Unifying Execution of Imperative and Declarative Code

Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, Daniel Jackson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{aleks, drayside, kuat, dnj}@csail.mit.edu

ABSTRACT

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 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, the heap is updated to reflect the solution, so the user can continue to manipulate the program heap in the usual imperative way. We show that this approach is not only convenient, but, for certain problems can also outperform a standard imperative implementation. We also present an optimization technique that allowed us to run our tool on heaps with almost 2000 objects.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environment; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.1.2 [**Programming Techniques**]: Automatic Programming

General Terms

Design, Languages

Keywords

formal methods, executable specifications, declarative programming, constraint-based languages

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 [23], an Alloy-like [11] language, that supports first-order relational logic with transitive closure, and standard 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. As we shall show, in some cases, this form of execution is competitive with hand-written imperative code; in others, it is not, but there are still contexts in which it makes sense to trade performance for other benefits [17].

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. The execution engine runs a solver when a declarative constraint is encountered, and automatically updates the heap to reflect the solution. Afterwards, the program continues to manipulate the heap in the usual imperative way. Without such a technology, the standard approach is to translate the problem into the language of an external solver with specialized hand-written code, run the solver, and then translate the solution back to the native programming language using more specialized code. This requires more work and is more error prone.

This paper presents the implementation of our framework, 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 imperative 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. It can be freely obtained from [2].

Contributions of this paper include:

- A framework for unified execution of imperative and declarative code, that combines an existing pre/post/invariant annotation language with a single extension (to 'magically' execute a procedure with a specifica-

ICSE '11, May 21–28, 2011, Honolulu, Hawaii, USA

tion but containing no code), and which requires no preprocessing and no change to the execution environment beyond inclusion of the Squander library;

- A treatment of data abstraction, using serialization techniques, that allows specifications to reference the state of abstract objects – in particular the collections of the Java library, as well as user-defined datatypes – in a representation-independent fashion;
- An optimization technique for large object heaps, that overcomes a key problem in translation to SAT (namely that the sheer number of potential objects leads, with a naive encoding, to an array of SAT variables whose highest index is greater than Java's largest integer);
- A series of small examples, illustrating the expressiveness of the declarative notation, and its performance advantages, for some puzzles and graph algorithms;
- A case study, in which an existing non-trivial program (that was previously implemented with a handwritten encoding) was retrofitted with the new framework, as a test of its application in a slightly larger setting.

2. EXAMPLE – SUDOKU SOLVER

As an example, consider a simple *Sudoku* solver. The solver is given a partially filled puzzle (Figure 1), and is expected to fill out the empty cells with integer values so 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 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: A random Sudoku puzzle

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

@Invariant("all v: int |
  v != 0 => lone this.cells.elems.val.v")
static class CellGroup {
 Cell[] cells;
 public CellGroup(int n) {this.cells=new Cell[n];}}

