# Executable Specifications for Java Programs

by

Aleksandar Milicevic

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
September 3, 2010

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

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

# Executable Specifications for Java Programs

by

Aleksandar Milicevic

Submitted to the Department of Electrical Engineering and Computer Science
on September 3, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

## Abstract

In this thesis, we present a unified environment for running declarative specifications in the context of an imperative object-oriented programming language. Specifications are Alloy-like, written in first-order relational logic with transitive closure, and the imperative language for this purpose is Java. By being able to mix imperative code with executable declarative specifications, the user can easily express constraint problems in-place, i.e. in terms of the existing data structures and objects on the heap. After a solution is found, our framework will automatically update the heap to reflect the solution, so the user can continue to manipulate the program heap in the usual imperative way, without ever having to manually translate the problem back and forth between the host programming environment and the solver language. We show that this approach is not only convenient, but, for certain problems, like puzzles or NP-complete graph algorithms, it can also outperform the manual implementation. We also present an optimization technique that allowed us to run our tool on heaps with almost 2000 objects.

Thesis Supervisor: Daniel N. Jackson
Title: Professor

# Acknowledgments

This research was funded in part by the National Science Foundation under grants 0541183 (Deep and Scalable Analysis of Software), and 0707612 (CRI: CRD Development of Alloy Tools, Technology and Materials), and by the Northrop Grumman Cybersecurity Research Consortium under the Secure and Dependable Systems by Design project.

I would like to thank Daniel Jackson for his expert guidance and all the help I received from him. His clever ideas and numerous suggestions improved this thesis significantly.

I am grateful to Darko Marinov for introducing me to research in software engineering. I learned quite a lot from his unique enthusiasm for research.

Derek Rayside and Robert Seater originally had the idea of executing Alloy-like specifications. They encouraged me to explore this idea further and turn it into a research project. Derek's help throughout all stages of this project was immeasurable. Greg Dennis helped me learn Forge, and Kuat Yessenov patiently answered all my questions about JForge. Their support was indeed invaluable.

And, to my friends and colleagues at MIT: Eunsuk Kang, Jean Yang, Joseph P. Near, Jonathan Edwards, Kuat Yessenov, Rishabh Singh, Sachi Hemachandra, and Sasa Misailovic. They made my day-to-day work fun and enjoyable.

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

SQUANDER is a framework that provides a unified environment for writing declarative constraints and imperative statements in the context of a single program. This is particularly useful for implementing programs that involve computations that are relatively easy to specify but hard to solve algorithmically. In such cases, declarative constraints can be a natural way to express the core computation, whereas imperative code is a natural choice for reading the input parameters, populating the working data structures, setting up the problem, and presenting the solution back to the user. The ability to switch smoothly back and forth between declarative logic and imperative programming makes it possible to implement this kind of program more elegantly and with less effort on the programmer's part.

We propose a technology that can execute declarative specifications without requiring the programmer to write a single line of imperative implementation. The supported specification language is JFSL [35], an Alloy-like [15] language, that supports first-order relational logic with transitive closure, together with Java expressions. The expressive power of JFSL makes it easy to succinctly write complex relational properties in terms of a program's data structures and reachable objects on the heap. Even when the performance of such execution doesn't match a carefully written manual implementation, we have suggested a variety of other applications in which the ability to run declarative specifications is useful [27].

This thesis presents the implementation of our framework for executing declarative specifications, discusses the benefits of the unified environment, and shows several illustrative examples in which the direct execution of a declarative specification outperforms a hand-written implementation. The framework is affectionately named SQUANDER, since it squanders computational resources, running an NP-complete boolean satisfiability (SAT) algorithm to execute all programs – including those that have much lower complexity.

## 1.1 Motivation for having executable specifications

In software development, specifications are traditionally used to formally define a desired behavior of a program, and thus eliminate any ambiguities that a natural-language description might have. Afterwards, it is entirely up to the programmer to implement the system such that it adheres to the given specification. After the implementation has been done, one of the many existing tools can be used to check (to some extent) the implementation against the given formal specification and either verify that the program is correct or discover places where the code violates the specification (i.e. find bugs).

We argue that in the presence of modern hardware and state-of-the-art satisfiability solvers, the performance issues are not the limiting factor anymore. Furthermore, we don't have to sacrifice the expressive power of the specification language either. Instead of trying to derive an imperative code for the given declarative JFSL specification, we translate the problem into a set of boolean constraints, and then run an off-the-shelf SAT solver to search for a solution. Since JFSL is fully translatable into boolean logic, we achieve the full expressive power of the executable specification language.

By being able to mix imperative code with executable declarative specifications, the user can easily express constraint problems in-place, i.e. in terms of the existing data structures and objects on the heap. They can then run our solver, which will find a solution to the given set of constraints (if one exists) and automatically update the heap to reflect the solution. Afterwards, the user can continue to manipulate the program heap in the usual imperative way. Without a technology like this one, the standard solution would be to manually translate the problem into the language of an external solver, run the solver, and then again, manually translate the solution back to the native programming language. This obviously requires more work, it is cumbersome, and after all, it is more error-prone.

## 1.2 Variety of applications

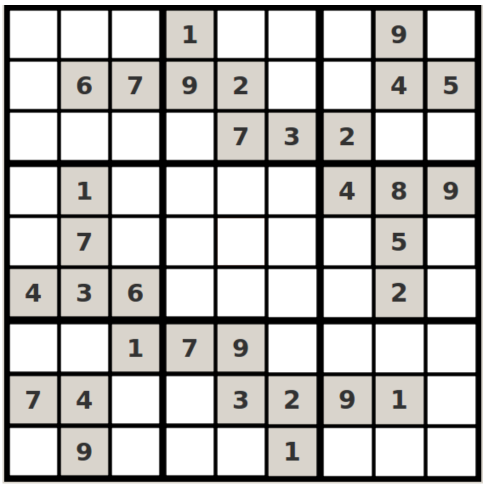
### 1.2.1 Specifying and solving constraint problems

Squander provides a unified environment for writing both declarative constraints and imperative statements. This is particularly useful for implementing algorithmically complex constraint problems within larger programs. Declarative constraints are used to conveniently specify the problem, i.e. to formally state *what* the desired solution should look like and not *how* the solution is to be computed (which is typically difficult for this kind of problems). On the other side, imperative code is a natural choice for the rest of the program, e.g. reading the input parameters, populating the working data structures, setting up the problem, and eventually presenting the solution back to the user. The ability to switch smoothly back and forth between declarative logic and imperative programming, makes it possible to implement this type of programs faster, more elegantly, and with much less effort on the programmer's part.

As an example, consider a simple *Sudoku* solver. The solver is given a partially filled puzzle (e.g. as in Figure 1-1), and is expected to fill out the empty cells with integer numbers such that the following constraints hold:

1. cell values must be in $\{1, 2, \cdots, n\}$ (where $n$ is the dimension of the puzzle – $n = 9$ in this example);

2. all cells within a given row, column, or highlighted sub-grid have distinct values.



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

Figure 1-1: A random Sudoku puzzle

A suitable Java data model for this problem is given in Figure 1-2. A `CellGroup` contains an array of `Cells` with no duplicate values; overlapping `CellGroups` are then defined in the class `Sudoku` for each row, column and sub-grid. The `init()` method (invoked from the `Sudoku`'s constructor) has the task of creating exactly $n \times n$ `Cell` objects, $n + n + n$ `CellGroup` objects, and properly establishing the sharing of `Cells` between `CellGroups`.

Now, onto the solving part. The invariant for `CellGroups`, given above, can be expressed in a single line of JFSL (Listing 1.4). The exact notation will be explained in one of the later sections, but basically, it says that for all integer values $v$ different from 0, select all `Cell` objects from the `CellGroup.cells` field with the value $v$ (`this.cells.elems.value.v`), and ensure that their count is either 0 or 1 (`lone`).

```
@Invariant("all v : int | v != 0 => lone this.cells.elems.value.v")
static class CellGroup {
```

Listing 1.4: CellGroup invariant

Class invariants assert properties of all members of a class, but cannot be executed *per se*. To establish a constraint, we define a standard Java method and annotate it with a specification, which includes:

- a precondition (`@Requires`), on the state before method invocation, assumed `true` if not specified;

- a postcondition (`@Ensures`), on the state after the method has been executed;

- a frame condition (`@Modifies`), indicating what parts of the state the method is allowed to modify.

In this case, the specification for the `solve()` method (Listing 1.5) simply says that in the post-state (i.e. after the method has been executed by SQUANDER), all cells must be filled out with non-zero values. The frame condition limits modifications to those `Cell.value` fields that are currently empty (`[{c: Cell | c.value == 0}]`), since we don't want to modify values given up-front. These constraints, implicitly conjoined with the class invariant, are sufficient to solve the Sudoku puzzle. The method body is simply a call to a utility method, namely `Squander.exe()`, which invokes the solver and attempts to satisfy the specification, by updating the cell values. Following execution of `solve`, assuming a solution is found, the program may proceed to, for example, print out the solved puzzle, using the usual imperative paradigm. Otherwise, an exception is thrown to signal that the specification could not be satisfied.

```
@Ensures("all c : Cell | c.value > 0 && c.value <= this.n")
@Modifies("Cell.value [{c : Cell | c.value == 0}]")
public void solve() {
  Squander.exe(this);
}
```

Listing 1.5: Spec for Sudoku.solve() method

### 1.2.2 Specifying data structures

Due to their high expressive power, first-order declarative specifications can be used to succinctly and formally specify operations on data structures. On the other side of the spectrum, there is a low-level, imperative implementation of the corresponding program, an algorithm that enumerates the steps needed to satisfy the high-level specification – this is usually a non-trivial task. For example, operations like deletion from a balanced tree or finding the longest path in a graph are much easier to specify in a declarative manner than to implement in Java.

Here we show how a Binary Search Tree (BST) can be implemented with SQUANDER. A BST has a single root node. Every node contains an integer key and pointers to its left and right nodes (pointers may be `null`). In order to be a valid BST, the representation invariant must hold: (1) there are no loops, and (2) for every node $n$ in the BST, the key of $n$ is strictly greater than all keys of all nodes in its left subtree and strictly less than all keys of all nodes in its right subtree.

```
@SpecField("this.nodes = this.root.*(left+right) - null")
public class BST {
  @Invariant({
   /* form a tree   */ "this !in this.^parent",
   /* left sorted   */ "all x: this.left.*(left+right) - null | x.key < this.key",
   /* right sorted */ "all x: this.right.*(left+right) - null | x.key > this.key"
  })
  public static class Node {
    Node left, right;
    int key;
  }

  private Node root;
}
```

Listing 1.6: Binary Search Tree skeletal code

The structure of the BST example is given in Listing 1.6. Specification statements (written as Java annotations) define class specific properties and contracts. The specification field named `this.nodes` is introduced here as a shortcut and will be used later to refer to all reachable nodes for a given tree. Reflexive transitive closure operator (`.*`) is used to conveniently specify all reachable nodes starting from the `root` node. The representation invariant for the class `Node` is written next. Note that this formal definition written in JFSL follows quite closely the informal, natural-language definition given above.

Having defined class invariants, we can now write specifications for some methods. Formal specifications for node insertion and node deletion are shown in Listing 1.7. They are both quite obvious and intuitive. In order to insert a node in a tree, before the insertion, a node with the same key may not exist in the tree, and after the insertion, the tree must contain the given node. Deletion is defined similarly. It is implicitly taken that the class invariant must hold before and after the execution of any method.

