Kodkod for Alloy Users

Emina Torlak and Greg Dennis
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139
{emina, gdennis}@mit.edu

ABSTRACT

The generality of Alloy's relational logic and the full automation of its analysis have prompted several attempts to use Alloy as a backend engine for other tools. However, these efforts have been hampered by the Alloy Analyzer's lack of 1) a clean API, 2) support for partial instances, and 3) a mechanism for sharing subformulas and subexpressions. Designed expressly as a plugin component, the Kodkod relational engine overcomes these limitations. Unlike the Analyzer, Kodkod provides a simple interface for constructing and analyzing Alloy formulas; it accepts user-provided partial instances of these formulas; and it employs a robust scheme for exploiting shared formulas and expressions.

This paper is an introduction to Kodkod for current Alloy practitioners. It describes the key differences between Kodkod and the Alloy Analyzer, including the meaning and utility of partial instances. A complete example that programmatically builds and analyzes an Alloy formula with Kodkod is presented, and results that compare Kodkod to the Alloy Analyzer on a series of benchmarks are discussed.

Keywords

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

1. INTRODUCTION

The Alloy Analyzer enables automatic analysis of formulas written in the relational logic of Alloy [4]. This automated analysis, coupled with the versatility of the logic as a general problem description language, has prompted several attempts to use Alloy as a basis for other automated tools. However, these efforts have been hampered by three key limitations of the Alloy Analyzer:

A clean API through which clients can build, manipulate, and analyze Alloy formulas. A clean programming interface was not among the original design considerations for the Alloy Analyzer. Designed as a desktop CAD application, the Analyzer's intended use was limited to the parsing of Alloy source files and their translation to boolean satisfiability (SAT) problems. To use the Analyzer as a backend engine, tools such as Jalloy [9] have resorted to generating Alloy source and feeding it to the Analyzer's parser. This process of generating textual models is typically slow, awkward, and error-prone. Even when accurate, the generation can yield models that the Analyzer cannot parse efficiently, e.g. a model with a single monolithic formula not decomposed into predicates, functions, or lets.

Support for partial instances of formulas. To grasp the meaning of a *partial instance*, consider the task of solving a Sudoku puzzle. The goal of Sudoku is to fill a 9×9 grid so that each digit (1 through 9) appears exactly once in each row, each column, and each of the nine 3×3 regions of the board. Every Sudoku puzzle contains some "given" cells that are assigned fixed digits at the outset such that the puzzle has exactly one solution. The "givens" can be viewed as a *partial instance* of this Sudoku problem. That is, they represent the part of the problem's solution that is known a priori. With the Alloy Analyzer, such partial solutions must be encoded as constraints on the model, and the analysis then must essentially rediscover that partial solution from the constraints. Performance of the analysis would thus be improved if the partial instance could instead be provided directly.

A mechanism for sharing common subformulas and subexpressions. The data structures used to represent formulas and expressions in the Alloy Analyzer are trees. Consequently, common subgraphs cannot be shared; they must be duplicated. This strict tree structure simplifies a number of the Analyzer's algorithms, but at a cost. Not only do the duplicate subgraphs consume additional memory, but they can also lead to duplicate SAT variables in the resulting boolean formula, causing performance to needlessly suffer as well. To compensate, the Analyzer employs a clever *template detection* scheme [8] that avoids doubly allocating boolean variables for some common subgraphs, but it still misses other sharing opportunities.

Kodkod is a new relational engine designed expressly as a plugin component that can easily be incorporated as a backend of another tool. Unlike the Alloy Analyzer, Kodkod provides a clean Java interface for constructing, manipulating, and analyzing an Alloy formula; it allows the user to specify a partial instance of the formula; it allows common subformulas and subexpressions to be directly shared; and it employs an improved scheme for detecting further sharing opportunities.

First Alloy Workshop November 2006, Portland, Oregon USA

Current applications of Kodkod include:

- Design analysis. Design specifications written for the Alloy Analyzer, including a model of the Mondex electronic purse [6], have been analyzed with Kodkod.
- Code analysis. A procedure can be checked against a declarative specification by translating its code to a relational constraint. Dennis and Chang [1] have applied this technique to check the correctness of Java data structures.
- Course scheduling. Given the prerequisite dependencies, overall requirements of a degree program, information about which terms particular courses are offered in, and a set of courses already taken, Kodkod can plan a student's course schedule. Yeung has built such a course scheduling application [11].
- Sudoku. The rules of this puzzle game can be encoded as relational constraints, and the "given" cells can be can be provided as a partial instance.
- Alloy Analyzer 4.0. An ongoing project seeks to replace the core of the Alloy Analyzer with Kodkod.