public class Sudoku {
 CellGroup[] rows, cols, grids; int n;
 public Sudoku(int n) { init(n); }
 @Ensures("all c:Cell | c.val>0 && c.val<=this.n")
 @Modifies("Cell.val [{c:Cell | c.val == 0}]")
 public void solve() { Squander.exe(this); }}
```

Listing 1: Sudoku Specification

A suitable Java data model for this problem is given in Listing 1; ignore the annotations for now. 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, `@Invariant` annotation). 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`).

Class invariants assert properties of all members of a class, but cannot be executed *per se*. To establish an executable 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 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.

3. BACKGROUND

3.1 Kodkod – A Solver for Relational Logic

Kodkod [21] is a constraint solver for relational logic. It takes a relational formula to be solved, along with definitions of a set of untyped relations, consisting of bounds for each relation, and a bounded universe of atoms from which the relations in the solution are populated. It translates the formula to a boolean satisfiability problem and applies an off-the-shelf SAT solver to search for a satisfying solution, which, if found, is translated back to the relational domain.

Relations in Kodkod are untyped, meaning that every relation can potentially contain any tuple drawn from the finite universe. 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, the faster the solving.

3.2 JFSL – JForge Specification Language

JFSL [23] is a formal specification language for Java. As in Alloy, all expressions evaluate to relations. JFSL provides

common relational algebra operators: join (`.`), transpose (`~`), tuple constructor (`->`), transitive closure (`^`), and reflexive transitive closure (`*`), as well as set algebra operators: union (`+`), difference (`-`), and intersection (`&`).

Like in Java, the universe of discourse consists of Java objects, `null`, and primitive values. Each type corresponds to a set of objects of **exactly** that type (e.g. `int` is the set of all primitive integers, while `Object` is the set of all instances of class `Object`, excluding subclasses). Each field is treated as a binary relation from its declarer type to its value type. The value type is represented by a union of the concrete types that are assignable from the field's declared type. An array type `T[]` is represented with a ternary relation `elems` of type `T[]->int->(T+Null)`. The bracket operator is conveniently treated as a left join, so `a[b]` is equivalent to `b.(a.elems)` if `a` is an array or `b.a`, otherwise. Formulas are built from expressions using boolean algebra, the cardinality operator (`#`)), set comprehensions (e.g. `{x:T | P(x)}`), and quantifications (e.g. `all x:T | P(x)`).

JFSL specifications are written as Java annotations. Besides the already mentioned `@Invariant`, `@Requires`, `@Ensures`, and `@Modifies` clauses, JFSL provides support for *specification fields* via the `@SpecField` annotation. Definition of a specification field consists of the type declaration, and optionally an abstraction function, which 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) – this is particularly useful for specifying abstract datatypes, such as Java collections.

The frame condition has an important role in precisely specifying the effects of a method. It takes the form `@Modifies("f [s][u]")`, with three parts. The first, and the only mandatory part, is the name of the modifiable field, `f`. It is optionally followed by the *instance selector* (`s`) which specifies instances for which the field may be modified (taken to be "all" if not specified), and the *upper bound* (`u`) of the modification (taken to be the full extent of the field's type if not specified). In the Sudoku example (Listing 1) we used an instance selector to specify that only the empty cells may be modified. In SQUANDER, the instance selector provides the only mean of specifying the modifiable instances. If for some reason we wanted to additionally constrain the values that may be assigned to the cells, we could either add constraints to the `@Ensures` clause or use the upper bound clause of the frame condition. The difference is that the former is less efficient: the frame condition's upper bound is translated directly to Kodkod bounds, which shrinks the search space at the beginning of the search, as opposed to leaving it large and constraining the search by adding more rules.

4. FROM OBJECTS TO RELATIONS

SQUANDER execution begins when the utility method `Squander.exe()` is called by the client code, and 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 hold 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.

4.1 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 Section 6), and allows users to easily customize behavior for user-defined abstractions.

4.1.1 Keeping Track of Type Parameters

Java collection classes make extensive use of parametric types (also called "generics" in Java terminology). This allows the programmer to indicate the type of objects a collection may contain, e.g. a set of nodes (`Set<Node>`) rather than a set of arbitrary objects (`Set`). Unfortunately, though, 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. 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 the type parameter, the extent of the elements of a set is the set of all objects on the heap. 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 (see Section 4.4), which may dramatically improve Kodkod's running time.

Java reflection does, however, provide static types of fields and method parameters, which include type parameter information. Say 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.

4.2 Reading, Parsing and Type Checking JFSL

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 [23].

4.3 Example of a Translation