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes + z)")
@Modifies("Node.left, Node.right, this.root")
public void insert(Node z) {
  Squander.exe(this, z);
}

@Requires("z in this.nodes")
@Ensures("this.nodes = @old(this.nodes) - z")
@Modifies("Node.left, Node.right, this.root")
public void remove(Node z) {
  Squander.exe(this, z);
}
```

Listing 1.7: Specification for the `insert` and `delete` methods

In case of complex data structures, like this one, the class invariant is what makes the manual implementation tricky. Complex data structures have non-trivial invariants, and the task of writing the implementation that maintains the invariant throughout the program execution (after every method invocation) is known to be difficult and prone to errors [20]. For example, implementing the insertion in a BST is certainly not trivial, but it is not too difficult either, because we know that the new node ought to be inserted at one of the leaf positions. However, an imperative procedure for the node removal operation is be notoriously painful, because of so many corner cases, all of which must be handled separately. With SQUANDER, the specifications given in Listing 1.7 are sufficient to execute node insertion or deletion on any binary tree.

Clearly, neither the specification for `insert` nor `delete` method can be executed as efficiently a carefully written imperative algorithm. However, performance is usually acceptable for unit-test size examples. For example, given a tree with 15 nodes, it takes a couple of seconds for SQUANDER to insert or delete a node, which can be quite useful during the early stages of the development process, including design, testing, fast prototyping, and similar.

### 1.2.3 Other applications

The technology that allows us to execute specifications in the context of an imperative programming language, also permits many other applications. To name a few, the same technology can be used for:

- **test input generation**: to generate test inputs (say valid Binary Search Trees), one could define a method that returns a BST, and optionally add additional constraints in the postcondition part of the specification (e.g. the resulting tree must have between $n$ and $m$ nodes). Typically, SQUANDER would non-deterministically find a solution that satisfies all constraints, but for this purpose, the implementation could easily be altered so that it keeps finding all possible solutions, or at least up to some number of satisfying solutions.

- **differential testing**: once we have both a specification for a method and an implementation, we would like to check whether they correspond to each other. One way would be to use a verification tool to prove that the two match. However, that might be too slow, or even intractable if the implementation is quite complicated. A more lightweight approach would be to run both the specification and the implementation on a suite of test cases and check whether the results match.

- **specification validation**: this is another interesting practical application. Specification can also contain errors, and the most intuitive way to test a specification would be to execute it on some concrete input and see if the result makes sense or not.

- **runtime assertion checking**: obviously, SQUANDER can be used just to check whether a given rich property holds at an arbitrary point during the execution of a program.

```
static class Cell {
  int value = 0; // 0 means empty
}
```

Listing 1.1: class Cell

```
static class CellGroup {
  Cell[] cells;
  public CellGroup(int n) {
    this.cells = new Cell[n];
  }
}
```

Listing 1.2: class CellGroup

```
public class Sudoku {
  private final int n;
  private CellGroup[] rows;
  private CellGroup[] cols;
  private CellGroup[] grids;

  public Sudoku(int n) {
    assert Math.sqrt(n) * Math.sqrt(n) == n : "n must be a square number";
    this.n = n;
    init();
  }

  private void init() {
    this.rows = new CellGroup[n];
    this.cols = new CellGroup[n];
    this.grids = new CellGroup[n];
    for (int i = 0; i < n; i++) {
      rows[i] = new CellGroup(n);
      cols[i] = new CellGroup(n);
      grids[i] = new CellGroup(n);
    }
    int m = (int) Math.sqrt(n);
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        Cell c = new Cell();
        rows[i].cells[j] = c;
        cols[j].cells[i] = c;
        int gridI = i / m;
        int gridJ = j / m;
        int gridIdx = gridI * m + gridJ;
        int gridCellI = i % m;
        int gridCellJ = j % m;
        int gridCellIdx = gridCellI * m + gridCellJ;
        grids[gridIdx].cells[gridCellIdx] = c;
      }
    }
  }
}
```

Listing 1.3: class Sudoku

Figure 1-2: Sudoku data model

# Chapter 2

# Background

## 2.1 Kodkod – A solver for relational logic

Kodkod [31–33] is an efficient constraint solver for relational logic. It requires a bounded universe, a set of untyped relations, bounds for every relation, and a relational formula. It then translates the given problem to a boolean satisfiability problem and applies an off-the-shelf SAT solver to search for a satisfying solution, which, if found, is finally translated back to the relational domain.

When created, relations in Kodkod are untyped, meaning that every relation can potentially contain any tuple drawn from the finite universe. However, it helps to think about the relation types from the beginning, just to have a sense of which atoms each relation can potentially contain. The actual set of tuples that a relation may contain is defined through Kodkod *bounds*. Two bounds need to be specified: *lower bound* to define tuples that a relation **must** contain, and *upper bound* to define tuples that a relation **may** contain. The size of these bounds is what primarily influences the search time – the fewer tuples there are in the difference of the upper and the lower bound, the smaller the search space is, the faster the solving is.

## 2.2 JFSL – JForge Specification Language

JFSL [35] is a formal lightweight specification language for Java. It supports relational and set algebra, as well as common Java operators. With the expressive power of relational algebra, JFSL makes it easy to succinctly and formally specify complex properties about Java programs, such as class invariants, method pre and post conditions, as well as method *frame conditions* (the portion of the heap the method is allowed to modify). It also supports *specification fields* which can be particularly useful for specifying abstract data types.

### 2.2.1 JFSL expressions

As in Alloy, all expressions in JFSL evaluate to relations. JFSL provides common relational algebra operators, together with integer and boolean operators. These are summarized in Tables 2.1 and 2.2. Quantified expressions are also supported. The list of supported quantifiers is shown in Table 2.3.

| Operator | Description |
|---|---|
| `~` | relational transpose or bitwise negation |
| `^` | transitive closure |
| `*` | reflexive transitive closure |
| `#` | set cardinality |
| `no` | "no" multiplicity (empty) |
| `lone` | "lone" multiplicity (zero or one) |
| `one` | "one" multiplicity (exactly one) |
| `some` | "some" multiplicity (one or more) |
| `!` | boolean negation |
| `-` | integer negation |
| `sum` | integer summation |

Table 2.1: Unary expressions supported by JFSL

### 2.2.2 JFSL annotations

JFSL specifications are written as Java annotations. These annotations are briefly summarized below:

- `@Invariant` — attached to classes and used to define conditions that must always (before and after every execution of a method) hold true for the given class.

- `@Requires` — attached to methods and used to specify constraints on the state before method invocation. The method is expected to execute correctly only if the precondition is satisfied immediately before invocation. Class invariants are implicitly added to method preconditions.

- `@Ensures` — attached to methods and used to specify constraints on the state after method invocation. In other words, it captures all effect the method is expected to produce. Class invariants are implicitly added to method postconditions.

- `@Modifies` — attached to methods and used to specify frame condition. For this thesis, we enhanced the frame condition annotation to hold up to 4 different pieces of specification (syntax: `@Modifies("f [s][l][u]")`). The first, and the only mandatory piece, is the name of the modifiable field, `f`. It is optionally followed by the *instance selector* (`s`), the *lower bound* and the *upper bound*. The instance selector specifies instances for which the field may be modified (assumed "all" if not specified). The lower bound (assumed "empty" if not specified) contains concrete field values for some objects in the post-state. The upper bound of the modification (assumed the extent of the field's type if not specified) holds possible fields values in the post-state.

- `@SpecField` — attached to classes and used to define specification fields. Definition of a specification field consists of a type declaration, and (optionally) an abstraction function. The abstraction function defines how the field value is computed in terms of other fields. For example, `@SpecField("x: one int | x = this.y - this.z")` defines a singleton integer field `x`, the value of which must be equal to the difference of `y` and `z`. Specifications fields are inherited from super-types and sub-types can override the abstraction function (by simply redefining it), a feature that is particularly useful for specifying abstract datatypes, such as Java collections.

In the context of executable specifications, the goal is to execute a method based on its specification. That assumes making modifications to the modifiable portion of the heap (as specified by the frame condition) so that the final state satisfies the postcondition.

| Operator | Description |
|---|---|
| `@+` | relational union |
| `@-` | relational difference |
| `@&` | relational intersection |
| `.` | relational join |
| `->` | relational product (tuple constructor) |
| `++` | relational override |
| `=`, `==` | relational equality |
| `in` | relational subset |
| `!in` | relational not-subset |
| `+` | integer addition or set union |
| `-` | integer subtraction or set difference |
| `*` | integer multiplication |
| \ | integer division |
| `%` | integer modulo division |
| `<` | integer less than |
| `>` | integer greater than |
| `<=` | integer less than or equal |
| `>=` | integer greater than or equal |
| `&` | integer bitwise "and" or set intersection |
| `\|` | integer bitwise "or" |
| `<<` | integer bit shift left |
| `>>` | integer bit shift right |
| `>>>` | integer unsigned bit shift right |
| `&&` | boolean conjunction |
| `\|\|` | boolean disjunction |
| `^^` | boolean exclusive disjunction |
| `=>` | boolean implication |
| `<=>` | boolean equivalence ("if and only if") |
| `?` | if-then-else ternary operator (as in Java) |

Table 2.2: Binary and ternary expressions supported by JFSL

| Quantifier | Description | Usage |
|---|---|---|
| `all` | universal quantifier | `all x: T \| P(x)` |
| `some` | existential quantifier | `some x: T \| P(x)` |
| `sum` | integer summation | `sum i: int \| a[i]` |
| `union` | set comprehension | `{x: T \| P(x)}` |

Table 2.3: Quantified expressions supported by JFSL

# Chapter 3

# From Object Heap to Relational Logic

Squander execution begins when the utility method `Squander.exe()` is called by the client code. In overview, execution involves the following steps:

- Assembling the relevant constraints, from the annotations comprising the method's specification, as well as class annotations corresponding to invariants of all relevant classes (determined by a traversal of the heap from the receiver object).
- Construction of relations representing the values of objects and their fields in the pre-state, and additional relations for modifiable fields to represent their values in the post-state, along with their Kodkod bounds;
- Parsing of the constraints and conversion to a single relational formula (handed to Kodkod for solving);
- If a solution is found, translation of the Kodkod result objects into updates of the Java heap state, by modification of the object fields.

## 3.1 Heap traversal and object serialization

The first concern for Squander is discovering the reachable portion of the heap. The traversal algorithm is a standard breath-first algorithm (although depth-first would suffice too), starting from a given set of root objects (the caller instance plus method arguments) and repeatedly visiting all children until all reachable objects have been visited. The interesting part is how to enumerate children, i.e. how to serialize a given object into a set of field values.

Squander provides a generic mechanism that allows for different object serializers based on the object's class. For example, the default object serializer simply returns values of an object's fields. This behavior is good in many cases, including user-defined classes. However, when serializing abstract types – such as an object of type `java.util.Set` – we would like to return only the members of the set, excluding objects that are artifacts of the representation (such as hash buckets). An *abstraction function* is needed to separate the actual content from the internal representation, and this is exactly what object serializers provide. Similarly, they also provide *concretization functions* that are used to restore an object's state from a given set of abstract values returned by the solver. Through this mechanism, SQUANDER provides support for Java collections and Java arrays (more details in Chapter 6), and allows users to easily customize behavior for user-defined abstractions.

### 3.1.1 Keeping track of type parameters

Java collection classes make extensive use of parametric types (also called "generics" in Java terminology). This lets the programmer declare the type of objects a collection is allowed to contain, e.g. "set of nodes" (`Set<Node>`) as opposed to "set of any objects" (`Set`). Unfortunately, however, since generics were a late addition to Java, they are implemented using *type erasure*, and the parameter information is only available at compile time. For ease of use, SQUANDER is a runtime mechanism that uses the standard JVM, so it has no access to the compile-time type information.

Knowing the exact types of objects, including type parameters, is important though. One reason is that we don't want to have to write explicit casts in our specifications every time we refer to an element of a collection (as one must do in Java when not using generics). Other reasons are mainly concerned about performance: without knowing its type parameter, the extent of a *any* set is always a set of all objects. If the set is actually a set of integers, than the actual extent is much smaller, which would result in a smaller bound if the set was modifiable (as explained in Section 3.3), which could dramatically improve the Kodkod's running time.

Java reflection does, however, provide static types of fields and method parameters, include type parameter information. For example, if there is a field declared as `Set<Node> nodes`, the fact that the field is a "set of Nodes" can be obtained at runtime. Consequently, if we know that some object `obj` was read as a value of the field `nodes`, we can conclude that the type of `obj` is actually `Set<Node>`. Almost all objects during the heap traversal are discovered by reading field values. It is only the caller instance whose origin is not known; all other objects are either passed as method parameters or read as field values, so complete type information can be obtained for all objects but the root.

## 3.2 Reading, parsing and type-checking JFSL specifications

When a new class is discovered during heap traversal, its specification is obtained by reflection. The specification of a class includes *class invariants* (`@Invariant`) and *specification fields* (`@SpecField`). These can be specified either directly in the source file using Java annotations or through a special *spec file*, which must be found in the classpath and whose name must correspond to its target class's full name. Next, text-based specifications are parsed and type-checked (e.g. to make sure that all identifiers can be resolved to actual classes/fields in the program, and that expressions have expected types, etc.) and eventually translated into relational expressions. For most of this task, SQUANDER borrows functionality from JForge [35].

## 3.3 Defining relations and bounds

Having traversed the heap, found all reachable objects, and discovered all classes/-fields referred to in the specification, we are ready to construct the relations that represent the state of the heap.

The translation does not use all fields, but rather considers only *relevant* fields, i.e. those that are explicitly mentioned in the specification for the current method. Similarly, not all reachable objects are needed; only objects reachable by following the relevant fields are included in the translation. These objects will be referred to as *literals*.

First we define a finite universe consisting of all literals, plus integers within the bound. For every literal, a unary relation is created. These relations are constant, i.e. they are given an exact bound (lower and upper bounds are equal) of a single unary tuple containing the corresponding literal.

For each Java type, one could either create a new relation (with appropriate bounds so that it contains the known literals), or one could construct a relational expression denoting the union of relations corresponding to all instance literals of that type. In our implementation, we took the former approach, since it results in more readable expressions (which helps debugging the framework) and has no performance impact.

For every field (including specification fields), a relation of type `fld`.*declType* $\rightarrow$ `fld`.*type*[1] is created to hold assignments of field values to objects. If the field is modifiable (inferred from its mention in a `@Modifies` clause), an additional relation is created, with the suffix "pre" appended to denote the pre-state value. Relations for unmodifiable fields are given an exact bound that reflects the current state of the heap. For the modifiable relations, the "pre" relation is given the same exact bound, and the "post" relation is bounded so that it may contain any tuple permitted by the field's type. Local variables, such as `this`, `return`, and method arguments are treated similarly to literals.

Table 3.1 summarizes how relations and bounds are created. Function `rel` takes a Java element and, depending whether the element is modifiable, returns either one or two relations (the "`[]`" notation means "list of", and **R** is the constructor for relations, taking a name and a type for the relation). Function `bound` takes a Java element and its corresponding relation, and returns a bound for the relation. The `Bound` data type contains both lower and upper bounds. If only one expression is passed to its constructor (**B**), both bounds are set to that value. Helper functions `is_mod`, `is_post` and `fldval` are used to check whether a field is modifiable, to check whether a relation refers to the post-state, and to return a literal that corresponds to the value of a given field of a given literal, respectively.

[1]`fld`.*declType* is the declaring class of `fld`

