solver or a special-purpose translator to some logic, used either with an off-the-shelf SAT solver or, since recently, an SMT solver [38, 53, 9, 29] that can also reason about linear integer and bitvector arithmetic. An expertly implemented special-purpose solver is likely to perform better than a translation-based alternative, simply because a custom solver can be guided with domain-specific knowledge that may be hard (or impossible) to use effectively in a translation. But crafting an efficient search algorithm is tricky, and with the advances in SAT solving technology, the performance benefits of implementing a custom solver tend to be negligible [53]. Even for a problem as simple as Sudoku, with many known special-purpose inference rules, SAT-based approaches [86, 144] are competitive with hand-crafted solvers (e.g. [141]).

Reducing a high-level problem description to SAT is not easy, however, since a boolean encoding has to contain just the right amount and kind of information to elicit the best performance from the SAT solver. If the encoding includes too many redundant formulas, the solver will slow down significantly [119, 139, 41]. At the same time, introducing certain kinds of redundancy into the encoding, in the form of symmetry breaking [27, 116] or reconvergence [150] clauses, can yield dramatic improvements in solving times.

The challenges of using SAT for declarative configuration and analysis are not limited to finding the most effective encoding. When a SAT solver fails to find a satisfying assignment for the translation of a problem, many applications need to

know what caused the failure and correct it. For example, if a software package cannot be installed because it conflicts with one or more existing packages, a SAT-based installation manager such as OPIUM [133] needs to identify (and remove) the conflicting packages. It does this by analyzing the proof of unsatisfiability produced by the SAT solver to find an unsatisfiable subset of the translation clauses known as an *unsatisfiable core*. Once extracted from the proof, the boolean core needs to mapped back to the conflicting constraints in the problem domain. The problem domain core, in turn, has to be *minimized* before corrective action is taken because it may contain constraints which do not contribute to its unsatisfiability.

This thesis presents a framework that facilitates easy and efficient use of SAT for declarative configuration and analysis. The user of the framework provides just a high-level description of the problem—in a logic that underlies many software design languages [2, 143, 123, 69]—and a partial solution, if one is available. The framework then does the rest: efficient translation to SAT, interpretation of the SAT instance in terms of problem-domain concepts, and, in the case of unsatisfiability, interpretation and minimization of the unsatisfiable core. The key algorithms used for SAT encoding [131] and core minimization [129] are the main technical contributions of this work; the main methodological contribution is the idea of separating the description of the problem from the description of its partial solution [130]. The embodiment of these contributions, called *Kodkod*, has so far been used in a variety of applications for declarative configuration [100, 149], design analysis [21], bounded code verification [31, 34, 126], and automated test-case generation [114, 134].

1.1 Bounded relational logic

Kodkod is based on the "relational logic" of Alloy [69], consisting essentially of a first-order logic augmented with the operators of the relational calculus [127]. The inclusion of transitive closure extends the expressiveness beyond standard first-order logics, and allows the encoding of common reachability constraints that otherwise could not be expressed. In contrast to specification languages (such as Z [123], B

[2], and OCL [143]) that are based on set-theoretic logics, Alloy's relational logic was designed to have a stronger connection to data modeling languages (such as ER [22] and SDM [62]), a more uniform syntax, and a simpler semantics. Alloy's logic treats everything as a relation: sets as relations of arity one and scalars as singleton sets. Function application is modeled as relational join, and an out-of-domain application results in the empty set, dispensing with the need for special notions of undefinedness. The use of multi-arity relations (in contrast to functions over sets) is a critical factor in Alloy being first order and amenable to automatic analysis. The choice of this logic for Kodkod was thus based not only on its simplicity but also on its analyzability.

Kodkod extends the logic of Alloy with the notion of *relational bounds*. A bounded *relational specification* is a collection of constraints on relational variables of any arity that are bound above and below by relational constants (i.e. sets of tuples). All bounding constants consist of tuples that are drawn from the same finite universe of uninterpreted elements. The upper bound specifies the tuples that a relation may contain; the lower bound specifies the tuples that it must contain.

Figure 1-2a shows a snippet of bounded relational logic[1] that describes the Sudoku puzzle from Fig. 1-1. It consists of three parts: the universe of discourse (line 1); the bounds on free variables that encode the assertional knowledge about the problem (lines 2-7), such as the initial state of the grid; and the constraints on the bounded variables that encode definitional knowledge about the problem (lines 10-21), i.e. the rules of the game.

The bounds specification is straightforward. The unary relation num (line 2) provides a handle on the set of numbers used in the game. As this set is constant, the relation has the same lower and upper bound. The relations r1, r2 and r3 (lines 4-6) partition the numbers into three consecutive, equally-sized intervals. The ternary relation grid (line 7) models the Sudoku grid as a mapping from cells, defined by their row and column coordinates, to numbers. The set $\{\langle 1, 1, 6\rangle, \langle 1, 4, 2\rangle, \ldots, \langle 9, 9, 7\rangle\}$ specifies the lower bound on the grid relation; these are the mappings of cells to

[1]Because Kodkod is designed as a Java API, the users communicate with it by constructing formulas, relations and bounds via API calls. The syntax shown here is just an illustrative rendering of Kodkod's abstract syntax graph, defined formally in Chapter 2.

numbers that are given in Fig. 1-1.[2] The upper bound on its value is the lower bound augmented with the bindings from the coordinates of the empty cells, such as the cell in the first row and second column, to the numbers 1 through 9.

The rest of the problem description defines the rules of Sudoku: each cell on the grid contains some value (line 10), and that value is unique with respect to other values in the same row, column, and 3×3 region of grid (lines 11-21). Relational join is used to navigate the grid structure: the expression 'grid[x][num\y]', for example, evaluates to the contents of the cells that are in the row x and in all columns except y. The relational join operator is freely applied to quantified variables since the logic treats them as singleton unary relations rather than scalars.

Having a mechanism for specifying precise bounds on free variables is not necessary for expressing problems like Sudoku and declarative configuration. Knowledge about partial solutions can always be encoded using additional constraints (e.g. the 'puzzle' formulas in Figs. 1-2b and 1-2c), and the domains of free variables can be specified using types (Fig. 1-2b), membership predicates (Fig. 1-2c), or both (Fig. 1-2d). But there are two important advantages to expressing assertional knowledge with explicit bounds. The first is methodological: bounds cleanly separate what is known to be true from what is defined to be true. The second is practical: explicit bounds enable faster *model finding*.

1.2 Finite model finding

A *model* of a specification, expressed as a collection of declarative constraints, is a binding of its free variables to values that makes the specification true. The bounded relational specification in Fig. 1-2a, for example, has a single model (Fig. 1-3) which maps the grid relation to the solution of the sample Sudoku problem. An engine that searches for models of a specification in a finite universe is called a *finite model finder*, or simply a *model finder*.

Traditional model finders [13, 25, 51, 68, 70, 91, 93, 117, 122, 151, 152] have

[2]The '...' symbol is not a part of the syntax. It is used in Fig. 1-1 and in text to mean 'etc'.

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}

num :1 [{⟨1⟩,⟨2⟩,⟨3⟩,⟨4⟩,⟨5⟩,⟨6⟩,⟨7⟩,⟨8⟩,⟨9⟩},
        {⟨1⟩,⟨2⟩,⟨3⟩,⟨4⟩,⟨5⟩,⟨6⟩,⟨7⟩,⟨8⟩,⟨9⟩}]
r1  :1 [{⟨1⟩,⟨2⟩,⟨3⟩}, {⟨1⟩,⟨2⟩,⟨3⟩}]
r2  :1 [{⟨4⟩,⟨5⟩,⟨6⟩}, {⟨4⟩,⟨5⟩,⟨6⟩}]
r3  :1 [{⟨7⟩,⟨8⟩,⟨9⟩}, {⟨7⟩,⟨8⟩,⟨9⟩}]
grid :3 [{⟨1, 1, 6⟩,⟨1, 4, 2⟩, ..., ⟨9, 9, 7⟩},
         {⟨1, 1, 6⟩,⟨1, 2, 1⟩,⟨1, 2, 2⟩, ..., ⟨1, 3, 9⟩,
          ⟨1, 4, 2⟩,⟨1, 5, 1⟩, ..., ⟨9, 9, 7⟩}]

∀ x, y: num   | some grid[x][y]
∀ x, y: num   | no (grid[x][y] ∩ grid[x][num\y])
∀ x, y: num   | no (grid[x][y] ∩ grid[num\x][y])
∀ x: r1, y: r1 | no (grid[x][y] ∩ grid[r1\x][r1\y])
∀ x: r1, y: r2 | no (grid[x][y] ∩ grid[r1\x][r2\y])
∀ x: r1, y: r3 | no (grid[x][y] ∩ grid[r1\x][r3\y])
∀ x: r2, y: r1 | no (grid[x][y] ∩ grid[r2\x][r1\y])
∀ x: r2, y: r2 | no (grid[x][y] ∩ grid[r2\x][r2\y])
∀ x: r2, y: r3 | no (grid[x][y] ∩ grid[r2\x][r3\y])
∀ x: r3, y: r1 | no (grid[x][y] ∩ grid[r3\x][r1\y])
∀ x: r3, y: r2 | no (grid[x][y] ∩ grid[r3\x][r2\y])
∀ x: r3, y: r3 | no (grid[x][y] ∩ grid[r3\x][r3\y])
```

(a) Sudoku in bounded relational logic

```
abstract sig Num {grid: Num->Num}
abstract sig R1, R2, R3 extends Num {}
one sig N1, N2, N3 extends R1 {}
one sig N4, N5, N6 extends R2 {}
one sig N7, N8, N9 extends R3 {}

fact rules {
  all x, y: Num    | some grid[x][y]
  all x, y: Num    | no grid[x][y] & grid[x][Num-y]
  all x, y: Num    | no grid[x][y] & grid[Num-x][y]
  all x: R1, y: R1 | no grid[x][y] & grid[R1-x][R1-y]
  all x: R1, y: R2 | no grid[x][y] & grid[R1-x][R2-y]
  all x: R1, y: R3 | no grid[x][y] & grid[R1-x][R3-y]
  all x: R2, y: R1 | no grid[x][y] & grid[R2-x][R1-y]
  all x: R2, y: R2 | no grid[x][y] & grid[R2-x][R2-y]
  all x: R2, y: R3 | no grid[x][y] & grid[R2-x][R3-y]
  all x: R3, y: R1 | no grid[x][y] & grid[R3-x][R1-y]
  all x: R3, y: R2 | no grid[x][y] & grid[R3-x][R2-y]
  all x: R3, y: R3 | no grid[x][y] & grid[R3-x][R3-y]
}

fact puzzle {
  N1->N1->N6 + N1->N4->N2 + ... +
  N9->N9->N7 in grid }
```

(b) Sudoku in Alloy

```
fof(at_most_one_in_each_row, axiom,
  (! [X, Y1, Y2] :
    ((grid(X, Y1) = grid(X, Y2)) => Y1 = Y2))).

fof(at_most_one_in_each_column, axiom,
  (! [X1,X2,Y] :
    ((grid(X1,Y) = grid(X2,Y)) => X1 = X2))).

fof(region_reflexive, axiom, (! [X] : region(X,X))).
fof(region_symmetric, axiom,
  (! [X,Y] : (region(X,Y) => region(Y,X)))).
fof(region_transitive, axiom,
  (! [X,Y,Z] :
    ((region(X,Y) & region(Y,Z)) => region(X,Z)))).
fof(regions, axiom,
  (region(1,2) & region(2,3) & region(4,5) &
   region(5,6) & region(7,8) & region(8,9) &
   ~region(1,4) & ~region(4,7) & ~region(1,7) )).
fof(all_different, axiom,
  (1 != 2 & 1 != 3 & 2 != 3 & 4 != 5 & 4 != 6 &
   5 != 6 & 7 != 8 & 7 != 9 & 8 != 9 )).
fof(at_most_one_in_each_region, axiom,
  (! [X1, Y1, X2, Y2] :
    ((region(X1,X2) & region(Y1,Y2) &
      grid(X1,Y1) = grid(X2,Y2)) =>
     (X1 = X2 & Y1 = Y2)))).

fof(puzzle, axiom,
  ( grid(1,1) = 6 & grid(1,4) = 2 & ... &
    grid(9,9) = 7 )).
```

(c) Sudoku in FOL (TPTP [124] syntax)

```
Given:
  type int Num
  Given(Num,Num,Num)
Find:
  Grid(Num,Num) : Num
Satisfying:
  ! r c n : Given(r,c,n) => Grid(r,c) = n.
  ! r c1 c2: Num(r) & Num(c1) & Num(c2) &
    Grid(r,c1) = Grid(r,c2) => c1 = c2.
  ! c r1 r2: Num(r1) & Num(r2) & Num(c) &
    Grid(r1,c) = Grid(r2,c) => r1 = r2.

declare { Same(Num,Num)
  Region(Num,Num,Num,Num) }

! r1 r2 c1 c2: (Grid(r1,c1) = Grid(r2,c2) &
  Region(r1,r2,c1,c2)) => (r1 = r2 & c1 = c2).
{ Same(n,n). Same(n1,n2) <- Same(n2,n1).
  Same(1,2). Same(1,3). Same(2,3). Same(4,5).
  Same(4,6). Same(5,6). Same(7,8). Same(7,9).
  Same(8,9). }
{ Region(r1,r2,c1,c2) <-
  Same(r1,r2) & Same(c1,c2). }

Data:
  Num = {1..9}
  Given = { 1,1,6; 1,4,2; ...; 9,9,7; }
```

(d) Sudoku in FOL/ID (IDP [88] syntax)

Figure 1-2: Sudoku in bounded relational logic, Alloy, FOL, and FOL/ID.

6	4	7	**2**	1	3	9	**5**	8
9	**1**	**8**	5	**6**	4	7	**2**	3
2	5	**3**	8	7	9	**4**	6	1
1	9	5	**6**	4	**7**	**8**	3	2
4	8	**2**	3	**5**	1	6	7	9
7	3	6	**9**	2	**8**	1	4	5
5	7	**4**	1	**9**	2	**3**	8	6
8	**2**	9	7	3	6	5	**1**	**4**
3	6	1	4	8	**5**	2	9	**7**

(a) Solution

num ↦ {⟨1⟩,⟨2⟩,⟨3⟩,⟨4⟩,⟨5⟩,⟨6⟩,⟨7⟩,⟨8⟩,⟨9⟩}
r1 ↦ {⟨1⟩,⟨2⟩,⟨3⟩}
r2 ↦ {⟨4⟩,⟨5⟩,⟨6⟩}
r3 ↦ {⟨7⟩,⟨8⟩,⟨9⟩}
grid ↦ {⟨1,1,6⟩,⟨1,2,4⟩,⟨1,3,7⟩,⟨1,4,2⟩,⟨1,5,1⟩,⟨1,6,3⟩,⟨1,7,9⟩,⟨1,8,5⟩,⟨1,9,8⟩,
⟨2,1,9⟩,⟨2,2,1⟩,⟨2,3,8⟩,⟨2,4,5⟩,⟨2,5,6⟩,⟨2,6,4⟩,⟨2,7,7⟩,⟨2,8,2⟩,⟨2,9,3⟩,
⟨3,1,2⟩,⟨3,2,5⟩,⟨3,3,3⟩,⟨3,4,8⟩,⟨3,5,7⟩,⟨3,6,9⟩,⟨3,7,4⟩,⟨3,8,6⟩,⟨3,9,1⟩,
⟨4,1,1⟩,⟨4,2,9⟩,⟨4,3,5⟩,⟨4,4,6⟩,⟨4,5,4⟩,⟨4,6,7⟩,⟨4,7,8⟩,⟨4,8,3⟩,⟨4,9,2⟩,
⟨5,1,4⟩,⟨5,2,8⟩,⟨5,3,2⟩,⟨5,4,3⟩,⟨5,5,5⟩,⟨5,6,1⟩,⟨5,7,6⟩,⟨5,8,7⟩,⟨5,9,9⟩,
⟨6,1,7⟩,⟨6,2,3⟩,⟨6,3,6⟩,⟨6,4,9⟩,⟨6,5,2⟩,⟨6,6,8⟩,⟨6,7,1⟩,⟨6,8,4⟩,⟨6,9,5⟩,
⟨7,1,5⟩,⟨7,2,7⟩,⟨7,3,4⟩,⟨7,4,1⟩,⟨7,5,9⟩,⟨7,6,2⟩,⟨7,7,3⟩,⟨7,8,8⟩,⟨7,9,6⟩,
⟨8,1,8⟩,⟨8,2,2⟩,⟨8,3,9⟩,⟨8,4,7⟩,⟨8,5,3⟩,⟨8,6,6⟩,⟨8,7,5⟩,⟨8,8,1⟩,⟨8,9,4⟩,
⟨9,1,3⟩,⟨9,2,6⟩,⟨9,3,1⟩,⟨9,4,4⟩,⟨9,5,8⟩,⟨9,6,5⟩,⟨9,7,2⟩,⟨9,8,9⟩,⟨9,9,7⟩}

(b) Kodkod model

Figure 1-3: Solution for the sample Sudoku puzzle.

no dedicated mechanism for accepting and exploiting partial information about a problem's solution. They take as inputs the specification to be analyzed and an integer bound on the size of the universe of discourse. The universe itself is implicit; the user cannot name its elements and use them to explicitly pin down known parts of the model. If the specification does have a *partial model*—i.e. a partial binding of variables to values—which the model finder should extend, it can only be encoded in the form of additional constraints. Some ad hoc techniques can be used to infer partial bindings from these constraints. For example, Alloy3 [117] infers that the relations `N1` through `N9` can be bound to distinct elements in the implicit universe because they are constrained to be disjoint singletons (Fig. 1-2b, lines 3-5). In general, however, partial models increase the difficulty of the problem to be solved, resulting in performance degradation.

In contrast to traditional model finders, *model extenders* [96], such as IDP1.3 [88] and Kodkod, allow the user to name the elements in the universe of discourse and to use them to pin down parts of the solution. IDP1.3 allows only complete bindings of relations to values to be specified (e.g. Fig. 1-2d, lines 23-24). Partial bindings for relations such as `Grid` must still be specified implicitly, using additional constraints (line 7). Kodkod, on the other hand, allows the specification of precise lower and upper bounds on the value of each relation. These are exploited with new techniques (Chapters 2-3) for translating relational logic to SAT so that model finding difficulty

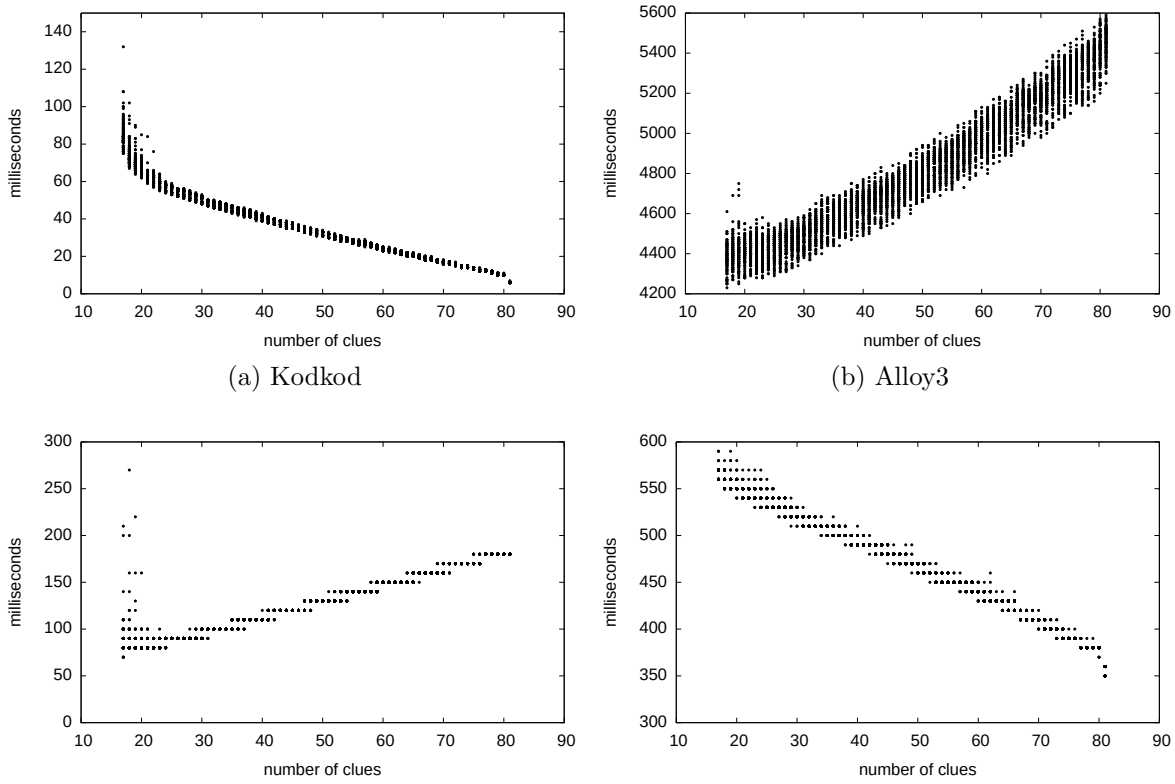
varies inversely with the size of the available partial model.

The impact of partial models on various model finders is observable even on small problems, like Sudoku. Figure 1-4, for example, shows the behavior of four state-of-the-art[3] SAT-based model finders on a progression of 6600 Sudoku puzzles with different numbers of givens, or *clues*. The puzzle progression was constructed iteratively from 100 original puzzles which were randomly selected from a public database of 17-clue Sudokus [110]. Each original puzzle was expanded into 66 variants, with each consecutive variant differing from its predecessor by an additional, randomly chosen clue. Using the formulations in Fig. 1-2 as templates, the puzzles were specified in the input languages of Paradox2.3 (first order logic), IDP1.3 (first order logic with inductive definitions), Alloy3 (relational logic), and Kodkod (bounded relational logic). The data points on the plots in Fig. 1-4 represent the CPU time, in milliseconds, taken by the model finders to discover the models of these specifications. All experiments were performed on a 2×3 GHz Dual-Core Intel Xeon with 2 GB of RAM. Alloy3, Paradox2.3, and Kodkod were configured with MiniSat [43] as their SAT solver, while IDP1.3 uses MiniSatID [89], an extension of MiniSat for propositional logic with inductive definitions.

The performance of the traditional model finders, Alloy3 (Fig. 1-4b) and Paradox2.3 (Fig. 1-4c), degrades steadily as the number of clues for each puzzle increases. On average, Alloy3 is 25% slower on a puzzle with a fully specified grid than on a puzzle with 17 clues, whereaes Paradox2.3 is twice as slow on a full grid. The performance of the model extenders (Figs. 1-4a and 1-4d), on the other hand, improves with the increasing number of clues. The average improvement of Kodkod is 14 times on a full grid, while that of IDP1.3 is about 1.5 times.

The trends in Fig. 1-4 present a fair picture of the relative performance of Kodkod and other SAT-based tools on a wide range of problems. Due to the new translation techniques described in Chapters 2-3, Kodkod is roughly an order of magnitude faster than Alloy3 with and without partial models. It is also faster than IDP1.3 and Para-

[3]With the exception of Alloy3, which has been superseded by a new version based on Kodkod, all model finders compared to Kodkod throughout this thesis represent the current state-of-the-art.



(a) Kodkod (b) Alloy3

(c) Paradox2.3 (d) IDP1.3

Figure 1-4: Effect of partial models on the performance of SAT-based model finders, when applied to a progression of 6600 Sudoku puzzles. The x-axis of each graph shows the number of clues in a puzzle, and the y-axis shows the time, in milliseconds, taken by a given model finder to solve a puzzle with the specified number of clues. Note that Paradox2.3, IDP1.3 and Kodkod solved many of the puzzles with the same number of clues in the same amount of time (or within a few milliseconds of one another), so many of the points on their performance graphs overlap.

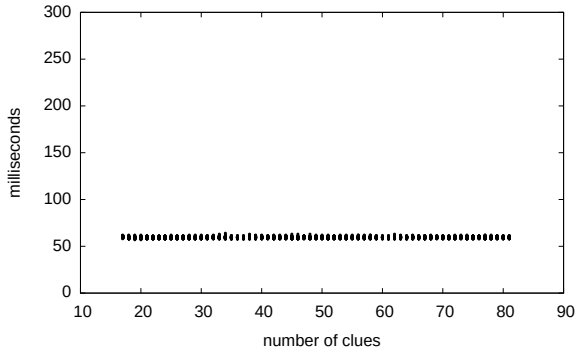


Figure 1-5: Effect of partial models on the performance of a dedicated Sudoku solver, when applied to a progression of 6600 Sudoku puzzles.

dox2.3 on the problems that this thesis targets—that is, specifications with partial models and intricate constraints over relational structures.[4] For a potential user of these tools, however, the interesting question is not necessarily how they compare to one another. Rather, the interesting practical question is how they might compare to a custom translation to SAT.

This question is hard to answer in general, but a comparison with existing custom translations is promising. Figure 1-5, for example, shows the performance of a dedicated, SAT-based Sudoku solver on the same 6600 puzzles solved with Kodkod and the three other model finders. The solver consists of 150 lines of Java code that generate Lynce and Ouaknine's optimized SAT encoding [86] of a given Sudoku puzzle, followed by an invocation of MiniSat on the generated file. The program took a few hours to write and debug, as the description of the encoding [86] contained several errors and ambiguities that had to be resolved during implementation. The Kodkod-based solver, in contrast, consists of about 50 lines of Java API calls that directly correspond to the text in Fig. 1-2a; it took an hour to implement.

The performance of the two solvers, as Figs. 1-4a and 1-5 show, is comparable. The dedicated solver is slightly faster on 17-clue Sudokus, and the Kodkod solver is faster on full grids. The custom solver's performance remains constant as the number of clues in each puzzle increases because it handles the additional clues by feeding extra *unit clauses* to the SAT solver: adding these clauses takes negligible

[4]Problems that are better suited to other tools than to Kodkod are discussed in Chapter 5.

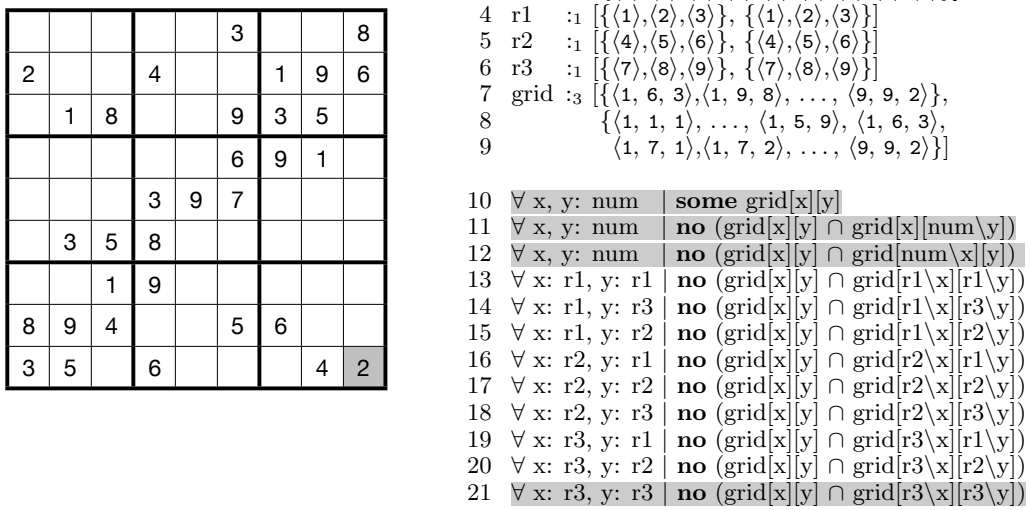
time, and, given that the translation time heavily dominates the SAT solving time, their positive effect on MiniSat's performance is unobservable. Both implementations were also applied to 16×16 and 25×25 puzzles, with similar outcomes.[5] The solvers based on other model finders were unable to solve Sudokus larger than 16×16.

1.3 Minimal unsatisfiable core extraction

When a specification has no models in a given universe, most model finders [25, 51, 68, 70, 88, 91, 122, 151, 152] simply report that it is *unsatisfiable* in that universe and offer no further feedback. But many applications need to know the cause of a specification's unsatisfiability, either to take corrective action (in the case of declarative configuration [133]) or to check that no models exist for the right reasons (in the case of bounded verification [31, 21]). A bounded verifier [31, 21], for example, checks a system description $s_1 \wedge \ldots \wedge s_n$ against a property p in some finite universe by looking for models of the formula $s_1 \wedge \ldots \wedge s_n \wedge \neg p$ in that universe. If found, such a model, or a *counterexample*, represents a behavior of the system that violates p. A lack of models, however, does not necessarily mean that the analysis was successful. If no models exist because the system description is overconstrained, or because the property is a tautology, the analysis is considered to have failed due to a vacuity error.

A cause of unsatisfiability of a given specification, expressed as a subset of the specification's constraints that is itself unsatisfiable, is called an *unsatisfiable core*. Every unsatisfiable core includes one or more *critical constraints* that cannot be removed without making the remainder of the core satisfiable. Non-critical constraints, if any, are irrelevant to unsatisfiability and generally decrease a core's utility both for diagnosing faulty configurations [133] and for checking the results of a bounded analysis [129]. Cores that include only critical constraints are said to be *minimal*.

[5] The bounded relational encoding of Sudoku used in these experiments (Fig. 1-2a) is the easiest to understand, but it does not produce the most optimal SAT formulas. An alternative encoding, where the multiplicity **some** on line 10 is replaced by **one** and each constraint of the form '$\forall$ x: r_i, y: r_j | **no** (grid[x][y] $\cap$ grid[r_i\x][r_j\y])' is loosened to 'num $\subseteq$ grid[r_i][r_j],' actually produces a SAT encoding that is more efficient than the custom translation across the board. For example, MiniSat solves the SAT formula corresponding to the alternative encoding of a 64×64 Sudoku ten times faster than the custom SAT encoding of the same puzzle.



					3			8
2			4			1	9	6
	1	8			9	3	5	
					6	9	1	
			3	9	7			
	3	5	8					
		1	9					
8	9	4			5	6		
3	5		6				4	2

(a) An unsatisfiable Sudoku puzzle

```
1  {1, 2, 3, 4, 5, 6, 7, 8, 9}

2  num :1 [{⟨1⟩,⟨2⟩,⟨3⟩,⟨4⟩,⟨5⟩,⟨6⟩,⟨7⟩,⟨8⟩,⟨9⟩},
3          {⟨1⟩,⟨2⟩,⟨3⟩,⟨4⟩,⟨5⟩,⟨6⟩,⟨7⟩,⟨8⟩,⟨9⟩}]
4  r1    :1 [{⟨1⟩,⟨2⟩,⟨3⟩}, {⟨1⟩,⟨2⟩,⟨3⟩}]
5  r2    :1 [{⟨4⟩,⟨5⟩,⟨6⟩}, {⟨4⟩,⟨5⟩,⟨6⟩}]
6  r3    :1 [{⟨7⟩,⟨8⟩,⟨9⟩}, {⟨7⟩,⟨8⟩,⟨9⟩}]
7  grid :3 [{⟨1, 6, 3⟩,⟨1, 9, 8⟩, ..., ⟨9, 9, 2⟩},
8           {⟨1, 1, 1⟩, ..., ⟨1, 5, 9⟩, ⟨1, 6, 3⟩,
9            ⟨1, 7, 1⟩,⟨1, 7, 2⟩, ..., ⟨9, 9, 2⟩}]