To illustrate translation, we'll first give an example even simpler than the Sudoku example from Section 2. Afterwards, we'll define the translation process more formally.

Consider an insertion into a binary search tree. For the purpose of translation, we don't need to give either the full specification or the full source code listing, since the structure of the heap alone is sufficient to explain most of the details. 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 (through the field `key`). The snapshot shown in Figure 2 gives an example of a heap consisting of a tree `t` (in the class `BST`), four node objects (of which only three are reachable), and four key values. 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 1. Relations in the upper section are unary, given relations, and represent objects found on the heap. The middle section contains relations that are also given, 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 an empty set and the upper bound is the extent of the field's type. Note, however, that SQUANDER also allows the users to tighten these bounds by specifying the exact objects that are allowed to be modified (using the *instance selector* clause of the frame condition, as defined in Section 3.2). For example, for the relation `left`, it is easy to specify in JFSL that the only left pointers that may be modified are those of nodes whose left pointers are currently set to null. As previously reported by Samimi et al. [18], this can lead to a significant speed-up in certain cases. To control the scope of the modification more finely, SQUANDER additionally allows the users to constrain the set of possible values for a field, by using the *upper bound* clause of the frame condition.

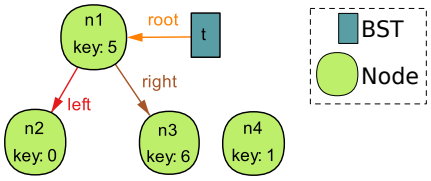

Figure 2: A snapshot for the pre-state of `t.insert(z)`

BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$
key:	$\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$				
root_pre:	$\{(t_1 \to n_1)\}$				
left_pre:	$\{(n_1 \to n_2), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$				
right_pre:	$\{(n_1 \to n_3), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$				
root:	$\{\}$,	$\{t_1\} \times \{n_1, n_2, n_3, n_4\}$			
left:	$\{\}$,	$\{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$			
right:	$\{\}$,	$\{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$			

Table 1: Translation of the heap from Figure 2 to bounds (a single exact bound or a pair of lower/upper bounds)

4.4 Defining Relations and Bounds

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 (and 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 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 in debugging) and has no performance impact.

For every field (including specification fields), a relation of type `fld`.*declType* $\to$ `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 2 summarizes how relations and bounds are created. Function `rel` takes a Java element and, depending on whether the element is modifiable, returns either one or two relations (the "`[]`" notation means "list of", and **R** is

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

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.

rel :: Element → [Relation]		
rel (Literal lit)	=	[**R**(lit.*name*, lit.*type*)]
rel (Type t)	=	[**R**(t.*name*, t)]
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 (Type t) (Relation r)	=	$\bigcup_{\text{lit<:t}}$ **B**(lit)
bound (Field fld) (Relation r)	=	
if is_mod(fld) ∧ is_post(r)		
B({}, ext(fld.*declType* × fld.*type*))		
else		
B($\bigcup_{\text{lit: Object}}$ lit × 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 : [])	=	$\bigcup_{\text{lit<:t}}$lit
ext (t : xs)	=	**ext** t × **ext** xs

Table 2: Translation of different Java constructs into relations (function `rel`) and bounds (function `bound`)

4.5 Restoring the Java Heap State

After running Kodkod to search for a solution, the Java heap state must be updated to reflect the outcome of the search. If Kodkod reports that a solution could not be found, SQUANDER simply throws an exception, which can be caught and handled by the client code. If there exist multiple solutions to the given problem, Kodkod will non-deterministically return one[2]. The solution is then restored to the Java heap by modification of the object fields through reflection. Finally, execution returns to the client code.

The process of restoring the heap is straightforward. During the translation to Kodkod, SQUANDER saves the mapping from Java objects to Kodkod atoms and from Java fields to Kodkod relations. Therefore, a Kodkod solution, containing values of all relations in the form of tuples of atoms, can be directly mapped back to the values of Java fields.

5. 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. For performance reasons, Kodkod uses a single sequential array indexed by a Java integer, and so the size of the matrix is limited to the largest integer values in Java (`Integer.MAX_VALUE`). Consequently, if the universe contains 1291 atoms or more, the matrix for a ternary relation would contain at least 1291^3 cells, which would exceed the size limit, so the relation could not be represented in Kodkod.

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 4.4 (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 Section 8), the heap contains more than 1900 objects.

5.1 "KodkodPart" Translation

Our KodkodPart translation achieves a universe with fewer atoms by establishing a mapping from Java objects (also called literals, as in Section 4.4) to Kodkod atoms that 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 2. Domains $\mathcal{D}$, literals $\mathcal{L}$, and assignments of literals to domains $\gamma : \mathcal{D} \to \mathcal{P}(\mathcal{L})$ for this example are summarized in Table 3.

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

Table 3: Summary of domains and instances