```
rel :: Element → [Relation]

rel (Literal lit)                    =   [R(lit.name, lit.type)]
rel (Type t)                         =   $\bigcup_{\text{lit}<:\text{t}}$ rel lit
rel (Field fld)                      =
    if is_mod(fld)
        [R(fld.name, fld.declType → fld.type)] ++
        [R(fld.name + "_pre", fld.declType → fld.type)]
    else
        [R(f.name, f.declType → f.type)]
rel (Local var)                      =   [R(var.name, var.type)]

bound :: Element, Relation → Bound

bound (Literal lit) (Relation r)     =   B(lit)
bound (Field fld) (Relation r)       =
    if is_mod(fld) $\wedge$ is_post(r)
        B({}, ext(fld.declType $\times$ fld.type))
    else
        B($\bigcup_{\text{lit: Object}}$ lit $\times$ fldval(lit, fld))
bound (Return ret) (Relation r)      =   B({}, ext(ret.type))
bound (Local var) (Relation r)       =   B(var)

ext :: [Type] → Expression                         (helper)

ext []                               =   {}
ext (t : [])                         =   $\bigcup_{\text{lit}<:\text{t}}$lit
ext (t : xs)                         =   ext t $\times$ ext xs
```

Table 3.1: Translation of different Java constructs into relations (function `rel`) and bounds (function `bound`)

## 3.4 Example: translation of `BST.insert`

To illustrate translation, consider the Binary Search Tree example introduced in Section 1.2.2. From the snapshot of the heap shown in Figure 3-1, we see that the class `BST` contains a single pointer to the root node, and the `Node` class contains pointers to left and right sub-trees, as well as a single integer value (field `key`). When a node is inserted, all node pointers may potentially be modified, so the specification for the `insert` method declares fields `root`, `left`, and `right` as modifiable.

The resulting set of relations is shown in Table 3.2. Relations in the upper section are unary, unmodifiable relations, and represent objects found on the heap. The middle section contains relations that are also unmodifiable, because they are used to either represent unmodifiable fields or values in the pre-state of modifiable fields. Finally, the relations in the bottom section represent the post-state of modifiable fields; these are the relations for which the solver will attempt to find appropriate values. By default, the lower bound is simply set to an empty set and the upper bound is the upper bound is set the extent of the field's type.

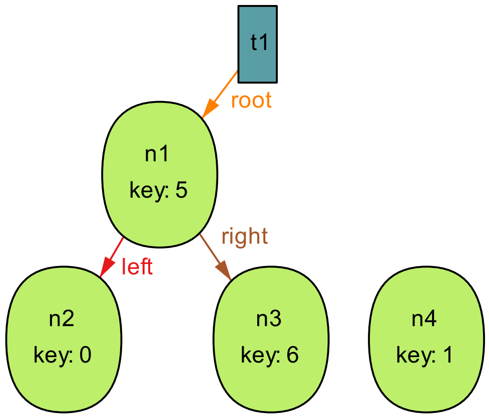


Figure 3-1: A snapshot for the pre-state of `t1.insert(n4)`

| | |
|---|---|
| **BST**: | $\{t_1\}$ |
| **N1**: | $\{n_1\}$ |
| **N2**: | $\{n_2\}$ |
| **N3**: | $\{n_3\}$ |
| **N4**: | $\{n_4\}$ |
| **null**: | $\{null\}$ |
| **BST_this**: | $\{t_1\}$ |
| **z**: | $\{n_4\}$ |
| **ints**: | $\{0, 1, 5, 6\}$ |
| **key**: | $\{(n_1 \rightarrow 5), (n_2 \rightarrow 0), (n_3 \rightarrow 6), (n_4 \rightarrow 1)\}$ |
| **root_pre**: | $\{(t_1 \rightarrow n_1)\}$ |
| **nodes_pre**: | $\{(t_1 \rightarrow n_1), (t_1 \rightarrow n_2), (t_1 \rightarrow n_3), (t_1 \rightarrow n_4)\}$ |
| **left_pre**: | $\{(n_1 \rightarrow n_2), (n_2 \rightarrow null), (n_3 \rightarrow null), (n_4 \rightarrow null)\}$ |
| **right_pre**: | $\{(n_1 \rightarrow n_3), (n_2 \rightarrow null), (n_3 \rightarrow null), (n_4 \rightarrow null)\}$ |
| **root**: | $\{\}, \quad \{t_1\} \times \{n_1, n_2, n_3, n_4\}$ |
| **nodes**: | $\{\}, \quad \{t_1\} \times \{n_1, n_2, n_3, n_4\}$ |
| **left**: | $\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$ |
| **right**: | $\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$ |

Table 3.2: Translation of the heap from Figure 3-1

## 3.5 Tightening the bounds

By declaring fields `left` and `right` as modifiable, as in the specification for `BST.insert` (Listing 1.7), we allow arbitrary modifications to the tree, as long as all constraints are satisfied. In effect, after the execution of the specification for the `insert` method, the tree will contain all old nodes plus the new node, but the shape of the tree may randomly change.

If we wanted to change the specification so that the tree topology is preserved and the new nodes can only be inserted at leaf positions, we could manually add additional clauses to the postcondition, specifying that left and right pointers of certain nodes must remain the same in the post state. However, a better (more efficient) approach would be to modify the frame condition to tighten the bounds for the `left` and `right` pointers, thus reducing the size of the search space and potentially significantly improving performance, as previously reported by Samimi et al. [28].

Consider the modified frame condition shown in Listing 3.1. This frame condition now specifies additional constrains on the modification of the `left` and `right` fields. It says that the value of the left pointer may change only for those nodes for which the value of the left pointer is currently set to `null` (and similarly for the right pointers). This way we make sure that all nodes are inserted at the leaf positions, just like a manual implementation would do.

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes + z)")
@Modifies({
  "Node.left  [{n: this.nodes | n.left == null}]",
  "Node.right [{n: this.nodes | n.right == null}]",
  "this.root"})
public void insert(Node z) {
  Squander.exe(this, z);
}
```

Listing 3.1: Modified frame condition for the `insert` method

To implement this new feature, a straightforward approach would be to automatically generate and add the following constraints to the method's postcondition.

```
all n: Node | n !in {n: this.nodes | n.left == null} => n.left = @old(n.left)
all n: Node | n !in {n: this.nodes | n.right == null} => n.right = @old(n.right)
```

Listing 3.2: Translation of the instance selector clause into constraints

As explained above, this would be an inefficient implementation. Instead, SQUANDER evaluates the instance selector clause (e.g. {`n: this.nodes | n.left == null`}) against the current heap (without running the solver) to obtain the list of modifiable objects (the evaluation mechanism is similar to that of MintEra [2]). With the list of modifiable objects, SQUANDER modifies the bounds for the corresponding field relation so that the current field values of the objects not to be modified are included in the lower bound, thus forcing the value to stay the same in the post state. For the heap shown in Figure 3-1, the modifiable objects for the `left` field are `n2` and `n3`, because their left pointers are currently set to null. Similarly, for the `right` field, the modifiable objects are also `n2` and `n3`. The updated bounds for these two fields are shown in Table 3.3. For the larger trees, this can make a big difference in scalability, as will be shown later in Section 7.2.4.

| | | |
|---|---|---|
| **left**: | $\{(n_1 \to n_2), (n_4 \to null)\}$, | $(n_1 \to n_2) \cup (n_4 \to null) \cup \{n_2, n_3\} \times \{n_1, n_2, n_3, n_4\}$ |
| **right**: | $\{(n_1 \to n_3), (n_4 \to null)\}$, | $(n_1 \to n_3) \cup (n_4 \to null) \cup \{n_2, n_3\} \times \{n_1, n_2, n_3, n_4\}$ |

Table 3.3: The updated bounds for the `left` and `right` relations

# Chapter 4

# Minimizing the Universe Size

To represent a relation $r$ of arity $k$, Kodkod allocates a matrix of size $n^k$, where $n$ is the number of atoms in the universe. Consequently, if the universe contains more than 1290 atoms, a ternary relation would contain $1291^3$ cells. These cells are stored (for performance) in a single sequential array, indexed by a Java integer, and since $1291^3$ is greater than the largest integer value in Java (`Integer.MAX_VALUE`), the relation cannot be represented.

In practice, this can be a problem. SQUANDER makes frequent use of ternary relations (e.g. for representing arrays, lists and maps), and heaps with more than 1290 objects are not uncommon for problems that we would like to be able to solve with SQUANDER, so a simple translation like the one described in Section 3.3 (which simply creates a new atom for every object it finds on the heap) is not feasible. As an illustration, in our case study on a course scheduling application for the MIT undergraduate degree program (explained in detail in Section 8), the heap contains more than 1900 objects.

In this chapter, we describe a different translation technique (named *KodkodPart*) that we developed to minimize the number of atoms that is used to represent the object heap in the relational world of Kodkod.

## 4.1 *KodkodPart* translation

Our KodkodPart translation achieves a universe with fewer atoms by establishing a mapping from Java objects (also called literals, as in Section 3.3) to Kodkod atoms which is not necessarily an *injective* function. In other words, multiple literals are allowed to map to a single atom, so that there can be fewer atoms than literals. The key requirement is, however, that there exists (in the larger context) an inverse function from atoms back to literals, so that the heap can be properly restored after a solution has been found. This inverse function, we will see, can be contructed with the help of available type information.