10 ∀ x, y: num   | some grid[x][y]
11 ∀ x, y: num   | no (grid[x][y] ∩ grid[x][num\y])
12 ∀ x, y: num   | no (grid[x][y] ∩ grid[num\x][y])
13 ∀ x: r1, y: r1 | no (grid[x][y] ∩ grid[r1\x][r1\y])
14 ∀ x: r1, y: r3 | no (grid[x][y] ∩ grid[r1\x][r3\y])
15 ∀ x: r1, y: r2 | no (grid[x][y] ∩ grid[r1\x][r2\y])
16 ∀ x: r2, y: r1 | no (grid[x][y] ∩ grid[r2\x][r1\y])
17 ∀ x: r2, y: r2 | no (grid[x][y] ∩ grid[r2\x][r2\y])
18 ∀ x: r2, y: r3 | no (grid[x][y] ∩ grid[r2\x][r3\y])
19 ∀ x: r3, y: r1 | no (grid[x][y] ∩ grid[r3\x][r1\y])
20 ∀ x: r3, y: r2 | no (grid[x][y] ∩ grid[r3\x][r2\y])
21 ∀ x: r3, y: r3 | no (grid[x][y] ∩ grid[r3\x][r3\y])
```

(b) Core of the puzzle (highlighted)

Figure 1-6: An unsatisfiable Sudoku puzzle and its core.

Figure 1-6 shows an example of using a minimal core to diagnose a faulty Sudoku configuration. The highlighted parts of Fig. 1-6b comprise a set of critical constraints that cannot be satisfied by the puzzle in Fig. 1-6a. The row (line 11) and column (line 12) constraints rule out '9' as a valid value for any of the blank cells in the bottom right region. The values '2', '4', and '6' are also ruled out (line 21), leaving five unique numbers and six empty cells. By the pigeonhole principle, these cells cannot be filled (as required by line 10) without repeating some value (which is disallowed by line 21). Removing '2' from the highlighted cell fixes the puzzle.

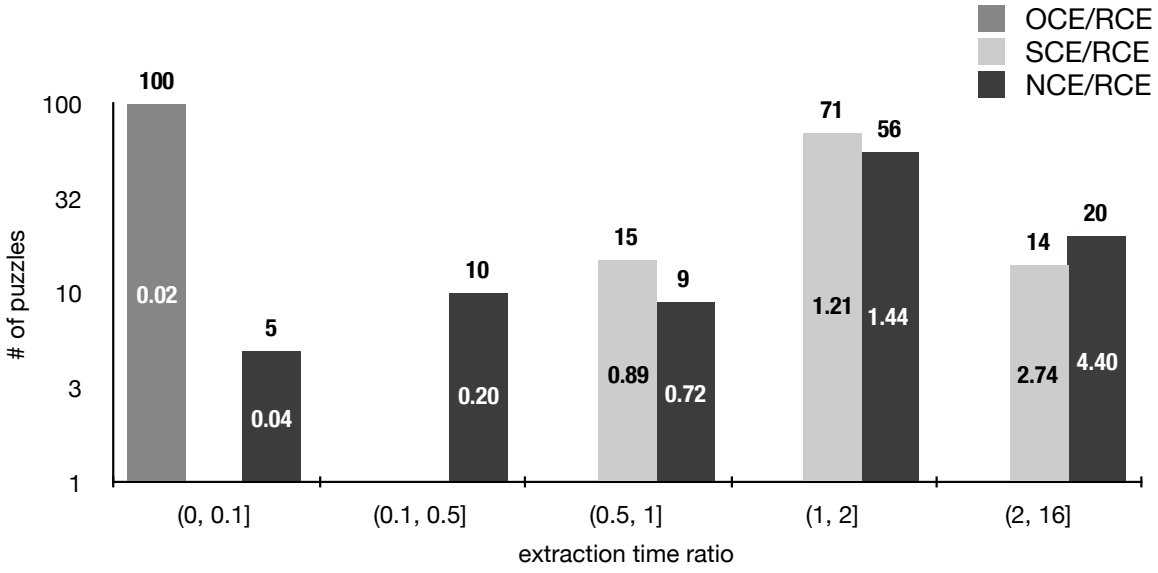
The problem of unsatisfiable core extraction has been studied extensively in the SAT community, and there are many efficient algorithms for finding small or minimal cores of propositional formulas [32, 60, 61, 59, 79, 85, 97, 102, 153]. A simple facility for leveraging these algorithms in the context of SAT-based model finding has been implemented as a feature of Alloy3. The underlying mechanism [118] involves translating a specification to a SAT problem; finding a core of the translation using an existing SAT-level algorithm [153]; and mapping the clauses from the boolean core

back to the specification constraints from which they were generated. The resulting specification-level core is guaranteed to be sound (i.e. unsatisfiable) [118], but it is not guaranteed to be minimal or even small.

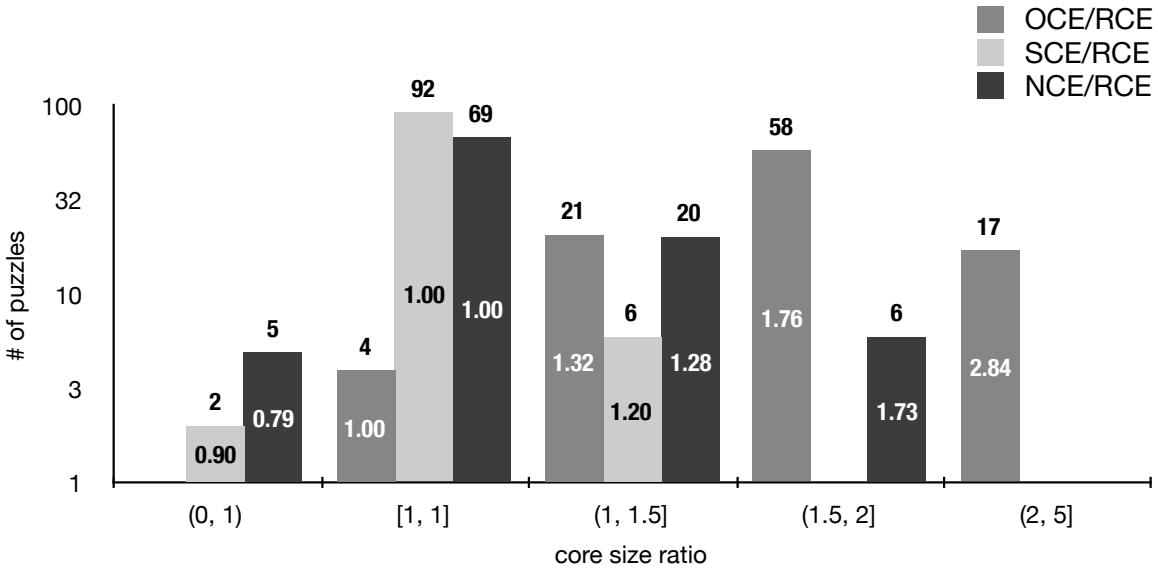
Recycling core extraction (RCE) is a new SAT-based algorithm for finding cores of declarative specifications that are both sound and minimal. It has two key ideas (Chapter 4). The first idea is to lift the minimization process from the boolean level to the specification level. Instead of attempting to minimize the boolean core, RCE maps it back and then minimizes the resulting specification-level core, by removing candidate constraints and testing the remainder for satisfiability. The second idea is to use the proof of unsatisfiability returned by the SAT solver, and the mapping between the specification constraints and the translation clauses, to identify the boolean clauses that were inferred by the solver and that still hold when a specification-level constraint is removed. By adding these clauses to the translation of a candidate core, RCE allows the solver to reuse previously made inferences.

Both ideas employed by RCE are straightforward and relatively easy to implement, but have dramatic consequences on the quality of the results obtained and the performance of the analysis. Compared to NCE and SCE [129], two variants of RCE that lack some of its optimizations, RCE is roughly 20 to 30 times faster on hard problems and 10 to 60 percent faster on easier problems (Chapter 4). It is much slower than Alloy3's core extractor, OCE [118], which does not guarantee minimality. Most cores produced by OCE, however, include large proportions of irrelevant constraints, making them hard to use in practice.

Figure 1-7, for example, compares RCE with OCE, NCE and SCE on a set of 100 unsatisfiable Sudokus. The puzzles were constructed from 100 randomly selected 16×16 Sudokus [63], each of which was augmented with a randomly chosen, faulty clue. Figure 1-7a shows the number of puzzles on which RCE is faster (or slower) than each competing algorithm by a factor that falls within the given range. Figure 1-7b shows the number of puzzles whose RCE cores are smaller (or larger) than those of the competing algorithms by a factor that falls within the given range. All four extractors were implemented in Kodkod, configured with MiniSat, and all experiments



(a) Extraction times



(b) Core sizes

Figure 1-7: Comparison of SAT-based core extractors on 100 unsatisfiable Sudokus. Figure (a) shows the number of puzzles on which RCE is faster, or slower, than each competing algorithm by a factor that falls within the given range. Figure (b) shows the number of puzzles whose RCE cores are smaller, or larger, than those of the competing algorithms by a factor that falls within the given range. Both histograms are shown on a logarithmic scale. The number above each column specifies its height, and the middle number is the average extraction time (or core size) ratio for the puzzles in the given category.

were performed on a 2×3 GHz Dual-Core Intel Xeon with 2 GB of RAM.

Because SCE is essentially RCE without the clause recycling optimization, they usually end up finding the same minimal core. Of the 100 cores extracted by each algorithm, 92 were the same (Fig. 1-7b). RCE was faster than SCE on 85 of the problems (Fig. 1-7a). Since Sudoku cores are easy to find[6], the average speed up of RCE over SCE is about 37%. NCE is the most naive of the three approaches and does not exploit the boolean-level cores in any way. It found a different minimal core than RCE for 31 of the puzzles. For 26 of those, the NCE core was larger than the RCE core, and indeed, easier to find, as shown in Fig. 1-7a. Nonetheless, RCE was, on average, 77% faster than NCE. OCE outperformed all three minimality-guaranteeing algorithms by large margins. However, only four OCE cores were minimal, and more than half the constraints in 75 of its cores were irrelevant.

1.4 Summary of contributions

This thesis contributes a collection of techniques (Fig. 1-8) that enable easy and efficient use of SAT for declarative problem solving. They include:

1. A new problem-description language that extends the relational logic of Alloy [69] with a mechanism for specifying precise bounds on the values of free variables (Chapter 2); the bounds enable efficient encoding and exploitation of *partial models*.

2. A new translation to SAT that uses sparse-matrices and auto-compacting circuits (Chapter 2); the resulting boolean encoding is significantly smaller, faster to produce, and easier to solve than the encodings obtained with previously published techniques [40, 119].

3. A new algorithm for identifying symmetries that works in the presence of arbitrary bounds on free variables (Chapter 3); the algorithm employs a fast greedy

[6] Chapter 4 describes a metric for approximating the difficulty of a given problem for a particular core extraction algorithm.

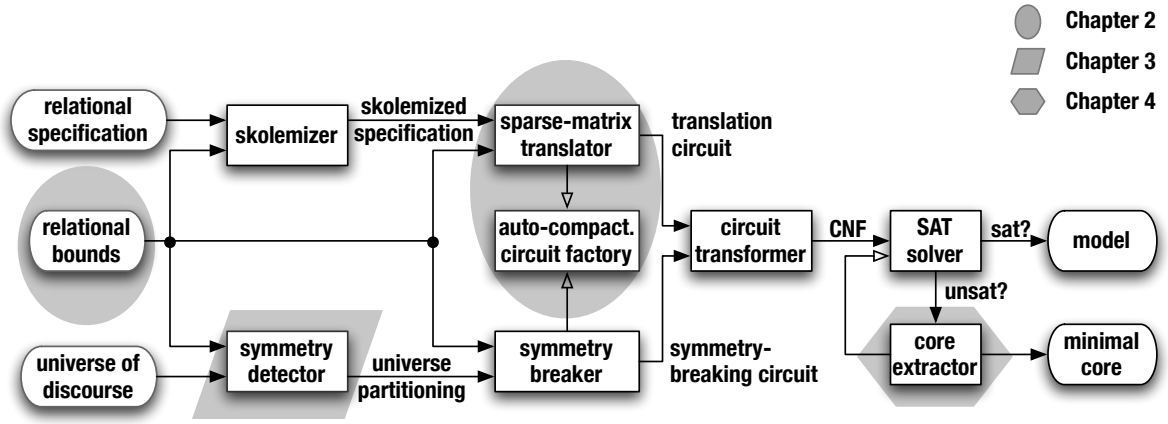


Figure 1-8: Summary of contributions. The contributions of this thesis are highlighted with gray shading; the remaining parts of the framework are implemented using standard techniques. Filled arrows represent data and control flow between components. Clear arrows represent usage relationships between components.

technique that is both effective in practice and scales better than a complete method based on graph automorphism detection.

4. A new algorithm for finding minimal unsatisfiable cores that recycles inferences made at the boolean level to speed up core extraction at the specification level (Chapter 4); the algorithm is much faster on hard problems than related approaches [129], and its cores are much smaller than those obtained with non-minimal extractors [118].

These techniques have been prototyped in Kodkod, a new engine for finding models and cores of large relational specifications. The engine significantly outperforms existing model finders [13, 25, 88, 93, 117, 152] on problems with partial models, rich type hierarchies, and low-arity relations. As such problems arise in a wide range of declarative configuration and analysis settings, Kodkod has been used in several configuration [149, 101], test-case generation [114, 135] and bounded verification [21, 137, 31, 126, 34] tools (Table 1.1). These applications have served as a comprehensive testbed for Kodkod, revealing both its strengths and limitations. The latter open up a number of promising directions for future work (Chapter 5).

bounded verification	**Alloy4 [21] analyzer for the Alloy language.** Alloy4 uses Kodkod for simulation and checking of software designs expressed in the Alloy modeling language. It is 2 to 10 times faster than Alloy3 and provides a precise debugger for overconstrained specifications that is based on Kodkod's core extraction facility. Like its predecessor, Alloy4 has been used for modeling and analysis in a variety of contexts, e.g. filesystems [75], security [81], and requirements engineering [113].
	Kato [136, 137] slicer for declarative specifications. Kato uses Kodkod to slice and solve Alloy specifications of complex data structures, such as red-black trees and doubly-linked lists. The tool splits a given specification into base and derived constraints using a heuristically selected slicing criterion. The resulting slices are then fed to Kodkod separately so that a model of the base slice becomes a partial model for the derived slice. The final model, if any, satisfies the entire specification. Because the subproblems are usually easier to solve than the entire specification, Kato scales better than Alloy4 on specifications that are amenable to slicing.
	Forge [30, 31], Karun [126], and Minatur [34] bounded code verifiers. These tools use Kodkod to check the methods of a Java class against rich structural properties. The basic analysis [138] involves encoding the behavior of a given method, within a bounded heap, as a set of relational constraints that are conjoined with the negation of the property being checked. A model of the resulting specification, if one exists, represents a concrete trace of the method that violates the property. All three tools have been used to find previously unknown bugs in open source systems, including a heavily tested job scheduler [126] and an electronic vote tallying system that had been already checked with a theorem prover [30].
test-case generation	**Kesit [134, 135] test-case generator for software product lines.** Kesit uses Kodkod to incrementally generate tests for products in a software product line. Given a product that is composed of a base and a set of features, Kesit first generates a test suite for the base by feeding a specification of its functionality to Kodkod. The tests from the resulting test-suite (derived from the models of the specification) are then used as partial models for the specification of the features. Because the product is specified by the conjunction of the base and feature constraints, the final set of models is a valid test-suite for the product as a whole. Kesit's approach to test-case generation has been shown to scale over 60 times better than previous approaches to specification-based testing [76, 77].
	Whispec [114] test-case generator for white-box testing. Whispec uses Kodkod to generate white-box tests for methods that manipulate structurally complex data. To test a method, Whispec first obtains a model of the constraints that specify the method's preconditions. The model is then converted to a test input, which is fed to the method. A path condition of the resulting execution is recorded, and new path conditions (for unexplored paths) are constructed by negating the branch predicates in the recorded path. Next, Kodkod is applied to the conjunction of the pre-condition and one of the new path conditions to obtain a new test input. This process is repeated until the desired level of code coverage is reached. Whispec has been shown to generate significantly smaller test suites, with better coverage, than previous approaches.
declarative configuration	**ConfigAssure [100, 101] system for network configuration.** ConfigAssure uses Kodkod for synthesis, diagnosis and repair of network configurations. Given a partially configured network and set of configuration requirements, ConfigAssure generates a relational satisfiability problem that is fed to Kodkod. If a model is found, it is translated back to a set of configuration assignments: nodes to subnets, IP addresses to nodes, etc. Otherwise, the tool obtains an unsatisfiable core of the configuration formula and repairs the input configuration by removing the configuration assignments that are in the core. ConfigAssure has been shown to scale to realistic networks with hundreds of nodes and subnets.
	A declarative course scheduler [148, 149]. The scheduler uses Kodkod to plan a student's schedule based on the overall requirements and prerequisite dependencies of a degree program; courses taken so far; and the schedule according to which particular courses are offered. The scheduler is offered as a free, web-based service to MIT students. Its performance is competitive with that of conventional planners.

Table 1.1: Recent applications of Kodkod.

Chapter 2

From Relational to Boolean Logic

The relational logic of Alloy [69] combines the quantifiers of first order logic with the operators of relational algebra. The logic and the language were designed for modeling software abstractions, their properties and invariants. But unlike the logics of traditional modeling languages [123, 143], Alloy makes no distinction between relations, sets and scalars: sets are relations with one column, and scalars are singleton sets. Treating everything as a relation makes the logic more uniform and, in some ways, easier to use than traditional modeling languages. Applying a partial function outside of its domain, for example, simply yields the empty set, eliminating the need for special undefined values.

The generality and versatility of Alloy's logic have prompted several attempts to use its model finder, Alloy3 [117], as a generic constraint solving engine for declarative configuration [99] and analysis [76, 138]. These efforts, however, were hampered by two key limitations of the Alloy system. First, Alloy has no notion of a partial model. If a partial solution, or a model, is available for a set of Alloy constraints, it can only be provided to the solver in the form of additional constraints. Because the solver is essentially forced to rediscover the partial model from the constraints, this strategy does not scale well in practice. Second, Alloy3 was designed for *small-scope analysis* [69] of hand-crafted specifications of software systems, so it performs poorly on problems with large universes or large, automatically generated specifications.

Kodkod is a new tool that is designed for use as a generic relational engine.

Its model finder, like Alloy3, works by translating relational to boolean logic and applying an off-the-shelf SAT solver to the resulting boolean formula. Unlike Alloy3, however, Kodkod scales in the presence of partial models, and it can handle large universes and specifications. This chapter describes the elements of Kodkod's logic and model finder that are key to its ability to produce compact SAT formulas, with and without partial models. Next chapter describes a technique that is used for making the produced formulas slightly larger but easier to solve.

2.1 Bounded relational logic

A specification in the relational logic of Alloy is a collection of constraints on a set of relational variables. A model of an Alloy specification is a binding of its free variables to relational constants that makes the specification true. These constants are sets of tuples, drawn from a common universe of uninterpreted elements, or *atoms*. The universe itself is implicit, in the sense that its elements cannot be named or referenced through any syntactic construct of the logic. As a result, there is no direct way to specify relational constants in Alloy. If a partial binding of relations to constants—i.e. a *partial model*—is available for a specification, it must be encoded indirectly, with constraints that use additional variables (e.g. `N1` through `N9` in Fig. 1-2b) as implicit handles to distinct atoms. While sound, this encoding of partial models is impractical because the additional variables and constraints make the resulting model finding problem larger rather than smaller.

The *bounded relational logic* of Kodkod (Fig. 2-1) extends Alloy in two ways: the universe of atoms for a specification is made explicit, and the value of each free variable is explicitly bound, above and below, by relational constants. A *problem* description in Kodkod's logic consists of an Alloy specification, augmented with a *universe declaration* and a set of *bound declarations*. The universe declaration specifies the set of atoms from which a model of the specification is to be drawn. The bound declarations bound the value of each relation with two relational constants drawn from the declared universe: an *upper bound*, which contains the tuples that the relation

problem := universe relBound* formula*

universe := { atom[, atom]* }
relBound := var :$_{\text{arity}}$ [constant, constant]
constant := {tuple[, tuple]*} | {}[×{}]*
tuple := ⟨atom[, atom]*⟩

atom, var := identifier
arity := positive integer

formula :=

no expr	*empty*
| **lone** expr	*at most one*
| **one** expr	*exactly one*
| **some** expr	*non-empty*
| expr ⊆ expr	*subset*
| expr = expr	*equal*
| ¬ formula	*negation*
| formula ∧ formula	*conjunction*
| formula ∨ formula	*disjunction*
| formula ⇒ formula	*implication*
| formula ⇔ formula	*equivalence*
| ∀ varDecls **|** formula	*universal*
| ∃ varDecls **|** formula	*existential*

expr :=

var	*variable*
| ˜expr	*transpose*
| ˆexpr	*closure*
| *expr	*reflex. closure*
| expr ∪ expr	*union*
| expr ∩ expr	*intersection*
| expr \ expr	*difference*
| expr . expr	*join*
| expr → expr	*product*
| formula ? expr : expr	*if-then-else*
| {varDecls **|** formula}	*comprehension*

varDecls := var : expr[, var : expr]*

(a) Abstract syntax

P : problem → binding → boolean
R : relBound → binding → boolean
F : formula → binding → boolean
E : expr → binding → constant
binding : var → constant

$P[\![\{a_1, \ldots, a_n\}\ r_1 \ldots r_j \quad f_1 \ldots f_m]\!]\mathrm{b} :=$
$\quad \mathrm{R}[\![r_1]\!]\mathrm{b} \wedge \ldots \wedge \mathrm{R}[\![r_j]\!]\mathrm{b} \wedge \mathrm{F}[\![f_1]\!]\mathrm{b} \wedge \ldots \wedge \mathrm{F}[\![f_m]\!]\mathrm{b}$

$\mathrm{R}[\![v :_k [l, u]]\!]\mathrm{b} := l \subseteq \mathrm{b}(v) \subseteq u$

$\mathrm{F}[\![$**no** $p]\!]\mathrm{b}$	:=	$\lvert \mathrm{E}[\![p]\!]\mathrm{b} \rvert = 0$
$\mathrm{F}[\![$**lone** $p]\!]\mathrm{b}$	:=	$\lvert \mathrm{E}[\![p]\!]\mathrm{b} \rvert \leq 1$
$\mathrm{F}[\![$**one** $p]\!]\mathrm{b}$	:=	$\lvert \mathrm{E}[\![p]\!]\mathrm{b} \rvert = 1$
$\mathrm{F}[\![$**some** $p]\!]\mathrm{b}$	:=	$\lvert \mathrm{E}[\![p]\!]\mathrm{b} \rvert > 0$
$\mathrm{F}[\![p \subseteq q]\!]\mathrm{b}$	:=	$\mathrm{E}[\![p]\!]\mathrm{b} \subseteq \mathrm{E}[\![q]\!]\mathrm{b}$
$\mathrm{F}[\![p = q]\!]\mathrm{b}$	:=	$\mathrm{E}[\![p]\!]\mathrm{b} = \mathrm{E}[\![q]\!]\mathrm{b}$
$\mathrm{F}[\![\neg f]\!]\mathrm{b}$	:=	$\neg \mathrm{F}[\![f]\!]\mathrm{b}$
$\mathrm{F}[\![f \wedge g]\!]\mathrm{b}$	:=	$\mathrm{F}[\![f]\!]\mathrm{b} \wedge \mathrm{F}[\![g]\!]\mathrm{b}$
$\mathrm{F}[\![f \vee g]\!]\mathrm{b}$	:=	$\mathrm{F}[\![f]\!]\mathrm{b} \vee \mathrm{F}[\![g]\!]\mathrm{b}$
$\mathrm{F}[\![f \Rightarrow g]\!]\mathrm{b}$	:=	$\mathrm{F}[\![f]\!]\mathrm{b} \Rightarrow \mathrm{F}[\![g]\!]\mathrm{b}$
$\mathrm{F}[\![f \Leftrightarrow g]\!]\mathrm{b}$	:=	$\mathrm{F}[\![f]\!]\mathrm{b} \Leftrightarrow \mathrm{F}[\![g]\!]\mathrm{b}$

$\mathrm{F}[\![\forall\ v_1 : e_1, ..., v_n : e_n \mid f]\!]\mathrm{b} :=$
$\quad \bigwedge_{s \in \mathrm{E}[\![e_1]\!]\mathrm{b}} (\mathrm{F}[\![\forall\ v_2 : e_2, ..., v_n : e_n \mid f]\!](\mathrm{b} \oplus v_1 \mapsto \{\langle s \rangle\})$
$\mathrm{F}[\![\exists\ v_1 : e_1, ..., v_n : e_n \mid f]\!]\mathrm{b} :=$
$\quad \bigvee_{s \in \mathrm{E}[\![e_1]\!]\mathrm{b}} (\mathrm{F}[\![\exists\ v_2 : e_2, ..., v_n : e_n \mid f]\!](\mathrm{b} \oplus v_1 \mapsto \{\langle s \rangle\})$

$\mathrm{E}[\![v]\!]\mathrm{b}$	:=	$\mathrm{b}(v)$
$\mathrm{E}[\![\tilde{}p]\!]\mathrm{b}$	:=	$\{\langle p_2, p_1 \rangle \mid \langle p_1, p_2 \rangle \in \mathrm{E}[\![p]\!]\mathrm{b}\}$
$\mathrm{E}[\![\hat{}p]\!]\mathrm{b}$	:=	$\{\langle p_1, p_n \rangle \mid \exists\ p_2, ..., p_{n-1} \mid \langle p_1, p_2 \rangle, ..., \langle p_{n-1}, p_n \rangle \in \mathrm{E}[\![p]\!]\mathrm{b}\}$
$\mathrm{E}[\![*p]\!]\mathrm{b}$	:=	$\mathrm{E}[\![\hat{}p]\!]\mathrm{b} \cup \{\langle \mathrm{p_1}, \mathrm{p_1} \rangle \mid \mathrm{true}\}$
$\mathrm{E}[\![p \cup q]\!]\mathrm{b}$	:=	$\mathrm{E}[\![p]\!]\mathrm{b} \cup \mathrm{E}[\![q]\!]\mathrm{b}$
$\mathrm{E}[\![p \cap q]\!]\mathrm{b}$	:=	$\mathrm{E}[\![p]\!]\mathrm{b} \cap \mathrm{E}[\![q]\!]\mathrm{b}$
$\mathrm{E}[\![p \setminus q]\!]\mathrm{b}$	:=	$\mathrm{E}[\![p]\!]\mathrm{b} \setminus \mathrm{E}[\![q]\!]\mathrm{b}$
$\mathrm{E}[\![p\ .\ q]\!]\mathrm{b}$	:=	$\{\langle p_1, ..., p_{n-1}, q_2, ..., q_m \rangle \mid \langle p_1, ..., p_n \rangle \in \mathrm{E}[\![p]\!]\mathrm{b} \wedge \langle \mathrm{q_1}, ..., \mathrm{q_m} \rangle \in \mathrm{E}[\![q]\!]\mathrm{b}\ \}$
$\mathrm{E}[\![p \to q]\!]\mathrm{b}$	:=	$\{\langle p_1, ..., p_n, q_1, ..., q_m \rangle \mid \langle p_1, ..., p_n \rangle \in \mathrm{E}[\![p]\!]\mathrm{b} \wedge \langle \mathrm{q_1}, ..., \mathrm{q_m} \rangle \in \mathrm{E}[\![q]\!]\mathrm{b}\ \}$
$\mathrm{E}[\![f\ ?\ p : q]\!]\mathrm{b}$	:=	if $\mathrm{F}[\![f]\!]\mathrm{b}$ then $\mathrm{E}[\![p]\!]\mathrm{b}$ else $\mathrm{E}[\![q]\!]\mathrm{b}$

$\mathrm{E}[\![\{v_1 : e_1, ..., v_n : e_n \mid f\}]\!]\mathrm{b} :=$
$\quad \{\langle s_1, ..., s_n \rangle \mid s_1 \in \mathrm{E}[\![e_1]\!]\mathrm{b} \wedge \mathrm{s_2} \in \mathrm{E}[\![e_2]\!](\mathrm{b} \oplus \mathrm{v_1} \mapsto \{\langle \mathrm{s_1} \rangle\})$
$\quad\quad \wedge \ldots \wedge s_n \in \mathrm{E}[\![e_n]\!](\mathrm{b} \oplus \bigcup_{i=1}^{n-1} v_i \mapsto \{\langle s_i \rangle\})$
$\quad\quad \wedge \mathrm{F}[\![f]\!](\mathrm{b} \oplus \bigcup_{i=1}^{n} v_i \mapsto \{\langle s_i \rangle\})\}$

(b) Semantics

Figure 2-1: Syntax and semantics of bounded relational logic. Because Kodkod is designed as a Java API, the users communicate with it by constructing universes, bounds, and formulas via API calls. The syntax presented here is for illustrative purposes only. Mixed and zero arity expressions are not allowed. The arity of a relation is the same as the arity of its bounding constants. There is exactly one bound declaration $v :_k [l, u]$ for each relation v that appears in a problem description. The empty set $\{\}$ has arity 1. The empty set of arity k is represented by taking the cross product of the empty set with itself k times, i.e. $\{\} \times \ldots \times \{\}$.

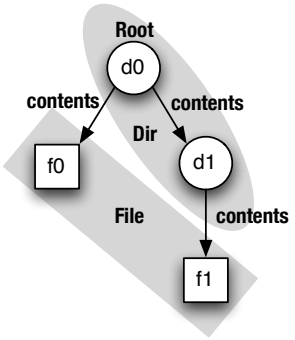
```
1  {d0, d1, f0, f1, f2}

2  File     :1 [{}, {⟨f0⟩,⟨f1⟩,⟨f2⟩}]
3  Dir      :1 [{}, {⟨d0⟩,⟨d1⟩}]
4  Root     :1 [{⟨d0⟩}, {⟨d0⟩}]
5  contents :2 [{⟨d0, d1⟩},
               {⟨d0, d0⟩, ⟨d0, d1⟩, ⟨d0, f0⟩, ⟨d0, f1⟩, ⟨d0, f2⟩,
                ⟨d1, d0⟩, ⟨d1, d1⟩, ⟨d1, f0⟩, ⟨d1, f1⟩, ⟨d1, f2⟩}]

6  contents ⊆ Dir → (Dir ∪ File)
7  ∀ d: Dir | ¬(d ⊆ d.^contents)
8  Root ⊆ Dir
9  (File ∪ Dir) ⊆ Root.*contents
```

(a) Problem description