Kodkod and the Alloy Analyzer share the same underlying technology: a translation from relational to boolean logic, and the application of an off-the-shelf SAT solver on the resulting boolean formula. Nevertheless, Kodkod outperforms Alloy dramatically, particularly on problems involving partial solutions. A number of important technical challenges were overcome to make this possible, but those are not the subject of this paper. Instead, this is intended as a guide for potential Kodkod practitioners, in particular those who are already familiar with Alloy.

2. KODKOD LOGIC & ANALYSIS

The logic accepted by Kodkod is a core subset of the Alloy modeling language. Kodkod formulas, like Alloy formulas, are constraints on *relational variables* of any arity. In Alloy, these relational variables are divided into "signatures" (for unary relations) and "fields" (for non-unary relations), and the signatures establish a type hierarchy. Kodkod, in contrast, makes no special distinction between unary and non-unary relational variables, and all relations are untyped. In addition, the purely syntactic conventions of the Alloy modeling language — predicates, functions, facts, etc — are not part of the Kodkod logic. Think of the Kodkod logic as Alloy, stripped of its syntactic sugar.

Kodkod requires the relational variables to be bound prior to analysis, as does the Alloy Analyzer. But unlike Alloy, Kodkod relations are not bound by integer limits on the number of atoms in each set. Instead, every relational variable in Kodkod (including non-unary ones) must be bound from above by a *relational constant*, a fixed set of tuples drawn from a universe of atoms. This *upper bound* consists of the tuples the variable *may* contain. Each relation in Kodkod must also be bound from below by a relational constant — a *lower bound* consisting of the tuples it *must* contain, i.e. a partial instance. An *instance* of a formula is a binding of the declared relational variables to relational constants, that makes the formula true. Kodkod's analysis will search for such an instance within the provided upper and lower bounds.

problem := univDecl relDecl* formula

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

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

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

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

expr :=	
expr + expr	union
| expr & expr	intersection
| expr – expr	difference
| expr . expr	join
| expr –> expr	product
| ~expr	transpose
| ^expr	closure
| {varDecl | formula}	comprehension
| relVar	relation
| quantVar	quantified variable

Figure 1: Abstract syntax of Kodkod logic

Figure 1 defines the abstract syntax of the Kodkod relational logic. Although this is presented as a syntax for pedagogical purposes, the formulas are constructed via Kodkod API calls, not via source text that is parsed. Also, the actual Kodkod data structures allow subformulas and subexpressions to be shared, so the syntax should be seen as describing a directed acyclic graph rather than a tree.

As shown in Figure 1, a *problem* in the logic is a *universe declaration*, a set of *relation declarations*, and a *formula* in which the declared relations appear as free variables. Each relation declaration specifies the arity of a relation variable and the constants that are its lower and upper bounds. The tuples of all constants are drawn from the problem's universe. The remaining syntactic productions describe a subset of the Alloy logic.

To illustrate, consider the following formulation of the pigeonhole principle. The principle says that n pigeons cannot be placed into $n-1$ holes so that each pigeons has a hole to itself. Taking n to be three, we can state the principle as a Kodkod problem using the following formulation:

```
{P1, P2, P3, H1, H2}

Pigeon :1 [{⟨P1⟩⟨P2⟩⟨P3⟩}, {⟨P1⟩⟨P2⟩⟨P3⟩}]
Hole   :1 [{⟨H1⟩⟨H2⟩}, {⟨H1⟩⟨H2⟩}]
nest   :2 [{}, {⟨P1, H1⟩⟨P1, H2⟩⟨P2, H1⟩⟨P2, H2⟩⟨P3, H1⟩⟨P3, H2⟩}]

(all p : Pigeon | one p.nest) and
(all h : Hole | lone nest.h)
```

The first line declares a universe of five atoms. We arbitrarily chose the first three of them to represent pigeons and the last two to represent holes. Because formulas cannot contain constants, a relational variable with the same upper and lower bound is declared for each constant in the formula. The variables Pigeon and Hole, for example, serve as handles to the constants {⟨P1⟩⟨P2⟩⟨P3⟩} and {⟨H1⟩⟨H2⟩}, the sets of all pigeons and holes, respectively. The binary relation nest encodes the placement of pigeons into holes. Its value is constrained to be an injection by the formula.