Consider the tree insertion example, shown in Figure 3-1. Domains $\mathcal{D}$, literals $\mathcal{L}$, and assignments of literals to domains $\gamma : \mathcal{D} \rightarrow \mathcal{P}(\mathcal{L})$ for this example are summarized in Table 4.1.

$$\begin{aligned}
\mathcal{D} &= \{\texttt{BST},\ \texttt{Node},\ \texttt{Null},\ \texttt{Integer}\}\\
\mathcal{L} &= \{bst_1,\ n_1,\ n_2,\ n_3,\ n_4,\ null,\ 0,\ 1,\ 5,\ 6\}\\
\gamma(\texttt{BST}) &= \{bst_1\}\\
\gamma(\texttt{Node}) &= \{n_1,\ n_2,\ n_3,\ n_4\ \}\\
\gamma(\texttt{Null}) &= \{null\}\\
\gamma(\texttt{Integer}) &= \{0,\ 1,\ 5,\ 6\}
\end{aligned}$$

Table 4.1: Summary of domains and instances for the `BST.insert` example

Recall that field types are represented as unions of base types (in this section also called *partitions*). For instance, the type of the field `BST.root` is `BST` → `Node` ∪ `Null`, because values of this field can be either instances of `Node` or the `null` constant. That means that all objects of class `Node` plus the constant `null` must be mapped to different atoms, so that it is possible to unambiguously restore the value of the field `root`. This is the basic idea behind the KodkodPart translation: *all literals within any given partition must be mapped to different atoms, whereas literals not belonging to a common partition may share atoms.* The inversion function can then work as follows: for a given atom, first select the correct partition based on the type of the field being restored, then unambiguously select the corresponding literal from that partition.

To complete the example, the set of all unary types used in the specification for this example is:

$$\mathcal{T} = \{\texttt{BST},\ \texttt{BST} \cup \texttt{Null},\ \texttt{Node},\ \texttt{Node} \cup \texttt{Null},\ \texttt{Null},\ \texttt{Integer}\}$$

This set is discovered simply by keeping track of types of all relations created for Java fields, and it automatically becomes the set of our partitions. A valid assignment of atoms to literals that uses only 5 atoms, as opposed to 10 which is how many the original translation would use, could be:

| | | | | |
|---|---|---|---|---|
| $bst_1 \rightarrow a_0$ | $n_1 \rightarrow a_0$ | $n_2 \rightarrow a_1$ | $n_3 \rightarrow a_2$ | $n_4 \rightarrow a_3$ |
| $null \rightarrow a_4$ | $0 \rightarrow a_0$ | $1 \rightarrow a_1$ | $5 \rightarrow a_2$ | $6 \rightarrow a_3$ |

As a limitation of this technique, if the class `Object` is used as a field type, or anywhere in the specification, it will result in one big partition containing all literals (because every class is a subclass of `Object`), making the algorithm equivalent to the original translation.

## 4.2 Partitioning algorithm

For a given set of base domains $\mathcal{D}$, literals $\mathcal{L}$, and partitions $\mathcal{T}$ ($\mathcal{T} = \mathcal{P}(\mathcal{D})$), and a given function $\gamma : \mathcal{D} \rightarrow \mathcal{P}(\mathcal{L})$ that maps domains to their instance literals, this algorithm produces a set of atoms $\mathcal{A}$ and a function $\alpha : \mathcal{L} \to \mathcal{A}$, such that for every partition $p$, function $\alpha$ returns different values for all instance literals of $p$. Formally:

$$(\forall p \in \mathcal{T})(\forall l_1, l_2 \in \psi(p))\ l_1 \neq l_2 \implies \alpha(l_1) \neq \alpha(l_2)$$

where $\psi$ is a function that for a given partition returns a comprehension of all instance literals of all of its domains:

$$\psi : \mathcal{T} \to \mathcal{P}(\mathcal{L}); \quad \psi(p) = \{\gamma(d) \mid d \in p\}$$

Obviously, a simple bijection would satisfy this specification, but such a solution wouldn't achieve its main goal, which is to minimize the number of atoms, because the number of atoms in this case would be exactly the same as the number of literals. In order to specify solutions that are actually "useful", we are going to require the algorithm to produce a result such that the cardinality of $\mathcal{A}$ (i.e. the total number of atoms) is minimal.

It is not immediately clear what the minimal number of atoms ought to be. One might think that no more atoms are required than the number of instances in the largest partition. However, this is not always true. Consider the case shown in Figure 4-1. The largest partitions are `P1` and `P4`, both having 5 literals. On the other hand, any pair of the domains `B`, `C`, and `D` have a partition in common, even though there is no single partition containing them all. They thus form a strongly connected component, and their literals must differ. There are 6 literals in total in these three domains, so 5 atoms cannot be enough. As a conclusion, the minimal number of atoms is indeed the number of literals in the largest partitions, but only after all strongly connected domains have been merged into a single partition.

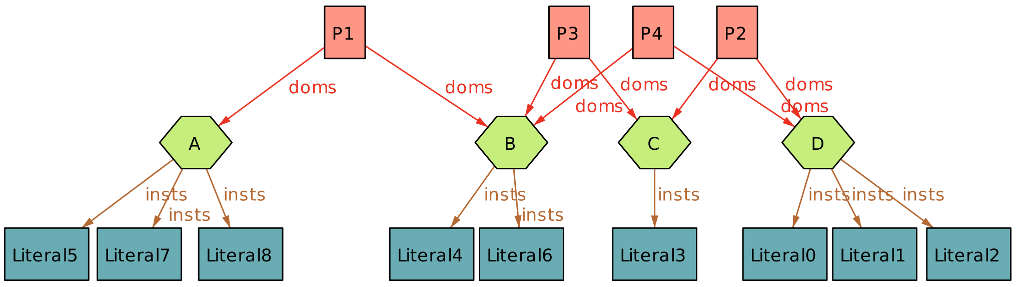


Figure 4-1: KodkodPart: an example where more than the number of literals of the largest partition is needed.

Luckily, cases like the one in Figure 4-1 never happen in Squander, so our implementation of the algorithm doesn't have to search for cliques and merge partitions. The reason this never happens is that domains are always Java classes, and partitions are types used to represent fields. A type of a field is a union type which includes the entire subclass hierarchy of the field's base type. For instance, if `C` and `D` are Java classes, `C` extends `D` (`C <: D`), and some field has declared type $D$, then the type of the field (in the relational world) will be `D` ∪ `C` ∪ `Null`, meaning that `D` ∪ `Null` is never going to be used as a partition for anything.

In summary, the actual implementation inside Squander works as follows:

1. Dependencies between domains are computed. A domain depends on all domains with which it shares a partition. Let the function $\delta : \mathcal{D} \rightarrow \mathcal{P}(\mathcal{D})$ express this:
$$\delta(d) = \{d_1 \mid d_1 \neq d \wedge ((\exists p \in \mathcal{T})\ d_1 \in p)\}$$
2. The largest partition $p_{max}$ is found such that
$$(\nexists p \in \mathcal{T})\ |\psi(p)| > |\psi(p_{max})|$$
3. For every literal $l$ in $\psi(p_{max})$ an atom $a$ is created, it is added to the universe $\mathcal{A}$ and assigned to $l$, such that $\alpha(l) = a$. From this point onwards, $\mathcal{A}$ is fixed.
4. For every other partition $p$ iteratively, for all literals $l_p \in p$ that do not already have an atom assigned, a set of possible atoms $\mathcal{A}_{l_p}$ is computed and the first value from this set is assigned to $l_p$. $\mathcal{A}_{l_p}$ is computed when atoms corresponding to all literals of all dependent domains is subtracted from $\mathcal{A}$, i.e.:
$$\mathcal{A}_{l_p} = \mathcal{A} \setminus \{\alpha(l) \mid l \in \mathcal{L}_d\}\text{ , where}$$
$$\mathcal{L}_d = \{\gamma(d) \mid d \in \mathcal{D}_d\}\text{ , where}$$
$$\mathcal{D}_d = \delta(d_l)\text{, where } d_l \in \mathcal{D} \wedge l_p \in \gamma(d_l)$$

# Chapter 5

# Specification Fields and the Abstract State

This chapter discusses how specification fields can be used to model abstract state in a modular fashion. It also explains how SQUANDER is capable of maintaining and managing the abstract state throughout the program execution. As a result, specification fields don't have to be defined in terms of the concrete state (by means of an abstraction function), i.e. they can be left purely abstract. This is particularly useful for implementing mock objects [5, 21]. For example, an abstract data type declares a certain number of specification fields to abstractly model the data type. It can't provide an abstraction function, because it is not aware of the concrete representation that will be used to represent the state. However, it can still formally specify the operations of the data type just in terms of its specification fields. Before a concrete implementation has been developed, SQUANDER can be used to run the abstract class just as if it were a regular Java class with concrete fields. By keeping track of the abstract state between method calls, and properly updating the abstract state after each method call, SQUANDER is essentially providing a mock implementation on the fly. After they have been developed, the concrete implementations of the abstract data type can simply override the definitions of the specification fields, and have their concrete state automatically be updated.

## 5.1 Mock objects example

Consider the implementation of an integer set data structure given in Listing 5.1. The base class, `IntSet`, declares a single specification field, namely `elems`, to model the content of the set, and provides the specification for several common set operations. Semantically, `IntSet` should be an abstract class, but we did not to declare it "abstract" in the Java sense, because we wanted to be able to create instances of `IntSet` and show that even though they have no concrete fields to store the actual elements of the set, with SQUANDER, they are ready to be used as mock objects (as in Listing 5.2), since SQUANDER will maintain the content of the abstract `elems` field for them.

```
@SpecField("elems: set int")
public /*abstract*/ class IntSet {
  public IntSet() { init(); }

  @Ensures("no this.elems")
  private void init()                { Squander.exe(this);  }

  @Ensures("return = e !in @old(this.elems) && this.elems = @old(this.elems) @+ e")
  @Modifies("this.elems")
  public boolean add(int e)          { return Squander.exe(this, e); }

  @Ensures("return = e in this.elems")
  public boolean contains(int e)     { return Squander.exe(this, e); }

  @Ensures("return = s.elems in this.elems")
  public boolean containsAll(IntSet s) { return Squander.exe(this, s); }

  @Ensures("return = e in @old(this.elems) && this.elems = @old(this.elems) @- e")
  @Modifies("this.elems")
  public boolean remove(int e)       { return Squander.exe(this, e); }

  @Ensures("return.elts = this.elems")
  @FreshObjects(cls=Set.class, typeParams={Integer.class}, num=1)
  @Modifies("return.elts")
  public Set<Integer> nodes()        { return Squander.exe(this); }
}
```

Listing 5.1: IntSet class

```
public static void main(String[] args) {
  IntSet s1 = new IntSet();
  IntSet s2 = new IntSet();
  s1.add(2);  s1.add(3);  s1.add(4);
  s2.add(3);  s2.add(2);
  System.out.println(s1.containsAll(s2)); // prints "true"
  s1.remove(2);
  System.out.println(s1.containsAll(s2)); // prints "false"
}
```

Listing 5.2: IntSet mock objects

Now, we can provide a concrete `IntSet` implementation backed with the `Set<Integer>` class (Listing 5.3). This class simply overrides the definition of the `elems` specification field and assigns an abstraction function to it to establish the correspondence between the specification field and the concrete Java field. It also defines the frame for the specification field (the "from" clause) so that the fields from the frame are automatically added to the list of modifiable fields when the specification field is declared as modifiable. Nothing else has to be changed. If we modify the `main` method from Listing 5.2 so that objects of class `SetIntSet` are created instead, calls to `insert` and `remove` will not modify the abstract state anymore, but will actually modify the `mySet` field.

```
@SpecField("elems: set int from this.mySet.elts | this.elems = this.mySet.elts")
public class SetIntSet extends IntSet {
  private Set<Integer> mySet = new HashSet<Integer>();
}
```

Listing 5.3: SetIntSet class

# Chapter 6

# User-Defined Abstractions for Library Types

Imagine we have to implement a program that solves a particular constraint problem, and that we want to delegate the task of constraint solving to SQUANDER. Suppose our program makes an extensive use of Java collections to store various problem elements. This is not uncommon at all. Imagine we are trying to implement a course scheduler: we would probably want to use a list to keep an ordered sequence of semesters, a map to keep assignments of courses to semesters, another map to keep course prerequisites, etc. If we were to solve such a problem with SQUANDER, we would like to have a way of accessing the contents of our collections from the specification. For instance, we would like to write a piece of specification that says that for every course assigned to a semester, all prerequisites for that course are assigned to some of the previous semesters. The problem is, however, that we don't known the internal representation of various implementations of `java.util.List` or `java.util.Map`, so we can't write specification statements that directly refer to Java fields, like we used to do in previous examples. We also don't want to change the structure of our program, and say use our own implementation of different collection classes. This is a common problem, and SQUANDER provides a generic solution by letting the users write *abstraction* and *concretization* functions for third party or library classes.