```
File     ↦ {⟨f0⟩,⟨f1⟩}
Dir      ↦ {⟨d0⟩,⟨d1⟩}
Root     ↦ {⟨d0⟩}
contents ↦ {⟨d0, d1⟩, ⟨d0, f0⟩, ⟨d1, f1⟩}
```



(b) A sample model

Figure 2-2: A toy filesystem.

may include, and a *lower bound*, which contains the tuples that the relation *must* include. Collectively, the lower bounds define a partial model, and the upper bounds limit the pool of values available for completing that partial model.

Figure 2-2a demonstrates the key features of Kodkod's logic on a toy specification of a filesystem. The specification (lines 6-9) has four free variables: the binary relation contents and the unary relations File, Dir, and Root. File and Dir represent the files and directories that make up the filesystem. The contents relation is an acyclic mapping of directories to their contents, which may be files or directories (line 6-7). Root represents the root of the filesystem: it is a directory (line 8) from which all files and directories are reachable by following the contents relation zero or more times (line 9).

The filesystem universe consists of five atoms (line 1). These are used to construct lower and upper bounds on the free variables (lines 2-5). The upper bounds on File and Dir partition the universe into atoms that represent directories (d0 and d1) and those that represent files (f0, f1, and f2); their lower bounds are empty. The Root relation has the same lower and upper bound, which ensures that all filesystem models found by Kodkod are rooted at d0. The bounds on the contents relation specify that it must contain the tuple ⟨d0, d1⟩ and that its remaining tuples, if any, must be drawn from the cross product of the directory atoms with the entire universe.

A model of the toy filesystem is shown in Fig. 2-2b. Root is mapped to {⟨d0⟩},

as required by its bounds. The contents relation includes the sole tuple from its lower bound and two additional tuples from its upper bound. File and Dir consist of the file and directory atoms that are related by contents, as required by the specification (Fig. 2-2a, lines 6, 8 and 9).

2.2 Translating bounded relational logic to SAT

Using SAT to find a model of a relational problem involves several steps (Fig. 1-8): translation to boolean logic, symmetry breaking, transformation of the boolean formula to conjunctive normal form, and conversion of a boolean model, if one is found, to a model of the original problem. The last two steps implement standard transformations [44, 68], but the first two use novel techniques which are discussed in this section and the next chapter.

2.2.1 Translation algorithm

Kodkod's translation algorithm is based on the simple idea [68] that a relation over a finite universe can be represented as a matrix of boolean values. For example, a binary relation drawn from the universe $\{a_0, \ldots, a_{n-1}\}$ can be encoded with an $n \times n$ bit matrix that contains a 1 at the index $[i, j]$ when the relation includes the tuple $\langle a_i, a_j \rangle$. More generally, given a universe of n atoms, the collection of possible values for a relational variable $v :_k [l, u]$ corresponds to a k-dimensional matrix m with

$$m[i_1, \ldots, i_k] = \begin{cases} 1 & \text{if } \langle a_{i_1}, \ldots, a_{i_k} \rangle \in l, \\ \mathcal{V}(v, \langle a_{i_1}, \ldots, a_{i_k} \rangle) & \text{if } \langle a_{i_1}, \ldots, a_{i_k} \rangle \in u \setminus l, \\ 0 & \text{otherwise,} \end{cases}$$

where $i_1, \ldots, i_k \in [0 \,..\, n)$ and $\mathcal{V}$ maps its inputs to unique boolean variables. These matrices can then be used in a bottom-up, compositional translation of the entire specification (Fig. 2-3): relational expressions are translated using matrix operations, and relational constraints are translated as boolean constraints over matrix entries.

Figure 2-4 illustrates the translation process on the constraint contents $\subseteq$ Dir $\rightarrow$ (Dir $\cup$ File) from the filesystem specification (Fig. 2-2a, line 6). The constraint and

$\mathrm{T_P}$: problem $\to$ bool
$\mathrm{T_R}$: relBound $\to$ universe $\to$ matrix
$\mathrm{T_F}$: formula $\to$ env $\to$ bool
$\mathrm{T_E}$: expr $\to$ env $\to$ matrix
env : var $\to$ matrix

bool := 0 | 1 | boolVar | $\neg$ bool | bool $\wedge$ bool | bool $\vee$ bool | bool ? bool : bool
boolVar := identifier
idx := $\langle$int[, int]$^*\rangle$

$\mathcal{V}$: var $\to$ $\langle$atom[, atom]$^*\rangle$ $\to$ boolVar — *boolean variable for a given tuple in a relation*
$(\!|\ |\!)$: matrix $\to$ {idx[, idx]*} — *set of all indices in a matrix*
$[\![\ ]\!]$: matrix $\to$ $\mathrm{int}^{\mathrm{int}}$ — *size of a matrix, (size of a dimension)$^{number\ of\ dimensions}$*
[ ] : matrix $\to$ idx $\to$ bool — *matrix value at a given index*

$\mathcal{M}$: $\mathrm{int}^{\mathrm{int}}$ $\to$ (idx $\to$ bool) $\to$ matrix
$\mathcal{M}(s^d, f)$:= new $m \in$ matrix where $[\![m]\!] = s^d \wedge \forall\ \vec{x} \in \{0, ..., s-1\}^d, m[\vec{x}] = f(\vec{x})$

$\mathcal{M}$: $\mathrm{int}^{\mathrm{int}}$ $\to$ idx $\to$ matrix
$\mathcal{M}(s^d, \vec{x}) := \mathcal{M}(s^d, \lambda\vec{y}.$ if $\vec{y} = \vec{x}$ then 1 else 0)

$\mathrm{T_P}[\{a_1, \ldots, a_n\}\ \ v_1 :_{k_1} [l_1, u_1] \ldots v_j :_{k_j} [l_j, u_j]\ \ f_1 \ldots f_m] := \mathrm{T_F}[\bigwedge_{i=1}^{m} f_i](\cup_{i=1}^{j} v_i \mapsto \mathrm{T_R}[v_i :_{k_i} [l_i, u_i], \{a_1, \ldots, a_n\}])$

$\mathrm{T_R}[v :_k [l, u], \{a_1, \ldots, a_n\}] := \mathcal{M}(n^k, \lambda\langle i_1, ...i_k\rangle.$ if $\langle a_{i_1}, \ldots, a_{i_k}\rangle \in l$ then 1
else if $\langle a_{i_1}, \ldots, a_{i_k}\rangle \in u \setminus l$ then $\mathcal{V}(v, \langle a_{i_1}, \ldots, a_{i_k}\rangle)$
else 0)

$\mathrm{T_F}[\mathbf{no}\ p]\mathrm{e} := \neg\mathrm{T_F}[\mathbf{some}\ p]\mathrm{e}$
$\mathrm{T_F}[\mathbf{lone}\ p]\mathrm{e} := \mathrm{T_F}[\mathbf{no}\ p]\mathrm{e} \vee \mathrm{T_F}[\mathbf{one}\ p]\mathrm{e}$
$\mathrm{T_F}[\mathbf{one}\ p]\mathrm{e} := \mathrm{let}\ m \leftarrow \mathrm{T_E}[p]\mathrm{e}\ \mathrm{in}\ \bigvee_{\vec{x} \in (\!|m|\!)} m[\vec{x}] \wedge (\bigwedge_{\vec{y} \in (\!|m|\!) \setminus \{\vec{x}\}} \neg m[\vec{y}])$
$\mathrm{T_F}[\mathbf{some}\ p]\mathrm{e} := \mathrm{let}\ m \leftarrow \mathrm{T_E}[p]\mathrm{e}\ \mathrm{in}\ \bigvee_{\vec{x} \in (\!|m|\!)} m[\vec{x}]$
$\mathrm{T_F}[p \subseteq q]\mathrm{e} := \mathrm{let}\ m \leftarrow (\neg\mathrm{T_E}[p]\mathrm{e} \vee \mathrm{T_E}[q]\mathrm{e})\ \mathrm{in}\ \bigwedge_{\vec{x} \in (\!|m|\!)} m[\vec{x}]$
$\mathrm{T_F}[p = q]\mathrm{e} := \mathrm{T_F}[p \subseteq q]\mathrm{e} \wedge \mathrm{T_F}[q \subseteq p]\mathrm{e}$
$\mathrm{T_F}[\mathbf{not}\ f]\mathrm{e} := \neg\mathrm{T_F}[f]\mathrm{e}$
$\mathrm{T_F}[f \wedge g]\mathrm{e} := \mathrm{T_F}[f]\mathrm{e} \wedge \mathrm{T_F}[g]\mathrm{e}$
$\mathrm{T_F}[f \vee g]\mathrm{e} := \mathrm{T_F}[f]\mathrm{e} \vee \mathrm{T_F}[g]\mathrm{e}$
$\mathrm{T_F}[f \Rightarrow g]\mathrm{e} := \neg\mathrm{T_F}[f]\mathrm{e} \vee \mathrm{T_F}[g]\mathrm{e}$
$\mathrm{T_F}[f \Leftrightarrow g]\mathrm{e} := (\mathrm{T_F}[f]\mathrm{e} \wedge \mathrm{T_F}[g]\mathrm{e}) \vee (\neg\mathrm{T_F}[f]\mathrm{e} \wedge \neg\mathrm{T_F}[g]\mathrm{e})$

$\mathrm{T_F}[\forall\ v_1 : e_1, ..., v_n : e_n \mid f]\mathrm{e} := \mathrm{let}\ m \leftarrow \mathrm{T_E}[e_1]\mathrm{e}\ \mathrm{in}\ \bigwedge_{\vec{x} \in (\!|m|\!)} (\neg m[\vec{x}] \vee \mathrm{T_F}[\forall\ v_2 : e_2, ..., v_n : e_n \mid f](\mathrm{e} \oplus v_1 \mapsto \mathcal{M}([\![m]\!], \vec{x})))$
$\mathrm{T_F}[\exists\ v_1 : e_1, ..., v_n : e_n \mid f]\mathrm{e} := \mathrm{let}\ m \leftarrow \mathrm{T_E}[e_1]\mathrm{e}\ \mathrm{in}\ \bigvee_{\vec{x} \in (\!|m|\!)} (m[\vec{x}] \wedge \mathrm{T_F}[\forall\ v_2 : e_2, ..., v_n : e_n \mid f](\mathrm{e} \oplus v_1 \mapsto \mathcal{M}([\![m]\!], \vec{x})))$

$\mathrm{T_E}[v]\mathrm{e} := \mathrm{e}(v)$
$\mathrm{T_E}[\tilde{}p]\mathrm{e} := (\mathrm{T_E}[p]\mathrm{e})^{\mathsf{T}}$
$\mathrm{T_E}[\hat{}p]\mathrm{e} := \mathrm{let}\ m \leftarrow \mathrm{T_E}[p]\mathrm{e}, s^d \leftarrow [\![m]\!], \mathrm{sq} \leftarrow (\lambda x.i.$ if $i = s$ then x else let $y \leftarrow \mathrm{sq}(x, i * 2)$ in $y \vee y \cdot y)$ in $\mathrm{sq}(m, 1)$
$\mathrm{T_E}[^*p]\mathrm{e} := \mathrm{let}\ m \leftarrow \mathrm{T_E}[\hat{}p]\mathrm{e}, s^d \leftarrow [\![m]\!]$ in $m \vee \mathcal{M}(s^d, \lambda\langle i_1, \ldots, i_d\rangle.$ if $i_1 = i_2 \wedge \ldots \wedge i_1 = i_k$ then 1 else 0)
$\mathrm{T_E}[p \cup q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \vee \mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[p \cap q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \wedge \mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[p \setminus q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \wedge \neg\mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[p\ .\ q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \cdot \mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[p \to q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \times \mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[f\ ?\ p : q]\mathrm{e} := \mathrm{let}\ m_p \leftarrow \mathrm{T_E}[p]\mathrm{e}, m_p \leftarrow \mathrm{T_E}[q]\mathrm{e}$ in $\mathcal{M}([\![m_p]\!], \lambda\vec{x}.\ \mathrm{T_F}[f]\mathrm{e}\ ?\ m_p[\vec{x}]\ :\ m_q[\vec{x}])$

$\mathrm{T_E}[\{v_1 : e_1, ..., v_n : e_n \mid f\}]\mathrm{e} := \mathrm{let}\ m_1 \leftarrow \mathrm{T_E}[e_1]\mathrm{e}, s^d \leftarrow [\![m_1]\!]$ in
$\mathcal{M}(s^n, \lambda\langle i_1, \ldots, i_n\rangle.$ let $m_2 \leftarrow \mathrm{T_E}[e_2](\mathrm{e} \oplus v_1 \mapsto \mathcal{M}(s, \langle i_1\rangle)), \ldots,$
$m_n \leftarrow \mathrm{T_E}[e_n](\mathrm{e} \oplus v_1 \mapsto \mathcal{M}(s, \langle i_1\rangle) \oplus \ldots \oplus v_{n-1} \mapsto \mathcal{M}(s, \langle i_{n-1}\rangle))$ in
$m_1[i_1] \wedge \ldots \wedge m_n[i_n] \wedge \mathrm{T_F}[f](\mathrm{e} \oplus v_1 \mapsto \mathcal{M}(s, \langle i_1\rangle) \oplus \ldots \oplus v_n \mapsto \mathcal{M}(s, \langle i_n\rangle)))$

Figure 2-3: Translation rules for bounded relational logic.

its subexpressions are translated in an environment that binds each free variable to a matrix that represents its value. The bounds on a variable are used to populate its representation matrix as follows: lower bound tuples are represented with 1s in the corresponding matrix entries; tuples that are in the upper but not the lower bound are represented with fresh boolean variables; and tuples outside the upper bound are represented with 0s. Translation of the remaining expressions is straightforward. The union of Dir and File is translated as the disjunction of their translations so that a tuple is in Dir ∪ File if it is in Dir or File; relational cross product becomes the generalized cross product of matrices, with conjunction used instead of multiplication; and the subset constraint forces each boolean variable representing a tuple in Dir → (Dir∪File) to evaluate to 1 whenever the boolean representation of the corresponding tuple in the contents relation evaluates to the same.

2.2.2 Sparse-matrix representation of relations

Many problems suitable for solving with a relational engine are typed: their universes are partitioned into sets of atoms according to a type hierarchy, and their expressions are bounded above by relations over these sets [39]. The toy filesystem, for example, is defined over a universe that consists of two types of atoms: the atoms that represent directories and those that represent files. Each expression in the filesystem specification (Fig. 2-2a) is bounded above by a relation over the types $T_{\text{dir}} = \{\texttt{d0}, \texttt{d1}\}$ and $T_{\text{file}} = \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}$. The upper bound on contents, for example, relates the directory type to both the directory and file types, i.e. $\lceil \mathsf{contents} \rceil = \{\langle T_{\text{dir}}, T_{\text{dir}} \rangle, \langle T_{\text{dir}}, T_{\text{file}} \rangle\} = \{\texttt{d0}, \texttt{d1}\} \times \{\texttt{d0}, \texttt{d1}\} \cup \{\texttt{d0}, \texttt{d1}\} \times \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}$.

Previous relational engines (§2.3.1) employed a type checker [39, 128], a source-to-source transformation [40], and a typed translation [68, 117], in an effort to reduce the number of boolean variables used to encode typed problems. Kodkod's translation, on the other hand, is designed to exploit types, provided as upper bounds on free variables, transparently: each relational variable is represented as an untyped matrix whose dimensions correspond to the entire universe, but the entries outside the variable's upper bound are zeroed out. The zeros are then propagated up the translation

$$e = \left\{ \mathsf{File} \mapsto \begin{bmatrix} 0 \\ 0 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix},\ \mathsf{Dir} \mapsto \begin{bmatrix} d_0 \\ d_1 \\ 0 \\ 0 \\ 0 \end{bmatrix},\ \mathsf{Root} \mapsto \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},\ \mathsf{contents} \mapsto \begin{bmatrix} c_0 & 1 & c_2 & c_3 & c_4 \\ c_5 & c_6 & c_7 & c_8 & c_9 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \right\}.$$

$$\mathrm{T_E}[\mathsf{Dir}]e = e(\mathsf{Dir}) = \begin{bmatrix} d_0 \\ d_1 \\ 0 \\ 0 \\ 0 \end{bmatrix},\ \mathrm{T_E}[\mathsf{File}]e = e(\mathsf{File}) = \begin{bmatrix} 0 \\ 0 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix},\ \mathrm{T_E}[\mathsf{contents}]e = e(\mathsf{contents}) = \begin{bmatrix} c_0 & 1 & c_2 & c_3 & c_4 \\ c_5 & c_6 & c_7 & c_8 & c_9 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

$$\mathrm{T_E}[\mathsf{Dir} \cup \mathsf{File}]e = \mathrm{T_E}[\mathsf{Dir}]e \vee \mathrm{T_E}[\mathsf{File}]e = \begin{bmatrix} d_0 \\ d_1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \vee \begin{bmatrix} 0 \\ 0 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix}.$$

$$\mathrm{T_E}[\mathsf{Dir} \to (\mathsf{Dir} \cup \mathsf{File})]e = \mathrm{T_E}[\mathsf{Dir}]e \times \mathrm{T_E}[\mathsf{Dir} \cup \mathsf{File}]e = \begin{bmatrix} d_0 & d_1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} d_0 \\ d_1 \\ f_0 \\ f_1 \\ f_2 \end{bmatrix}$$

$$= \begin{bmatrix} d_0 \wedge d_0 & d_0 \wedge d_1 & d_0 \wedge f_0 & d_0 \wedge f_1 & d_0 \wedge f_2 \\ d_1 \wedge d_0 & d_1 \wedge d_1 & d_1 \wedge f_0 & d_1 \wedge f_1 & d_1 \wedge f_2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

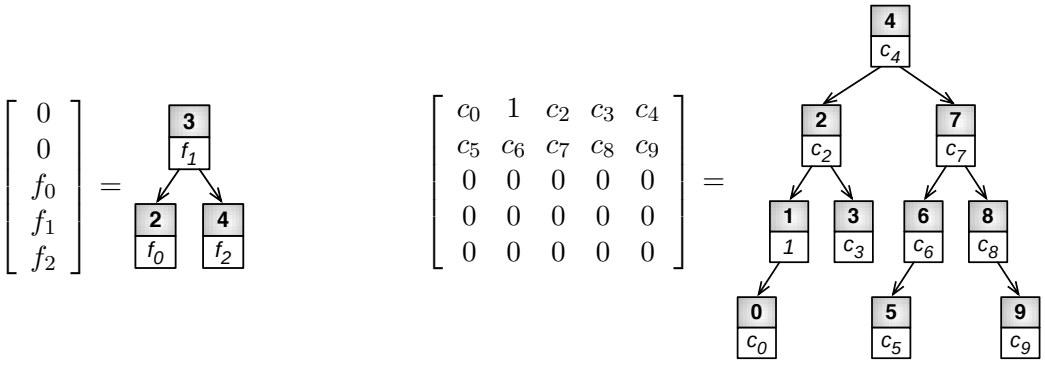
$$\mathrm{T_F}[\mathsf{contents} \subseteq \mathsf{Dir} \to (\mathsf{Dir} \cup \mathsf{File})]e = \bigwedge\ (\neg\, \mathrm{T_E}[\mathsf{contents}]e \vee \mathrm{T_E}[\mathsf{Dir} \to (\mathsf{Dir} \cup \mathsf{File})]e)$$

$$= \bigwedge \left(\neg \begin{bmatrix} c_0 & 1 & c_2 & c_3 & c_4 \\ c_5 & c_6 & c_7 & c_8 & c_9 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \vee \begin{bmatrix} d_0 \wedge d_0 & d_0 \wedge d_1 & d_0 \wedge f_0 & d_0 \wedge f_1 & d_0 \wedge f_2 \\ d_1 \wedge d_0 & d_1 \wedge d_1 & d_1 \wedge f_0 & d_1 \wedge f_1 & d_1 \wedge f_2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \right)$$

$$= \bigwedge \begin{bmatrix} \neg c_0 \vee (d_0 \wedge d_0) & \neg 1 \vee (d_0 \wedge d_1) & \neg c_2 \vee (d_0 \wedge f_0) & \neg c_3 \vee (d_0 \wedge f_1) & \neg c_4 \vee (d_0 \wedge f_2) \\ \neg c_5 \vee (d_1 \wedge d_0) & \neg c_6 \vee (d_1 \wedge d_1) & \neg c_7 \vee (d_1 \wedge f_0) & \neg c_8 \vee (d_1 \wedge f_1) & \neg c_9 \vee (d_1 \wedge f_2) \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= (\neg c_0 \vee (d_0 \wedge d_0)) \wedge (\neg 1 \vee (d_0 \wedge d_1)) \wedge (\neg c_2 \vee (d_0 \wedge f_0)) \wedge (\neg c_3 \vee (d_0 \wedge f_1)) \wedge (\neg c_4 \vee (d_0 \wedge f_2)) \wedge$$
$$(\neg c_5 \vee (d_1 \wedge d_0)) \wedge (\neg c_6 \vee (d_1 \wedge d_1)) \wedge (\neg c_7 \vee (d_1 \wedge f_0)) \wedge (\neg c_8 \vee (d_1 \wedge f_1)) \wedge (\neg c_9 \vee (d_1 \wedge f_2)).$$

Figure 2-4: A sample translation. The shading highlights the redundancies in the boolean encoding.



(a) $\mathrm{T_E}[\mathsf{File}]e$ (b) $\mathrm{T_E}[\mathsf{contents}]e$

Figure 2-5: Sparse representation of the translation matrices $\mathrm{T_E}[\mathsf{File}]e$ and $\mathrm{T_E}[\mathsf{contents}]e$ from Fig. 2-4. The upper half of each tree node holds its key, and the lower half holds its value. Matrices are indexed starting at 0.

chain, ensuring that no boolean variables are wasted on tuples guaranteed to be outside an expression's valuation. The upper bound on the expression $\mathsf{Dir} \to (\mathsf{Dir} \cup \mathsf{File})$, for example, is $\{\langle T_{\text{dir}}, T_{\text{dir}}\rangle, \langle T_{\text{dir}}, T_{\text{file}}\rangle\}$, and the regions of its translation matrix (Fig. 2-4) that correspond to the tuples outside of its 'type' are zeroed out.

This simple scheme for exploiting both types and partial models is enabled by a new multidimensional sparse-matrix data structure for representing relations. As noted in previous work [40], an untyped translation algorithm cannot scale if based on the standard encoding of matrices as multi-dimensional arrays, because the number of zeros in a k-dimensional matrix over a universe of n atoms grows proportionally to n^k. Kodkod therefore encodes translation matrices as balanced trees that store only non-zero values. In particular, each tree node corresponds to a non-zero cell (or a range of cells) in the full n^k matrix. The cell at the index $[i_1, \ldots, i_k]$ that stores the value v becomes a node with $\sum_{j=1}^{k} i_j n^{k-j}$ as its key and v as its value, where the index-to-key conversion yields the decimal representation of the n-ary number $i_1 \ldots i_k$. Nodes with consecutive keys that store a 1 are merged into a single node with a range of keys, enabling compact representation of lower bounds.

Figure 2-5 shows the sparse representation of the translation matrices $\mathrm{T_E}[\mathsf{File}]e$ and $\mathrm{T_E}[\mathsf{contents}]e$ from Fig. 2-4. The File tree contains three nodes, with keys that correspond directly to the indices of the non-zero entries in the matrix. The contents

tree consists of ten nodes, each of which corresponds to a non-empty entry $[i, j]$ in the contents matrix, with $i * 5 + j$ as its key and the contents of $[i, j]$ as its value. Both of the trees contain only nodes that represent exactly one cell in the corresponding matrix. It is easy to see, however, that the nodes 1 through 3 in the contents tree, for example, could be collapsed into a single node with $[1\,..\,3]$ as its key and 1 as its value if the entries $[0, 2]$ and $[0, 3]$ of the matrix were replaced with 1s.

Operations on the sparse matrices are implemented in a straightforward way, so that the cost of each operation depends on the number of non-zero entries in the matrix and the tree insertion, deletion, and lookup times. For instance, the disjunction of two matrices with m_1 and m_2 non-zero entries takes $O((m_1 + m_2) \log(m_1 + m_2))$ time. It is computed simply by creating an empty matrix with the same dimensions as the operands; iterating over the operands' nodes in the increasing order of keys; computing the disjunction of the values with matching keys; and storing the result, under the same key, in the newly created matrix. A value with an unmatched key is inserted directly into the output matrix (under its key), since the absence of a key from one of the operands is interpreted as its mapping that key to zero. Implementation of other operations follows the same basic idea.

2.2.3 Sharing detection at the boolean level

Relational specifications are typically built out of expressions and constraints whose boolean encodings contain many equivalent subcomponents. The expression $\mathsf{Dir} \rightarrow (\mathsf{Dir} \cup \mathsf{File})$, for example, translates to a matrix that contains two entries with equivalent but syntactically distinct formulas: $d_0 \wedge d_1$ at index $[0, 1]$ and $d_1 \wedge d_0$ at index $[1, 0]$ (Fig. 2-4). The two formulas are propagated up to the translation of the enclosing constraint and, eventually, the entire specification, bloating the final SAT encoding and creating unnecessary work for the SAT solver. Detecting and eliminating structural redundancies is therefore crucial for scalable model finding.

Prior work (§2.3.2) on redundancy detection for relational model finding produced a scheme that captures a class of redundancies detectable at the problem level. This class is relatively small and does not include the kind of low-level redundancy high-