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 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:

[2]Kodkod can also enumerate all possible solutions (one at a time) and that functionality is exported by SQUANDER. This feature is for example useful for generating test instances that satisfy a certain property.

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

5.2 Partitioning Algorithm

For a given set of base domains $\mathcal{D}$, literals $\mathcal{L}$, and partitions $\mathcal{T}$ ($\mathcal{T}$ is of type $\mathcal{P}(\mathcal{D})$), and a given function $\gamma : \mathcal{D} \to \mathcal{P}(\mathcal{L})$ that maps domains to their instance literals, this algorithm produces a set of atoms $\mathcal{A}$ and a function $\alpha : \mathcal{L} \to \mathcal{A}$, so that for every partition p, function α 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 ψ 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 actually improve performance, we require the algorithm to produce a result that minimizes the cardinality of $\mathcal{A}$ (i.e. the total number of atoms).

In summary, the algorithm works as follows:

1. Dependencies between domains are computed. A domain depends on all domains with which it shares a partition. Let the function $\delta : \mathcal{D} \to \mathcal{P}(\mathcal{D})$ 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, so that $\alpha(l) = a$. From this point onwards, $\mathcal{A}$ is fixed.
4. For every other partition p iteratively, for all literals $l_p \in \psi(p)$ 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 obtained when atoms corresponding to all literals of all dependent domains is subtracted from $\mathcal{A}$.

For a discussion of why it is sufficient to simply find the largest partition and allocate that many atoms, see [14, §4.2]

A limitation of this technique is that, 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`), so that the result is equivalent to the original translation. Similarly, if there exists a single partition with more than 1290 literals, the problem will still be unrepresentable.

6. USER-DEFINED ABSTRACTIONS FOR LIBRARY TYPES

When specifying a program, we would like to be able to refer to the content of library types, to iterate over the elements of a Java set, for example. Clearly we do not wish to expose to the internals of the various `Set` implementations, so we cannot write declarative statements that directly refer to the fields of `HashSet`, for example. We also do not want our specification to depend on a concrete implementation of these classes. This is a common problem, and SQUANDER provides a generic solution by letting the users write *abstraction* and *concretization* functions for library classes.

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

The `.jfspec` files are written in JFSL. They contain abstract fields (the same `@SpecField` annotation is used) and invariants (`@Invariant`). Some abstract fields may have 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. The abstraction function (`absFunc`) takes a concrete object and returns values of its abstract fields. The concretization function (`concrFunc`) takes a concrete object and a value of an abstract field, and is expected to restore that value to the concrete object, by mutation. The code for serializers is straightforward and is omitted for lack of space.

SQUANDER provides built-in support for Java collections and Java arrays through this mechanism.

6.1 Specification for Java Sets and Maps

The abstract representation of a set is a set of elements, which is captured in a single `SpecField` named `elts`; similarly for maps, a binary relation also named `elts` is used to represent the mapping from keys to values (Listing 2). Additional fields (`size`, `keys`, `vals`) are defined for convenience; an abstraction functions is also provided so that the serializer is not required to implement abstraction and concretization functions for them. Additionally, an invariant is specified for maps constrain each key to map to at most one value.

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

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 2: Specification for `Set` and `Map`

6.2 Specifications for Java Lists and Arrays

To capture the abstract representation of a list, we again declare a single field, again named `elts`, but of type `int->E` (Listing 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 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.

Java arrays are likewise supported through this mechanism, and their specification is almost identical to that given in Listing 3. 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 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 3: Specification for `List`

7. EXAMPLES AND EVALUATION

In this section we demonstrate how our framework is efficient enough to be used – at least in some cases – as a primary implementation mechanism, where, in contrast, previous uses of solving during execution have focused on specialized applications (such as redundant computation as a fallback [18], or for exploring behavior in the early stages of the development process [17]).

7.1 Solving Hard Problems

If a problem is solvable in polynomial time, a careful manual implementation is likely to outperform a SQUANDER implementation. But if the problem itself is difficult, a solution with SQUANDER may turn out – due to the efficiency of SAT solvers – to perform better than a hand-written algorithm.

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. In Tables 4 and 5 we give total running time of our tool (`Squander`), translation time to Kodkod (`tTransl`), translation time from Kodkod to CNF (`tKodkod`) and SAT solving time (`tSAT`).

7.1.1 *"Hamiltonian Path" Algorithm*

A Hamiltonian path in a graph is one that visits each node in the graph exactly once. Listing 4 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.