The task of supporting an arbitrary third party class consists of: (1) writing a `.jfspec` file containing abstract field definitions, and (2) writing an *object serializer*, as an implementation of `IObjSer` interface (Listing 6.1) that provides abstraction and concretization functions for the abstract fields.

The `.jfspec` files are written in JFSL. They contain a number of abstract fields (the same `@SpecField` annotation is used) and a number of invariants (`@Invariant`). Some abstract fields may be left with only a type declaration, whereas others may also be given an abstraction function (written in JFSL), expressed in terms of the existing fields. The accompanying object serializer must provide concrete implementations of abstraction and concretization functions for only those abstract fields not already having an abstraction function defined in the `.jfspec` file.

Object serializers must implement four methods. The abstraction function (`absFunc`) takes a concrete object[1] and produces values for its abstract fields. Each `FieldValue` contains the name of the abstract field and a set of tuples to which the field evaluates. The concretization function (`concrFunc`) takes a concrete object and a value of an abstract field, and is supposed to restore that value onto the concrete object, by modifying the given argument `obj`. In addition to these two essential methods, serializers have must be able to tell what classes of objects they support (`accepts`) and also must be able to create new instances of the classes they support (`newInstance`).

```
public interface IObjSer {
  public boolean accepts(Class<?> clz);
  public Object newInstance(Class<?> cls);
  public List<FieldValue> absFunc(JavaScene javaScene, Object obj);
  public void concrFunc(Object obj, FieldValue fieldValue);
}
```

Listing 6.1: `IObjSer` interface

Squander provides built-in support for Java collections and Java arrays through this mechanism. These are explained in the following sections.

## 6.1 Supporting Java collections

### 6.1.1 Specification for `java.util.Set`

The abstract representation of a set is a set of elements, which is captured in a single `SpecField` named `elts` (Listing 6.2). An additional field, `size`, is defined for convenience. Since fields `elts` and `size` are not independent, an abstraction function is given for `size`, to constrain its value to be the cardinality of the set `elts`, and Squander can make use of this, without requiring the serializer to provide abstraction and concretization functions for `size`. The code for the serializer is straightforward and is given in Appendix A.

```
interface Set<K> {
  @SpecField("elts : set K")
  @SpecField("size : one int | this.size = #this.elts")
}
```

Listing 6.2: Specification for `java.util.Set`

### 6.1.2 Specification for `java.util.List` and Java arrays

To capture the abstract representation of a list, we again declare a single field, again named `elts`, but of type `int -> E` (Listing 6.3). This time, however, we must include an additional constraint to ensure that these fields represent a valid list: there should be exactly one element for every index from 0 (inclusive) to the size of the list (exclusive), and no elements at any other index. We write this constraint as an invariant in the same `jfspec` file. As before, we define field `size` to represent the number of

[1]It actually takes an instance of `JavaScene` as well, which may be needed in order to access the class's specification

elements in the list. Finally, we define an extra field, `prev`, defining the reverse ordering, whose use is illustrated in the case study described later. The abstraction function for this field makes use of a constant relation, namely `DEC`, which is built-in to SQUANDER and evaluates to all pairs $\{i, i-1\}$, where both $i$ and $i-1$ are integers drawn from the finite universe.

Including extra specification fields does not exact any performance penalty, since only those fields that are actually used in a specification for a particular method will be used in translation.

```
interface List<E> {
  @SpecField("elts  : int -> E")
  @SpecField("size : one int | this.size = #this.elts")
  @SpecField("prev : E -> E  | this.prev = (~this.elts) . DEC . (this.elts)")
  @Invariant({
    "all i : int | (i >= 0 && i < this.size) ? one this.elts[i] : no this.elts[i]"
  })
}
```

Listing 6.3: Specification for `java.util.List`

Java arrays are also supported through this mechanism. The specification is given in Listing 6.4 and is very similar to the one for Java lists. Even though the name of this specification file is "Object[]", the implementation knows to make an exception in this case and use it for all types of arrays. This mechanism automatically supports multi-dimensional arrays, because in Java, multi-dimensional arrays are simply arrays of arrays, and our mechanism for defining specifications is inherently compositional.

```
interface Object[]<E> {
  @SpecField("elems  : int -> E")
  @SpecField("length : one int | this.length = #this.elems")
  @Invariant({
    "all i : int | (i >= 0 && i < this.length) ? one this.elems[i] : no this.elems[i]"
  })
}
```

Listing 6.4: Specification for Java arrays

### 6.1.3 Specification for `java.util.Map`

The specification for Java maps is given in Listing 6.5. A binary relation named `elts` is used to represent the mapping from keys to values. A Java map, unlike an unconstrained relation in Alloy, can map a key to at most one value, so an invariant is needed to constrain the field `elts` accordingly.

```
interface Map<K,V> {
  @SpecField("elts : K -> V")
  @SpecField("size : one int | this.size = #this.elts")
  @SpecField("keys : set K   | this.keys = this.elts.(V)")
  @SpecField("vals : set V   | this.vals = this.elts[K]")
  @Invariant({
    "all k : K | k in this.elts.V => one this.elts[k]"
  })
}
```

Listing 6.5: Specification for `java.util.Map`

# Chapter 7

# Examples and Evaluation

## 7.1 Solving hard problems

If a problem is solvable in polynomial time, a careful manual implementation is likely to outperform a Squander implementation, because Squander always resorts to running a boolean satisfiability algorithm, which is NP-complete. But if the problem itself is difficult – due to the efficiency of modern SAT solvers – Squander may turn out to be more efficient than typical hand-written algorithms.

Of course, Squander will not always offer the most efficient solution; most of these problems have been well studied, and highly specialized heuristics have been developed for solving them. Nevertheless, it is perhaps surprising how competitive a SAT-based solution is – even including Squander's overhead of encoding and decoding – with many hand-written solutions. For our comparison, we used standard textbook solutions to the benchmark problems, which are typically based on backtracking with pruning.

### 7.1.1 "Hamiltonian Path" algorithm

A Hamiltonian path in a graph is one that visits each node in the graph exactly once. Listing 7.1 shows both the data representation that we used for graphs and the specification for this problem. To find a solution, the framework must create a fresh array of nodes to hold the result; this is specified explicitly using the `@FreshObjects` annotation. The specification asserts that the returned path contains all nodes in the graph, and that for every two consecutive nodes in the path, there exists an edge between them in the graph.

The textbook backtracking algorithm uses an adjacency matrix to represent a graph. We took this particular implementation from the web site of the Cornell course on Algorithms and Data Structures [1].

In our experiment, we generated two categories of directed graphs: (1) graphs without any Hamiltonian paths, and (2) graphs containing one or more Hamiltonian paths. For each graphs size, we ran the experiment on 10 different graphs of that size, measured the execution times, and calculated the average. All experiments included 2 warmup runs to neutralize possible effects of class loading, etc.

The following procedure was used to generate graphs:

1. generate and add $n$ nodes to the graph
2. generate a random permutation of nodes and add edges between the neighboring nodes in the permutation, including the edge between the last and the first node. At this point, the graph contains a Hamiltonian cycle.
3. randomly choose a number between 30 and 90 percent of the maximum number of nodes ($n(n+1)$) and keep adding random edges until the number of edges in the graph is equal to the chosen number.
4. randomly choose a node and remove all its incoming edges. At this point, the graph still contains at least one Hamiltonian path, the one that starts from the node selected in this step.
5. if the goal is to generate graphs with no Hamiltonian paths, remove all outgoing edges of the node selected in the previous step.

```
public class Graph {
 public static class Node { int value; }
 public static class Edge { Node src, dst; }

 private Set<Node> nodes;
 private Set<Edge> edges;

 @Ensures({
  "return[int] in this.edges.elts",
  "return[int].(src + dst) = this.nodes.elts",
  "return.length = #this.nodes.elts - 1",
  "all i: int | i >= 0 && i < return.length - 1 => return[i].dst = return[i+1].src"
 })
 @Modifies({"return.length", "return.elems"})
 @FreshObjects(cls = Edge[].class, num = 1)
 public Edge[] hp() { return Squander.exe(this); }
}
```

Listing 7.1: Hamiltonian Path Specification

The results are shown in Table 7.1. For the manual implementation, establishing the absence of a Hamiltonian path is harder than finding one (if it exists), since this requires exploring all paths from the first node (i.e. whichever node it chooses first). Sometimes, it can happen that the first node has no outgoing edges, in which case the manual algorithm terminates instantly, but on average, the problem becomes hard for a random graph with 15 or more nodes. In contrast, the boolean solver seems to easily locate the isolated node, no matter where it is found in the graph, and can thus prove nonexistence of a Hamiltonian path more easily. Finding a path when one exists is harder, but on average, the declarative solution still scales better than the backtracking algorithm.

### 7.1.2 The *N-Queens* problem

The problem of *N-Queens* involves placing $N$ queens on an $N \times N$ chess board so that no queen can take any of the others. E. W. Dijkstra, in his book on structured

| | Graphs **without** Hamiltonian paths | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 10 | 14 | 15 | 20 | 25 | 30 | 35 | 40 |
| Manual | 0.02 | 96.92 | t/o | t/o | t/o | t/o | t/o | t/o |
| Squander | 0.34 | 0.3 | 0.34 | 0.33 | 0.68 | 1.8 | 2.8 | 4.1 |

| | Graphs **with** Hamiltonian paths | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 10 | 14 | 15 | 20 | 25 | 30 | 35 | 40 |
| Manual | 0.01 | 50.17 | 214.96 | t/o | t/o | t/o | t/o | t/o |
| Squander | 0.24 | 0.29 | 0.37 | 0.75 | 4.88 | 119.81 | t/o | t/o |

Table 7.1: Hamiltonian path execution times

programming [9], describes a backtracking solution with pruning, which we implemented in Java for the purpose of our experiment. This algorithm keeps track of rows, columns and diagonals that have been taken by the queens already placed on the board, so every time it has to pick a position for the next queen, it avoids all conflicting cells, thus pruning a large portion of the search space. (There is a known polynomial time algorithm for N-Queens [30], which first guesses a solution, and then performs a local search using a gradient-based heuristic to move certain queens around until all conflicts have been resolved.)

```
@Requires("result.length == n")
@Ensures({
 "all k: int | k>=0 && k<n => lone (Cell@i) . k",
 "all k: int | k>=0 && k<n => lone (Cell@j) . k",
 "all q1: result.elts | no q2: result.elts - q1 |"+
 "  q1.i = q2.i || q1.i-q1.j = q2.i-q2.j || q1.j = q2.j || q1.i+q1.j = q2.i+q2.j"
})
@Modifies({
 "Cell.i [][][{k: int | k>=0 && k<n}]",
 "Cell.j [][][{k: int | k>=0 && k<n}]"
})
public static void nqueens(int n, Set<Cell> result) {
    Squander.exe(null, n, result);
}
```

Listing 7.2: NQueens Specification

Listing 7.2 gives the specification for N-Queens. The `nqueens` method takes an integer `n`, and a set already containing exactly `n` `Cell`[1] objects, and is expected to modify the coordinates of the given cells so that they represent a valid positioning of `n` queens.[2] The frame condition specifies that only cell coordinates are modifiable. The three bracketed subexpressions respectively mean that: (1) all `Cell` instances are modifiable, (2) the lower bound is empty, and (3) the upper bound is $\{0, \cdots, n-1\}$ (values for cell coordinates). In the post-condition, the third "`all`" clause asserts that no two different cells (queens) in the resulting set may be in the same row, column or

[1]`Cell` is a simple wrapper class for `i` and `j` coordinates of the chess board.