```
REACH(op : binary operator, v : vertex, k : integer)
 1  if op = OP(v) ∧ SIZEOF(v) = 2 ∧ k > 1 then
 2    L ← REACH(op, LEFT(v), k − 1)
 3    R ← REACH(op, RIGHT(v), k − |L|)
 4    return L ∪ R
 5  else
 6    return {v}
```

Figure 2-6: Computing the d-reachable descendants of a CBC node. The functions OP and SIZEOF return the operator and the number of children of a given vertex. The functions LEFT and RIGHT return the left and right children of a binary vertex. The d-reachable descendants of a vertex v are given by $\text{REACH}(\text{OP}(v), v, 2^d)$.

lighted in Fig. 2-4. Kodkod uses a different approach and exploits redundancies at the boolean level, with a new circuit data structure called *Compact Boolean Circuits* (CBCs). CBCs are related to several other data structures (§2.3.4) which were developed for use with model checking tools (e.g. [49]) and so do not work as well with a relational translator (§2.4).

A Compact Boolean Circuit is a partially canonical, directed, acyclic graph (V, E, d). The set V is partitioned into operator vertices $V_{\text{op}} = V_{\text{AND}} \cup V_{\text{OR}} \cup V_{\text{NOT}} \cup V_{\text{ITE}}$ and leaves $V_{\text{leaf}} = V_{\text{VAR}} \cup \{\mathsf{T}, \mathsf{F}\}$. The AND and OR vertices have two or more children, which are ordered according to a total ordering on vertices $<_v$; an if-then-else (ITE) vertex has three children; and a NOT vertex has one child. Canonicity at the level of children is enforced for all operator vertices. That is, two distinct vertices of the same type must differ by at least one child, and no vertex can be simplified to another by applying an equivalence law from Table 2.1 to its children. Beyond this, partial canonicity is enforced based on the circuit's *binary compaction depth* $d \geq 1$. In particular, no binary vertex $v \in V$ can be transformed into another vertex $w \in V$ by applying the law of associativity to the *d-reachable* descendants of v, computed as shown in Fig. 2-6.

An example of a non-compact boolean circuit and its CBC equivalents is shown in Fig. 2-7. Part (a) displays the formula $(x \wedge y \wedge z) \Leftrightarrow (v \wedge w)$ encoded as $(\neg(x \wedge y \wedge z) \vee (v \wedge w)) \wedge (\neg(w \wedge v) \vee (x \wedge (y \wedge z)))$. Part (b) shows an equivalent CBC with the binary compaction depth of $d = 1$, which enforces partial canonicity at the level

	Rule	Condition
NOT	$\neg\neg a \to a$	

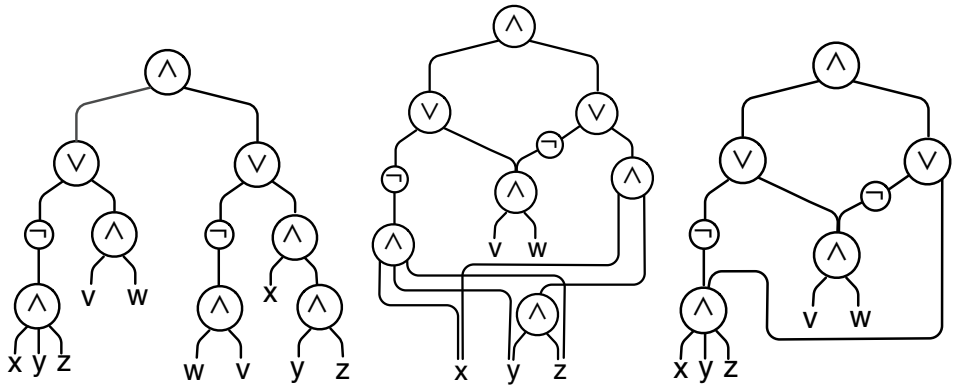
ITE	$i\,?\,t : e \to t$	$i = 1 \vee t = e$
	$i\,?\,t : e \to e$	$i = 0$
	$i\,?\,t : e \to i \vee e$	$t = 1 \vee i = t$
	$i\,?\,t : e \to \neg i \wedge e$	$t = 0 \vee \neg i = t$
	$i\,?\,t : e \to \neg i \vee t$	$e = 1 \vee \neg i = e$
	$i\,?\,t : e \to i \wedge t$	$e = 0 \vee i = e$

AND	$\bigwedge_{i\in[1..n]} a_i \to \bigwedge_{i\in[1..i)\cup(i..n]} a_i$	$\exists i \in [1\,..\,n], a_i = 1$
	$\bigwedge_{i\in[1..n]} a_i \to \bigwedge_{i\in[1..i)\cup(i..n]} a_i$	$\exists i, j \in [1\,..\,n], i \neq j \wedge a_i = a_j$
	$\bigwedge_{i\in[1..n]} a_i \to 0$	$\exists i \in [1\,..\,n], a_i = 0$
	$\bigwedge_{i\in[1..n]} a_i \to 0$	$\exists i, j \in [1\,..\,n], a_i = \neg a_j$
	$(\bigwedge_{i\in[1..n]} a_i) \wedge b \to 0$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = \neg b \vee b = \neg a_i$
	$\neg(\bigvee_{i\in[1..n]} a_i) \wedge b \to \neg(\bigvee_{i\in[1..n]} a_i)$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = \neg b \vee b = \neg a_i$
	$\neg(\bigvee_{i\in[1..n]} a_i) \wedge b \to 0$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = b$
	$(\bigwedge_{i\in[1..n]} a_i) \wedge b \to (\bigwedge_{i\in[1..n]} a_i)$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = b$
	$(\bigvee_{i\in[1..n]} a_i) \wedge b \to b$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = b$
	$(\bigwedge_{i\in[1..n]} a_i) \wedge (\bigwedge_{j\in[1..m]} b_j) \to (\bigwedge_{i\in[1..n]} a_i)$	$n \leq 2^d \wedge m \leq 2^d \wedge \forall j \in [1\,..\,m], \exists i \in [1\,..\,n], a_i = b_j$
	$(\bigvee_{i\in[1..n]} a_i) \wedge (\bigvee_{j\in[1..m]} b_j) \to (\bigvee_{i\in[1..m]} b_j)$	$n \leq 2^d \wedge m \leq 2^d \wedge \forall j \in [1\,..\,m], \exists i \in [1\,..\,n], a_i = b_j$
	$(\bigwedge_{i\in[1..n]} a_i) \wedge (\bigvee_{j\in[1..m]} b_j) \to (\bigwedge_{i\in[1..n]} a_i)$	$n \leq 2^d \wedge m \leq 2^d \wedge \exists j \in [1\,..\,m], \exists i \in [1\,..\,n], a_i = b_j$

OR	$\bigvee_{i\in[1..n]} a_i \to \bigvee_{i\in[1..i)\cup(i..n]} a_i$	$\exists i \in [1\,..\,n], a_i = 0$
	$\bigvee_{i\in[1..n]} a_i \to \bigvee_{i\in[1..i)\cup(i..n]} a_i$	$\exists i, j \in [1\,..\,n], i \neq j \wedge a_i = a_j$
	$\bigvee_{i\in[1..n]} a_i \to 1$	$\exists i \in [1\,..\,n], a_i = 1$
	$\bigvee_{i\in[1..n]} a_i \to 1$	$\exists i, j \in [1\,..\,n], a_i = \neg a_j$
	$(\bigvee_{i\in[1..n]} a_i) \vee b \to 1$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = \neg b \vee b = \neg a_i$
	$\neg(\bigwedge_{i\in[1..n]} a_i) \vee b \to \neg(\bigwedge_{i\in[1..n]} a_i)$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = \neg b \vee b = \neg a_i$
	$\neg(\bigwedge_{i\in[1..n]} a_i) \vee b \to 1$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = b$
	$(\bigvee_{i\in[1..n]} a_i) \vee b \to (\bigvee_{i\in[1..n]} a_i)$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = b$
	$(\bigwedge_{i\in[1..n]} a_i) \vee b \to b$	$n \leq 2^d \wedge \exists i \in [1\,..\,n], a_i = b$
	$(\bigvee_{i\in[1..n]} a_i) \vee (\bigvee_{j\in[1..m]} b_j) \to (\bigvee_{i\in[1..n]} a_i)$	$n \leq 2^d \wedge m \leq 2^d \wedge \forall j \in [1\,..\,m], \exists i \in [1\,..\,n], a_i = b_j$
	$(\bigwedge_{i\in[1..n]} a_i) \vee (\bigwedge_{j\in[1..m]} b_j) \to (\bigwedge_{i\in[1..m]} b_j)$	$n \leq 2^d \wedge m \leq 2^d \wedge \forall j \in [1\,..\,m], \exists i \in [1\,..\,n], a_i = b_j$
	$(\bigvee_{i\in[1..n]} a_i) \vee (\bigwedge_{j\in[1..m]} b_j) \to (\bigvee_{i\in[1..n]} a_i)$	$n \leq 2^d \wedge m \leq 2^d \wedge \exists j \in [1\,..\,m], \exists i \in [1\,..\,n], a_i = b_j$

Table 2.1: Simplification rules for a CBC with depth d. The rules that depend on d are tested for applicability under two conditions. If the relevant operand is a (negated) binary conjunction or a disjunction, then the rule is tested for applicability to its d-reachable descendants. If the operand is a (negated) nary conjunction or a disjunction with up to 2^d children, then the rule is tested for applicability to those children. The rule is not tested for applicability otherwise, which ensures that all rules that depend on d are applicable in constant $O(2^d)$ time.

of inner nodes' children. That is, the depth of $d = 1$ offers only the basic canonicity guarantee, forcing the subformula $(v \wedge w)$ to be shared. Part (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.



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

Figure 2-7: A non-compact boolean circuit and its compact equivalents.

Partial canonicity of CBCs is maintained by a factory data structure that synthesizes and caches all CBCs that are a part of the same graph (V, E, d). Given an operator and o operands with at most c children each, the factory sorts the operands and performs the applicable simplifications from Table 2.1. This takes $O(o \log o)$ time, since the multi-operand rules can be applied in $O(o)$ time, and the two-operand rules, which are only applied to operands with at most 2^d children (or d-reachable descendants), take constant 2^d time. If the simplification yields a boolean constant or one of the operands, that result is returned. Otherwise, the operands are hashed, and made into a new circuit only if the factory's cache does not already contain a circuit with the same inputs (or d-reachable descendants). Assuming collision resistant hashing, checking the cache hits for (d-reachable) equality to the operands takes $O(\max(2^d, o)) = O(o)$ time, since d is a small fixed constant.

2.3 Related work

The body of research on SAT-based model finding for relational logic spans nearly two decades. The first half of this section provides an overview of that work, with a focus on prior techniques for exploiting types (§2.3.1) and for sharing subformulas in the boolean encoding (§2.3.2). The second half covers the work that is not specific to model finding but that is nonetheless closely related to the techniques employed in Kodkod, namely other sparse-matrix (§2.3.3) and auto-compacting circuit (§2.3.4) representations.

2.3.1 Type-based representation of relations

Early versions of the Alloy language [67, 68, 71] employed a simple type system in which the universe of atoms was partitioned into a set of top-level types, and the type of each expression was given as a product of these types. The language was translated to SAT using rules [68, 71] that mirror those of Kodkod, but that are applied to rectangular matrices with typed dimensions instead of square matrices with dimensions that range over the entire universe. Under this scheme, a k-ary relation of type $T_1 \rightarrow \ldots \rightarrow T_k$ is translated to a matrix with dimensions $|T_1| \times \ldots \times |T_k|$. For example, the Dir relation from the filesystem problem corresponds to a vector of length 2, containing the boolean variables d_0 and d_1, and the File relation corresponds to a vector of length 3, containing the variables f_0 through f_2.

The typed approach has the advantage of producing dense matrices that can be represented simply as multidimensional arrays. But the disadvantage of using irregularly-shaped matrices is that they restrict the kind of operations that can be performed on expressions of different types. For example, the relations Dir and File cannot be added together because the matrix disjunction operator that is used for translating unions requires its arguments to have identical dimensions.

In the early versions of Alloy [68, 71], the dimension mismatch problem was addressed by merging some of the types into a single supertype. To make the expression $\mathsf{Dir} \cup \mathsf{File}$ legally typed and translatable, the types T_{dir} and T_{file} are merged (manu-

ally, by changing the specification) into a new type, T_{object}, that contains all files and directories in the universe. The relations Dir and File are then both declared to have T_{object} as their type, resulting in two equally-sized translation vectors that can be combined with the disjunction operator. The downside of this process, however, is that it expands the upper bound on both Dir and File to the entire universe, doubling the number of boolean variables needed to encode their values.

Alloy3 [117] addressed the dimension mismatch problem with two new features: a type system [39, 128] that supports subtypes and union types, and a source-to-source transformation [40] for taking advantage of the resulting type information without changing the underlying translation (i.e. [68, 71]). These features are implemented as two additional steps that happen before the translation. First, the typechecker partitions the universe into a set of *base types*, and checks that the type of each expression can be expressed as a relation over these base types. Next, the expression types are used to *atomize* the specification into a set of equivalent constraints involving expressions that range strictly over the base types. Finally, the new constraints, being well-typed according to the old type system, are reduced to SAT as before [68, 71].

For example, suppose that the filesystem specification consists of a single constraint, namely $\mathsf{contents} \subseteq \mathsf{Dir} \rightarrow (\mathsf{Dir} \cup \mathsf{File})$. To translate this specification to SAT, Alloy3 first partitions the filesystem universe into two base types, T_{dir} and T_{file}, to produce the following binding of expressions to types: $\mathsf{Dir} \mapsto \{\langle T_{\text{dir}}\rangle\}$, $\mathsf{File} \mapsto \{\langle T_{\text{file}}\rangle\}$, $\mathsf{Dir} \cup \mathsf{File} \mapsto \{\langle T_{\text{dir}}\rangle, \langle T_{\text{file}}\rangle\}$, $\mathsf{contents} \mapsto \{\langle T_{\text{dir}}, T_{\text{dir}}\rangle, \langle T_{\text{dir}}, T_{\text{file}}\rangle\}$, and $\mathsf{Dir} \rightarrow (\mathsf{Dir} \cup \mathsf{File}) \mapsto \{\langle T_{\text{dir}}, T_{\text{dir}}\rangle, \langle T_{\text{dir}}, T_{\text{file}}\rangle\}$. It then atomizes the contents variable into two new variables, $\mathsf{contents} = \mathsf{contents}_d \cup \mathsf{contents}_f$, that range over the types $\{\langle T_{\text{dir}}, T_{\text{dir}}\rangle\}$ and $\{\langle T_{\text{dir}}, T_{\text{file}}\rangle\}$. In the next and final stage, the specification $\mathsf{contents} \subseteq \mathsf{Dir} \rightarrow (\mathsf{Dir} \cup \mathsf{File})$ is expanded to $(\mathsf{contents}_d \cup \mathsf{contents}_f) \subseteq (\mathsf{Dir} \rightarrow \mathsf{Dir}) \cup (\mathsf{Dir} \rightarrow \mathsf{File})$, which is logically equivalent to $\mathsf{contents}_d \subseteq (\mathsf{Dir} \rightarrow \mathsf{Dir}) \wedge \mathsf{contents}_f \subseteq (\mathsf{Dir} \rightarrow \mathsf{File})$. Each expression that appears in this new specification ranges over the base types, and the specification as a whole can be translated with the type-based translator using the same number of boolean variables as the translation shown in Fig. 2-4.

The atomization approach, however, does not work well for specifications with

transitive closure. Because there is no efficient way to atomize transitive closure expressions [40] such as ^contents in the toy filesystem, Alloy3 handles problems with closure by merging the domain and range types of each closure expression into a single base type. In the case of the filesystem problem, this leads to the merging of file and directory types into a single type and, consequently, to the doubling in the number of boolean variables needed to represent each relation.

2.3.2 Sharing detection at the problem level

Formal specifications of software systems often make use of quantified formulas whose ground form contains many identical subcomponents. For example, the specification of a filesystem that disallows sharing of contents among directories (e.g. via hard links) may include the constraint $\forall$ p, d: Dir | $\neg$p=d $\Rightarrow$ **no** (p.contents $\cap$ d.contents), which grounds out to

$$
\begin{aligned}
&(\neg\{\langle \mathsf{d0}\rangle\}=\{\langle \mathsf{d0}\rangle\} \Rightarrow \textbf{no}\ (\{\langle \mathsf{d0}\rangle\}.\mathsf{contents} \cap \{\langle \mathsf{d0}\rangle\}.\mathsf{contents})) \wedge \\
&(\neg\{\langle \mathsf{d0}\rangle\}=\{\langle \mathsf{d1}\rangle\} \Rightarrow \textbf{no}\ (\{\langle \mathsf{d0}\rangle\}.\mathsf{contents} \cap \{\langle \mathsf{d1}\rangle\}.\mathsf{contents})) \wedge \\
&(\neg\{\langle \mathsf{d1}\rangle\}=\{\langle \mathsf{d0}\rangle\} \Rightarrow \textbf{no}\ (\{\langle \mathsf{d1}\rangle\}.\mathsf{contents} \cap \{\langle \mathsf{d0}\rangle\}.\mathsf{contents})) \wedge \\
&(\neg\{\langle \mathsf{d1}\rangle\}=\{\langle \mathsf{d1}\rangle\} \Rightarrow \textbf{no}\ (\{\langle \mathsf{d1}\rangle\}.\mathsf{contents} \cap \{\langle \mathsf{d1}\rangle\}.\mathsf{contents}))
\end{aligned}
$$

in a two-directory universe. Alloy3's sharing detection mechanism [117, 119] focuses on ensuring that all occurrences of a given ground component, such as {⟨d0⟩}.contents, share the same boolean encoding in memory.

The scheme works in two stages: a detection phase, in which the AST of a specification is traversed and each node is annotated with a syntactic *template*, and a translation phase, in which the template annotations are used to ensure sharing. For the above example, the first phase produces a set of template annotations that includes ¬p=d $\mapsto$ "¬? = ?", p.contents $\mapsto$ "?.contents", and d.contents $\mapsto$ "?.contents".[1] In the next phase, the translator keeps a cache for each template, which stores the translations of all ground forms that instantiate the template. A ground form for a node is then translated only if its translation is missing from the cache of the

[1] Alloy3 creates a unique ID for each template rather than a syntactic representation such as "?.contents"; the syntactic representation is used in the text for presentational clarity.

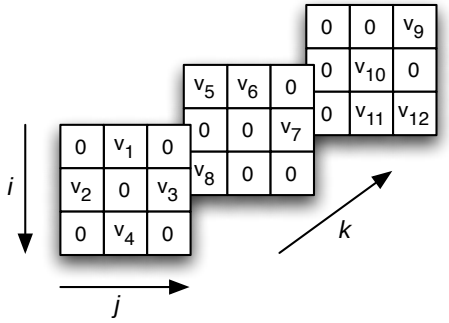
node's template. For example, when visiting d.contents in the environment {p↦{⟨d0⟩}, d↦{⟨d0⟩}}, the translator will find that the cache of "?.contents" contains a translation for {⟨d0⟩}.contents, because p.contents, which is translated first, has the same ground form in the given environment. The end result is that each occurrence of the ground expression {⟨d0⟩}.contents (and, similarly, {⟨d1⟩}.contents) translates to the same matrix in memory.

Unlike the template approach, Kodkod's CBC approach ensures not only that the translations for syntactically equivalent ground expressions are shared, but also that ¬{⟨d0⟩}= {⟨d1⟩} and ¬{⟨d1⟩}={⟨d0⟩} are translated to the same circuit. The template scheme misses the latter opportunity for sharing because the two instantiations of the template ¬?=? are syntactically distinct. In general, the set of redundancies detected by CBCs is a superset of those detectable with templates. Even if the templates incorporated semantic information to detect that ¬{⟨d0⟩}={⟨d1⟩} and ¬{⟨d1⟩}={⟨d0⟩} are equivalent, the scheme would still miss the lower-level redundancies captured by CBCs, such as the formulas highlighted in Fig. 2-4.

2.3.3 Multidimensional sparse matrices

Many representations have been developed for two-dimensional sparse matrices, including the Coordinate (COO), Compressed Row/Column Storage (CRS/CCS), Jagged Diagonal (JAD), Symmetric Sparse Skyline (SSS), and Java Sparse Array (JSA) formats [12, 84]. The problem of representing multidimensional sparse matrices, on the other hand, has received relatively little attention. Existing high-performance approaches for representing k-dimensional sparse data include the generalized CRS/CCS (GCRS/GCCS) format and the extended Karnaugh map CRS/CCS (ECRS/ECCS) format [82].

Figure 2-8 illustrates the key ideas behind the GCRS and ECRS representations on a sample three dimensional matrix. The GCRS format is based on the canonical data layout for multidimensional arrays [24], in which an n^k matrix is represented using n one-dimensional arrays of size n^{k-1}, with the matrix index $[i_1, \ldots, i_k]$ mapping to the index $\sum_{j=2}^{k} i_j n^{j-2}$ of the i_1^{th} array (Fig. 2-8b). The ECRS format is based on



(a) A 2D view of a sample 3D matrix m

0	v_1	0	v_5	v_6	0	0	0	v_9
v_2	0	v_3	0	0	v_7	0	v_{10}	0
0	v_4	0	v_8	0	0	0	v_{11}	v_{12}

(b) Canonical data layout of m

0	v_5	0	v_1	v_6	0	0	0	v_9
v_2	0	0	0	0	v_{10}	v_3	v_7	0
0	v_8	0	v_4	0	v_{11}	0	0	v_{12}

(c) Extended Karnaugh map layout of m

C_1	0	4	8	12							
C_2	1	1	2	2	0	2	2	1	1	0	1	2
C_3	0	1	1	2	0	0	1	2	0	1	2	2
V	v_1	v_5	v_6	v_9	v_2	v_3	v_7	v_{10}	v_4	v_8	v_{11}	v_{12}

(d) GCRS representation of m

R	0	4	8	12							
C	1	2	3	8	0	5	6	7	1	3	5	8
V	v_5	v_1	v_6	v_9	v_2	v_{10}	v_3	v_7	v_8	v_4	v_{11}	v_{12}

(e) ECRS representation of m

Figure 2-8: GCRS and ECRS representations for multidimensional sparse matrices. GCRS is the compressed row storage representation of a multidimensional matrix stored using the canonical data layout. ECRS is the compressed row storage representation of a multidimensional matrix stored using the extended Karnaugh map layout.

the extended Karnaugh map layout [83] which represents multidimensional arrays with collections of two-dimensional matrices. An $n \times n \times n$ matrix, for example, is represented with n matrices of size $n \times n$, where the 3-dimensional index $[i_1, i_2, i_3]$ maps to the index $[i_1, i_3]$ of the i_2^{th} matrix. The three matrices are laid out in memory with n arrays of length n^2, with the index $[i_1, i_2, i_3]$ mapping to the index $i_2 n + i_3$ of the i_1^{th} array (Fig. 2-8c).

Once a multidimensional sparse matrix is laid out in memory, it can be compressed using a variation on the standard CRS format for 2-dimensional arrays. Given an n^k matrix m, GCRS encodes its canonical data representation CDL(m) with arrays C_1, ..., C_k and V (Fig. 2-8d). The V array stores the non-zero values of the matrix, obtained by scanning the rows of CDL(m) from left to right. The C_1 array stores the locations in V that start a row. In particular, if $V[l] = m[i_1, \ldots, i_k]$ then $C_1[i_1] \leq l < C_1[i_1 + i]$. By convention, C_1 has $n+1$ elements with $C_1[n] = |V|$, where $|V|$ is the length of V. Arrays $C_2, \ldots, C_k$ store the non-row indices of the elements in V; i.e., if $V[l] = m[i_1, i_2, \ldots, i_k]$, then $C_2[l] = i_2, \ldots, C_k[l] = i_k$. ECRS (for three-dimensional matrices) encodes EKM(m), the extended Karnaugh map representation of m, with three arrays: R, C, and V (Fig. 2-8e). The R and V arrays are defined in the same way as the C_1 and V arrays produced by GCRS. The C array stores the column indices of the non-zero elements in EKM(m); that is, $V[l] = m[i_1, i_2, i_3]$ then $C[l] = i_2 n + i_3$.

Compared to Kodkod's sparse matrices, GCRS and ECRS provide better average access time for a given index.[2] For an n^k matrix with an average of d non-zeroes per row, the value at a given index can be retrieved in $O(\log d)$ time for both GCRS and ECRS; the Kodkod representation requires $O(\log nd)$ time. The Kodkod representation, however, has several features that make it more suitable for use with a relational translator than GCRS or ECRS. First, Kodkod matrices are mutable, which is crucial for the translation of expressions (e.g. set comprehensions) whose matrices must be filled incrementally. Neither GCRS nor ECRS support mutation. Second, the representation of a Kodkod matrix shrinks as the number of consecutive

[2]Worst case access time for all three encodings is logarithmic in the number of non-zero elements.

1s in its cells increases, which is important for efficient translation of problems that have large partial models or that use built-in constants such as the universal relation. GCRS and ECRS, for example, need $O(n^k)$ space to represent the universal relation of arity k over an n-atom universe; Kodkod needs just one node. Finally, disjunction, multiplication and other operations are straightforward to implement for Kodkod matrices. Corresponding operations on GCRS and ECRS matrices, in contrast, are much trickier to design: the published addition and multiplication algorithms for these matrices work only when one of the operands is represented sparsely and the other fully [82].

2.3.4 Auto-compacting circuits

The choice of circuit representation in a model checker affects the efficiency of the overall tool in two ways [18]. First, a compact representation saves memory, enabling faster processing of larger problems. Second, it eliminates redundancies in the boolean encoding, which is particularly important for SAT-based tools. These concerns have been addressed in the literature with three different circuit representations, designed to work with different model checking backends: *Boolean Expression Diagrams* (BEDs) [6], which are designed for use with BDD-based model checkers [95]; *Reduced Boolean Circuits* (RBCs) [1]; and *And-Inverter Graphs* (AIGs) [19, 78], which are used in SAT-based tools for bounded model checking [49, 73].

BEDs, RBCs and AIGs are similar to CBCs in that they define a graph over operator and leaf vertices which satisfies a set of partial canonicity invariants. Of the four, BEDs support the largest number of operators (with every 2-input boolean function represented by its own BED vertex), and AIGs support the fewest operators (allowing only conjunctions and negations). RBCs support three operators: conjunction, negation, and equivalence. CBCs are the only representation that supports multi-input AND and OR gates, which is crucial for efficient translation of quantified formulas. In particular, a quantified formula such as $\forall x_1, \ldots, x_k : X \mid F(x_1, \ldots, x_k)$ is translated as a conjunction of $|X|^k$ circuits that encode the ground forms of $F(x_1, \ldots, x_k)$. CBCs represent this conjunction using a single gate with $|X|^k$ inputs. The other three

representations use $|X|^k - 1$ binary gates.

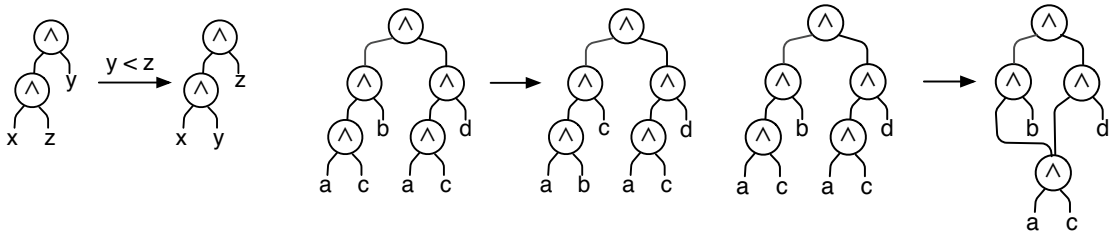
In addition to supporting multi-input gates, CBCs also use different simplification rules than BEDs, AIGs, and RBCs. Both BEDs and AIGs employ *two-level rewrite rules* (i.e. rules that take into account operands' children), which have been shown to inhibit rather than promote subformula sharing [18]. Figure 2-9 displays an example of such a rule, and how it interferes with sharing. A recent implementation of AIGs [19] limited its use of two-level rules to those that do not interfere with sharing. These are a superset of the rules employed by CBCs (Table 2.1). RBCs do not perform any two-level simplifications. Rather, the two-level redundancies missed during the construction of RBCs can be caught with an optional post-construction step [18], which reduces the number of gates in an RBC by up to 10%.

Of the four circuit representations, RBCs are fastest to construct, since their simplification rules take constant time to apply. The construction of a given CBC is also constant, if only binary gates are used. The use of n-ary gates slows the construction of a specific circuit down to $O(\max(i \log i, c))$ time, where i is the number of inputs to the circuit and c is the number of those inputs' children. This, however, is not a bottleneck in practice (§2.4). The complexity of BED and AIG construction is not known [18], because the two-level rules used by BED and AIGs can trigger an unknown number of recursive calls to the circuit creation procedure. The only CBC rules that may trigger a recursive call are the last four ITE rules in Table 2.1, but the resulting recursion bottoms out immediately since the only rules applicable in the recursive call are the AND and OR rules, which are implemented without calling the factory.[3]

2.4 Experimental results

This section concludes the chapter with an empirical evaluation of the presented techniques against related work. The techniques are compared on a collection of 30 problems, analyzed in universes of 4 to 120 atoms (Table 2.2). Twelve of the prob-

[3]Note that each AND and OR rule, when applicable, simply eliminates some or all of the inputs.



(a) A rewrite rule (b) Using rule (a) (c) Using CBC rules

Figure 2-9: An example of a non-optimal two-level rewrite rule. An expression of the form $(x \wedge z) \wedge y$ is re-written to $(x \wedge y) \wedge z$ when $y < z$ according to the vertex ordering $<$. When the rule is applied to the circuit of the form $((a \wedge c) \wedge b) \wedge ((a \wedge c) \wedge d)$, the sharing of the subexpression $(a \wedge c)$ is destroyed because $((a \wedge c) \wedge b)$ is rewritten to $((a \wedge b) \wedge c)$.

lems are Alloy benchmarks, ranging from toy specifications of linked lists and trees to complete specifications of the Mondex transfer protocol [106], Dijkstra's mutual exclusion algorithm [33], a ring leader selection algorithm [20], and a hotel locking scheme [69]. All but two of the Alloy benchmarks (FileSystem and Handshake) are unsatisfiable. The remaining eighteen problems are drawn from the TPTP library [124] of benchmarks for automated theorem provers and model finders. Most of these are rated as "hard" to solve; two (ALG195 and NUM378) have large partial instances; and all of them are unsatisfiable.

The "Kodkod" columns of Table 2.2 show the performance of Kodkod, configured to use CBCs with the sharing detection parameter d set to 3. The performance is measured in terms of the translation and SAT solving times, both given in seconds, and the size of the boolean encoding, given as the number of variables and clauses fed to the SAT solver. The middle region of the table shows the changes in Kodkod's performance when it is configured to use a different circuit representation instead of CBCs with $d = 3$. The "RBC" columns of the SET948 row, for example, show that the use of RBCs on this problem decreases the translation time by a factor of 1.3; increases the size of the boolean encoding by 258,528 variables and 324,666 clauses; and increases the SAT solving time by a factor of 36.69. The last region of the table compares the performance of Alloy3 to that of Kodkod (as given in the "Kodkod" column). The notation "t/o" means that a process did not complete within 5 minutes, and "m/o" means that it ran out of memory. All experiments were performed using

			Kodkod & CBCs with d=3				Kodkod & CBCs with d=1				Kodkod & RBCs				Alloy3			
	problem	univ.	transl.(s)	solve(s)	variables	clauses	transl	solve	variables	clauses	transl	solve	variables	clauses	transl.	solve	variables	clauses
Alloy benchmarks	AWD.A241	51	0.47	109.64	38459	91389	×1.12	×1	+0	+0	×1.17	×1.25	+19362	+28557	×29.97	×0.63	+12467	+325355
	AWD.OpTotal	51	0.25	0.00	0	0	×1	0 sec	+0	+0	×1.12	0 sec	+0	+0	×54.56	2.33 sec	+43256	+381458
	AWD.Ignore	51	0.39	15.01	30821	62457	×0.93	×1	+0	+0	×1.02	×0.77	+14285	+17644	×30.46	×0.28	+12592	+324355
	AWD.Transfer	41	0.36	86.51	21605	48809	×0.92	×1	+0	+0	×0.97	×0.98	+10146	+13837	×41.9	×1.21	+6951	+140325
	Dijkstra	60	41.03	17.91	310718	1770316	×0.99	×1	+0	+0	×0.93	×1.74	+400448	-462284	m/o	m/o	m/o	m/o
	FileSystem	90	13.31	2.19	256025	702536	×0.98	×0.72	+150	+420	×1.09	×0.15	+215224	+329458	×12.77	×6.61	+200325	+1693334
	Handshake	10	0.23	0.50	4913	11313	×0.96	×0.99	+0	+0	×1.07	×2.25	+7724	+20257	×6.55	×0.89	+3499	+27172
	List.Empties	120	65.88	12.11	2547227	7165003	×0.97	×0.29	+360	+300	×1.32	×2.48	+2342214	+3380966	m/o	m/o	m/o	m/o
	List.Reflexive	28	0.58	10.14	34925	92537	×1.03	×1.14	+56	+42	×1.18	×1.7	+24020	+35523	×23.52	t/o	+19501	+238791
	List.Symm	16	0.23	19.17	7285	18199	×1.03	×0.82	+24	+16	×0.94	×1.12	+3962	+5808	×13.88	×0.5	+5856	+51675
	RingElection	16	0.66	22.85	30303	98305	×0.98	×0.89	+24	+24	×1.04	×0.64	+20709	+18789	×15.27	×2.17	+12755	+208211
	Trees	7	9.53	54.84	407399	349797	×0.69	×1.45	+27083	+82719	×0.66	×2.48	+331355	+212278	m/o	m/o	m/o	m/o
TPTP benchmarks	ALG195	14	0.46	0.04	32184	96606	×0.98	×0.67	+8	+196	×1.2	×0.71	+20178	+33710	×101.87	×48.79	+163224	+737902
	ALG197	14	0.42	0.02	28556	83370	×1.04	×0.63	+43	+547	×1.17	×1.46	+18259	+30026	×25.93	×27.63	+21468	+112411
	ALG212	7	6.95	47.31	1072205	1032354	×0.98	×1	+0	+0	×1.4	×2.08	+721464	+395691	t/o	t/o	t/o	t/o
	COM008	12	0.59	5.46	9624	15827	×0.95	×1	+0	+0	×0.98	×2.35	+6803	+8123	×8.73	×0.45	+9823	+84740
	GEO091	20	8.82	57.50	106350	206029	×1.02	×1.68	+32880	+48150	×1.2	t/o	+112559	+153919	×1.08	t/o	+63529	+683959
	GEO092	16	2.79	5.11	48521	92844	×1.01	×1.17	+12818	+18586	×1.02	×1.46	+47626	+64194	×10.56	×1.87	+34444	+499784
	GEO115	18	5.06	46.13	109022	190839	×1.08	×0.86	+25818	+40242	×1.45	t/o	+99463	+132544	t/o	t/o	t/o	t/o
	GEO158	16	2.77	11.07	46673	89825	×1.02	×1.2	+12818	+18586	×0.95	×4.15	+46598	+63166	×10.72	×33.28	+34040	+488887
	GEO159	16	8.83	27.54	87241	200377	×1.01	×1.51	+16738	+25418	×0.96	×2.04	+74058	+107618	t/o	t/o	t/o	t/o
	LAT258	7	1.52	7.56	205647	337033	×0.95	×1.22	+14	+0	×1.39	×1.51	+116703	+117305	m/o	m/o	m/o	m/o
	MED007	35	1.62	39.30	130817	268666	×1	×1	+0	+0	×1.28	×1.66	+107203	+116548	×22.32	×3.23	+5968	+569927
	MED009	35	1.51	44.07	130817	268667	×1.08	×1.01	+0	+0	×1.4	×1.61	+107203	+116548	×26.75	×1.22	+5969	+569928
	NUM374	5	0.65	34.44	62112	198128	×1.16	×1.01	+0	+0	×1.24	×1.19	+33861	+61646	×13.94	×1.02	+8117	+93445
	NUM378	22	0.42	0.00	0	0	×0.86	0 sec	+0	+0	×1.19	0 sec	+0	+0	m/o	m/o	m/o	m/o
	SET943	7	0.41	11.25	16009	39950	×0.99	×1.2	+7	+28	×0.99	×2.16	+10493	+14488	×29	×11.35	+9115	+61090
	SET948	14	4.10	2.08	339151	870110	×0.97	×23.52	+14	+56	×1.3	×116.87	+258528	+324666	×35.44	×19.06	+143243	+1122974
	SET967	4	0.37	0.09	14659	45941	×0.98	×0.92	+16	+48	×1.02	×1.54	+6416	+11425	×8.58	×4.24	+5135	+29063
	TOP020	10	18.03	48.93	2554124	4264114	×0.99	×1	+0	+0	×1.35	×1.18	+1417260	+1522500	m/o	m/o	m/o	m/o
						average:	**×0.99**	**×1.7**			**×1.12**	**×5.63**			**×24.94**	**×8.78**		

Table 2.2: Evaluation of Kodkod's translation optimizations. The first region of the table shows the baseline performance of Kodkod (using CBCs of depth 3), with the translation and solving times given in seconds. The next two regions compare the performance of Kodkod with basic CBCs ($d = 1$) and with RBCs to the baseline. The last region compares Alloy3 to the baseline, where m/o means that a process ran out of memory and t/o means that it did not complete within 5 minutes.

MiniSat on a 2×3 GHz Dual-Core Intel Xeon with 2 GB of RAM.

The results in Table 2.2 demonstrate three things. First, CBCs detect more sharing when d is greater than 1, with a negligible slowdown in the translation time and a significant speed up in the SAT solving time. Most of the sharing detected by CBCs with $d = 3$ that is missed by CBCs with $d = 1$ is undetectable by other circuit representations, since they perform equivalence checking exclusively at the level of children (i.e. at $d = 1$). Second, while the construction of CBCs takes slightly more time than that of RBCs, a CBC-based translation is faster in practice because fewer gates need to be constructed. The CBC circuits are also solved, on average, 5.63 times faster than the corresponding RBCs (not counting the two RBCs on which the SAT solver timed out). Third, the comparison between Alloy3 and Kodkod shows that a translation based on boolean-level sharing detection and sparse matrices is roughly an order of magnitude more effective than a translation that uses problem-level sharing detection and typed matrices. Alloy3 either timed out or ran out of memory on 11 out of 30 problems. It translated the remaining ones 24.94 times slower than Kodkod, producing significantly larger boolean formulas that, in turn, took 8.78 times longer to solve.

Chapter 3

Detecting Symmetries

Many problems exhibit symmetries. A symmetry is a permutation of atoms in a problem's universe that takes models of the problem to other models and non-models to other non-models. The toy filesystem (Fig. 2-2), for example, has six symmetries, each of which permutes the file atoms and maps the directory atoms to themselves. Figures 3-1 and 3-2 show the action of these symmetries on two sets of bindings from filesystem variables to constants: all bindings in Fig. 3-1 satisfy the filesystem constraints and all bindings in Fig. 3-2 violate them. Because symmetries partition a problem's state space into classes of equivalent bindings, a model, if one exists, can be found by examining a single representative of each equivalence class. For problems with large state spaces but few classes of equivalent bindings, exploiting, or *breaking*, symmetries can lead to exponential gains in model finding efficiency.

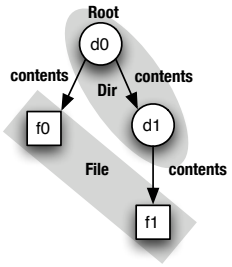
Model finders exploit symmetries in two ways. Dynamic techniques explore the state space of a problem directly, with a dedicated algorithm that employs symmetry-based heuristics to guide the search [70, 93, 151, 152]. Static approaches use a black-box search engine, such as a SAT solver, on a problem that has been augmented with *symmetry breaking predicates* [27, 55, 116]. These predicates consist of constraints that are true of at least one binding in each equivalence class (but not all); adding them to the translation of a problem blocks a backtracking solver from exploring regions of the search space that contain no representative bindings. Kodkod, like other SAT-based model finders [117, 25], takes the static approach to symmetry breaking.

To use either approach, however, a model finder must first find a set of symmetries to break. In the case of logics accepted by traditional model finders, symmetry detection is straightforward. Given a universe of typed atoms, every permutation that maps atoms of the same type to one another is a symmetry of any problem defined over that universe [70]. Atoms of the same type can be freely interchanged because traditional logics provide no means of referring to individual atoms. But logics accepted by model extenders such as Kodkod and IDP1.3 do, which makes interchangeable atoms, and therefore symmetries, hard to identify. This chapter presents a new greedy technique for finding a useful subset of the available symmetries in the context of model extension. The technique is shown to be correct, locally optimal, effective in practice, and more scalable than a complete symmetry discovery method based on graph automorphism detection.

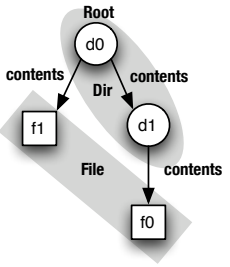
3.1 Symmetries in model extension

When configured or analyzed declaratively, systems with interchangeable components give rise to model finding (or extension) problems with sets of symmetric atoms. These symmetries, in turn, induce equivalences or isomorphisms among bindings in the problem's state space. More precisely, a symmetry is a permutation of atoms in a problem's universe that acts on a model of the problem to produce a model and on a non-model to produce a non-model. The action of a symmetry on a binding from variables to relational constants is defined as follows: a symmetry acts on a binding by acting on each of its constants; on a constant by acting on each of its tuples; and on a tuple by acting on each of its atoms. Two bindings that are related by a symmetry are said to be *isomorphic*.

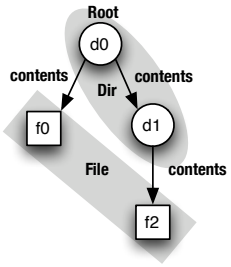
The set of symmetries of a problem P, denoted by $\mathrm{Sym}(P)$, defines an equivalence relation on the bindings that comprise the state space of P. Two bindings b and b' are equivalent if $b' = l(b)$ for some $l \in \mathrm{Sym}(P)$. Because equivalent bindings all satisfy or all violate P, it suffices to test one binding in each equivalence class induced by $\mathrm{Sym}(P)$ to find a model of P. Many practical problems, especially in bounded



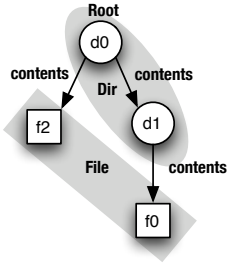
(a) Original model (Fig. 2-2b)



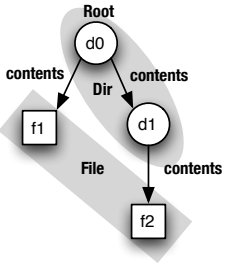
(b) Permutation (`f0 f1`)



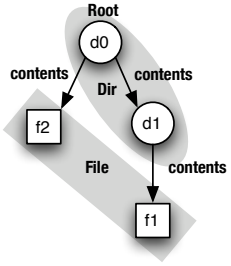
(c) Permutation (`f1 f2`)



(d) Permutation (`f0 f2 f1`)

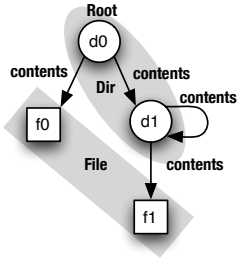


(e) Permutation (`f0 f1 f2`)

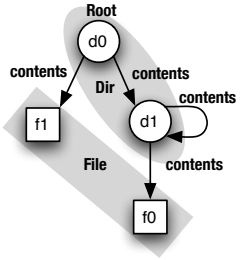


(f) Permutation (`f0 f2`)

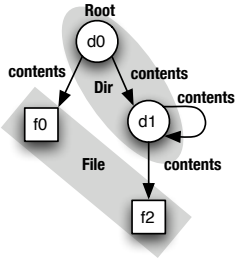
Figure 3-1: Isomorphisms of the filesystem model. Atom permutations are shown in the cycle notation for permutations [7], where each element in a pair of parenthesis is mapped to the one following it. The last element is mapped to the first, and the excluded elements are mapped to themselves.



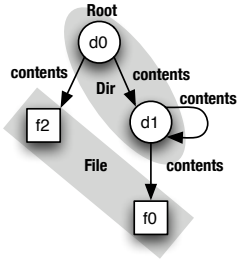
(a) An invalid binding



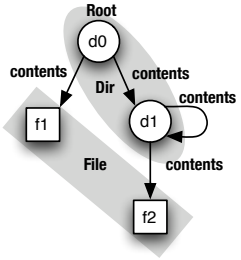
(b) Permutation (`f0 f1`)



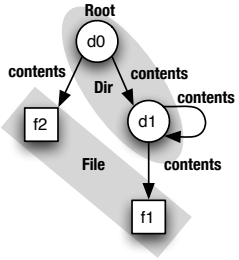
(c) Permutation (`f1 f2`)



(d) Permutation (`f0 f2 f1`)



(e) Permutation (`f0 f1 f2`)



(f) Permutation (`f0 f2`)

Figure 3-2: Isomorphisms of an invalid binding for the toy filesystem. The bindings shown here are invalid because the contents relation maps `d1` to itself, violating the acyclicity constraint. Atom permutations are shown in cycle notation.

verification, have an exponentially large state space but only a polynomial number of equivalence classes [117]. Exploitation of symmetries, or symmetry breaking, is crucial for making such problems tractable [27, 111].

Until recently, much of the work on symmetries in model finding has focused on the design of better symmetry breaking methods [70, 93, 151, 152, 116]. Symmetry detection has received less attention because discovering symmetries of a traditional model finding problem is straightforward. If a problem P is specified in an unsorted logic, then $\mathrm{Sym}(P)$ contains every permutation of atoms in the universe of P; if the logic is typed, then $\mathrm{Sym}(P)$ consists of all permutations that map atoms of the same type to one another [70]. These results, however, no longer hold when a standard logic is extended to support partial models. If P is specified as a model extension problem that references individual atoms, as it may be in bounded relational logic, finding $\mathrm{Sym}(P)$ becomes intractable.

Consider, for example, the toy filesystem from the previous chapter and two candidate symmetries, $l_1 = (\texttt{f0}\,\texttt{f2})$ and $l_2 = (\texttt{d0}\,\texttt{d1})$.[1] Both permutations respect the intrinsic "types" in the filesystem universe, mapping directories to directories and files to files. When the two are applied to the model in Fig. 2-2b, however, only l_1 yields a model (Fig. 3-1f). The permutation l_2 yields the following binding, which satisfies the filesystem specification but violates the bound constraints on the contents and Root relations: $\{\mathsf{File} \mapsto \{\langle\texttt{f0}\rangle, \langle\texttt{f1}\rangle\}, \mathsf{Dir} \mapsto \{\langle\texttt{d1}\rangle, \langle\texttt{d0}\rangle\}, \mathsf{Root} \mapsto \{\langle\texttt{d1}\rangle\}, \mathsf{contents} \mapsto \{\langle\texttt{d1,d0}\rangle, \langle\texttt{d1,f0}\rangle, \langle\texttt{d0,f1}\rangle\}\}$.

This example reveals two important properties of untyped model extension problems.[2] First, *any* permutation is a symmetry of a specification, by the result from traditional model finding (Lemma 3.1). The permutation l_2, for example, is a symmetry of the filesystem specification even though it is not a symmetry of the entire problem. Second, a permutation is a symmetry of a set of bound constraints only if it maps each constant within those bounds to itself (Lemma 3.2). The permutation l_1 is

[1] The cycle notation for permutations is explained in Fig. 3-1.

[2] All results from this chapter apply to other model extension logics, such as IDP, which restrict the use of constants to type or partial model constraints (i.e. $v = c$, $v \subseteq c$, and $c \subseteq v$ where v is a variable and c is a constant). If constants can be used freely, then Thm. 3.1 describes a subset of $\mathrm{Sym}(P)$ rather than all of it; that is, only the $\Leftarrow$ direction of the theorem holds.

a symmetry of the filesystem bounds and of the entire problem. As a result, $\mathrm{Sym}(P)$ for a Kodkod problem P is the set of all permutations that map each constant in P's bounds to itself (Thm. 3.1). This, however, means that finding $\mathrm{Sym}(P)$ is as hard as graph automorphism detection (Thm. 3.2), which has no known polynomial time solution [8].

Lemma 3.1 (Symmetries of specifications)**.** *If S is a bounded relational specification over a (finite) universe U, then every permutation $l : U \to U$ is a symmetry of S.*

Proof. Since S is a specification in a standard untyped logic (i.e. it contains no constants), the symmetry result from traditional model finding applies [70]. □

Lemma 3.2 (Symmetries of bounds)**.** *Let U be a finite universe and let B be a set of bounds of the form $v :_k [c_L, c_U]$, where v is a variable of arity k and $c_L, c_U \in 2^{U^k}$. A permutation $l : U \to U$ is a symmetry of B if and only if $l(c) = c$ for each relational constant c that occurs in B.*

Proof. ($\Leftarrow$, by contradiction). Suppose that $l(c) = c$ for all c in B and that l is not a symmetry of B. Then, there must be some binding b from variables in B to sets of tuples that satisfies B, written as $b \models B$, but $l(b) \not\models B$. For this to be true, there must also be some $v :_k [c_L, c_U]$ in B such that $c_L \not\subseteq l(b(v))$ or $l(b(v)) \not\subseteq c_U$. Since $b \models B$, $c_L \subseteq b(v) \subseteq c_U$. Let x be the difference between $b(v)$ and c_L, i.e. $x = b(v) \setminus c_L$. This leads to $l(b(v)) = l(c_L \cup x) = l(c_L) \cup l(x) = c_L \cup l(x)$, which implies that $c_L \subseteq l(b(v))$. So it must be the case that $l(b(v)) \not\subseteq c_U$. But this cannot be true, because the fact that $b(v) \subseteq c_U$ and that $l(c_U) = c_U$ together imply that $l(b(v)) \subseteq l(c_U) \subseteq c_U$.

($\Rightarrow$, by contradiction). Suppose that B is a set of bounds and that l is a symmetry of B with $l(c) \neq c$ for some constant c in B. Then, there must be some $v :_k [c_L, c_U]$ in B such that $c_L = c$ or $c_U =$ c. Let b be the binding that maps v to c and every other variable in B to its lower bound. By construction, b is a model of B, written as $b \models B$. Because $l(c) \neq c$ and l is a permutation, it must be the case that $l(c) \not\subseteq c$ and $c \not\subseteq l(c)$. If $c = c_U$, then $b(v) = c$ implies that $l(b(v)) = l(c) = l(c_U) \not\subseteq c_U$. This means that $l(b) \not\models v :_k [c_L, c_U]$ and therefore $l(b) \not\models B$, contradicting the assumption

that l is a symmetry of B. Similarly, if $c = c_L$, then $c_L \not\subseteq l(c_L) = l(b(v))$, which, once again, means that $l(b) \not\models v :_k [c_L, c_U]$, $l(b) \not\models B$, and l is not a symmetry of B. □

Theorem 3.1 (Symmetries of problems)**.** *Let P be a bounded relational problem over a (finite) universe U. A permutation $l : U \to U$ is a symmetry of P if and only if $l(c) = c$ for all constants c that occur in the bounds of P.*

Proof. ($\Leftrightarrow$). Suppose that P and l are a problem and a permutation with the stated properties. Let b be a binding that maps the free variables in P to constants drawn from U. By the semantics of the logic, $[\![P]\!]_b = [\![B]\!]_b \wedge [\![S]\!]_b$, where B are the bound constraints and S is the specification of P. By Lemmas 3.1 and 3.2, $[\![S]\!]_b = [\![S]\!]_{l(b)}$ and $[\![B]\!]_b = [\![B]\!]_{l(b)}$. Hence, $[\![P]\!]_b = [\![P]\!]_{l(b)}$. Because a symmetry of P has to be a symmetry of all its constraints, and by Lemma 3.2, only permutations that fix constants are symmetries of bound constraints, all symmetries of P fix the constants that occur in B. □

Theorem 3.2 (Hardness of symmetry detection for model extension)**.** *Let P be a bounded relational problem over a (finite) universe U. Finding* $\mathrm{Sym}(P)$ *is as hard as graph automorphism detection.*

Proof. (by reduction) Let $G = (V, E)$ be some graph; U a universe consisting of the nodes in V; and c a binary relation that contains the tuple $\langle v_i, v_j \rangle$ (and the tuple $\langle v_j, v_i \rangle$, if G is undirected) whenever E contains an edge from v_i to v_j. By Thm. 3.1, the symmetries of the problem $P = g :_2 [c, c]$ are the automorphisms of G. □

3.2 Complete and greedy symmetry detection

Many graphs that arise in practice are amenable to efficient symmetry detection with tools like Nauty [94], Saucy [28] and Bliss [72]. Relational bounds for small to medium-sized problems, for example, correspond to graphs with easily detectable symmetries. Bounds for large problems, however, are much harder for graph automorphism tools; the cost of exact symmetry detection for a large problem generally rivals the cost of finding its model with a SAT solver (§3.3).

This section therefore presents two new symmetry detection algorithms for model extension problems. One is a complete method based on graph automorphism detection, which works for smaller problems. The other is a greedy approach that scales to large problems. Unlike the complete method, which is guaranteed to find all symmetries for all problems, the greedy algorithm finds all symmetries for some problems and a subset of the symmetries for others. This, as the next section shows, is not an issue in practice because the problems that are most likely to benefit from symmetry breaking are those on which the greedy algorithm is most effective.

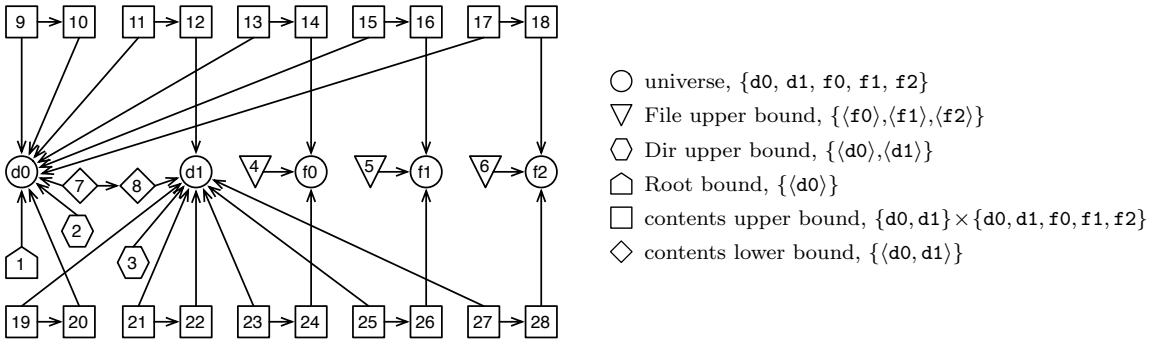
3.2.1 Symmetries via graph automorphism detection

Figure 3-3a shows the pseudocode for a complete symmetry discovery method based on graph automorphism detection. The intuition behind the algorithm is that a relational constant of arity k can be represented as a set of linked lists of length k, each of which encodes a tuple in the obvious way. A set of n relational constants that make up the bounds of a problem P can then be converted into a directed colored graph $G = (V, E)$ as follows (Fig. 3-3b). The vertices of the graph are partitioned into $n + 1$ sets, $V_0, \ldots, V_n$. All vertices in the same partition have the same color, which differs from the colors of all other partitions. The set V_0 consists of the atoms that make up the universe of P. The set V_i $(i > 0)$ consists of the buckets of the lists that comprise the i^{th} constant. The set E encodes the list pointers as graph edges. There is an edge between v and v' in V_i $(i > 0)$ only if the bucket v points to the bucket v' in the list representation of a given tuple. Similarly, E contains an edge between $v \in V_i$ $(i > 0)$ and $v' \in V_0$ only if the bucket v contains the atom v'.

When the graph G is fed to Nauty [94], the result is a set of permutations that map G to itself and that map vertices of the same color to one another. Because there is a one-to-one correspondence between the constants in P and the (colored) vertices and edges of G, the set of all automorphisms of G, denoted as $\text{Aut}(G)$, is exactly the set of symmetries of P when the action of each symmetry on the buckets is omitted (Thm. 3.3). Figure 3-3c shows the automorphisms of the graph that represents the filesystem bounds (Fig. 3-3b), as given by Nauty.