Kodkod and the Alloy Analyzer perform the same basic analysis. They both translate relational logic to boolean logic and invoke an off-the-shelf SAT solver on the resulting boolean formula. If a solution to the boolean formula is found, it is converted into an instance of the relational formula, i.e. a binding of the formula's relational variables to constants.

Kodkod improves upon this basic analysis by applying new techniques and optimizations to the translation from relational to boolean logic. For starters, it exploits the user-provided partial instances to reduce the size of the resulting SAT problem. In addition, Kodkod replaces the Alloy Analyzer's symmetry-breaking scheme [7] with a more general technique that works in the presence of the arbitrary upper and lower bounds provided for each relation.

One of Kodkod's more powerful optimizations is its detection of common subformulas and subexpressions. Not only does Kodkod allow explicit sharing of subgraphs by aliasing the data structures themselves, but it also employs a robust scheme for finding and eliminating duplicate subgraphs that could have been directly shared. This is accomplished efficiently with a single depth-first search down the formula graph. For any shared subgraph – either aliased explicitly or found through the search – boolean variables will be allocated only once, thus reducing the SAT problem and improving the performance of the analysis.

3. KODKOD API

The Kodkod API is a Java library that embodies the logic and analysis described in the previous section. At roughly 12,000 source lines of code (SLOC), it's a relatively small piece of software. To demonstrate the features of the API and how it differs from the Alloy Analyzer, we apply both tools to the Sudoku problem shown in Figure 2. Shaded entries are the puzzle's "givens" and the white entries show its solution — the solution that Kodkod and the Analyzer need to generate to satisfy the rules of the game.

3.1 Sudoku in Alloy

Figure 3 shows an Alloy formulation of the sample Sudoku problem. Signature Number introduces the set of numbers used in the game. It is partitioned into three regions, each of which is further partitioned into three singletons. These declarations ensure that an instance of the model contains exactly nine Number atoms, singletons N1 through N9.

The field data establishes a two-dimensional grid that maps each row and column pair to the digits in that cell on the grid. The rules predicate constrains data to be a proper Sudoku solution. The first rule states that data is a function from pairs of Numbers to at most one Number. (That this function is total is implied by the remaining rules.) The second rule ensures that each row of the grid is "complete", i.e. that all the numbers are found in that row. The remaining rules ensure that each column and each 3×3 region of the grid is complete as well.

The puzzle predicate encodes the given cells (shaded in Figure 2) for the sample puzzle. Because the Alloy Analyzer lacks support for partial instances, these givens must be encoded as constraints on the data field. For example, the constraint N1–>N4–>N2 in data ensures that the solution maps the cell (1, 4) to the number 2. The game predicate conjoins the puzzle and the rules; running it amounts to asking the Alloy Analyzer to solve the sample problem.

3.2 Sudoku in Kodkod

Figure 4 shows a formulation of the sample Sudoku problem with the Kodkod API. The formula and expression productions from Figure 1 are represented by immutable Java classes. Being immutable, the classes require that any subformulas and subexpressions be provided on construction, thus ensuring that the abstract syntax graph is acyclic.

A problem's universe is given as a Universe object constructed from a user-provided Collection of Objects. Each Universe provides a TupleFactory for creating constants, represented by TupleSet objects, from atoms drawn from that Universe. The relation declarations are embodied by the Bounds abstraction, which maps Relation objects to upper and lower bound TupleSets. A configurable Solver object provides access to the facilities for solving a Formula with respect to the given Bounds. A Solver is configured through its Options component, which enables the selection of various SAT solvers and tuning of various translation parameters.

Lines 2-6 of Figure 4 declare a ternary relation data and four sets, Number and regions[0..2]. Note that unlike their Alloy equivalents, these relations are untyped. The limit on the values each relation can contain are specified separately through the Bounds. Java methods can be used to encapsulate and parameterize reusable formulas and expressions — the same role predicates and functions play in Alloy. The methods complete and rules, for example, correspond to the predicates of the same name in the Alloy model.

Unlike its Alloy equivalent, the puzzle method encodes the problem's partial instance (lines 47-75) in the Bounds rather than as constraints. Line 38 constructs a Universe out of a set of 9 Integers, and from the Universe a TupleFactory is obtained to build a Bounds for the problem. The boundExactly method assigns a Relation the same lower and upper bounds, thereby ensuring it is a constant. The Number relation, for example, is assigned the set of all tuples of arity 1 (the set of all 9 integers), while region[0] is the set containing exactly 1, 2 and 3. The only Relation in the sample problem with different lower and upper bounds is data. Its lower bound is the TupleSet givens which contains the tuples corresponding to the shaded cells in Figure 2 (lines 47-74). Its upper bound is the set of all tuples of arity 3 (line 75).