```
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 4: Hamiltonian Path specification

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.

The results are shown in Table 4. 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 programming [7], 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 [20], which first guesses a solution, and then performs a local search using a

	Graphs without Hamiltonian paths								Graphs with Hamiltonian paths							
	10	14	15	20	25	30	35	40	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	0.01	50.17	214.96	t/o	t/o	t/o	t/o	t/o
Squander	0.19	0.29	0.28	0.37	0.57	1.2	1.8	3.59	0.22	0.45	0.31	0.87	8.76	98.08	t/o	t/o
tTransl	0.10	0.10	0.12	0.12	0.11	0.12	0.14	0.20	0.12	0.20	0.15	0.20	0.38	0.83	t/o	t/o
tKodkod	0.09	0.18	0.15	0.24	0.43	1.01	1.49	3.07	0.1	0.18	0.12	0.18	0.53	1.05	t/o	t/o
tSAT	0.0	0.01	0.01	0.01	0.03	0.07	0.17	0.32	0.0	0.07	0.04	0.49	7.85	96.2	t/o	t/o

Table 4: Hamiltonian path execution times

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)
```

Listing 5: NQueens Specification

Listing 5 gives the specification for N-Queens. The `nqueens` method takes an integer `n`, and a set already containing exactly `n` `Cell`[3] objects, and is expected to modify the coordinates of the given cells so that they represent a valid positioning of `n` queens.[4] The frame condition specifies that only cell coordinates are modifiable. The two bracketed subexpressions respectively mean that: (1) all `Cell` instances are modifiable, (2) 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 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. To improve clarity, we could introduce a special annotation, e.g. `@AdviceSpec`, to hold such redundant constraints which don't add anything new to the problem specification, but only help the performance.

n =	16	28	32	34	36	68
Manual	0.01	0.49	15.94	428.94	t/o	t/o
Squander	0.64	4.88	10.32	11.58	16.02	269.09
tTransl	0.18	0.57	0.93	1.1	1.34	17.57
tKodkod	0.38	1.45	2.59	3.08	3.54	32.71
tSAT	0.08	2.86	6.8	7.4	11.14	218.81

Table 5: N-Queens execution times (in seconds)

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

[4] 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.

Table 5 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.

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 [24] used the Kodkod [21] 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. Conceptually, the complexity of coming up with correct specification for the problem remains the same; the difference is that the user can now write about 30 lines of human-readable specifications, instead of manually writing 1500 lines of code to walk the heap, construct all the Kodkod relations, create all the constraints through the Kodkod API, and finally restore the solution onto the heap.

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 collection classes, 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 (`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 Section 5 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.