[2]The reason why this method takes a set of cells (as opposed to creating a new set with `n` `Cells` in it) is non-essential: SQUANDER can't arbitrarily create new objects; instead it requires the user to explicitly pass the number of new objects via `FreshObjects` annotations. Unfortunately, annotations cannot take variables as arguments, only constants

either diagonal. The first two universal quantifier clauses are redundant; they state that every row and every column must contain exactly one `Cell` object, which follows from the third constraint. Even though they are not required for correct execution, redundant constraints often (as here) improve the performance of the solver.

| n = | 8 | 16 | 28 | 32 | 34 | 36 | 54 |
|---|---|---|---|---|---|---|---|
| Manual | 0.0 | 0.0 | 0.5 | 15.9 | 428.9 | t/o | t/o |
| Squander | 0.4 | 0.6 | 4.7 | 10.0 | 11.1 | 15.7 | 89.1 |

Table 7.2: N-Queens execution times (in seconds)

Table 7.2 shows results for different values of `n`. For smaller values (up to 28), the (manual) backtracking algorithm performs better (although SQUANDER's performance is not terrible). For larger values of `n`, SQUANDER scales considerably better. It computes a solution for 54 queens in 89 seconds, whereas the manual algorithm begins to time out (that is, exceed the five minute limit we set) at only 34 queens.

## 7.2 Textbook data structures

In this section, we'll first show on several examples how textbook data structures can be easily and succinctly specified in JFSL, and then we'll run some mini benchmarks and show the execution times with SQUANDER.

### 7.2.1 Binary Search Tree

We'll use the same binary search tree we used in the introduction (Section 1.2.2, Listing 1.6). We evaluated both versions of the specification for the `insert` method: the one that allows arbitrary modifications to the tree (Listing 1.7) and the one that forces that the new nodes are inserted at leaf positions by making use of the instance selector clause in the frame condition (Listing 3.1, named `insert_fast` here).

### 7.2.2 Balanced Binary Search Tree

Having specified the binary search tree, it is very easy to define a subclass of it and add an extra invariant to implement a balanced search tree. The extra invariant (Listing 7.3) is used to enforce that for each node, the sizes of its left and right subtrees differ by at most 1. Specifications for the `insert` and `delete` methods (Listing 1.7) remain unchanged and they are automatically inherited from the superclass. Note that the specification for the `insert` method that localizes the scope of modification cannot be used here, because it forces the nodes to be inserted at leaf positions, which is in collision with the balancedness constraint.

```
@Invariant("all n: this.nodes |
  (#n.left.^(left+right) - #n.right.^(left+right)) in this.diff[int]")
public class BalancedBST extends BST {
   private final int[] diff = new int[] {-1, 0, 1};
}
```

Listing 7.3: Specification for Balanced Binary Search Tree

### 7.2.3 Linked List

The specification for the `LinkedList` class is shown in Listing 7.4. For a given node, the specification field `succ` evaluates to all nodes reachable from that node following the `next` pointer (including the starting node). The specification field `nodes` evaluates to all nodes of a list, and is simply a shortcut for `header.succ`.

The specification for the `add` and `remove` is pretty straightforward. The `add` method requires that the list rooted at the given node is not already in the list, and ensures that the resulting list contains all nodes reachable from the given node. We also want to ensure that these nodes are inserted at the end of the list, so in the frame condition we assert that the only node whose `next` pointer may be modified is the last node of the list, effectively forcing the beginning of the list to stay untouched. Similarly, for the `remove` method we assert that the only node that is modifiable is the immediate predecessor of the node to be removed.

```
@SpecField("nodes : set Node | this.nodes = this.header.succ")
public class LinkedList {

  @Invariant("this !in this.^next")
  @SpecField("succ : set Node | this.succ = this.*next - null")
  public static class Node {
    private int value;
    private Node next;
  }

  private Node header;

  @Requires("n.succ !in this.nodes")
  @Ensures("this.nodes = @old(this.nodes) + n.succ")
  @Modifies({
    "this.header",
    "Node.next [{nn : this.nodes | nn.next == null}]"
  })
  public void add(Node n) { Squander.exe(this, n); }

  @Requires("n in this.nodes")
  @Ensures("this.nodes = @old(this.nodes) - n")
  @Modifies({
    "this.header",
    "Node.next [{nn : this.nodes | nn.next == n}]"
  })
  public void remove(Node n) { Squander.exe(this, n); }
}
```

Listing 7.4: Specification for Linked List

### 7.2.4 Benchmarks

We measured the time it took Squander to execute specifications of some common operations on several different types of complex data structures. The algorithmic complexity of the chosen operations is in polynomial time, so it is unreasonable to expect Squander to perform as good as carefully written manual implementations. However, the results show that the running times are acceptable for inputs of small size, so the applicability of this approach is typically limited to the development phase.

For all benchmarks presented here, we had 1 warmup run and 3 test runs. The purpose of the initial warmup run was to eliminate possible "cold start" effects, such as JVM initialization and class loading. In the subsequent 3 test runs, we executed the operation under test on 3 different (randomly generated) instances of a given size, measured the total execution time (which includes both serialization/deserialization done by Squander and solving done by Kodkod + SAT solver), and reported the average. The timeout threshold was set to 1 minute for all experiments, and the maximum Java heap size was left to the default value of 256MB. All benchmarks were run a Linux box, with Intel® Core™2 Duo CPU @ 2.93GHz, 4GB of RAM, running Ubuntu 9.10.

To generate random binary trees of size $n$ for insertion, we executed a manual implementation of the `insert(Node n)` operation $n-1$ times. Every time we would pass a node with a random (but previously unseen) value. The same way, we would then generate the $n$-th node, the node to be inserted, and execute the specification for the `insert` method with Squander. For deletion, we would similarly generate a tree with $n$ nodes, randomly pick a node from the tree to be deleted, and measure the time of execution of the specification for the `remove` method.

To generate inputs for the "Balanced BST" benchmark, we used exactly the same procedure the generate unbalanced trees, and the require that the tree is balanced after the insertion/deletion. Since the invariant for these trees is often violated in the pre-state (since the way we generate them doesn't make them necessarily balanced), we had to configure Squander so that it doesn't check the precondition.

Finally, to generate linked list, we simply generated random sequences of nodes. For the benchmarks of size $n$, the `insert` benchmark inserts the $n$-th node in a list already containing $n-1$ nodes, and the `remove` benchmark removes a node from a list of $n$ nodes.

The results are shown in Tables 7.3, 7.4, and 7.5. For the hard search problems (insertion/deletion from a (balanced) binary tree), executing specifications becomes infeasible for trees with more than 10-15 nodes. These are hard search problems, because the bounds for the modifiable relations corresponding the `left` and `right` fields grow rapidly as the number of nodes increases. In these cases, the time taken by the SAT solver dominates the total time. In other benchmarks (`BST.insert_fast`, linked list insertion/deletion) it is possible to localize the modifications to only several objects, which results in much better scalability. Interestingly, for these benchmarks, the problem is not the SAT solving time anymore. Once the boolean formula has been constructed, it is easy for the SAT solver to find a solution, because the formula is now fairly smaller, since only a smaller part of the state needs to be solved for and Kodkod does a pretty good job of optimizing the final boolean formula. Instead, Kodkod translation time becomes dominant now because the sole translation from relational to boolean logic takes a lot of time when the bounds for the relations are big. As an intuition, for a tree with 100 nodes, the upper bound for the `left` and `right` fields counts $100 \times 100 = 10,000$ tuples. A matrix with 10,000 elements is used to represent each of these two fields. Translation of relational expressions to boolean formulas involves many different matrix operations, so when matrices are big (i.e. contain many elements) these operations can be very slow.

| BST | 5 | 10 | 15 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `insert_fast` | 0.09 | 0.19 | 0.29 | 0.69 | 1.16 | 3.10 | 6.52 | 12.29 | 24.53 | 38.02 | 55.58 |
| `insert` | 0.18 | 0.29 | 0.38 | 0.75 | t/o | t/o | t/o | t/o | t/o | t/o | t/o |
| `remove` | 0.01 | 0.33 | 1.27 | t/o | t/o | t/o | t/o | t/o | t/o | t/o | t/o |

Table 7.3: Binary Search Tree benchmarks

| Balanced BST | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|
| `insert` | 0.3 | 0.35 | 0.43 | 0.5 | 5.01 | 5.45 | 15.31 | 34.13 | t/o |
| `remove` | 0.39 | 0.21 | 0.66 | 0.92 | 1.07 | 12.46 | 5.64 | 43.58 | 50.4 |

Table 7.4: Balanced Binary Search Tree benchmarks

| Linked List | 20 | 60 | 100 | 140 | 180 | 200 | 250 | 300 |
|---|---|---|---|---|---|---|---|---|
| `add` | 0.29 | 0.79 | 3.2 | 13.17 | 24.6 | 29.7 | 51.84 | m/o |
| `remove` | 0.3 | 0.57 | 1.91 | 6.82 | 17.33 | 29.99 | 35.19 | m/o |

Table 7.5: Linked List benchmarks

# Chapter 8

# Course Scheduler Case Study

As a larger case study, we re-implemented an existing application – a course scheduler that helps students select courses to complete graduation requirements. Given a student's current standing, it finds a path to graduation that meets the program requirements for the undergraduate degree in EECS at MIT. The MIT program offers around 300 courses, defines prerequisites for more than 150 courses, and also specifies some additional requirements (e.g. mandatory courses, selections of multiple options from groups of courses, etc). The original implementation [36] used the Kodkod [31] constraint solver directly via its API.

About 1500 lines of code were written to translate the student's standing and the set of MIT requirements to relational constraints, run Kodkod to find a solution, and finally translate the Kodkod solution back to the original data structures. It is these lines of code that SQUANDER is intended to eliminate.

The goal of this case study was to assess the usability of SQUANDER on a real world program, whose core lies in solving a constraint problem. A key goal was to make minimal changes to the existing data structures of the original application, so that the rest of the application (e.g. GUI, I/O, etc.) might be reused without modification.

SQUANDER's built-in abstractions for the Java library classes (Chapter 6), used extensively in the data model, were essential in reducing the annotation burden. We had to annotate user-defined classes with invariants, define additional specification fields when necessary, and introduce a single new method (named `solve`), with a specification capturing the course requirements.

A second goal was to show that the framework could scale to a large heap. The novel translation presented in Chapter 4 enabled us to handle heaps with almost 2000 objects. The time it takes SQUANDER to find a solution for a problem of this size is less than 5 seconds. The original implementation still runs faster (it takes about 1 second) but the cost of its development was much higher.

## 8.1 Data Model

The full data model for the scheduler application is given in Listing 8.1.

The top-most class in the hierarchy is the class `Problem`. Aside from references to a `DegreeProgram` and a `Schedule`, the `Problem` class contains a set of additional constraints provided by the student, e.g. "don't schedule a course", "schedule a course after a given semester". Departmental requirements are associated with `DegreeProgram`.

A `DegreeProgram` contains information about the selected program. The purpose of `allCourses` and `prereqMap` fields is intuitive: the former simply contains all courses offered by the department, and the latter contains prerequisites for certain courses. A prerequisite for a course is defined (in disjunctive normal form) as sets of sets of courses (in other words, the students choose a single set of courses as a prerequisite). This class also contains all course groupings defined by this program (`groupings`) and the top-level grouping (`rootGrouping`) which in the end resolves to all courses that a student must pass (course groupings are explained below).

A `Schedule` contains a list of semesters (which is given in advance and must not be changed during the process of solving), a mapping from semesters to courses (`sCourses`) and an auxiliary field, namely `prereqUsed`, holding the choice of course used to satisfy a course's prerequisite (recall that the prerequisites map can possibly contain several possibilities for a single course). The `sCourses` map may already contain some entries (courses that have already been taken). The goal of this application is to compute the content of this map such that it preserves the existing assignments and satisfies all requirements.

Semesters can have a set of attributes associated to it (e.g. "Fall", "Spring", etc.) and can also be flagged as "past semester". Assignment of courses for past semesters must not change.

Courses can be grouped in `CourseGrouping` so that the requirements can be written in terms of course groups. Moreover, every course is also put in a singleton group, for which `isSingleCourse` field is set to `true` and the value of the `courses` field is a singleton set containing that one course. For others, the `courses` field is to be computed according to the requirements associated with the grouping.