The main method plays the role of the Alloy run command. It constructs and customizes a Kodkod Solver (lines 81-82), applies it to the problem (line 84), and prints the result.

4. RESULTS

We have compared Kodkod's performance (KK) to that of the Alloy Analyzer (AA) on three sets of problems:[1]

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

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

Figure 2: Sample Sudoku Puzzle

```
abstract sig Number { data: Number -> Number }

abstract sig Region1, Region2, Region3 extends Number {}

one sig N1, N2, N3 extends Region1 {}
one sig N4, N5, N6 extends Region2 {}
one sig N7, N8, N9 extends Region3 {}

pred complete(rows: set Number, cols: set Number) {

  Number in cols.(rows.data) }

pred rules() {
  all x, y: Number { lone y.(x.data) }
  all row: Number { complete(row, Number) }
  all col: Number { complete(Number, col) }
  complete(Region1, Region1)
  complete(Region1, Region2)
  complete(Region1, Region3)
  complete(Region2, Region1)
  complete(Region2, Region2)
  complete(Region2, Region3)
  complete(Region3, Region1)
  complete(Region3, Region2)
  complete(Region3, Region3)
}

pred puzzle() {
  N1->N1->N1 + N1->N4->N2 + N1->N7->N3 +
  N2->N2->N2 + N2->N5->N3 + N2->N8->N4 +
  N3->N3->N3 + N3->N6->N4 + N3->N9->N5 +
  N4->N1->N6 + N4->N4->N4 + N4->N7->N5 +
  N5->N2->N7 + N5->N5->N5 + N5->N8->N6 +
  N6->N3->N8 + N6->N6->N6 + N6->N9->N7 +
  N7->N1->N8 + N7->N4->N9 + N7->N7->N7 +
  N8->N2->N9 + N8->N5->N1 + N8->N8->N8 +
  N9->N3->N1 + N9->N6->N2 + N9->N9->N4 in data
}

pred game() { rules() && puzzle() }

run game
```

Figure 3: Sudoku in Alloy

```
1  public final class Sudoku {
2    private final Relation Number = Relation.unary("Number");
3    private final Relation data = Relation.ternary("data");
4    private final Relation[] regions = new Relation[] {
5      Relation.unary("Region1"), Relation.unary("Region2"),
6      Relation.unary("Region3") };
7
8    public Formula complete(Expression rows, Expression cols) {
9      return Number.in(cols.join(rows.join(data))); }
10
11   public Formula rules() {
12     final Variable x = Variable.unary("x");
13     final Variable y = Variable.unary("y");
14     final Formula f1 = y.join(x.join(data)).lone().
15       forAll(x.oneOf(Number).and(y.oneOf(Number)));
16
17     final Variable row = Variable.unary("row");
18     final Formula f2 = complete(row, Number).
19       forAll(row.oneOf(Number));
20
21     final Variable col = Variable.unary("col");
22     final Formula f3 = complete(Number, col).
23       forAll(col.oneOf(Number));
24
25     Formula rules = f1.and(f2).and(f3);
26     for(Relation rx: regions) {
27       for(Relation ry: regions) {
28         rules = rules.and(complete(rx,ry));
29       }
30     }
31     return rules;
32   }
33
34   public Bounds puzzle() {
35     final Set<Integer> atoms = new LinkedHashSet<Integer>(9);
36     for(int i = 1; i <= 9; i++) { atoms.add(i); }
37
38     final Universe u = new Universe(atoms);
39     final Bounds b = new Bounds(u);
40     final TupleFactory f = u.factory();
41
42     b.boundExactly(Number, f.allOf(1));
43     b.boundExactly(regions[0], f.setOf(1, 2, 3));
44     b.boundExactly(regions[1], f.setOf(4, 5, 6));
45     b.boundExactly(regions[2], f.setOf(7, 8, 9));
46
47     final TupleSet givens = f.noneOf(3);
48     givens.add(f.tuple(1, 1, 1));
49     givens.add(f.tuple(1, 4, 2));
       ...
74     givens.add(f.tuple(9, 9, 4));
75     b.bound(data, givens, f.allOf(3));
76
77     return b;
78   }
79
80   public static void main(String[] args) {
81     final Solver solver = new Solver();
82     solver.options().setSolver(SATFactory.MiniSat);
83     final Sudoku sudoku = new Sudoku();
84     final Solution sol = solver.solve(sudoku.rules(), sudoku.puzzle());
85     System.out.println(sol);
86   }
87 }
```