```
@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; }
```

```
public class Problem {
 private DegreeProgram dp; private Schedule schedule; private Set<Requirement> additionalReqs;
 @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 "semester to set of courses" map, but don't change the mapping for "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]",
 public void solve() { Squander.exe(this); }}
```

Listing 6: Partial specification for the course scheduler

8.1 Specification

The total number of lines of specification we wrote for this case study was less than 30. Here, we illustrate only the most interesting portion; the rest can be found in [14, ch. 8].

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. It may contain some existing assignments (e.g. courses already taken) which are not to be modified. Course prerequisites are defined (in disjunctive normal form) as sets of sets of courses, with an additional field (`prereqUsed`)) holding the choice of course used to satisfy a course's prerequisite. The specification that captures these invariants (and others) is shown in Listing 6.

The core of the specification for the scheduler is associated with the method `solve()` (Listing 6). 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", etc; departmental requirements are associated with `DegreeProgram`. The post-condition says that: (1) all requirements must hold, and (2) the schedule must include all courses specified by the department.

To express the first property, without having to know about all subclasses of `Requirement`, we simply defined a boolean specification field (named `cond`) and asserted that it evaluates to `true`. Specific classes that implement `Requirement` are expected to override the definition of `cond` to impose their own constraints. That way, we were able to specify several kinds of requirement (e.g. mandatory courses, minimal number of courses from a certain course group, etc.), but detailed description is beyond the scope of this paper.

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.

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 [10] 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 [9] argue that direct execution of specifications would inevitably lead to a decrease in the expressive power of the specification language. On the other side, Fuchs [8] 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).

In our previous paper [17], we suggested how executing specifications might play a useful role in an agile development process [3] (for fast prototyping, test input generation, creation of mock objects directly from interfaces, etc.), generalizing an idea that originally appeared in TestEra [13] in which test cases generated by solving a representation invariant were concretized into heap structures. Since then, we have developed the techniques discussed in this paper,

and turned the prototype implementation mentioned in our previous paper into a robust framework [2].

In the meantime, Samimi et al. implemented a tool [18], called PBnJ, that takes ideas from our earlier paper, 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 [17], 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 [16], comprising a combination of conventional imperative statements and declarative *specification statements* [15]. Wahls et al. are working on executing JML [5] (a specification language very similar to JFSL) by translation to constraint programs and using backtracking search [6, 12]. Oz also integrates constraint programming with high-level programming language and does so by using constraint propagation and search combinators [19]. Yang's LogLog tool [22] employs runtime constraint solving to automatically impute values for missing data based on declarative constraints.

10. CONCLUSIONS AND FUTURE WORK

In this paper, 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 paper (a) that we are now able – for a non-trivial class of problems – to use the mechanism 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.

In the future, we are hoping to explore a different translation mechanism, that would not only minimize the number of atoms in the universe, but the number of relations as well. For many problems, we compared the existing handwritten translations to Kodkod with those produced by SQUANDER. In almost all cases, the handwritten translations were more compact and used fewer relations. This is because object graphs usually contain many unmodifiable "links" that are only used to navigate from the root objects to the modifiable portion of the heap. A more clever translation could short-circuit some of those links, thus reduce the size of the object graph, and potentially decrease the solving time. We also plan to compare different techniques for solving declarative constraints, e.g. the backtracking ones as in JMLe [12] or Korat [4], with our current, SAT-solver based one.

11. REFERENCES

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

[2] Squander Home Page. `http://people.csail.mit.edu/aleks/squander`.

[3] K. Beck. *Extreme Programming Explained*. 1999.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *ISSTA'02*.

[5] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[6] N. Cata no and T. Wahls. Executing JML specifications of Java card applications: a case study. In *SAC'09*, pages 404–408, 2009.

[7] O. J. Dahl, E. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press, 1972.

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

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

[10] C. A. R. Hoare. An Overview of Some Formal Methods for Program Design. *IEEE Computer*, 20(9):85–91, 1987.

[11] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Apr. 2006.

[12] B. Krause and T. Wahls. jmle: A tool for executing JML specifications via constraint programming. In *FMICS'06*, Aug. 2006.

[13] D. Marinov and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *ASE'01*, 2001.

[14] A. Milicevic. Executable Specifications for Java Programs. Master's thesis, MIT, Sept. 2010.

[15] C. Morgan. The Specification Statement. *ACM Trans. Prog. Lang. Syst.*, 10(3), 1988.

[16] C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1998.

[17] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *Onward'09*, 2009.

[18] H. Samimi, E. D. Aung, and T. D. Millstein. Falling Back on Executable Specifications. In *ECOOP'10*.

[19] C. Schulte, G. Smolka, and J. Würtz. Encapsulated Search and Constraint Programming in Oz. In *PPCP*, pages 134–150. Springer-Verlag, 1994.

[20] R. Sosic and J. Gu. A polynomial time algorithm for the N-Queens problem. *SIGART B.*, 1(3):7–11, 1990.

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

[22] J. Yang. Specification-Enhanced Execution. Master's thesis, MIT, May 2010.

[23] K. Yessenov. A Light-weight Specification Language for Bounded Program Verification. Master's thesis, MIT, 2009.

[24] V. S. Yeung. Declarative Configuration Applied to Course Scheduling . Master's thesis, MIT, 2006.