To understand course groups better, and how requirements are always defined in terms of course groups, consider the following requirement

```
<minSizeSubsetReq>
  <size>2</size>
  <subset>
    <member>6.101</member>
    <member>6.105</member>
    <member>bio-lab</member>
  </subset>
</minSizeSubsetReq>
```

where "bio-lab" is a course group containing two or more courses, and "6.101" and "6.105" are singleton course groups. This requirement means that the student must pick 2 out of these three groups, and pass all courses from those 2 selected groups. In other words, the requirement is not satisfied if the student passes "6.101" and one course from "bio-lab".

Finally, a `Course` has a name and potentially some attributes. In order for a course to be scheduled for a semester, that semester's set of attributes must include all attributes of the course.

```
public class Problem {
 private DegreeProgram dp;
 private Schedule schedule;
 private Set<Requirement> additionalReqs;
}

public class DegreeProgram {
 private Set<Course> allCourses;
 private PrereqMap prereqMap;
 private Set<CourseGrouping> groupings;
 private CourseGrouping rootGrouping;
}

public class Schedule {
 private List<Semester> semesters;
 private Map<Semester, Set<Course>> sCourses;
 private Map<Course, Set<Course>> prereqUsed;
}

public class Semester {
 private String name;
 private boolean isPastSemesters;
 private Set<Attribute> attributes;
}

public class CourseGrouping {
 private String name;
 private boolean isSingleCourse;
 private Set<Requirement> groupingReqs;
 private Set<Course> courses;
}

public class Course {
 private String name;
 private Set<Attribute> attributes;
}

public class PrereqMap {
 protected Map<Course, Set<Set<Course>>> prereqs;
}

public abstract class Requirement {
 protected CourseGrouping cg;
}
```

Listing 8.1: Course Scheduler data model

## 8.2 Specification

### 8.2.1 Invariants for the schedule

Class `Schedule` is the primary class. It contains a list of semesters (given in advance and not to be modified), and a mapping from semesters to courses (`sCourses`) to be computed. The content of the `prereqUsed` fields is also to be computed so that it holds the choice of course used to satisfy a course's prerequisite.

There are several invariants that must hold for a schedule to be valid. They are all listed in Listing 8.2 and explained individually below.

- *Schedule a course's prerequisites for semesters preceding the semester of that course.* This constraint can be expressed with several nested universal quantifiers, but a more complicated, closed form relation expression is used instead for efficiency reasons. This expression says that all tuples of type `Course` × `Course` found in the `prereqUsed` map (the left-hand-side of the `in` expression) can also be found in the relation holding the mapping of courses to previously taken courses (the right-hand-side of the `in` expression; makes use of the `prev` field defined for lists).

- *Include prerequisites for courses.* All scheduled courses, for which there exist at least one entry in the departmental map of course prerequisites (`PrereqMap`), must appear as keys in the map of used prerequisites. This is constraint is needed to force the content of the `prereqUsed` map, whereas the previous constraint enforces that the course prerequisites are actually assigned to preceding semesters.

- *Course and semester attributes match.* This one simply constrains courses to be assigned only to semesters whose attributes include all of the course's attributes.

- *Don't skip semesters.* It is usually not desired to have semesters with no courses, so this rule is used to disallow that.

- *Don't assign courses more than once.* Set intersection is used to ensure that for every semester, the set of courses assigned for that semester and the set of courses assigned for all other semesters are disjoint.

- *Don't include null.* Since `null` is a valid value for all reference types in Java, `null`s must be explicitly disallowed when desired.

```
@Invariant({
 /* for all courses, prereqs must be taken in the previous semesters */
 "this.prereqUsed.elts.elts in
   ((~this.sCourses.elts.elts).(^(this.semesters.prev)).(this.sCourses.elts.elts))",
 /* include prerequisites for courses */
 "(this.sCourses.vals.elts & PrereqMap.prereqs.keys) in this.prereqUsed.keys",
 /* course and semester attributes match */
 "all sem: Semester | this.sCourses.elts[sem].elts.attributes.elts in
                      sem.attributes.elts",
 /* don't skip semesters */
 "all sem: Semester | some this.sCourses.elts[sem].elts &&
    !this.semesters.prev[sem].isPast
      => some this.sCourses.elts[this.semesters.prev[sem]].elts",
 /* don't assign courses more than once */
 "all sem: Semester |
    no (this.sCourses.elts[sem].elts & this.sCourses.elts[Semester - sem].elts)",
 /* don't add "null" */
 "null !in this.sCourses.vals.elts"})
public class Schedule {
 private List<Semester> semesters;
 private Map<Semester, Set<Course>> sCourses;
 private Map<Course, Set<Course>> prereqUsed;
}
```

Listing 8.2: The invariant for the class `Schedule`

### 8.2.2 Specification for the `solve()` method

The core of the specification for the scheduler is associated with the method `solve()` (Listing 8.3).

The post-condition says that: (1) all requirements must hold, and (2) the schedule must include all courses from the root grouping (`dp.rootGrouping`). To express the first property without having to know about all subclasses of `Requirement`, we simply defined a boolean specification field (named `cond`) for the `Requirement` class and asserted that it evaluates to `true`. Concrete implementations of `Requirement` are expected to override the definition of `cond` to impose their own constraints. Detailed explanation of different types of requirements is given later in Section 8.2.3.

The frame condition for `solve` requires more than just listing the modifiable fields. For example, not only must the `Set<Course>.data` field be modifiable (because we are searching for suitable values for the `Schedule.sCourses` map), but an instance selector must also be provided to specify that only those sets of courses that are not associated with the past semesters may be modified. The content of the "used prerequisites" map (`Map<Course, Set<Course>>`) must also be modifiable, but this time, however, it is essential to tighten the upper bound for this field so that its content is a subset of the constant map of course prerequisites defined by the department (which has less than 300 entries). Otherwise, the bound for this field would have gone up to 90,000 atoms, since there are 300 distinct courses and 300 distinct sets of courses on the heap, causing a huge performance setback. Finally, sets of courses associated with course groupings are also modifiable, but only for those groups that don't represent a single course.

```
@Ensures({
 "all req: this.additionalReqs.elts +
           this.dp.groupings.elts.groupingReqs.elts | req.cond",
 "this.dp.rootGrouping.courses.elts in this.schedule.sCourses.vals.elts"
})
@Modifies({
 /* modify sCourses map, but don't change the mapping for the "past" semesters */
 "this.schedule.sCourses.vals.elts
      [{st: java.util.Set<Course> | no sem: Semester |
       sem.isPast && ((sem->st) in this.schedule.sCourses.elts)}]",
 /* modify the used prerequisites map */
 "this.schedule.prereqUsed.elts [] [] [PrereqMap.prereqs.elts.elts]",
 /* and courses assigned to course groups */
 "CourseGrouping.courses.elts [{cg: CourseGrouping | !cg.isSingleCourse}.courses]"
)}
public void solve() {
 Squander.exe(this);
}
```

Listing 8.3: Specification for the `solve` method

### 8.2.3 Specifying the requirements

Specification fields are particularly useful for defining specification in a modular fashion, thus reducing the coupling between modules. As mentioned above, at the place where the post-condition for the `solve()` method is defined, all existing types of requirements are not known. In order to abstractly say that all requirements hold, without knowing the concrete requirement rules for any of them, we defined a boolean specification field for the `Requirement` class, namely `cond` as shown in Listing 8.4. In this section, we give concrete abstraction functions for all different kinds of requirements.

```
@SpecField("cond : one boolean")
public abstract class Requirement {
  protected CourseGrouping cg;
}
```

Listing 8.4: Specification field for the `Requirement` class

### MandatoryCourseReq

`MandatoryCourseReq` contains a set of mandatory courses and simply asserts that all those courses must be found in the set of courses of the group which is associated with this requirement.

```
@SpecField("cond : one boolean |
              this.requiredCourses.elts.courses.elts in this.cg.courses.elts")
public class MandatoryCourseReq extends Requirement {
 private Set<CourseGrouping> requiredCourses;
}
```

Listing 8.5: Specification for the `MandatoryCourseReq` class

### MinimumSizeSubsetReq

`MinimumSizeSubsetReq` contains a set of course groups and the minimum number of those groups that must be included. The abstraction function asserts that there exists a subset of `choices` with at least `minSubsetSize` elements such that all corresponding courses are included in the courses for this requirement's group.

```
@SpecField("cond : one boolean | exists s : set CourseGrouping |
              #s == this.minSubsetSize && s in this.choices.elts &&
              s.courses.elts in this.cg.courses.elts")
public class MinimumSizeSubsetReq extends Requirement {
 private Set<CourseGrouping> choices;
 private int minSubsetSize;
}
```

Listing 8.6: Specification for the `MinimumSizeSubsetReq` class

### NoOverlapReq

`NoOverlapReq` contains a set of course groups and asserts that those courses must not overlap with the courses assigned to this requirement's group.

```
@SpecField("cond: one boolean | no this.members.elts.courses & this.cg.courses.elts")
public class NoOverlapReq extends Requirement {
 private Set<CourseGrouping> members;
}
```

Listing 8.7: Specification for the `NoOverlapReq` class

### NeverScheduleReq

The course scheduler application allows the students to specify if they don't want to take certain courses at all. To represent this requirement, class `NeverScheduleReq` is used. Classes used to represent the students' requirements don't have an associated course group, so their abstraction functions must refer directly to the schedule and the scheduled courses. So in this case, the abstraction function simply asserts that the given course is not found in the `sCourses` map of the `Schedule` object.

```
@SpeccField("cond : one boolean | this.course !in Schedule.sCourses.vals.elts")
public class NeverScheduleReq extends Requirement {
 private final Course course;
}
```

Listing 8.8: Specification for the `NeverScheduleReq` class

### TimeReq

The students are also allowed to explicitly specify time requirements for certain courses. For example, they can specify that a certain course must be scheduled either exactly at, before, after, not before or not after a given semester.

```
@SpecField("cond : one boolean | " +
 /* AT */
 "this.opVal = 2 ? (this.course in Schedule.sCourses.elts[this.semester].elts) : (" +
 /* BEFORE */
 "this.opVal = 0 ? (some s: Semester | this.course in Schedule.sCourses.elts[s].elts
     && Schedule.semesters.elts.(s) < Schedule.semesters.elts.(this.semester)) : (" +
 /* NOT_AFTER */
 "this.opVal = 1 ? (no s: Semester | this.course in Schedule.sCourses.elts[s].elts
     && Schedule.semesters.elts.(s) > Schedule.semesters.elts.(this.semester)) : (" +
 /* NOT_BEFORE */
 "this.opVal = 3 ? (no s: Semester | this.course in Schedule.sCourses.elts[s].elts
     && Schedule.semesters.elts.(s) < Schedule.semesters.elts.(this.semester)) : (" +
 /* AFTER */
 "this.opVal = 4 ? (some s: Semester | this.course in Schedule.sCourses.elts[s].elts
      && Schedule.semesters.elts.(s) > Schedule.semesters.elts.(this.semester)) :
  false))))")
public class TimeReq extends Requirement {
 private final int opVal;
 private final Course course;
 private final Semester semester;
}
```

Listing 8.9: Specification for the `TimeReq` class

# Chapter 9

# Related Work

The idea of executable specifications is not a new one. However, it has been widely assumed that any implementation would be hopelessly inefficient, and thus not feasible for practical applications. Hoare [14] acknowledges the benefits that such technology would have, but also predicts that computers would never be powerful enough to carry out any interesting computation in this way. Hayes and Jones [13] argue that direct execution of specifications would inevitably lead to a decrease in the expressive power of the specification language, undoing much of the advantage of specifications. On the other side of the spectrum, Fuchs [12] claims that declarative specifications can be made executable by intuitive (manual) translation to either a functional programming language (such as ML) or a logic programming language (like Prolog).

Rayside et al. [27] suggested how executing specifications might play a useful role in an agile development process [4]: for fast prototyping, test input generation, creation of mock objects directly from interfaces, etc.

Samimi et al. implemented a tool [28], called PBnJ, that borrows most of the ideas from the work of Rayside et al. [27], but applies them in a different context: using executable specifications as a fallback mechanism. Like Squander, PBnJ provides a unified environment for imperative and declarative code but it lacks Squander's expressive power and the ability to handle abstract types (and in particular, library classes). Their *spec methods* are similar to the *spec fields* of [27], but do not accommodate arbitrary declarative formulas – rather, only those for which a straightforward translation to imperative code exists. As a consequence, spec methods can express something like "all nodes in a graph", or "all nodes such that each one of them has some property", but cannot express "some set of nodes that form a clique". Another limitation is that spec methods cannot be recursive. By creating a separate relation for every spec field, Squander solves all these problems: whatever abstraction function is given to a spec field, it will be translated into a relational constraint on the corresponding relation, and Kodkod will find a suitable value for it. PBnJ comes with custom classes for sets, lists and maps, but provides no mechanism for the user to extend the support to other abstract types.