```
SYM-COMPLETE(U : universe, C : set of constants drawn from U)
 1  V ← COLOR-FRESH(U)
 2  E ← {}
 3  for all c ∈ C do
 4    V_c ← {}
 5    for all ⟨a_0, a_1, ..., a_{k-1}⟩ ∈ c do
 6      t ← array of k fresh vertices
 7      for all 0 ≤ i < k do
 8        V_c ← V_c ∪ {t[i]}
 9        E ← E ∪ {⟨t[i], a_i⟩}
10      for all 0 ≤ i < k − 1 do
11        E ← E ∪ {⟨t[i], t[i + 1]⟩}
12    COLOR-FRESH(V_c)
13    V ← V ∪ V_c
14  return AUTOMORPHISMS(V, E)
```

(a) Pseudocode for the detection algorithm. The procedure COLOR-FRESH colors all vertices in a given set with the same color, which has not been used before. The procedure AUTOMORPHISMS calls an automorphism detection tool such as Nauty on the given colored graph.



(b) Graph representation of the filesystem bounds. For clarity, vertex shape is used in place of color to partition the vertices.

(f0 f1)(4 5)(13 15)(14 16)(23 25)(24 26)
(f1 f2)(5 6)(15 17)(16 18)(25 27)(26 28)
(f0 f2)(4 6)(13 17)(14 18)(23 27)(24 28)
(f0 f1 f2)(4 5 6)(13 15 17)(14 16 18)(23 25 27)(24 26 28)
(f0 f2 f1)(4 6 5)(13 17 15)(14 18 16)(23 27 25)(24 28 26)

(c) Automorphisms of the filesystem graph. The identity permutation is not shown.

Figure 3-3: Complete symmetry detection via graph automorphism.

Theorem 3.3 (Soundness and completeness)**.** *Let U be a finite universe and C a collection of relational constants drawn from U. Applying* SYM-COMPLETE *to U and C yields a colored graph $G = (V, E)$ such that the set of all automorphisms of G, denoted as* $\mathrm{Aut}(G)$*, is exactly* $\mathrm{Sym}(C)$ *when the action of each symmetry in* $\mathrm{Aut}(G)$ *on $V \setminus U$ is ignored.*

Proof. Let $\mathrm{Aut}(G)_U$ denote the set of all permutations obtained by ignoring the action of each symmetry in $\mathrm{Aut}(G)$ on $V \setminus U$. By the construction of $G = (V, E)$, there is an exact correspondence between the constants in C and the nodes and edges in G. In particular, G is the representation of each $c \in C$ as a set of lists, where each list encodes a tuple $t \in c$ in the obvious way. Let $l \in \mathrm{Aut}(G)$ be a symmetry of G. By the definition of a symmetry and the construction of G, l maps the set-of-lists representation of each $c \in C$ to itself. Consequently, when the action of l is restricted to the vertices in U (by ignoring its action on other vertices in V), the resulting permutation is a symmetry of each $c \in C$. In other words, $\mathrm{Aut}(G)_U \subseteq \mathrm{Sym}(C)$. It follows trivially from the construction of G that $\mathrm{Sym}(C) \subseteq \mathrm{Aut}(G)_U$. □

3.2.2 Symmetries via greedy base partitioning

While SYM-COMPLETE works for small and medium-sized problems, it performs relatively poorly on large relational problems that this thesis targets. Kodkod therefore uses a greedy algorithm, called *Greedy Base Partitioning* (GBP), to find a subset of the available symmetries that can be detected in polynomial time. This subset is complete (i.e. includes all symmetries) for *pure* model finding problems, in which the use of constants is limited to the encoding of type information. The algorithm is incomplete for problems with partial models or arbitrary upper bounds on free variables. Generally, the larger the partial model, the fewer symmetries are discovered.

This approach is practical for two reasons. First, the problems that are most likely to benefit from symmetry breaking are unsatisfiable and have large state spaces with few isomorphism classes. Pure model finding problems, which typically encode bounded verification tasks, tend to have both of these properties. Second, for any

```
1  {d0, d1, f0, f1, f2}

2  File      :1 [{}, {⟨f0⟩,⟨f1⟩,⟨f2⟩}]
3  Dir       :1 [{}, {⟨d0⟩,⟨d1⟩}]
4  Root      :1 [{}, {⟨d0⟩,⟨d1⟩}]
5  contents :2 [{}, {⟨d0, d0⟩, ⟨d0, d1⟩, ⟨d0, f0⟩, ⟨d0, f1⟩, ⟨d0, f2⟩,
                     ⟨d1, d0⟩, ⟨d1, d1⟩, ⟨d1, f0⟩, ⟨d1, f1⟩, ⟨d1, f2⟩}]

6  one Root
7  Root ⊆ Dir
8  contents ⊆ Dir → (Dir ∪ File)
9  ∀ d: Dir | ¬(d ⊆ d.^contents)
10 (File ∪ Dir) ⊆ Root.*contents
```

(a) Problem description

```
(f0 f1)
(f1 f2)
(f0 f2)
(f0 f1 f2)
(f0 f2 f1)
(d0 d1)
(d0 d1)(f0 f1)
(d0 d1)(f1 f2)
(d0 d1)(f0 f2)
(d0 d1)(f0 f1 f2)
(d0 d1)(f0 f2 f1)
```

(b) Symmetries of the new filesystem

Figure 3-4: A toy filesystem with no partial model. This formulation differs from the original (Fig. 2-2) in that there are no lower bounds; the upper bound on Root is extended to range over all directories; and Root is explicitly constrained to be a singleton. The effect is a doubling in the number of symmetries. (The identity symmetry is not shown.)

problem to benefit from symmetry breaking, relatively few symmetries need to be broken. Problems with partial models, which typically encode declarative configuration tasks, already have many of their symmetries implicitly broken by the partial model. If the toy filesystem, for example, were re-written to have no partial model (Fig. 3-4a), it would have 12 symmetries (Fig. 3-4b). The presence of the partial model, however, cuts that number in half.

GBP (Fig. 3-5a) is essentially a type inference algorithm. It works by partitioning the universe of a problem P into sets of atoms such that each constant in P can be expressed as a union of products of those sets (Def. 3.1). This is called a *base partitioning* (Def. 3.2), and every permutation that maps base partitions to themselves is a symmetry of P (Thm. 3.4). Since large partitions have more permutations than small partitions, GBP produces the coarsest base partitioning for P (Thms. 3.5-3.6). For a pure model finding problem, these partitions correspond to the types that comprise the problem's universe and hence capture all available symmetries. The coarsest base partitioning for the filesystem formulation in Fig. 3-4a, for example, is {{d0, d1}, {f0, f1, f2}}, which defines all 12 symmetries of the problem. When a partial model is present, the partitions found by GBP are guaranteed to capture a subset of Sym(P); they may capture all of Sym(P) if the partial model is fairly simple, as is

the case for the original toy filesystem (Fig. 3-5b).

Definition 3.1 (Flat product)**.** *Let c and c' be relational constants. The flat product of c and c', denoted by $c \otimes c'$, is defined as $\{\langle a_1, \ldots, a_k, b_1, \ldots, b_{k'}\rangle \mid \langle a_1, \ldots, a_k\rangle \in c \wedge \langle b_1, \ldots, b_{k'}\rangle \in c'\}$. The flat product is also applicable to sets, which are treated as unary relations. That is, every set s is lifted to the unary relation $\{\langle a\rangle \mid a \in s\}$ before the flat product is applied to it.*

Definition 3.2 (Base partitioning)**.** *Let U be a finite universe, c a non-empty relational constant drawn from U, and $S = \{s_1, \ldots, s_n\}$ a set of sets that partition U. The set S is a* base partitioning *of U with respect to c if $c = \bigcup_{i=1}^{n} s_{i1} \otimes \ldots \otimes s_{ik}$ for some $s_{11}, \ldots, s_{nk} \in S$. Base partitionings of U with respect to c, denoted by $\beta_U(c)$, are partially ordered by the refinement relation $\sqsubseteq$, which relates two partitionings R and S, written as $R \sqsubseteq S$, if every partition in S is a union of partitions in R.*

Theorem 3.4 (Soundness)**.** *Let U be a finite universe, P a problem over U, and $S = \{s_1, \ldots, s_n\}$ a base partitioning of U with respect to the constants that occur in P. If a permutation $l : U \to U$ maps each $s \in S$ to itself, then $l \in \mathrm{Sym}(P)$.*

Proof. Let c be a constant that occurs in P. Because l maps each partition in S to itself, and c is a union of products of some partitions in S, $l(c) = c$. This, by Thm. 3.1, implies that $l \in \mathrm{Sym}(P)$. □

Theorem 3.5 (Local optimality of GBP)**.** *Let U be a finite universe and C a collection of relational constants drawn from U. Applying* GBP *to U and C yields the coarsest base partitioning of U with respect to C.*

Proof. (by induction on the size of C). If C is empty, lines 2-3 of GBP never execute, so the algorithm returns $\{U\}$. Suppose that C is non-empty, and let c_i and S_i designate the values of the variables c and S at the end of the i^{th} loop iteration. Since S is initialized to $\{U\}$, S_1 is, by Thm. 3.6, the coarsest base partitioning of U with respect to c_1. Suppose that S_{n-1} is the coarsest base partitioning of U with respect to $c_1, \ldots, c_{n-1}$. It follows from Def. 3.2 and the coarseness of S_{n-1} that any base partitioning of U with respect to $c_1, \ldots, c_{n-1}$ must be a refinement of S_{n-1}. By Thm.