Figure 4: Sudoku in Kodkod

	Sudoku (9 × 9)			Tough Nut (8 × 8)		
solver	time	vars	clauses	time	vars	clauses
AA	3	11,618	44,152	60	0	0
KK	0	1,833	2,398	0	0	0

	Ceilings and Floors					
scope	6 men, 6 platforms			10 men, 10 platforms		
solver	time	vars	clauses	time	vars	clauses
AA	1	2,723	11,704	10	9,987	46,740
KK	0	1,749	3,289	4	6,477	12,449

	Pigeonhole Problem					
scope	10 pigeons, 9 holes			30 pigeons, 29 holes		
solver	time	vars	clauses	time	vars	clauses
AA	1	2,899	12,182	11	42,543	207,046
KK	0	1,133	1,983	1	20,473	38,013

	Mutex Ordering					
scope	30 atoms			45 atoms		
solver	time	vars	clauses	time	vars	clauses
AA	65	74,818	722,236	> 300	-	-
KK	2	20,080	120,097	15	67,695	543,597

	Ring Leader Election					
scope	15 atoms			24 atoms		
solver	time	vars	clauses	time	vars	clauses
AA	4	14,272	78,031	121	91,594	662,188
KK	1	8,665	29,590	43	45,136	183,484

Table 1: Results (time in seconds)

- CONSTRAINED PROBLEMS include a Sudoku puzzle and the Tough Nut puzzle [5]. The Sudoku puzzle has exactly one solution. The Tough Nut puzzle is unsatisfiable. It proves that an 8×8 checkerboard with two opposite corners deleted cannot be tiled with dominos.
- SYMMETRIC PROBLEMS consist of two instances of the pigeonhole problem and two instances of the 'Ceilings and Floors' problem bundled with the Alloy Analyzer distribution. Like the pigeonhole problem, it is unsatisfiable and exhibits a high degree of symmetry.
- DESIGN PROBLEMS include the formulations of Dijkstra's algorithm for mutex ordering [2] and the ring leader election algorithm described in [4]. We check that Dijkstra's algorithm prevents deadlocks, and that the leader election algorithm elects at most one leader.

The results are shown in Table 1. For each example, it shows the size of the problem's CNF encoding and the total analysis time rounded to the nearest second. The Sudoku and Tough Nut problems have fixed universes of size nine and eight, respectively. Other problems have been analyzed in universes of varying sizes. All analyses were performed on a 2×3GHz Dual Core Intel Xeon with 2 GB RAM. The Analyzer and Kodkod were both configured to use the MiniSat [3] SAT solver. Analyses that did not complete within five minutes (> 300 seconds) were interrupted.

According to the data in Table 1, Kodkod significantly outperforms the Analyzer on both the problems with partial solutions (Sudoku and Pigeonhole) and classic relational specifications for which Alloy was designed (Mutex Ordering and Ring Leader Election). On Tough Nut, Kodkod's variable and clause entries in the table are 0, indicating that its internal simplifications were alone sufficient to determine that the problem is unsatisfiable.

On these and many other problems, Kodkod was used as a standalone model finder. However, our primary goal is to design a relational engine that is not only scalable and lightweight, but easily integrated into domain-specific applications. At least two applications have used Kodkod in this capacity with success. Dennis and Chang designed a static analysis in which Java code is automatically translated into relational logic, and Kodkod is invoked to find executions of the code that violate a user-provided specification. Their tool automatically checked several implementations of the java.util.List interface against a full specification of its behavior. Yeung built a course scheduler [11], now available as a web application [10], that accepts as input a set of courses a student has taken, degree requirements, and a listing of offered subjects. These constraints are translated to our logic, and Kodkod finds a potential schedule for the student.

5. REFERENCES

[1] G. Dennis, F. Chang, and D. Jackson. Modular verification of code. In *ISSTA*, Portland, Maine, July 2006.

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

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

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

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

[6] T. Ramananandro. Mondex with the Alloy model-finding method, 2006.

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

[8] I. Shlyakhter, M. Sridharan, R. Seater, and D. Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. In *SAT*, Portofino, Italy, May 2003.

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

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

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