Squander can be considered as an implementation of the Carroll Morgan's mixed interpreter [25], comprising a combination of conventional imperative statements and declarative *specification statements* [24].

The PAISLey project [37] can execute specifications that represent both data manipulation and control flow. This form of specification is well suited for specifying process control systems, but is different from the first order relational logic supported by SQUANDER.

In later work, Wahls et al. are working on executing JML [7] (which a specification language very similar to JFSL) by translating them to constraint programs and using backtracking to search for a solution [8, 18] (as opposed to using a SAT solver). A comparison between these two techniques was outside the scope of this thesis. Yang's LogLog tool [34] employs runtime constraint solving to automatically impute values for missing data based on declarative constraints.

Forge [10, 11] is a bounded software verifier that allows an imperative procedure written in Forge Intermediate Representation (FIR) to be checked, up to a given bound on the heap size and the number of loop unrollings, against a rich specification. Its front-end for Java, JForge [35], can check a Java procedure against a JFSL specification. As in refinement calculus [3], FIR supports declarative *specification statements*. SQUANDER could exploit that fact and alternatively use Forge as a back-end solver. In that case, SQUANDER would construct a Forge universe that corresponds to the current state of the Java program, and a Forge procedure that contains a single specification statement corresponding to the specification being executed by SQUANDER. It would then ask Forge to check whether the given procedure is equivalent to `false`. If the answer is yes, that means that the specification is not satisfiable; otherwise, Forge returns a concrete counter-example describing a state in which the given program is not equivalent to `false`, i.e. a state in which the specification statement is `true`, which is exactly what SQUANDER is supposed to find. We decided not to use Forge, because going directly to Kodkod gave us more flexibility and options for optimizations.

# Chapter 10

# Conclusions and Future Work

## 10.1 Challenges/Difficulties

The main challenge to successfully completing this project was choosing the right level of abstraction. A solution that works for the simplest examples, like the Binary Search Tree example, is relatively simple. The basic idea – to use relations to represent field values, and then use Kodkod to search for field assignments – was also pretty intuitive, at least for someone with an Alloy mind-set. The difficult part was identifying the entire class of problems that are suitable for this general approach of mixing imperative and declarative code, and making sure that SQUANDER can support most, if not all, of them. For example, we didn't realizer that a basic object serializer that simply follows all object's fields wasn't sufficient until we tried to use SQUANDER to modify an existing application (namely the course scheduler application). The importance of the "KodkodPart" translation also became clear from this case study. We also realized that we needed a different abstraction for types. Initially, we equated our notion of a type with the Java `Class` class. As soon as we added support for Java collections, it was clear that knowing the actual type parameters of objects of generic classes was essential; otherwise, SQUANDER would have not scaled enough to solve the real MIT course requirements in our case study. Finally, we learned that in the interest of performance, we must leave the user the option to be more specific about frame conditions and specify fine-grained bounds (both lower and upper) for the modifiable fields, as well as the exact instances whose fields may be modified.

## 10.2 Limitations

There are several limitations of our framework worth mentioning:

- First of all, everything has to be bounded. As a consequence, SQUANDER cannot be expected to generate an arbitrary number of new objects needed to satisfy a specification; instead, the exact number of new objects of each class must be specified by the user. This fact doesn't make SQUANDER very suitable for functional style programs, where instead of mutating the current state, one would want to create a bunch of fresh objects and modify their state only.

- Another consequence of the bounded nature of SQUANDER is that integers must also be bounded to a small bitwidth. Using 32 bits to represents integers (as in Java) would make Kodkod intractable. Using a small bitwidth can occasionally cause subtle integer overflow problems, which are typically hard to find.

- Issues with equality. The current implementation uses observational equivalence [19, 26] only for strings and primitive types, and referential equality for all other types. That means that it is currently impossible to write a specification which asserts that two objects are equal in the sense of Java `equals`, i.e. that `obj1.equals(obj2)`.

- Higher-order expressions. Even though both JFSL and Kodkod API allow (in a somewhat limited form) higher-order expressions, the Kodkod engine won't accept them. As an example where this can be a problem, consider the longest path in a graph problem. It is not possible to write a specification that says "*find a path in the graph such that there is no other path in the graph longer than it*" and solve it with SQUANDER. It is possible, however, to express and solve "*find a path in the graph with at least `k` nodes*", which is computationally as hard as the previous problem, because a binary search can be used to efficiently find the maximum `k` for which a solution exists.

## 10.3 Future Work

In the future, we are hoping to explore a different translation mechanism, which would not only minimize the number of atoms in the universe, but the number of relations as well. For many problems, we compared translations to Kodkod that were handwritten with those produced by SQUANDER. In almost all cases, the handwritten translations were more compact and used fewer relations. Object graphs usually contain many unmodifiable fields ("links") that are only used to navigate from the root objects to the modifiable portion of the heap. Currently, our translation creates a relation for each of them. A more clever translation could short-circuit some of those links, therefore use fewer relations to represent the heap, which is likely to decrease the solving time and improve overall performance.

We also plan to compare different techniques for solving declarative constraints, e.g. the backtracking ones as in JMLe [18] or Korat [6, 22], with our current, SAT-solver based one. Boyapati et al. reported in [6] that the Korat search algorithm, which takes an imperative implementation of `repOk` [20] to check the class invariant, generates complex structures much faster than Alloy Analyzer [16], which is based on checking declarative constraints very similar to those used by SQUANDER. SQUANDER could translate the method's postcondition to an imperative `repOk` method (using a technique similar to that of MintEra [2]) and then run the Korat algorithm to search for a solution that satisfies the postcondition. Another benefit of this approach would be that all integers would automatically be supported, instead of only a bounded subset (which is required by Kodkod).

Our framework can also be used for test input generation. One can simply define a method with a return type of a class whose instances are to be generated and execute that method with SQUANDER. Provided that the invariant for the return type is defined, SQUANDER will find an instance of that type that satisfies the class invariant. That instance represents a valid test input. SQUANDER can also enumerate all possible solutions, i.e. generate all valid test inputs, by using the corresponding feature of Kodkod. Kodkod adds symmetry breaking [29, 33] predicates to return (mostly) non-isomorphic instances. TestEra [17, 23] takes a similar approach for generating test inputs from Alloy-like specifications, but it generates a much larger number of instances than Korat. Our current implementation of SQUANDER suffers from the same problem, which makes it less practical to use. It would be interesting to see if that can be improved by adding more symmetry breaking predicates at the SQUANDER level, instead of at the boolean level inside Kodkod.

## 10.4 Conclusion

In this thesis, we presented SQUANDER, a framework that unifies both writing and executing imperative and declarative code. With the optimizations described above, and specification extensions to support data abstraction, we have shown in this thesis (a) that we are now able – for a non-trivial class of problems – to use the mechanism for executing specifications as a standard runtime, and (b) that the framework is expressive enough to specify and completely eliminate the manual encodings and decodings of a moderately-sized course scheduling application we had previously implemented.

# Appendix A

# Source Code for SetSer.java

```
public class SetSer implements IObjSer {
  public static final String DATA = "elts";

  public boolean accepts(Class<?> clz) {
    return Set.class.isAssignableFrom(clz);
  }

  public Set newInstance(Class<?> cls) {
    return new HashSet();
  }

  public List<FieldValue> absFunc(JavaScene javaScene, Object obj) {
    ClassSpec cls = javaScene.classSpecForObj(obj);
    List<FieldValue> result = new LinkedList<FieldValue>();
    Set set = (Set) obj;
    JField dataField = cls.findField(DATA);
    if (dataField != null) {
      FieldValue fvElems = new FieldValue(dataField, 2);
      for (Object elem : set) {
        fvElems.addTuple(new ObjTuple(obj, elem));
      }
      result.add(fvElems);
    }
    return result;
  }

  public Object concrFunc(Object obj, FieldValue fieldValue) {
    String fldName = fieldValue.jfield().name();
    if (DATA.equals(fldName))
      return restoreElems(obj, fieldValue);
    else
      throw new RuntimeException("Unknown field name for Java Set: " + fldName);
  }

  private Object restoreElems(Object obj, FieldValue fieldValue) {
    ObjTupleSet value = fieldValue.tupleSet();
    assert value.arity() == 2;
    Set set = (Set) obj;
    set.clear();
    for (ObjTuple ot : value)
      set.add(ot.get(1));
    return set;
  }
}
```

Listing A.1: Serializer code for `java.util.Set` class

# Bibliography

[1] Hamiltonian path algorithm. `http://moodle.cornellcollege.edu/0809/mod/resource/view.php?id=8993`. CSC213-8 Course.

[2] Basel Y. Al-Naffouri. Mintera: A testing environment for java programs. Master's thesis, Massachusetts Institute of Technology, 2004.

[3] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978. Report A–1978–4.

[4] Kent Beck. *Extreme Programming Explained*. 1999.

[5] Kent Beck. *Test-Driven Development*. 2003.

[6] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated Testing Based on Java Predicates.

[7] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[8] Néstor Cata no and Tim Wahls. Executing jml specifications of java card applications: a case study. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 404–408, New York, NY, USA, 2009. ACM.

[9] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., London, UK, UK, 1972.

[10] Greg Dennis. *A Relational Framework for Bounded Program Verification*. PhD thesis, 2009.

[11] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 130–145, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, September 1992.

[13] Ian Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, 1989.

[14] C. A. R. Hoare. An overview of some formal methods for program design. *IEEE Computer*, 20(9):85–91, 1987.

[15] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* April 2006.

[16] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In *In Proceedings of the 22nd. International Conference on Software Engineering.* Society, ACM Press, 2000.

[17] Sarfraz Khurshid. *Generating structurally complex tests from declarative constraints.* PhD thesis, 2004.

[18] Ben Krause and Tim Wahls. jmle: A tool for executing jml specifications via constraint programming. In L. Brim, editor, *Formal Methods for Industrial Critical Systems (FMICS'06)*, volume 4346, pages 293–296, August 2006.

[19] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* 1986.

[20] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* 2001.

[21] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects.

[22] Darko Marinov. *Automatic testing of software with structurally complex inputs.* PhD thesis, Cambridge, MA, USA, 2005. Supervisor-Rinard, Martin C.

[23] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 22, Washington, DC, USA, 2001. IEEE Computer Society.

[24] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), 1988.

[25] Carroll Morgan. *Programming from Specifications.* 2nd edition, 1998. First edition 1990.

[26] Derek Rayside, Zev Benjamin, Rishabh Singh, Joseph P. Near, Aleksandar Milicevic, and Daniel Jackson. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 342–352, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson. Agile specifications. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 999–1006, New York, NY, USA, 2009. ACM.

[28] Hesam Samimi, Ei Darli Aung, and Todd D. Millstein. Falling back on executable specifications. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 552–576. Springer, 2010.

[29] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Appl. Math.*, 155(12):1539–1548, 2007.

[30] Rok Sosic and Jun Gu. A polynomial time algorithm for the n-queens problem. *SIGART Bull.*, 1(3):7–11, 1990.

[31] Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications.* PhD thesis, 2008.

[32] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM '08: Proceedings of the 15th international symposium on Formal Methods*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] Emina Torlak and Daniel Jackson. Kodkod: a relational model finder. In *TACAS'07: Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, pages 632–647, Berlin, Heidelberg, 2007. Springer-Verlag.

[34] Jean Yang. Specification-enhanced execution. Master's thesis, Massachusetts Institute of Technology, May 2010.

[35] Kuat Yessenov. A light-weight specification language for bounded program verification. Master's thesis, Massachusetts Institute of Technology, May 2009.

[36] Vincent S. Yeung. Declarative configuration applied to course scheduling. Master's thesis, 2006.

[37] Pamela Zave. An insider's evaluation of paisley. *IEEE Transactions on Software Engineering*, 17:212–225, 1991.