```
GBP(U: universe, C: set of constants drawn from U)
 1  S ← {U}
 2  for all c ∈ C do
 3    S ← PART(c, S)
 4  return S

PART(c: constant, S: set of universe partitions)
 1  k ← arity(c)
 2  c_first ← {a_1 | ⟨a_1, ..., a_k⟩ ∈ c}
 3  for all s ∈ S such that s ⊈ c_first ∧ s ∩ c_first ≠ ∅ do
 4    S ← (S \ s) ∪ {s ∩ c_first} ∪ {s \ c_first}
 5  if k > 1 then
 6    C ← {}
 7    for all s ∈ S such that s ∩ c_first ≠ ∅ do
 8      S ← S \ s
 9      while s ≠ ∅ do
10        x ← chooseElementFrom(s)
11        x_rest ← {⟨a_2, ..., a_k⟩ | ⟨x, a_2, ..., a_k⟩ ∈ c}
12        x_part ← {a ∈ s | {⟨a_2, ..., a_k⟩ | ⟨a, a_2, ..., a_k⟩ ∈ c} = x_rest}
13        s ← s \ x_part
14        S ← S ∪ {x_part}
15        C ← C ∪ {x_rest}
16    for all c' ∈ C do
17      S ← PART(c', S)
18  return S
```

(a) Pseudocode for greedy base partitioning. Variables whose names begin with a capital letter point to values whose type is a set of sets. Variable whose names begin with a lower case letter point to scalars or sets (of tuples or atoms).

line	description	S
1	initialize S to $\{\{\texttt{d0}, \texttt{d1}, \texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$	$\{\{\texttt{d0}, \texttt{d1}, \texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$
3	PART$(S, \{\langle\texttt{f0}\rangle, \langle\texttt{f1}\rangle, \langle\texttt{f2}\rangle\})$	$\{\{\texttt{d0}, \texttt{d1}\}, \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$
3	PART$(S, \{\langle\texttt{d0}\rangle, \langle\texttt{d1}\rangle\})$	$\{\{\texttt{d0}, \texttt{d1}\}, \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$
3	PART$(S, \{\langle\texttt{d0}\rangle\})$	$\{\{\texttt{d0}\}, \{\texttt{d1}\}, \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$
3	PART$(S, \{\langle\texttt{d0}, \texttt{d1}\rangle\})$	$\{\{\texttt{d0}\}, \{\texttt{d1}\}, \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$
	PART$(S, \{\langle\texttt{d1}\rangle\})$	$\{\{\texttt{d0}\}, \{\texttt{d1}\}, \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$
3	PART$(S, \{\texttt{d0}, \texttt{d1}\} \times \{\texttt{d0}, \texttt{d1}, \texttt{f0}, \texttt{f1}, \texttt{f2}\})$	$\{\{\texttt{d0}\}, \{\texttt{d1}\}, \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$
	PART$(S, \{\langle\texttt{d0}\rangle, \langle\texttt{d1}\rangle, \langle\texttt{f0}\rangle, \langle\texttt{f1}\rangle, \langle\texttt{f2}\rangle\})$	$\{\{\texttt{d0}\}, \{\texttt{d1}\}, \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$
4	return $\{\{\texttt{d0}\}, \{\texttt{d1}\}, \{\texttt{f0}, \texttt{f1}, \texttt{f2}\}\}$	

(b) Applying GBP to the toy filesystem. The figure shows a condensed representation of the trace produced by GBP on the toy filesystem. The first two columns give the line number and a description of the executed GBP statement. The last column shows the effect of the given statement on the value of S from the preceding row. Recursive calls to PART are indented.

Figure 3-5: Symmetry detection via greedy base partitioning.

3.6, S_n is the coarsest refinement of S_{n-1} such that $S_n \in \beta_U(c_n)$. Hence, S_n must be the coarsest base partitioning of U with respect to $c_1, \ldots, c_{n-1}, c_n$. □

Theorem 3.6 (Local optimality of PART)**.** *Let U be a finite universe, c a relational constant drawn from U, and S a set of sets that partition U. Applying* PART *to c and S yields the coarsest refinement of S that is a base partitioning of U with respect to c.*

Proof. (by induction on the arity of c). Suppose that $\text{arity}(c) = 1$. Let S_0 denote the value of S that is passed to the procedure, and let S_m denote the value of S after the first loop (lines 3-4) has terminated. Since an iteration of the loop simply splits a partition $s \in S_0$ into two partitions that add up to s, $S_m \sqsubseteq S_0$. The loop condition ensures that all partitions in S_m are either contained in c or do not intersect with it, so $S_m \in \beta_U(c)$. Finally, S_m is the coarsest refinement of S_0 that is a base partitioning with respect to c since none of the partitions created by the loop can be merged without taking away one of the partitions that makes up c.

For the inductive case, suppose that the theorem holds for all constants of arity $k-1$, and suppose that $\text{arity}(c) = k$. Let S_y denote the value of S after the second loop (lines 7-15) has terminated; and let S_z be the value of S after the third loop (lines 16-17) has terminated. The first loop refines S_0 into S_m, which is the coarsest base partitioning of U with respect to the first column of c. The second loop splits each $s \in S_m$ that intersects with c_{first} into the smallest number of subpartitions such that c maps all atoms in a subpartition, denoted by x_{part}, to the same constant of arity $k-1$, denoted by x_{rest}. This ensures that S_y is the coarsest refinement of S_0 such that c is a union of products of each $x_{\mathit{part}} \in S_y \setminus S_m$ with its corresponding $x_{\mathit{rest}} \in C$. By the inductive hypothesis, S_z is the coarsest refinement of S_y that is a base partitioning of U with respect to each $x_{\mathit{rest}} \in C$. Hence, by Lemma 3.3, S_z is the coarsest refinement of S_0 that is a base partitioning of U with respect to c. □

Lemma 3.3 (Partitioning invariant)**.** *Let U be a finite universe, c a constant of arity $k > 1$ drawn from U, and S a set of sets that partitions U. Let R be the coarsest refinement of S such that $c = \bigcup_{i=1}^{n} p_i \otimes c_i$ for some distinct partitions $p_1, \ldots, p_n \in R$*

and some distinct constants $c_1, \ldots, c_n$ of arity $k-1$. If Q is a refinement of S that is a base partitioning with respect to c, then Q is a refinement of R that is a base partitioning of U with respect to $c_1, \ldots, c_n$.

Proof. The proof follows directly from Def. 3.2 and the definition of R. □

3.3 Experimental results

Table 3.1 shows the results of applying GBP and SYMM-COMPLETE to a collection of 40 benchmarks. Thirty are the Alloy and TPTP benchmarks introduced in the previous chapter. The remaining ten are drawn from a public database of graph coloring problems [132]. The graphs to be colored are encoded as (large) partial models, and the k-colorability problem for each graph is encoded as a relational formula over a universe with k color atoms. All ten problems are unsatisfiable. The benchmarks in the "mulsol" and "zeroin" families represent register allocation problems for variables in real code; the "school" benchmarks are class scheduling problems.

The first two columns of Table 3.1 display the name of each benchmark and the size of its partial model, given as the number of known tuples. The third column shows the size of each problem's state space, given as the number of bits needed to encode all possible bindings that extend the problem's partial model (if any). The state space of AWD.A241, for example, consists of 2^{2512} distinct bindings. The "GBP" and "Symm-Complete" columns display the time, in milliseconds, taken by the corresponding algorithm to find the given number of symmetries, expressed in bits. The last two columns hold the SAT solving time for each problem, in milliseconds, with and without symmetry breaking. The notation "t/o" indicates that a given process failed to produce a result within 5 minutes (300,000 ms).

All experiments were performed on a 2×3 GHz Dual-Core Intel Xeon with 2 GB of RAM, using Kodkod's translation algorithm to reduce the benchmarks to SAT and MiniSat to solve the resulting boolean formulas. GBP is implemented as a part of Kodkod and works on the tool's internal representation of relational constants as sets of integers. The SYMM-COMPLETE implementation converts each Kodkod constant

				GBP		Symm-Complete		SAT time (ms)	
	problem	partial model	state space	time (ms)	symmetries (log base 2)	time (ms)	symmetries (log base 2)	with a SBP	without a SBP
Alloy benchmarks	AWD.A241	2	2512	8	105.63	246286	105.63	131331	t/o
	AWD.Op_total	2	2492	7	105.63	247628	105.63	0	0
	AWD.Ignore	2	2501	8	105.63	250810	105.63	18803	t/o
	AWD.Transfer	2	1370	7	73.50	20205	73.50	108280	t/o
	Dijkstra	0	16040	8	183.23	t/o	t/o	18070	182834
	FileSystem	1	3810	6	318.22	281662	318.22	2197	76
	Handshake	12	200	5	15.30	26	15.30	496	20
	Lists.Empties	0	14640	9	544.27	t/o	t/o	13803	t/o
	Lists.Reflexive	0	854	5	72.69	848	72.69	11824	22729
	Lists.Symmetric	0	288	4	30.60	46	30.60	23406	t/o
	RingElection	0	640	5	30.60	1911	30.60	23185	23281
	Trees	0	56	3	12.30	7	12.30	55645	t/o
TPTP benchmarks	ALG195	37	1052	3	0.00	128	0.00	43	42
	ALG197	37	1064	3	0.00	137	0.00	24	35
	ALG212	0	2436	7	12.30	107828	12.30	55879	t/o
	COM008	11	397	4	25.25	91	25.25	5753	t/o
	GEO091	0	2470	6	43.58	102381	43.58	74334	248153
	GEO092	0	1344	6	30.60	5893	30.60	6379	5363
	GEO115	0	8415	9	36.94	t/o	t/o	104182	59454
	GEO158	0	1312	5	30.60	6321	30.60	13231	7404
	GEO159	0	5408	7	30.60	t/o	t/o	48082	18090
	LAT258	0	792	8	9.49	948	9.49	15863	t/o
	MED007	0	1890	8	132.92	9510	132.92	41145	48853
	MED009	0	1890	7	132.92	9500	132.92	44604	53226
	NUM374	0	415	5	6.91	156	6.91	35016	t/o
	NUM378	462	2024	5	0.00	366	0.00	0	0
	SET943	0	511	5	12.30	278	12.30	11312	t/o
	SET948	0	6314	8	36.34	t/o	t/o	2931	t/o
	SET967	0	400	6	4.58	127	4.58	89	118
	TOP020	0	3620	8	21.79	140462	21.79	49346	55902
Graph coloring benchmarks	mulsol.i.1	8074	5319	20	395.11	t/o	t/o	5380	t/o
	mulsol.i.2	7985	5076	24	178.30	t/o	t/o	18078	t/o
	mulsol.i.3	8043	4968	24	159.84	t/o	t/o	19654	t/o
	mulsol.i.4	8104	4995	23	159.84	t/o	t/o	10236	t/o
	mulsol.i.5	8159	5022	23	161.84	t/o	t/o	27772	t/o
	school1_nsh	29589	4576	79	33.54	t/o	t/o	11269	t/o
	school1	38588	5005	98	33.54	t/o	t/o	215	t/o
	zeroin.i.1	8438	5697	20	535.59	t/o	t/o	9444	t/o
	zeroin.i.2	7321	5908	21	373.65	t/o	t/o	32945	t/o
	zeroin.i.3	7313	5562	20	343.51	t/o	t/o	3510	t/o

Table 3.1: Evaluation of symmetry detection algorithms. Partial model size is given as the number of known tuples. The size of a problem's state space is measured in the number of bits needed to encode all possible bindings that extend its partial model. The number of symmetries is also expressed in bits. All times are given in milliseconds; "t/o" means that the given process failed to complete within 5 minutes.

to a graph and uses Nauty [94] for automorphism detection.[3] Each benchmark was solved twice, once without any symmetry breaking and once with a predicate based on GBP's output.

The resulting data illustrate the key points of this chapter: (1) many model finding and extension problems are infeasible without symmetry breaking; (2) complete symmetry detection is impractical for large problems; and (3) the subset of symmetries detected by greedy base partitioning is sufficient for effective symmetry breaking. SYMM-COMPLETE timed out on 35% of the problems, and it was, on average, 8125 times slower than GBP on the remaining ones. GBP found all symmetries for Alloy and TPTP benchmarks, which are either pure model finding problems or have small partial models. It produced only a subset of the available symmetries[4] for the graph coloring benchmarks. Each of these subsets, however, captured enough symmetries to enable MiniSat to solve the corresponding problem within the time limit.

3.4 Related work

Symmetry breaking has been extensively studied in both traditional model finding and in constraint programming. Most of the work on symmetries in traditional model finding has focused on the design of better symmetry breaking heuristics, dynamic and static. Detection of symmetries is performed implicitly by type inference algorithms. The first half of this section summarizes that work. The second half covers the work on symmetry breaking and detection in constraint programming. The latter is of particular interest since detecting symmetries of constraint programs is, in many ways, harder than detecting symmetries of both model finding and model extension problems.

[3] The more recent graph automorphism detection tools [28, 72] work only on undirected graphs.

[4] All ten graphs are undirected and have symmetries detectable with Bliss [72] that are not detected by GBP.

3.4.1 Symmetries in traditional model finding

Most search-based model finders [70, 93, 122, 151, 152] use a variant of the Least Number Heuristic (LNH) [151] to reduce the number of isomorphic bindings that are examined during search. The basic idea behind LNH is simple. At a given point in the search, a type T is partitioned into elements $\{e_1, \ldots, e_i\}$ that appear in the current partial assignment from variables to values and the elements $\{e_{i+1}, \ldots, e_n\}$ that do not. The unconstrained elements, $\{e_{i+1}, \ldots, e_n\}$, are symmetric, so only one of them (e.g. e_{i+1}) needs to be considered in the subsequent assignments. In particular, suppose that C is a set of (ground) first order clauses, A is a partial assignment of variables constrained by C to values, and v is an unassigned variable of type T. To find a model of C in this scenario, it is sufficient to test only the bindings that extend A with an assignment from v to an element in $\{e_1, \ldots, e_{i+1}\}$. In practice, LNH improves search times by orders of magnitude for many problems [152].

Unlike search-based model finders, SAT-based tools [25, 117] cannot exploit symmetries dynamically. Instead, they employ a static symmetry breaking technique that involves augmenting the SAT encoding of a given problem with symmetry breaking predicates. These are constraints that hold for at least one binding in each isomorphism class in the problem's state space. The lex-leader predicate [27], for example, is true only of the lexicographically smallest binding in an equivalence class. It has the form $V \leq l(V)$, where l is a symmetry and V is an ordering of the bits that make up the state space of a given problem. The formula $d_0d_1f_0f_1f_2c_0c_2c_3c_4c_5c_6c_7c_8c_9 \leq d_0d_1f_1f_0f_2c_0c_3c_2c_4c_5c_6c_8c_7c_9$, for instance, is a lex-leader predicate for the symmetry (`f0 f1`) of the toy filesystem (Fig. 2-5). Most SAT-based model finders [25, 117] employ some variant of the lex-leader predicate. Alloy3 and Kodkod use it in addition to predicates [116] that break symmetries on relations with specific properties, such as acyclicity.

Model finders that accept typed formulas [69, 152, 117] use the provided sorts or types to identify symmetries. Those that accept untyped formulas take one of two approaches to symmetry detection. Mace4 [93] works on untyped problems directly,

taking their symmetries to be all permutations of the universe of interpretation. Paradox2.3 [25], on the other hand, first employs a sort inference algorithm to transform an untyped input problem into an equisatisfiable typed problem. It then solves the new problem, taking its symmetries to be all permutations that respect the inferred types. The inference algorithm works as follows. First, it assigns unrelated sorts to all predicate and function symbols that appear in a given set of unsorted clauses. Next, it forces all occurrences of a variable within a clause to have the same sort. Finally, it forces each pair of functions related by the '=' symbol to have the same sort. The result is a sound typing (as defined in [70]) that can be applied to the input problem without affecting its satisfiability.

The process of typing unsorted problems, and then breaking symmetries on the resulting types, can lead to dramatic improvements in model finding performance. To see why, consider the problem P specified by the clause "$\forall x, f(x) \neq g(x)$," where f and g are uninterpreted functions with the universal typing $f, g : U \rightarrow U$. In a two atom universe $\{a_0, a_1\}$, P has 2^8 possible bindings since four bits are needed to represent f and four to represent g. These bindings are partitioned into equivalence classes by two symmetries: $(a_0\ a_1)$ and the identity. Now consider the problem P' specified by the same clause as P but using the typing $f : A \rightarrow B$ and $g : A \rightarrow B$ inferred by Paradox2.3. In the universe $\{a_0, a_1, b_0, b_1\}$ where each of the types has 2 atoms, P' has 2^8 bindings, like P, and it has a model only if P does. The key difference between the two is that P' has twice as many symmetries as P, i.e. $(a_0\ a_1), (b_0\ b_1), (a_0\ a_1)(b_0\ b_1)$, and the identity. Since more symmetries lead to fewer isomorphism classes, a model finder needs to examine half as many (representative) bindings to solve P' as it does to solve P.

3.4.2 Symmetries in constraint programming

As in traditional model finding, symmetry breaking methods used in constraint programming fall into two categories: static and dynamic. Static approaches include a number of predicates based on the lex-leader method (e.g. [47, 50, 104]). Dynamic approaches include Symmetry Breaking During Search [10, 11], Symmetry Breaking

via Dominance Detection [45, 48], and Symmetry Breaking Using Stabilizers [103]. Gent et al. [55] provide a detailed overview of these techniques, both static and dynamic.

The research on the identification of symmetries in constraint solving problems (CSPs) has focused mostly on methods [54, 64] for enabling the user to specify commonly occurring symmetries directly, such as "all rows of the matrix m can be interchanged." Detecting the symmetries of a CSP automatically is tricky for two reasons. First, there are nearly a dozen different definitions of a symmetry for a CSP [26, 55]. Of these, the most liberal definition is that of a *solution symmetry* [26], defined as a permutation of ⟨variable, value⟩ pairs that preserves the set of solutions. The symmetries of model finding and extension problems, in contrast, are defined strictly as permutations of values. Second, there is no practical criterion for identifying the full symmetry group of a large CSP with non-binary constraints [26, 55].

There are, however, two automatic techniques [105, 107] for capturing a subset of the solution symmetries of a CSP. Both are closely related and work by applying a graph automorphism detector to the microstructure graph [26] of each constraint in a CSP. For example, consider the CSP "$x = y$," where x and y are integers in the range $[1\,..\,3]$. The microstructure graph of this problem is given in Fig. 3-6. The nodes A1 through A3 represent all possible assignments of the values 1 through 3 to the variables x and y that satisfy the constraint $x = y$. The graph has 48 symmetries, which capture all permutations of assignments from variables to values that preserve the set of solutions; e.g. $(x{=}2\ x{=}3)(y{=}2\ y{=}3)$. A potential bottleneck [55] for these approaches is their reliance on graph automorphism detection, since the size of a microstructure graph grows quickly in the presence of non-binary constraints.

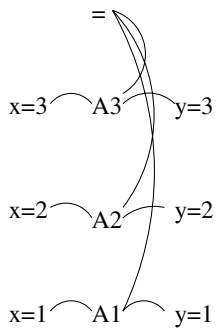


Figure 3-6: Microstructure of a CSP, as given by Puget [105].

Chapter 4

Finding Minimal Cores

A specification that has no models in a given universe is said to be *unsatisfiable* in that universe. Most model finders offer no feedback on unsatisfiable specifications apart from reporting that no models were found. But many applications need to know the cause of a specification's unsatisfiability, either to take corrective action (in the case of declarative configuration [133]) or to check that no models exist for the right reasons (in the case of bounded verification [31, 21]). In bounded verification [31, 21], for example, unsatisfiability is a necessary but insufficient indicator of success. If the verification formula $s_1 \wedge \ldots \wedge s_n \wedge \neg p$ has a model, such a model is a witness that the system described by $s_1 \wedge \ldots \wedge s_n$ violates the property p. But a lack of models is not a guarantee of success. If the formula is unsatisfiable because the system description $s_1 \wedge \ldots \wedge s_n$ is overconstrained, or because p is a tautology, then the analysis is considered to have failed due to a 'vacuity' error.

A cause of unsatisfiability of a given specification is called an *unsatisfiable core*. It is expressed as a subset of the specification's constraints that is itself unsatisfiable. Every such subset includes one or more *critical constraints* that cannot be removed without making the remainder of the core satisfiable. Non-critical constraints, if any, are irrelevant to unsatisfiability and generally decrease a core's utility both for diagnosing faulty configurations [133] and for checking the results of a bounded analysis [129]. Cores that include only critical constraints are said to be *minimal*.

This chapter presents a new technique for extracting minimal unsatisfiable cores,

called *recycling core extraction* (RCE). The technique was developed in the context of SAT-based model finding and relies on the ability of modern SAT solvers to produce checkable proofs of unsatisfiability. It is, however, more generally applicable. The rest of this chapter therefore describes RCE in broader terms, as a technique for extracting minimal cores of specifications that are translatable to the input logic of some resolution engine (e.g. a SAT solver or a resolution-based theorem prover). The technique is shown to be sound, minimal, and effective in practice. Its effectiveness is evaluated on a set of benchmarks of varying difficulties, where the difficulty of finding a minimal core is given as a function of its size, the size of the overall problem, and the time taken (by a resolution engine) to prove that the overall problem is unsatisfiable.

4.1 A small example

The first chapter demonstrated the use of minimal cores for diagnosis and repair of a faulty Sudoku configuration. This section shows how they can be used to expose three kinds of coverage errors that commonly arise in bounded verification: overconstraining the system description, so that legitimate behaviors are excluded; checking the system against a weak property, so that bad behaviors are accepted; and setting the analysis bounds too small, so that the model finder does not examine a sufficiently large space of possibilities. The last problem is specific to bounded verification and is not suffered by analyses based on theorem provers. These analyses are not immune to the first two problems, however, so the techniques presented here will work for them too.

4.1.1 A toy list specification

As an example, consider the toy formalization of LISP-style lists given in Fig. 4-1. There are two distinct kinds of lists (lines 10-11): cons cells and nil lists. Nil lists do not have any structure. A cons cell, on the other hand, is defined by two functions, car and cdr. The car function maps each cons cell to the thing stored in that cell (lines 12-13). The cdr function maps it to the rest of the list (lines 14-15). Every list ends with a Nil list (line 16).

The equivTo (line 17) and prefixes (line 19) relations define what it means for two lists to be equivalent to each other and what it means for one list to be a prefix of another. The equivalence definition is simple. Two lists are equivalent only if they store the same thing and have equivalent tails (line 18). The prefix definition consists of three constraints. First, a nil list is a prefix of every list (line 20); second, a cons cell is never a prefix of a nil list (line 21); and third, one cons cell is a prefix of another only if the two store the same thing and the tail of the first cell is a prefix of the tail of the second cell (line 22).

The universe in which the toy list specification is analyzed generally depends on the property being checked. The universe and bounds declarations in Fig. 4-1 show a sample analysis set-up involving three lists and three things. All relations have empty lower bounds, and their upper bounds reflect their types. List, Nil and Cons all range over the list atoms (lines 2-4). The upper bound on Thing consists of the "thing" atoms (line 5). The binary relation car is bound above by the cross product of lists and things (line 6). The remaining relations all range over the cross product of list atoms with themselves (line 7-9).

4.1.2 Sample analyses

Bounded verification of a system against a given property involves checking that the conjunction of the system constraints and the negation of the property is unsatisfiable. The check is performed within a user-provided finite *scope*, i.e. a finite universe and a corresponding set of bounds on free variables. If the property is invalid in the given scope, a model finder will produce a *counterexample*. This is a model of the verification formula, and, as such, it represents a behavior of the system that violates the property. The absence of counterexamples, on the other hand, can indicate any of the following:

1. the system satisfies the property in the way that the user intended;
2. the property holds vacuously;
3. the property holds but it is weaker than what was intended; and, finally,

```
1  {l0, l1, l2, t0, t1, t2}

2  List      :1 [{}, {⟨l0⟩,⟨l1⟩,⟨l2⟩}]
3  Nil       :1 [{}, {⟨l0⟩,⟨l1⟩,⟨l2⟩}]
4  Cons      :1 [{}, {⟨l0⟩,⟨l1⟩,⟨l2⟩}]
5  Thing     :1 [{}, {⟨t0⟩,⟨t1⟩,⟨t2⟩}]
6  car       :2 [{}, {⟨l0,t0⟩, ⟨l0,t1⟩, ⟨l0,t2⟩, ⟨l1,t0⟩, ⟨l1,t1⟩, ⟨l1,t2⟩, ⟨l2,t0⟩, ⟨l2,t1⟩, ⟨l2,t2⟩}]
7  cdr       :2 [{}, {⟨l0,l0⟩, ⟨l0,l1⟩, ⟨l0,l2⟩, ⟨l1,l0⟩, ⟨l1,l1⟩, ⟨l1,l2⟩, ⟨l2,l0⟩, ⟨l2,l1⟩, ⟨l2,l2⟩}]
8  equivTo   :2 [{}, {⟨l0,l0⟩, ⟨l0,l1⟩, ⟨l0,l2⟩, ⟨l1,l0⟩, ⟨l1,l1⟩, ⟨l1,l2⟩, ⟨l2,l0⟩, ⟨l2,l1⟩, ⟨l2,l2⟩}]
9  prefixes  :2 [{}, {⟨l0,l0⟩, ⟨l0,l1⟩, ⟨l0,l2⟩, ⟨l1,l0⟩, ⟨l1,l1⟩, ⟨l1,l2⟩, ⟨l2,l0⟩, ⟨l2,l1⟩, ⟨l2,l2⟩}]

10 List = Cons ∪ Nil
11 no Cons ∩ Nil

12 car ⊆ Cons→Thing
13 ∀ a: Cons | one a.car

14 cdr ⊆ Cons→List
15 ∀ a: Cons | one a.cdr
16 ∀ a: List | ∃ e: Nil | e ⊆ a.^cdr

17 equivTo ⊆ List→List
18 ∀ a, b: List | (a ⊆ b.equivTo) ⇔ (a.car = b.car ∧ a.cdr.equivTo = b.cdr.equivTo)

19 prefixes ⊆ List→List
20 ∀ e: Nil, a: List | e ⊆ a.prefixes
21 ∀ e: Nil, a: Cons | ¬ (a ⊆ e.prefixes)
22 ∀ a, b: Cons | (a ⊆ b.prefixes) ⇔ (a.car = b.car ∧ a.cdr in b.cdr.prefixes)
```

Figure 4-1: A toy list specification, adapted from a sample distributed with Alloy4 [21].

4. the system satisfies the property in the given scope but the scope is too small to reveal a counterexample, if one exists.

In practice, each of these cases leads to an identifiable pattern of minimal cores, described below.

Example 1: A vacuity error

A vacuity error happens when the property being checked is a tautology or when a part of the system description is overconstrained, admitting either no solutions or just the uninteresting ones. The toy list specification in Fig. 4-1, for example, is overconstrained. The problem is revealed by checking, in the universe with 3 atoms per type, that two lists are equivalent only if they have identical prefixes:

$$\forall \text{ a, b: List} \mid (\text{a} \subseteq \text{b.prefixes} \land \text{b} \subseteq \text{a.prefixes}) \Leftrightarrow \text{a} \subseteq \text{b.equivTo}$$

Kodkod confirms that the property has no counterexamples in the given scope, and produces the following minimal core:

11 **no** Cons ∩ Nil

14 cdr ⊆ Cons→List

16 ∀ a: List | ∃ e: Nil | e ⊆ a.^cdr

23 ¬(∀ a, b: List | (a ⊆ b.prefixes ∧ b ⊆ a.prefixes) ⇔ a ⊆ b.equivTo)

Increasing the analysis scope to a universe and bounds with 5, 10 or 20 atoms of each type yields the same result. Neither the definition of prefixes nor equivalence is needed to prove the property. Why?

Examining the core more closely reveals that the constraint on line 16 is too strong. It says that, for every list, traversing the cdr pointer one or more times leads to some nil list. But nil lists have no tails (lines 11 and 14). Consequently, there are no lists, nil or otherwise, that satisfy the toy list description. That is, the specification is consistent, but it only admits trivial models in which the List relation is empty. A revised definition of what it means for a list to be nil-terminated is given below:

16 ∀ a: List | ∃ e: Nil | e ⊆ a.*cdr

It states that, for every list, traversing the cdr pointer zero or more times leads to some nil list.

Example 2: A successful analysis

Checking a valid system description against a strong property results in minimal cores that include the negation of the property and most of the system constraints. When the revised toy list is checked against the previous assertion, Kodkod, once again, finds no counterexamples in the scope of 3 atoms per type. But the extracted core now includes the negation of the property and the constraints on lines 12, 14, 16, 18, and 20-22. When the scope is increased to 5 atoms per type, this core grows to include the constraint on line 17. Further increases in scope (e.g. up to 9 atoms per type) have no effect on the contents of the extracted core, suggesting that the toy list satisfies the property as intended.

Example 3: A weak property

A valid assertion that exercises only a small portion of a system is said to be *weak*. By themselves, weak assertions are not harmful, but they can be misleading. If the system analyst believes that a weak assertion covers all or most of the system, he can miss real errors in the parts of the system that are not exercised. For example, the following property is supposed to exercise both equivalence and prefix definitions. It states that a list which is equivalent to all of its prefixes has a nil tail:

$$\forall \text{ a: List} \mid (\text{a.prefixes} = \text{a.equivTo}) \Rightarrow \text{a.cdr} \subseteq \text{Nil}$$

Kodkod verifies the property against the revised specification in the scope of 3 atoms per type, producing the following minimal core:

11 **no** Cons ∩ Nil

14 cdr ⊆ Cons→List

16 ∀ a: List | ∃ e: Nil | e ⊆ a.*cdr

18 ∀ a, b: List | (a ⊆ b.equivTo) ⇔ (a.car = b.car ∧ a.cdr.equivTo = b.cdr.equivTo)

20 ∀ e: Nil, a: List | e ⊆ a.prefixes

23 ¬(∀ a: List | (a.prefixes = a.equivTo) ⇒ a.cdr ⊆ Nil)

The tool's output remains the same as the scope is increased to 5, 10 and 20 atoms per type. This is problematic because it shows that the property exercises only one constraint in the prefix definition, when the intention was to exercise the definition in its entirety. In other words, the property was intended to fail if any part of the prefix definition was wrong, but it will, in fact, hold even if the constraints on lines 21 and 22 are replaced with the constant "true."

Example 4: An insufficient scope

Many properties require a certain minimum scope for a bounded analysis to be meaningful. If a property applies only to lists of length 3 or more, establishing that it holds for lists with fewer than 3 elements is not useful. But determining a minimum useful scope can be tricky. Consider, for example, the following property, which states that the prefix relation is transitive over cons cells:

$$\forall \text{ a, b, c: Cons} \mid (\text{a} \subseteq \text{b.prefixes} \wedge \text{b} \subseteq \text{c.prefixes}) \Rightarrow \text{a} \subseteq \text{c.prefixes}$$

Since the property includes three quantified variables, it seems that 3 atoms per type should be the minimum useful scope for checking the property against the revised list specification. The corresponding core, however, shows that either this scope or the property is insufficient to cover the prefix definition:

15 $\forall$ a: Cons | **one** a.cdr

16 $\forall$ a: List | $\exists$ e: Nil | e $\subseteq$ a.*cdr

19 prefixes $\subseteq$ List$\rightarrow$List

21 $\forall$ e: Nil, a: Cons | $\neg$ (a $\subseteq$ e.prefixes)

22 $\forall$ a, b: Cons | (a $\subseteq$ b.prefixes) $\Leftrightarrow$ (a.car = b.car $\wedge$ a.cdr in b.cdr.prefixes)

23 $\neg(\forall$ a, b, c: Cons | (a $\subseteq$ b.prefixes $\wedge$ b $\subseteq$ c.prefixes) $\Rightarrow$ a $\subseteq$ c.prefixes)

When the analysis is repeated in the scope of 4 atoms per type, the core increases to include the rest of the prefix definition (line 20), demonstrating that the property is strong enough but that the scope of 3 is too small. This makes sense. Because every list must be nil-terminated, at least four lists are needed to check the property against a scenario in which the variables `a`, `b` and `c` are bound to distinct cons cells. Moreover, the core remains the same as the scope is increased (up to 8 atoms per type), suggesting that 4 atoms per type is indeed the minimum useful scope.

4.2 Core extraction with a resolution engine

The core extraction facility featured in the previous section implements a new technique for finding minimal cores, called *recycling core extraction* (RCE). RCE relies on the ability of modern SAT solvers to produce checkable proofs of unsatisfiability, expressed as DAGs. These proofs encode inferences made by the solver that lead from a subset, or a core, of the input clauses to a contradiction. Exploiting SAT proofs, however, is challenging because their cores are not guaranteed to be minimal or even small. Moreover, mapping a small boolean core back to specification constraints—a technique called *one-step core extraction* (OCE) [118]—may still yield

a large specification-level core. Shlyakhter [117], for example, found that reducing the size of the boolean core prior to mapping it back had little to no effect on the size of the resulting specification-level core.

RCE is based on two key ideas. The first idea is to minimize the core at the specification rather than the boolean level. That is, the initial boolean core is mapped to a specification level core, which is then pruned by removing candidate constraints and testing the remainder for satisfiability. The second idea is to use the boolean proof, and the mapping between the specification constraints and the translation clauses, to identify the inferences made by the solver that are still valid when a specification-level constraint is removed. By adding these inferences (expressed as clauses) to the translation of a candidate core, RCE allows the solver to reuse the work from previous invocations.

Although RCE was developed in the context of SAT-based model finding for relational logic, the technique is more widely applicable. This section therefore presents the RCE algorithm, and the proofs of its soundness and minimality, in the context of an abstract *resolution-based analysis framework*. The framework captures the general structure of SAT solvers [43, 56, 87], SAT-based model finders [25, 117, 131], and resolution-based theorem provers [74, 108, 145]. Abstractly, they all work by translating a declarative specification to a clausal logic and applying a resolution procedure to the resulting clauses.[1] The framework provides a high-level formalization of the key properties of these specifications, translations, and resolution engines.

4.2.1 Resolution-based analysis

Resolution [109] is a complete technique for proving that a set of clauses is unsatisfiable. It involves the application of the resolution rule (Fig. 4-2) to a set of clauses until the *empty clause* is derived, indicating unsatisfiability, or until no more clauses can be derived, indicating satisfiability. This process is guaranteed to terminate on an unsatisfiable input with a proof of its unsatisfiability, which takes the form of a *resolution refutation* graph (Def. 4.1). Its behavior on satisfiable inputs depends on

[1] SAT solvers can be viewed as using the identity as their translation function.

the decidability of the underlying logic. SAT solvers, for example, are *resolution engines* (Def. 4.2) for propositional logic, which is decidable, and they will terminate on every set of propositional clauses given enough time. Theorem provers, on the other hand, will run forever on some inputs since first order logic is undecidable.

SAT solvers are unique among resolution-based tools in that their input language is a clausal logic. Other tools that are based on resolution, such as theorem provers and model finders, accept syntactically richer languages, which they translate to first order or propositional clauses. The details of these languages and their translations differ from one tool to another, but they all share two basic properties. First, the input of each tool is a set of declarative constraints, or a *specification* (Def. 4.3), which is tested for unsatisfiability in some implicit or explicit universe of values. Second, translations employed by resolution-based tools are *regular* (Def. 4.4). That is, a specification and its translation are either both satisfiable or both unsatisfiable; the translation of a specification is the union of the clauses produced for each constraint; and the clauses produced for a given constraint in the context of different specifications are the same, up to a renaming of identifiers (i.e. variable, constant, predicate and function names).

Figure 4-3 shows an example of a resolution-based analysis of the specification $(a \Leftrightarrow b) \wedge (b \Leftrightarrow c) \wedge \neg(a \Rightarrow c)$. The specification is encoded in non-clausal propositional logic, and the translation $\mathcal{T}$ applies standard inference rules to transform this language to clauses. It is easy to see that both the specification and the translation satisfy the definitions 4.3 and 4.4. Because the specification is unsatisfiable, feeding its translation to the resolution engine $\mathcal{E}$ produces a proof of unsatisfiability in the form of a resolution refutation graph. The nodes of the graph are the translation clauses and the *resolvent* clauses inferred by the engine. The edges denote the application of the resolution rule. For example, the edges incident to $\neg b$ indicate that $\neg b$ is the result of applying the resolution rule to the clauses $\neg b \vee c$ and $\neg c$. The translation clauses that are connected to the empty clause $c_\emptyset$ form an unsatisfiable core of the translation.

Definition 4.1 (Resolution refutation)**.** *Let C and R be sets of clauses such that $R \backslash C$*

$$\frac{(a_1 \vee \ldots \vee a_i \vee \ldots \vee a_n),\ (b_1 \vee \ldots \vee b_j \vee \ldots \vee b_m),\ a_i = \neg b_j}{a_1 \vee \ldots \vee a_{i-1} \vee a_{i+1} \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_{j-1} \vee b_{j+1} \ldots \vee b_m}$$

Figure 4-2: The resolution rule for propositional logic.

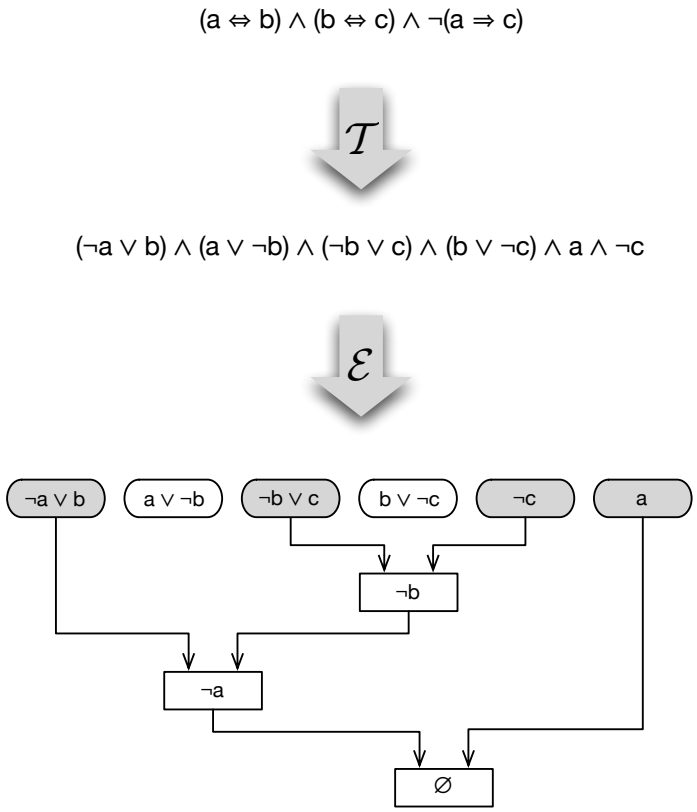


Figure 4-3: Resolution-based analysis of $(a = b) \wedge (b = c) \wedge \neg(a \Rightarrow c)$. The specification is first translated to a set of propositional clauses, using a regular translation $\mathcal{T}$. The resulting clauses are then fed to a resolution engine $\mathcal{E}$, which produces a proof of their unsatisfiability. The unsatisfiable core of the proof is shown in gray. The $\emptyset$ square designates the empty clause.

contains the empty clause, denoted by $c_\emptyset$. A directed acyclic graph $G = (C, R, E)$ is a resolution refutation of C iff

1. *each $r \in R$ is the result of resolving some clauses $x_1, \ldots, x_n \in C \cup R$;*

2. *the only edges in E are $\langle x_1, r\rangle, \ldots, \langle x_n, r\rangle$ for each $r \in R$; and,*

3. *$c_\emptyset$ is the sink of G.*

The sources of G are denoted by $\{c \in C \mid c_\emptyset \in E^*(\!|c|\!)\}$, where E^* is the reflexive transitive closure of E and $E^*(\!|c|\!)$ is the relational image of c under E^*. These form an unsatisfiable core of C.

Definition 4.2 (Resolution engine)**.** *A resolution engine $\mathcal{E} : \mathbb{P}(C) \rightharpoonup G$ is a partial function that maps each unsatisfiable clause set C to a resolution refutation graph $G = (C, R, E)$. The remaining clause sets on which $\mathcal{E}$ is defined are taken to resolution graphs that do not include $c_\emptyset$, indicating satisfiability.*

Definition 4.3 (Declarative specification)**.** *A declarative specification S is a set of constraints $\{s_1, \ldots, s_n\}$, interpreted in a universe of values U.*[2] *That is, all identifiers that appear in S denote either values from U or variables that range over U. The constraints $s_i \in S$ represent boolean functions whose conjunction is the meaning of S as a whole.*

Definition 4.4 (Regular translation)**.** *A procedure $\mathcal{T} : \mathcal{L} \to \mathbb{P}(C)$ is a regular translation of specifications in the language $\mathcal{L}$ to the clausal logic $\mathbb{P}(C)$ iff*

1. *a specification $S \in \mathcal{L}$ is unsatisfiable whenever $\mathcal{T}(S)$ is unsatisfiable;*

2. *the translation of a specification $S \in \mathcal{L}$ is the union of the translations of its constraints: $\mathcal{T}(S) = \mathcal{T}_S(roots(S)) = \cup_{s \in roots(S)} \mathcal{T}_S(s)$, where $\mathcal{T}_S(s)$ is the translation of the constraint s in the context of the specification S; and,*

[2]The term 'universe of values' refers to all possible values that the variables in a specification can take. For example, in the case of bounded relational logic, the 'universe of values,' as opposed to the 'universe of atoms,' refers to the entire state space of a specification.

3. *the translation of the constraints $S' \subseteq S$ is context independent up to a renaming: $\mathcal{T}_S(S') = r(\mathcal{T}(S'))$ for some bijection r from the identifiers that occur in $\mathcal{T}_S(S')$ to those that occur in $\mathcal{T}(S')$, lifted to clauses and sets of clauses in the obvious way.*

4.2.2 Recycling core extraction

Recycling core extraction is closely related to three simpler techniques: *naive*, *one-step*, and *simple core extraction* (NCE, OCE, and SCE). NCE and OCE have been described in previous work, the former for propositional logic [32] and linear programs [23] and the latter for declarative languages reducible to SAT [118]. SCE is a new technique that combines NCE and OCE, and RCE is a refinement of SCE. The pseudo-code for all four algorithms is shown in Fig. 4-4.

NCE (Fig. 4-4a) is the most basic method for extracting minimal cores in that it uses the resolution engine solely for satisfiability testing. The algorithm starts with an initial core K that contains the entire specification (line 1). The initial core is then pruned, one constraint at a time, by discarding all constraints u for which a regular translation of $K \setminus \{u\}$ is unsatisfiable (lines 3-8). This pruning step is sound since the regularity of the translation guarantees that $\mathcal{T}(K \setminus \{u\})$ and $K \setminus \{u\}$ are equisatisfiable. In the end, K contains a minimal core of S.

Because it calls the resolution procedure once for each constraint, NCE tends to be impractical for large specifications with small, hard cores. Shlyakhter et al. [118] addressed this problem with OCE (Fig. 4-4b), which sacrifices minimality for scalability. OCE simply returns all constraints in S whose translation contributes clauses to the unsatisfiable core of $\mathcal{E}(\mathcal{T}(S))$. The set of constraints computed in this way is an unsatisfiable core of S (§4.2.3), but it is usually not minimal.

SCE (Fig. 4-4c) combines the pruning loop of NCE with the extraction step of OCE. In particular, SCE is NCE with the following modifications: initialize K with a core of S instead of S (line 2), and reduce K to a core of $K \setminus \{u\}$ instead of $K \setminus \{u\}$ in the iterative step (line 9). These modifications eliminate unnecessary calls to the resolution engine without affecting either the soundness or the minimality of

```
NCE(S: 𝓛, 𝒯: 𝓛 → ℙ(C), ℰ: ℙ(C) ⇀ G)
 1  K ← S
 2  M ← {}
 3  while K ⊈ M do
 4    u ← pick(K \ M)
 5    M ← M ∪ {u}
 6    (C, R, E) ← ℰ(𝒯(K \ {u}))
 7    if c∅ ∈ R then
 8      K ← K \ {u}
 9  return K
```

(a) Naive core extraction

```
OCE(S: 𝓛, 𝒯: 𝓛 → ℙ(C), ℰ: ℙ(C) ⇀ G)
 1  (C, R, E) ← ℰ(𝒯(S))
 2  K ← {s∈S | c∅ ∈ E*(|𝒯_S(s)|)}
 3  return K
```

(b) One-step core extraction

```
SCE(S: 𝓛, 𝒯: 𝓛 → ℙ(C), ℰ: ℙ(C) ⇀ G)
 1  (C, R, E) ← ℰ(𝒯(S))
 2  K ← {s∈S | c∅ ∈ E*(|𝒯_S(s)|)}
 3  M ← {}
 4  while K ⊈ M do
 5    u ← pick(K \ M)
 6    M ← M ∪ {u}
 7    (C, R, E) ← ℰ(𝒯_S(K \ {u}))
 8    if c∅ ∈ R then
 9      K ← {s ∈ K\{u} | c∅ ∈ E*(|𝒯_S(s)|)}
10  return K
```

(c) Simple core extraction

```
RCE(S: 𝓛, 𝒯: 𝓛 → ℙ(C), ℰ: ℙ(C) ⇀ G)
 1  (C, R, E) ← ℰ(𝒯(S))
 2  K ← {s∈S | c∅ ∈ E*(|𝒯_S(s)|)}
 3  M ← {}
 4  while K ⊈ M do
 5    u ← pick(K \ M)
 6    M ← M ∪ {u}
 7    C' ← 𝒯_S(K \ {u})
 8    R' ← R \ E*(|C \ C'|)
 9    if c∅ ∈ R' then
10      K ← K \ {u}
11    else
12      (C'', R'', E'') ← ℰ(C' ∪ R')
13      if c∅ ∈ R'' then
14        (C, R, E) ← (C', R'∪R'', E''∪(E▷R'))
15        K ← {s : K\{u} | c∅ ∈ E*(|𝒯_S(s)|)}
16  return K
```

(d) Recycling core extraction

Figure 4-4: Core extraction algorithms. S is an unsatisfiable specification, $\mathcal{T}$ is a regular translation, and $\mathcal{E}$ is a resolution engine. Star (*) means reflexive transitive closure, $r(\!|X|\!)$ is the relational image of X under r, and $\triangleright$ is range restriction.

the resulting core (§4.2.3). But they still do not eliminate all unnecessary work. By applying $\mathcal{E}$ solely to $\mathcal{T}_S(K \setminus \{u\})$ on line 7, SCE forces the engine to re-learn some or all the clauses that it has already learned about $\mathcal{T}_S(K \setminus \{u\})$ while refuting $\mathcal{T}_S(K)$.

RCE (Fig. 4-4d) extends SCE with a mechanism for reusing learned clauses. Line 7 of RCE constructs $\mathcal{T}(K \setminus \{u\})$ from the already computed translations of the constraints in S, and line 8 collects the resolvents that $\mathcal{E}$ had already learned about $\mathcal{T}(K \setminus \{u\})$. These are simply all resolvents reachable from $\mathcal{T}(K \setminus \{u\})$ but not from the other clauses previously fed to $\mathcal{E}$. If they include the empty clause $c_\emptyset$, u is discarded (line 10) because there must be some other constraint in $K \setminus \{u\}$ whose translation contributes the same or a larger set of clauses to the core of C as u. Otherwise, line 12 applies $\mathcal{E}$ to $\mathcal{T}(K \setminus \{u\})$ and its resolvents. If the result is a refutation, the invalidity of K can be proved without u. The core of (C'', R'', E''), however, does not necessarily correspond to a core of S because the former may include resolvents for $\mathcal{T}(K \setminus \{u\})$. Consequently, lines 14-15 fix the proof and set K to the corresponding core, which excludes at least u.

4.2.3 Correctness and minimality of RCE

The soundness and minimality of SCE and RCE are proved below. Theorem 4.1 establishes that the step they share with OCE is sound. This is then used in the proof of Thm. 4.2 to show that, if it terminates, RCE produces a minimal unsatisfiable core of its input. It is easy to see that RCE terminates if its input engine $\mathcal{E}$ terminates on each invocation. Since RCE reduces to SCE when R' is set to the empty set on line 8, Thm. 4.2 also establishes the soundness and minimality of SCE.

Theorem 4.1 (Soundness of OCE)**.** *Let $G = (C, R, E)$ be a resolution refutation for $C = \mathcal{T}(S)$, a regular translation of the unsatisfiable specification S. Then, $K = \{s \in S \mid c_\emptyset \in E^*(\!|\mathcal{T}_S(s)|\!)\}$ is an unsatisfiable core of S.*

Proof. It follows from the definition of K that $K \subseteq S$ and $\{c \in C \mid c_\emptyset \in E^*(\!|c|\!)\} \subseteq \mathcal{T}_S(K)$. By Def. 4.1, the graph $G_K = (\mathcal{T}_S(K), R, E)$ is a resolution refutation of $\mathcal{T}_S(K)$. Because $\mathcal{T}$ is regular, $\mathcal{T}(K) = r(\mathcal{T}_S(K))$ for some renaming r. Let $r(G_K)$

denote G_K with r applied to all of its vertices. By Def. 4.1, $r(G_K)$ is a resolution refutation of $r(\mathcal{T}_S(K)) = \mathcal{T}(K)$. This completes the proof, since $\mathcal{T}(K)$ and K are equisatisfiable due to the regularity of $\mathcal{T}$. □

Theorem 4.2 (Soundness and minimality of RCE)**.** *If it terminates, RCE(S, $\mathcal{T}$, $\mathcal{E}$) returns a minimal unsatisfiable core of S, where S is an unsatisfiable specification, $\mathcal{T}$ is a regular translation, and $\mathcal{E}$ a resolution engine.*

Proof. To establish that K is unsatisfiable on line 16, it suffices to show that K_i is unsatisfiable after the i^{th} execution of lines 10 and 15. The soundness of the base case (line 2) follows from Thm. 4.1. Suppose that K_{i-1} is unsatisfiable and that the condition on line 9 is true. Then, by Def. 4.1 and construction of R', $(\mathcal{T}_S(K_{i-1} \setminus \{u\}), R', E)$ is a refutation of $\mathcal{T}(K_{i-1} \setminus \{u\})$, which means that $K_{i-1} \setminus \{u\}$ is unsatisfiable. Setting K_i to $K_{i-1} \setminus \{u\}$ on line 10 is therefore justified. Now suppose that the condition on line 9 is false. For line 15 to execute, the condition on line 13 must hold. If it does, line 14 executes first, establishing (C_i, R_i, E_i) as a resolution refutation for $\mathcal{T}(K_{i-1} \setminus \{u\})$ (Defs. 4.1 and 4.4, line 7). This and Thm. 4.1 ensure that the constraints assigned to K_i on line 15 are unsatisfiable.

Suppose that K is sound but not minimal on line 16. Then, there is a constraint $s \in K$ such that $K \setminus \{s\}$ is unsatisfiable. Lines 4 and 6 ensure that s is picked on line 5 during some iteration i of the loop. Because $K \subseteq K_{i-1}$ and $K \setminus \{s\}$ is unsatisfiable, it must be the case that $K_{i-1} \setminus \{s\}$ is also unsatisfiable. Consequently, either the condition on line 9 or the one on line 13 is true in the i^{th} iteration, causing s to be removed from K_i. But this is impossible since $K \subseteq K_i$. □

4.3 Experimental results

NCE, OCE, SCE and RCE have all been implemented in Kodkod, with MiniSat [43] as the resolution engine and the algorithm from Chapter 2 as the regular translation from bounded relational logic to propositional clauses. Table 4.1 shows the results of applying these implementations to a subset of the benchmarks introduced in Chapter

	problem	size	scope	variables	clauses	transl (sec)	solve (sec)	init core	min core	OCE (sec)	NCE (sec)	SCE (sec)	RCE (sec)
Alloy benchmarks	AWD.A241	10	10	38457	88994	1	167	6	1	3	t/o	t/o	125
	AWD.AbIgnore	10	10	30819	60071	0	19	5	1	0	168	59	42
	AWD.AbTransfer	10	7	15247	33217	0	23	7	4	1	112	40	59
	Dijkstra	9	25	1606605	7725002	165	84	9	6	9	623	166	81
	Hotel	59	5	22407	55793	0	0	53	31	0	30	20	14
	Lists.Empties	12	60	2547216	7150594	72	12	7	6	7	351	115	146
	Lists.Reflexive	12	14	34914	91393	1	23	10	5	2	133	88	97
	Lists.Symmetric	12	8	7274	17836	0	27	12	7	2	173	173	149
	RingElection	10	8	59447	187381	1	48	9	9	2	55	6	7
	Trees	4	7	407396	349384	9	97	4	4	0	7	7	6
TPTP benchmarks	ALG212	6	7	1072200	1026997	7	69	6	5	1	101	101	107
	COM008	14	9	6154	9845	0	1	14	10	0	271	274	301
	GEO091	26	10	106325	203299	9	117	24	7	2	2486	1575	176
	GEO092	26	8	48496	91281	3	7	24	7	0	191	102	41
	GEO115	27	9	108996	188776	5	71	24	7	2	1657	1023	116
	GEO158	26	8	46648	88234	3	38	25	7	1	404	242	86
	GEO159	28	8	87214	195200	9	56	24	7	1	1060	197	70
	LAT258	27	7	205621	336912	2	11	26	20	0	303	258	191
	MED007	41	35	130777	265702	2	84	24	7	0	t/o	t/o	71
	MED009	41	35	130777	265703	2	73	24	7	0	t/o	t/o	89
	NUM374	14	3	6869	18933	0	0	14	5	0	4	4	4
	SET943	18	5	5333	12541	0	0	14	4	0	109	46	39
	SET948	20	14	339132	863889	5	36	10	6	1	309	204	205
	SET967	20	4	14640	45111	0	0	10	2	0	2599	254	247
	TOP020	14	10	2554114	4262733	21	112	2	2	5	1008	8	8

Table 4.1: Evaluation of minimal core extractors. The notation "t/o" means that an algorithm was unable to produce a core for the specified problem in the given scope within one hour. Gray shading highlights the best running time among NCE, SCE, and RCE.

2. The chosen problems come from a variety of disciplines (software engineering, medicine, geometry, etc.), include 4 to 59 constraints, and exhibit a wide range of behaviors. In particular, twelve are theorems (i.e. unsatisfiable conjunctions of axioms and negated conjectures); four are believed to be satisfiable but have no known finite models; two are unsatisfiable in small universes and satisfiable in larger ones; and seven have no known finite models.

Each problem p was tested for satisfiability in scopes of increasing sizes until a failing scope $s_{\text{fail}}(p)$ was reached in which either a model was found or all three minimality-guaranteeing algorithms failed to produce a result for that scope within 5 minutes (300 seconds). The algorithms were then re-tested on each problem using a scope of $s_{\text{fail}}(p) - 1$ and a cut-off time of one hour (3600 seconds). All experiments were performed on a 2×3 GHz Dual-Core Intel Xeon with 2 GB of RAM.

The first three columns of Table 4.1 show the name of each problem, the num-

	problem	N-score	NCE / RCE	average speed up
easy	NUM374	-0.19	1.07	
	SET943	0.12	2.80	
	SET967	0.21	10.51	3.27
	Trees	0.58	1.09	
	COM008	0.60	0.90	
medium	Hotel	1.05	2.21	
	RingElection	1.72	7.59	
	ALG212	1.86	0.94	
	Lists.Empties	1.87	2.40	
	LAT258	1.88	1.58	
	Dijkstra	1.94	7.72	
	AWD.AbTransfer	2.10	1.89	3.21
	Lists.Symmetric	2.12	1.16	
	GEO092	2.13	4.68	
	Lists.Reflexive	2.21	1.37	
	AWD.AbIgnore	2.24	4.00	
	SET948	2.70	1.51	
	GEO158	2.85	4.70	
hard	GEO159	3.08	15.06	
	TOP020	3.13	131.46	
	GEO115	3.15	14.34	
	AWD.A241	3.18	28.81	42.14
	GEO091	3.35	14.09	
	MED009	3.40	40.33	
	MED007	3.45	50.90	

(a) RCE versus NCE

	problem	S-score	SCE / RCE	average speed up
easy	NUM374	-0.19	1.08	
	SET967	-0.14	1.03	
	SET943	-0.03	1.19	
	TOP020	0.35	1.00	
	Trees	0.58	1.09	1.07
	COM008	0.60	0.91	
	RingElection	0.64	0.86	
	Hotel	0.95	1.44	
	Lists.Empties	1.11	0.79	
medium	AWD.AbTransfer	1.80	0.67	
	LAT258	1.81	1.35	
	ALG212	1.86	0.94	
	AWD.AbIgnore	1.89	1.40	
	Dijkstra	1.94	2.06	
	Lists.Reflexive	2.06	0.91	3.63
	GEO092	2.08	2.51	
	Lists.Symmetric	2.12	1.16	
	SET948	2.16	1.00	
	GEO158	2.83	2.81	
	AWD.A241	2.92	28.81	
	GEO159	2.99	2.80	
hard	GEO115	3.08	8.86	
	MED009	3.13	40.33	27.25
	MED007	3.13	50.90	
	GEO091	3.30	8.93	

(b) RCE versus SCE

Table 4.2: Evaluation of minimal core extractors based on problem difficulty. If an algorithm timed out on a given problem (after an hour), its running time for that problem is taken to be one hour.

ber of constraints it contains, and the scope in which it was tested. The next two columns contain the number of propositional variables and clauses produced by the translator. The "transl (sec)" and "solve (sec)" columns show the time, in seconds, taken by the translator to generate the problem and the SAT solver to produce the initial refutation. The "initial core" and "min core" columns display the number of constraints in the initial core found by OCE and the minimal core found by the minimality-guaranteeing algorithms. The remaining columns show core extraction time, in seconds, for each algorithm.

On average, RCE outperforms NCE and SCE by a factor of 14 and 7, respectively. These overall averages, however, do not take into account the wide variance in difficulty among the tested problems. A more useful comparison of the minimality-guaranteeing algorithms is given in Table 4.2, where the problems are classified according to their difficulty for NCE (Table 4.2a) and SCE (Table 4.2b). Tables 4.2 and 4.2b show a difficulty rating for each problem; the speed-up of RCE over NCE or SCE on that problem; and the average speed-up of RCE on all problems in a given category.

A core extraction problem is rated as easy for NCE if its *N-score* is less than 1, hard if the score is 3 or more, and moderately hard otherwise. The N-score for a problem is defined as $\log_{10}((s-m)*t+m*t*.01)$, where s is the size of the input specification, m is the size of its minimal core, and t is the time, in seconds, taken by the SAT solver to determine that S is unsatisfiable. The N-score formula predicts how much work NCE has to do to eliminate irrelevant constraints from a specification, by predicting that NCE will take $(s-m)*t$ seconds to prune away the $(s-m)$ irrelevant constraints. The formula allocates only 1 percent of the initial solving time to the testing of a critical constraint because satisfiable formulas are solved quickly in practice. The difficulty of a problem for SCE is computed in a similar way; the S-score of a given problem is $\log_{10}((s'-m)*t+m*t*.01)$, where s' is the size of the initial (one-step) core.

Unsurprisingly, OCE outperforms both SCE and RCE in terms of execution time. However, it generates cores that are on average 2.6 times larger than the corresponding

minimal cores. For 21 out of the 25 tested problems (84%), the OCE core included more than 50% of the original constraints. In contrast, only 8 out of 25 (32%) minimal cores included more than half of the original constraints.

4.4 Related work

The problem of finding unsatisfiable cores has been studied in the context of linear programming [23], propositional satisfiability [32, 57, 79, 85, 97, 102, 153], and finite model finding [118]. The first half of this section presents an overview of that work. The second half summarizes the work on bounded model checking [52, 120] and other applications of SAT [42, 65, 112, 115, 121, 146] that, like RCE, recycle learned clauses for faster solving of closely related SAT instances.

4.4.1 Minimal core extraction

Among the early work on minimal core extraction is Chinneck and Dravnieks' [23] deletion filtering algorithm for linear constraints. The algorithm is similar to NCE: given an infeasible linear program LP, it tests each functional constraint for membership in an Irreducible Infeasible Subset (i.e. minimal unsatisfiable core) by removing it from LP and applying a linear programming solver to the result. If the reduced LP is infeasible, the constraint is permanently removed, otherwise it is kept. The remaining algorithms in [23] are specific to linear programs, and there is no obvious way to adapt them to other domains.

Most of the work on extracting small unsatisfiable cores comes from the SAT community. Several practical algorithms [57, 102, 153] have been proposed for finding small, but not necessarily minimal, cores of propositional formulas. Zhang and Malik's algorithm [153], for example, works by extracting a core from a refutation, feeding it back to the solver, and repeating this process until the size of the extracted core no longer decreases. A few proposed algorithms provide strong optimality guarantees—such as returning the smallest minimal core [85, 97] or all minimal cores [79, 80, 59, 61, 60] of a boolean formula—at the cost of scaling to problems that are orders

of magnitude smaller than those handled by the approximation algorithms. The Complete Resolution Refutation (CRR) algorithm by Dershowitz et al. [32] strikes an attractive balance between scalability and optimality: it finds a single minimal core but scales to large real-world formulas. CRR was one of the inspirations for RCE and is, in fact, an instantiation of RCE for propositional logic, with a SAT solver as a resolution engine and the identity function as the translation procedure.

The work by Shlyakhter et al. [118] is most closely related to the techniques presented in this chapter. It proposes one step core extraction (OCE) for declarative specifications in a language reducible to propositional logic. The algorithm translates an unsatisfiable specification to propositional logic, obtains an unsatisfiable core of the translation from a SAT solver, and returns snippets of the original formula that correspond to the extracted boolean core. Unlike RCE, the OCE method provides no optimality guarantees. In fact, the core it produces is equivalent to the initial core computed by RCE. As discussed in the previous section, this initial core is relatively cheap to find, but it tends to be much larger than a minimal core for most specifications.

4.4.2 Clause recycling

Many applications of SAT [42, 52, 65, 112, 115, 120, 121, 146] employ techniques for reusing learned clauses to speed up solving of related boolean formulas. Silva and Sakallah [121] proposed one of the early clause recycling schemes in the context of automatic test pattern generation (ATPG). The ATPG problem involves solving a SAT instance of the form $C \wedge F \wedge D$, where C is the set of clauses that encodes the correct behavior of a given circuit, F is the description of a faulty version of that circuit, and D states that C and F produce different outputs. A model of an ATPG formula, known as a *test pattern*, describes an input to the circuit being tested that distinguishes between the correct behavior C and the faulty behavior F. Generating test patterns for multiple faults leads to a series of SAT instances of the form $C \wedge F_i \wedge D_i$. Silva and Sakallah's [121] recycling method is based on reusing

conflict clauses[3] implied by C, which they call *pervasive clauses*. This approach does not exploit all valid learned clauses, however. If the faulty circuits F_i and F_{i+i} overlap, clauses learned from $F_i \cap F_{i+i}$ are not reused.

Shtrichman [120] generalized the idea of pervasive clauses and applied it in the context of SAT-based bounded model checking (BMC) [15, 16]. BMC is the problem of checking if a finite state machine m has a trace of length k or less that violates a temporal property p. Shtrichman's [120] work focuses on checking properties that hold in all states of a given state machine (written as $\mathbf{AG}p$ in temporal logic). The check is performed by applying a SAT solver to the formula $I_0 \wedge M_k \wedge P_k$, where I_0 encodes the initial state of m, M_k describes all executions of m of length k, and P_k asserts that p does not hold in at least one of the states described by M_k. A model of the BMC formula represents an execution of m (of length at most k) which violates the property p. If there are no counterexamples of length k, the check is repeated for executions of length $k+1$, leading to a series of overlapping SAT instances. Strichman's method reuses all conflict clauses that are implied by $(I_0 \wedge M_k \wedge P_k) \cap (I_0 \wedge M_{k+1} \wedge P_{k+1})$. In the case of BMC, these are the clauses learned from $I_0 \wedge M_k$.

Several clause-recycling techniques have also been proposed in the context of incremental SAT solving [42, 65, 146]. Hooker [65] proposed the first incremental technique for solving a series of SAT instances of the form $C_0 \subseteq \ldots \subseteq C_k$. The technique involves reusing all conflict clauses implied by C_i when solving C_{i+1}. Whittemore et al. [146] extended Hooker's definition of incremental SAT to allow clause removal. Their solver accepts a series of SAT instances $C_1, \ldots, C_k$, where each C_{i+1} can be obtained from C_i by adding or removing clauses. If clauses are removed from C_i, the solver discards all learned clauses that depend on $C_i \setminus C_{i+1}$ and reuses the rest. Identifying the clauses that need to be discarded is expensive, however, and it requires extra bookkeeping [146] during solving. Eén and Sörensson [42] addressed this problem with an incremental solver that provides limited support for clause removal. In particular, $C_{i+1} \setminus C_i$ can contain any clause, but $C_i \setminus C_{i+1}$ must consist of unit clauses. Since the clauses in $C_i \setminus C_{i+1}$ are treated as decisions by the search procedure, all

[3]Conflict clauses are resolvents that encode root causes of conflicts detected during search [90].

conflict clauses learned from C_i can be reused when solving C_{i+1}.

Chapter 5

Conclusion

Design, implementation, analysis and configuration of software all involve reasoning about relational structures: organizational hierarchies in the problem domain, architectural configurations in the high level design, or graphs and linked list in low level code. Until recently, however, engines for solving relational constraints have had limited applicability. Designed to analyze small, hand-crafted models of software systems, current frameworks perform poorly on large specifications, especially in the presence of partial models.

This thesis presented a framework for solving large relational problems using SAT. The framework integrates two facilities, a *finite model finder* and a *minimal unsatisfiable core extractor*, accessible through a high-level specification language that extends relational logic with a mechanism for specifying partial models (Chapters 1-2). Both facilities are based on new techniques. The model finder uses a new translation to SAT (Chapter 2) and a new algorithm for detecting symmetries in the presence of partial models (Chapter 3), while the core extractor uses a new method for recycling inferences made at the boolean level to speed up core minimization at the specification level (Chapter 4).

The work presented here has been prototyped and evaluated in *Kodkod*, a new relational engine with recent applications to declarative configuration [101, 149], test-case generation [114, 135] and bounded verification [21, 31, 34, 126, 137]. The adoption and use of Kodkod by the wider community has highlighted both strengths and limitations

of its techniques. The former include efficiency on specifications with partial models, low-arity relations and rich type hierarchies; the latter include limited scalability in the presence of high-arity relations, transitive closure and deeply nested quantifiers. This chapter concludes the thesis with a summary of the key contributions behind Kodkod, a discussion of the tool's applicability compared to other solvers, and an outline of directions for future work.

5.1 Discussion

The key contributions of this work are a new relational language that supports partial models; a new translation from relational logic to SAT that uses sparse matrices and auto-compacting circuits; a new algorithm for detecting symmetries in the presence of partial models; and a new algorithm for finding minimal cores with resolution-based engines. The language and the techniques combine into a framework that is well-suited to solving problems in both declarative configuration and analysis. Kodkod is currently the only engine (Table 5.1) that scales both in the presence of symmetries (which is crucial for declarative analysis) and partial models (which is crucial for declarative configuration). It is also the only model finding engine that incorporates a facility for the extraction of minimal cores.

This thesis focused on efficient analysis of specifications that commonly arise in declarative configuration and bounded verification. Such specifications are characterized by the use of typed relations with low arity, set-theoretic operators and join, shallowly nested (universal) quantifiers, and state spaces with partial models or many symmetries. Kodkod outperforms existing tools significantly on specifications with these features (e.g. the Sudoku examples from Chapter 1). It is, however, less effective than some of the existing solvers on other classes of problems.

DarwinFM [13], for example, is the most efficient model finder for specifications comprised of first order clauses over high-arity relations. It works by translating a formula in unsorted first order logic to a set of function-free first order clauses, which are solved using the Darwin [14] decision procedure. A key advantage of this

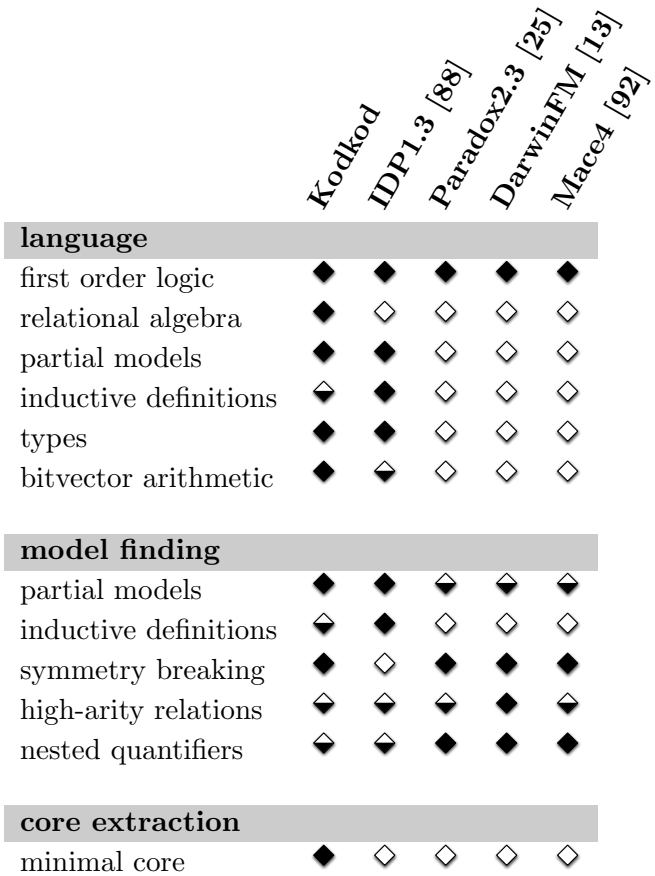


Table 5.1: Features of state-of-the-art model finders. Filled diamonds indicate that a specified feature is fully supported by a solver; half-filled diamonds indicate limited support; and empty diamonds indicate no support.

approach over SAT-based approaches is its space efficiency. In particular, the size of propositional encodings for first order or relational formulas grows exponentially with universe size and relation arity. The size of Darwin encodings, on the other hand, is linear in universe size. As a result, DarwinFM can handle formulas with high-arity relations (e.g. arity 56) on which other model finders run out of resources [125].

IDP1.3 [88] is, like Kodkod, a model extender. Unlike Kodkod, however, it targets specifications in sorted first order logic with inductive definitions (FOL/ID). Given a specification in FOL/ID, the tool translates it to a set of clauses in propositional logic with inductive definitions. These clauses are then tested for satisfiability using MiniSatID [89]. Because its underlying engine supports inductive definitions, IDP1.3 performs much better in their presence than Kodkod. It can, for example, find a Hamiltonian cycle in a graph with 200 nodes and 1800 edges in less than a second [147]. Kodkod, in contrast, scales only to graphs with about 30 nodes and 100 edges.

Paradox2.3 [25] is a traditional SAT-based model finder for unsorted first order logic. Unlike other SAT-based tools, Paradox2.3 searches for models incrementally. In particular, it translates the input formula to SAT in a universe of size k and feeds the result to an instance of MiniSat [43]. If the solver finds a model, it is translated to a model of the original specification and the search terminates. Otherwise, the clauses specific to the universe size are retracted from the solver's database, followed by the addition of new clauses that turn the solver's database into a valid translation of the original specification in a universe of size $k + 1$. This enables the SAT solver to reuse search information between consecutive universe sizes, making the tool well-suited to analyses whose goal is to find a model in some universe rather than test a specification's satisfiability in a specific universe. Paradox2.3 also employs a number of heuristics [25] that enable it to translate specifications in unsorted FOL much more efficiently than Kodkod, especially in the presence of deeply nested quantifiers.

Mace4 [92] implements a dedicated search algorithm for finding models of specifications in unsorted first order logic. Its search engine performs on-the-fly symmetry breaking using a variant of the Least Number Heuristic [151]. Search combined with dynamic symmetry breaking is particularly effective on specifications that arise in

group theory. Such specifications have highly symmetric state spaces and consist of formulas that compare deeply nested terms for equality. Predecessors of Mace4 [51, 151, 152] have all been used to solve open problems in abstract algebra, and search-based tools remain the most efficient model finders for this class of problems.

5.2 Future work

The techniques presented here have significantly extended the scalability and applicability of relational model finding. Kodkod is orders of magnitude faster than prior relational engines [68, 70, 117] and includes a unique combination of features that make it suitable for both declarative configuration [101, 149] and analysis [21, 31, 34, 114, 126, 135, 137]. But a number of scalability and usability problems still remain. The engine currently implements only rudimentary techniques for deciding bitvector arithmetic; its support for inductive definitions is very limited; it includes no special handling for simpler fragments of its input language; and it does not incorporate any heuristics for choosing good variable and constraint orderings either in the context of model finding or core extraction.

5.2.1 Bitvector arithmetic and inductive definitions

The need for better handling of arithmetic and inductive definitions arose in several applications [21, 31, 149] of Kodkod. At the moment, the tool handles both arithmetic and transitive closure by direct translation to SAT. This approach, known as "bit-blasting," is effective for some problems. For example, Kodkod is as efficient as state-of-the-art SMT solvers on specifications in which bitvectors are combined with low-level operators (shifts, ands, and ors) and compared for (in)equality [46]. In general, however, bit-blasting yields large and hard to solve boolean formulas.

An interesting direction for future work would be to explore if bit-blasting can be avoided, or at least minimized, with the aid of two recently developed SMT techniques [53, 89]. The first [53] is a linear solver for eliminating redundant variables from bitvector expressions prior to bit-blasting; the second [89] is a new SAT solver that

supports inductive definitions. The former would have to be adapted for use with Kodkod because most bitvector variables that appear in Kodkod formulas are not free: they represent cardinalities of relations or sums of integer-valued atoms in a set. Exploiting the latter would require extending CBCs (Chapter 2) with a new circuit that represents definitional implication [89] and changing the translation procedure to take advantage of this circuit.

5.2.2 Special-purpose translation for logic fragments

Because sets and scalars are treated as relations in bounded relational logic, Kodkod's translator represents every expression as a matrix of boolean values. Operations on expressions are translated using matrix operations: join corresponds to matrix multiplication, union to disjunction, intersection to conjunction, etc. The resulting translation algorithm is effective on specifications that make heavy use of relational operators. It is, however, much less efficient than the translators employed in Paradox2.3 and IDP1.3 on specifications that are essentially fragments of first order logic.

In particular, given a specification in which the only operations on relations are function application and membership testing, Kodkod's translator expends a large amount of resources on creating matrices and multiplying them when all that is needed to translate such a specification is a series of table lookups. This overhead is negligible for small and medium-sized problems. But for large problems with nested quantifiers, it makes a difference between generating a translation in a few seconds and several minutes. For example, Kodkod takes nearly 3 minutes to translate a specification that says a graph [147] with 7,260 vertices and 21,420 edges is 3-colorable. IDP1.3, in contrast, translates the same specification in less than a second. Similar performance can be elicited from Kodkod only if the specification is re-written so that all joins are eliminated.

An important direction for future work is the development of special-purpose translation techniques for inputs that are specified in simpler fragments of Kodkod's input language. This would require solving two problems: identifying specifications (or parts thereof) that are eligible for faster translation, and translating the identified

fragments in a more efficient manner. The latter could be accomplished either by adapting the techniques used in other solvers or by incorporating those solvers into Kodkod. The hybrid approach seems particularly appealing, since Kodkod's users would benefit automatically from any improvements to the underlying solvers, just as they automatically benefit from improvements in the SAT technology.

5.2.3 Heuristics for variable and constraint ordering

It is well-known that SAT solvers are sensitive to the order in which variables and clauses are presented to them, with small perturbations in input ordering causing large differences in runtime. MiniSat, for example, can take anywhere between 1.5 and 5 minutes to solve the problem AWD.A241 from Chapter 2, depending on the order in which its constraints are translated. Similarly, the order in which constraints are selected for pruning during core extraction (Chapter 4) often make the difference between finding a minimal core in a few seconds and giving up after a few hours.

The problem of choosing a good (variable) ordering has been studied extensively in the SAT community [3, 4, 5, 17, 35, 37, 36, 66, 98]. Heuristics that work at the boolean level, however, do not take into account domain-level knowledge. Recent work on choosing variable orderings for specific problems, such as graph coloring [140] or bounded model checking [142], has shown that heuristics based on domain knowledge can significantly outperform more generic approaches. Corresponding heuristics have not yet been developed for SAT-based model finding. The same is true for core extraction; no heuristics have been developed for choosing a good minimization order for a given core. Both of these are exciting and promising areas for future research, with a potential to further extend the efficiency and applicability of declarative problem solving.

Bibliography

[1] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on sat-solvers. In *TACAS '00*, pages 411–425, London, UK, 2000. Springer-Verlag.

[2] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Mince: a static global variable-ordering for SAT and BDD. In *Proc. Int'l Workshop on Logic and Synthesis*, University of Michigan, June 2001.

[4] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 116–119, New York, NY, USA, 2003. ACM.

[5] E. Amir and S. McIlraith. Solving satisfiability using decomposition and the most constrained subproblem. In *SAT'01*, 2001.

[6] Henrik Reif Andersen and Henrik Hulgaard. Boolean expression diagrams. In *LICS '97*, Warsaw, Poland, June 1997.

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

[8] L. Babai, W. M. Kantor, and E. M. Luks. Computational complexity and the classification of finite simple groups. In *IEEE SFCS*, pages 162–171. IEEE CSP, 1983.

[9] Domagoj Babic and Frank Hutter. Spear. In *SAT 2007 Competition*, 2007.

[10] R. Backofen and S. Will. Excluding symmetries in concurrent constraint programming. In *Workshop on Modeling and Computing with Concurrent Constraint Programming*, 1998.

[11] R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *Proc. Principles and Practice of Constraint Programming (CP'99)*, volume LNCS 1713, pages 73–87. Springer, 1999.

[12] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 2nd edition, 1994.

[13] Peter Baumgartner, Alexander Fuchs, Hans de Nivelle, and Cesare Tinelli. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*, 2007.

[14] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the model evolution calculus. *International Journal of Artificial Intelligence Tools*, 15(1):21–52, 2006.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Shtrichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[16] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS'99: Proc. of the 5th Int'l Conf. on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[17] P. Bjesse, J. Kukula, R. Damiano, T. Stanion, and Y. Zhu. Guiding SAT diagnosis with tree decompositions. In *SAT '03*, 2003.

[18] Per Bjesse and Arne Boralv. DAG-aware circuit compression for formal verification. In *Proc. 2004 IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD'04)*, pages 42–49, Washington, DC, USA, 2004. IEEE Computer Society.

[19] Robert Brummayer and Armin Biere. Local two-level and-inverter graph minimization without blowup. In *Proc. 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS '06)*, October 2006.

[20] E. J. H. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, 1979.

[21] Felix Chang. Alloy analyzer 4.0. http://alloy.mit.edu/alloy4/, 2007.

[22] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[23] John W. Chinneck and Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal of Computing*, 3(2):157–158, 1991.

[24] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. Technical Report TR-542, Univ. of Rochester, November 1994.

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

[26] David Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara Smith. Symmetry definitions for constraint satisfaction problems. In *Proc. Principles and Practice of Constraint Programming (CP'05)*, volume LNCS 3709, pages 17–31. Springer Berlin, 2005.

[27] James Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *KR'96*, pages 148–159. Morgan Kaufmann, San Francisco, 1996.

[28] Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proc. 45th Design Automation Conference*, Anaheim, California, June 2008.

[29] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, Budapest, Hungary, 2008.

[30] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In *VSTTE'08*, Toronto, Canada, October 2008.

[31] Gregory Dennis, Felix Chang, and Daniel Jackson. Modular verification of code. In *ISSTA '06*, Portland, Maine, July 2006.

[32] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *SAT '06*, LNCS, pages 36–41. Springer Berlin, 2006.

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

[34] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a sat solver. In *ESEC-FSE '07*, pages 195–204, New York, NY, USA, 2007. ACM.

[35] V. Durairaj and P. Kalla. Guiding cnf-sat search via efficient constraint partitioning. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 498–501, Washington, DC, USA, 2004. IEEE Computer Society.

[36] Vijay Durairaj and Priyank Kalla. Variable ordering for efficient SAT search by analyzing constraint-variable dependencies. In *SAT '05*, pages 415–422. Springer, 2005.

[37] Vijay Durairaj and Priyank Kalla. Guiding CNF-SAT search by analyzing constraint-variable dependencies and clause lengths. *High-Level Design Validation and Test Workshop, 2006. Eleventh Annual IEEE International*, pages 155–161, Nov. 2006.

[38] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV'06*, LNCS, pages 81–94. Springer-Verlag, 2006.

[39] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A type system for object models. In *SIGSOFT '04/FSE-12*, pages 189–199, New York, NY, USA, 2004. ACM Press.

[40] Jonathan Edwards, Daniel Jackson, Emina Torlak, and Vincent Yeung. Faster constraint solving with subtypes. In *ISSTA '04*, pages 232–242, New York, NY, USA, 2004. ACM Press.

[41] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT '05*, 2005.

[42] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *Proc. of the 1st Int'l Workshop on Bounded Model Checking (BMC'03)*, Boulder, Colorado, July 2003.

[43] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT'03*, volume LNCS 2919, pages 502–518, 2004.

[44] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. In *SBMC*, volume 2, pages 1–26, 2006.

[45] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *Proc. Principles and Practice of Constraint Programming (CP'01)*, volume LNCS 2239, pages 93–107. Springer, 2001.

[46] Mendy Fisch. Using satisfiability solvers to test network configuration, October 2008.

[47] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In P. van Hentenryck, editor, *Proc. Principles and Practice of Constraint Programming (CP'02)*, volume LNCS 2470, pages 462–476. Springer-Verlag, 2002.

[48] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proc. Principles and Practice of Constraint Programming (CP'01)*, volume LNCS 2239, pages 72–92. Springer, 2001.

[49] Formal Methods Group at ITC-IRST, Model Checking Group at CMU, Mechanized Reasoning Group at U. of Genova, and Mechanized Reasoning Group at U. of Trento. NuSMV: a new symbolic model checker. http://nusmv.irst.itc.it/, August 2008.

[50] A. M. Frisch, C. Jefferson, and I. Miguel. Symmetry breaking as a prelude to implied constraints: a constraint modelling pattern. *IJCAI*, pages 109–116, 2005.

[51] Masayuki Fujita, John Slaney, and Frank Bennett. Automating generation of some results in finite algebra. In *13th IJCAI*, Chambéry, France, 1993.

[52] Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Beyond safety: customized SAT-based model checking. In *DAC'05: Proc. of the 42nd Conf. on Design Automation*, pages 738–743, New York, NY, USA, 2005. ACM.

[53] Vijay Ganesh. *Decision Procedures for Bit-Vectors, Arrays and Integers*. PhD thesis, Stanford University, Stanford, CA, September 2007.

[54] Ian P. Gent and Iain McDonald. NuSBDS: Symmetry breaking made easy. In Barbara Smith, Ian P. Gent, and Warwick Harvey, editors, *Proc. 3rd Int'l Workshop on Symmetry in CSPs*, pages 153–160, 2003.

[55] Ian P. Gent, Karen E. Petrie, and Jean-Francois Puget. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 10. Elsevier Science Publishers B. V., 2006.

[56] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *DATE '02*, pages 142–149, 2002.

[57] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE '03*, 2003.

[58] Peter Green. *200 Difficult Sudoku Puzzles*. Lulu Press, 2nd edition, July 2005.

[59] Eric Grégoire, Bertrand Mazure, and Cédric Piette. Extracting MUSes. In *ECAI'06*, pages 387–391, Trento (Italy), August 2006.

[60] Eric Grégoire, Bertrand Mazure, and Cédric Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *IJCAI'07*, volume 2, pages 2300–2305, Hyderabad (India), January 2007.

[61] Eric Grégoire, Bertrand Mazure, and Cédric Piette. Local-search extraction of MUSes. *Constraints Journal*, 12(3):324–344, 2007.

[62] Michael Hammer and Dennis McLeod. The semantic data model: a modelling mechanism for data base applications. In *SIGMOD'78: Proceedings of the 1978 ACM SIGMOD Int'l Conf. on Management of Data*, pages 26–36, New York, NY, USA, 1978. ACM.

[63] Vegard Hanssen. Sudoku puzzles. http://www.menneske.no/sudoku/eng/, July 2008.

[64] Warwick Harvey, Tom Kelsey, and Karen Petrie. Symmetry group expressions for CSPs. In Barbara Smith, Ian P. Gent, and Warwick Harvey, editors, *Proc. 3rd Int'l Workshop on Symmetry in CSPs*, pages 86–96, 2003.

[65] J. N. Hooker. Solving the incremental satisfiability problem. *J. Log. Program.*, 15(1-2):177–186, 1993.

[66] J. Huang and A. Darwiche. A structure-based variable ordering heuristic for SAT. In *Proc. 1st Int'l Joint Conference on Artificial Intelligence (IJCAI)*, pages 1167–1172, August 2003.

[67] Daniel Jackson. An intermediate design language and its analysis. In *FSE'98*, November 1998.

[68] Daniel Jackson. Automating first order relational logic. In *FSE '00*, San Diego, CA, November 2000.

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

[70] Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free model enumeration: a new method for checking relational specifications. *ACM TPLS*, 20(2):302–343, March 1998.

[71] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *ESEC / SIGSOFT FSE '01*, pages 62–73, 2001.

[72] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *ALENEX 2007*, New Orleans, Lousiana, January 2007.

[73] T. Jussila and Armin Biere. Compressing BMC encodings with QBF. In *Proc. BMC'06*, 2006.

[74] John A. Kalman. *Automated Reasoning with Otter*. Rinton Press, 2001.

[75] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In *ABZ'08*, London, UK, September 2008.

[76] Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Massachusetts Institute of Technology, February 2003.

[77] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of java programs using sat. *ASE '00*, 11(4):403–434, 2004.

[78] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on CAD*, 21(12):1377–1394, December 2002.

[79] Mark H. Liffiton and Karem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *SAT '05*, 2005.

[80] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, January 2008.

[81] Amerson Lin, Mike Bond, and Joylon Clulow. Modeling partial attacks with Alloy. In *Security Protocols Workshop (SPW'07)*, Brno, Czech republic, April 2007.

[82] Chun-Yuan Lin, Yeh-Ching Chung, and Jen-Shiu Liu. Efficient data compression methods for multidimensional sparse array operations. In *Proc. 1st Int'l Symp. on Cyber Worlds (CW'02)*, Tokyo, Japan, 2002.

[83] Chun-Yuan Lin, Jen-Shiu Liu, and Yeh-Ching Chung. Efficient representation scheme for multidimensional array operations. *IEEE Transactions on Computers*, 51(3), March 2002.

[84] Mikel Lujan, Anila Usman, Patrick Hardie, T.L. Friedman, and John R. Gurd. Storage formats for sparse matrices in Java. In *ICCS 2005*, number 3514 in LNCS, pages 365–371, 2005.

[85] Inês Lynce and Jo ao Marques-Silva. On computing minimum unsatisfiable cores. In *SAT '04*, 2004.

[86] Inês Lynce and Jöel Ouaknine. Sudoku as a SAT problem. In *9th Int. Symp. on AI and Mathematics*, January 2006.

[87] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. zchaff2004: An efficient SAT solver. In *SAT (Selected Papers)*, pages 360–375, 2004.

[88] Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declarative problem solving. In E. Giunchiglia, V. Marek, D. Mitchell, and

E. Ternovska, editors, *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.

[89] Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. SAT(ID): Satisfiability for propositional logic with inductive definitions. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *LCNS*, pages 211–224. Springer, 2008.

[90] J. P. Marques-Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *Transactions on Computers*, 48(5):506–521, 1999.

[91] William McCune. A Davis-Putnam program and its application to finite first-order model search: quasigroup existence problem. Technical report, ANL, 1994.

[92] William McCune. Mace4 reference manual and guide. Technical Memorandum 264, Argonne National Laboratory, August 2003.

[93] William McCune. Prover9 and MACE4. http://www.cs.unm.edu/~mccune/prover9/, June 2008.

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

[95] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, 1993.

[96] David Mitchell and Eugenia Ternovska. A framework for representing and solving np search problems. In *AAAI'05*, pages 430–435. AAAI Press / MIT Press, 2005.

[97] Maher Mneimneh, Inês Lynce, Zaher Andraus, Jo ao Marques-Silva, and Karem A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable formulas. In *SAT '05*, 2005.

[98] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *DAC'01*, pages 530–535, 2001.

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

[100] Sanjai Narain. ConfigAssure. http://www.research.telcordia.com/bios/narain/, December 2007.

[101] Sanjai Narain, Gary Levin, Vikram Kaul, and Sharad Malik. Declarative infrastructure configuration synthesis and debugging. *Journal of Network Systems and Management, Special Issue on Security Configuration*, 2008.

[102] Yoonna Oh, Maher Mneimneh, Zaher Andraus, Karem Sakallah, and Igor Markov. Amuse: A minimally-unsatisfiable subformula extractor. In *DAC*, pages 518–523. ACM/IEEE, June 2004.

[103] J. F. Puget. Symmetry breaking using stabilizers. In F. Rossi, editor, *Proc. Principles and Practice of Constraint Programming (CP'03)*, volume LNCS 2833, pages 585–599. Springer, 2003.

[104] J. F. Puget. Breaking symmetries in all-different problems. *IJCAI*, pages 272–277, 2005.

[105] Jean-Francois Puget. Automatic detection of variable and value symmetries. In P. van Beek, editor, *Proc. Principles and Practice of Constraint Programming (CP'05)*, volume LNCS 3709, pages 475–489. Springer-Verlag, 2005.

[106] Tahina Ramananandro. The Mondex case study with Alloy. http://www.eleves.ens.fr/home/ramanana/work/mondex/, 2006.

[107] Arathi Ramani and Igor L. Markov. Automatically exploiting symmetries in constraint programming. In B. Faltings et al., editor, *Proc. CSCLP'04*, volume LNAI 3419, pages 98–112. Springer-Verlag, 2005.

[108] A. Riazanov. *Implementing an Efficient Theorem Prover*. PhD Thesis, The University of Manchester, Manchester, July 2003.

[109] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[110] Gordon Royle. Minimum sudoku. http://people.csse.uwa.edu.au/gordon/sudokumin.php, July 2008.

[111] Ashish Sabharwal. SymChaff: A structure-aware satisfiability solver. In *AAAI '05*, pages 467–474, Pittsburgh, PA, July 2005.

[112] Sean Safarpour, Andreas Veneris, Gregg Baeckler, and Richard Yuan. Efficient SAT-based boolean matching for FPGA technology mapping. In *DAC'06: Proc. of the 43rd Conf. on Design Automation*, pages 466–471, New York, NY, USA, 2006. ACM.

[113] Robert Seater, Daniel Jackson, and Rohit Gheyi. Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering Journal (REJ'07)*, 2007.

[114] Danhua Shao, Sarfraz Khurshid, and Dewayne Perry. Whispec: White-box testing of libraries using declarative specifications. In *LCSD'07*, Montreal, October 2007.

[115] ShengYu Shen, Ying Qin, and SiKun Li. Minimizing counterexample with unit core extraction and SAT. In *VMCAI'05*, Paris, France, January 2005.

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

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

[118] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *ASE '03*, 2003.

[119] Ilya Shlyakhter, Manu Sridharan, Robert Seater, and Daniel Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. In *SAT '03*, Portofino, Italy, May 2003.

[120] Ofer Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *CHARME'01: Proc. of the 11th IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods*, pages 58–70, London, UK, 2001. Springer-Verlag.

[121] Joao P. Marques Silva and Karem A. Sakallah. Robust search algorithms for test pattern generation. In *FTCS'97: Proc. of the 27th Int'l Symposium on Fault-Tolerant Computing*, page 152, Washington, DC, USA, 1997. IEEE Computer Society.

[122] John K. Slaney. Finder: Finite domain enumerator - system description. In *CADE-12*, pages 798–801, London, UK, 1994. Springer-Verlag.

[123] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 1992.

[124] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[125] Geoff Sutcliffe. The CADE ATP system competition. http://www.cs.miami.edu/ tptp/CASC/J4/, October 2008.

[126] Mana Taghdiri. *Automating Modular Program Verification by Refining Specifications*. PhD thesis, Massachusetts Institute of Technology, 2007.

[127] Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6:73–89, 1941.

[128] Emina Torlak. Subtyping in alloy. Master's thesis, MIT, May 2004.

[129] Emina Torlak, Felix Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM '08*, May 2008.

[130] Emina Torlak and Greg Dennis. Kodkod for Alloy users. In *First ACM Alloy Workshop*, Portland, Oregon, November 2006.

[131] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS '07*, Braga, Portugal, 2007.

[132] Michael Trick. Graph coloring instances. http://mat.gsia.cmu.edu/COLOR/instances.html, October 2008.

[133] Chris Tucker, David Schuffelton, Ranjitt Jhala, and Sorin Lerner. OPIUM: Optimizing package install/uninstall manager. In *Proc. 29th Int. Conf. in Soft. Eng. (ICSE'07)*, Minneapolis, Minnesota, May 2007.

[134] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. A specification-based approach to testing software product lines. In *ESEC-FSE '07*, pages 525–528, New York, NY, USA, 2007. ACM.

[135] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. Testing software product lines using incremental test generation. In *ISSRE'08*, Seattle / Redmond, WA, November 2008.

[136] Engin Uzuncaova and Sarfraz Khurshid. Kato: A program slicing tool for declarative specifications. In *ICSE'07*, pages 767–770, Washington, DC, USA, 2007. IEEE Computer Society.

[137] Engin Uzuncaova and Sarfraz Khurshid. Constraint prioritization for efficient analysis of declarative models. In *FM '08*, Turku, Finland, 2008.

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

[139] Miroslav N. Velev. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *ASP-DAC '04*, pages 310–315, January 2004.

[140] Miroslav N. Velev. Exploiting hierarchy and structure to efficiently solve graph coloring as sat. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 135–142, Piscataway, NJ, USA, 2007. IEEE Press.

[141] Pete Wake. Sudoku solver by logic. http://www.sudokusolver.co.uk/, September 2005.

[142] Chao Wang, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. Refining the SAT decision ordering for bounded model checking. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 535–538, New York, NY, USA, 2004. ACM.

[143] Jos Warmer and Anneke Kelppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[144] Tjark Weber. A SAT-based sudoku solver. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR-12*, pages 11–15, December 2005.

[145] Christoph Weidenbach. Combining superposition, sorts and splitting. *Handbook of automated reasoning*, pages 1965–2013, 2001.

[146] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: a new incremental satisfiability engine. In *DAC'01: Proc. of the 38th Conf. on Design Automation*, pages 542–545, New York, NY, USA, 2001. ACM.

[147] Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In *LaSh'08*, Leuven, Belgium, November 2008.

[148] Vincent Yeung. Course scheduler (http://optima.csail.mit.edu:8080/scheduler/), 2006.

[149] Vincent Yeung. Declarative configuration applied to course scheduling. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2006.

[150] Yinlei Yu, Cameron Brien, and Sharad Malik. Exploiting circuit reconvergence through static leraning in CNF SAT solvers. In *VLSI Design '08*, pages 461–468, 2008.

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

[152] Jian Zhang and Hantao Zhang. SEM: a system for enumerating models. In *IJCAI95*, Montreal, August 1995.

[153] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *SAT '03*, 2003.