An Imperative Extension to Alloy and a Compiler for its Execution

by

Joseph P. Near

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

© Massachusetts Institute of Technology 2010. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 7, 2010

Certified by . . . .
Daniel Jackson
Professor
Thesis Supervisor

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

An Imperative Extension to Alloy and a Compiler for its Execution

by

Joseph P. Near

Submitted to the Department of Electrical Engineering and Computer Science
on May 7, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis presents an extension of the Alloy specification language with the standard imperative programming constructs, allowing for the natural specification of dynamic systems. Using this extension, programmers can express stateful behavior directly, mixing declarative and imperative styles as desired. A relational semantics for the new imperative constructs will ensure that specifications written using the extension are translatable into the original Alloy language, allowing their analysis using the existing Alloy Analyzer.

The thesis also presents a compiler from the extended Alloy language to Prolog so that specifications may be efficiently executed. While the Alloy Analyzer's SAT-based analysis engine is incredibly fast in exploring a wide search tree, Prolog's unification-based strategy has the ability to delve very deeply into highly constrained search trees. Many specifications of dynamic systems have this property, making Prolog a perfect engine for executing them.

This combination of a language extension and a compiler for its execution represents an end-to-end solution for programming. The Alloy Analyzer allows the programmer to check properties of a high-level specification of the desired behavior, and the Prolog-based compiler allows the execution of that specification; if the compiled program is not fast enough, the programmer may refine *the specification* to make it faster, and the Alloy Analyzer will check that the refinement step has not introduced errors.

Thesis Supervisor: Daniel 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 am grateful to Daniel Jackson, whose expert guidance made this work possible.

To my friends and colleagues: Aleks, Derek, Eunsuk, Jean, Jonathan, Kuat, Rishabh, and Sasa. Their enthusiasm has given me immeasurable encouragement.

And to Marcie, whose love has made me a better person.

Contents

1 Introduction **7**
1.1 Dynamic Systems . . . 8
1.2 Language Extension . . . 8
1.3 Execution using Prolog . . . 9

2 The Alloy Language **10**
2.1 Relational Logic . . . 10
2.2 Commands & Analysis . . . 12
2.3 Dynamic Idioms . . . 13

3 Language Extension **15**
3.1 Dynamic Fields . . . 15
3.2 Named Actions . . . 15
3.3 Action Language . . . 16
3.4 Temporal Quantifiers . . . 17

4 Examples **18**
4.1 River Crossing . . . 18
4.2 Filesystem . . . 19
4.3 Insertion Sort . . . 21

5 Translation to Alloy **24**
5.1 Dynamic Idiom . . . 24
5.2 Translation . . . 24
5.3 Semantic Implications . . . 26
5.4 Semantic Correspondence . . . 26

6 Executing Alloy Specifications in Prolog **30**
6.1 Relational Expressions . . . 30
6.2 Imperative Constructs . . . 31
6.3 “New” vs. “Exists” . . . 32

7 Executing the Examples **33**
7.1 Address Book . . . 33
7.2 River Crossing . . . 35
7.3 Filesystem . . . 36

7.4 Insertion Sort . . . 38

8 Compiling to Prolog **41**
8.1 Expressions . . . 41
8.2 Formulas . . . 43
8.3 Actions . . . 43

9 Related Work **46**

10 Conclusions and Future Work **48**

List of Figures

2-1 Alloy Language Semantics 11

4-1 Alloy Analyzer Vizualization of the First, Second, and Final States of the River Crossing Solution 19

5-1 Rules for Translating the Action Language to Alloy 25
5-2 Address Book Example (Left) and its Translation (Right) 29
5-3 Structural Operational Semantics for Lookup, Assignment, and Sequencing 29

7-1 Scalability Comparison between Alloy Analyzer and Compiled Prolog: Sorting Concrete Sequences 40

8-1 Rules for Compiling Alloy Expressions into Prolog 42
8-2 Rules for Compiling Alloy Formulas into Prolog 43
8-3 Rules for Compiling Alloy Actions into Prolog 44

Chapter 1

Introduction

Alloy [21] is a modeling language based on first-order relational logic with transitive closure. It is designed to be simple but expressive, and to be amenable to automatic analysis. As such, few features are provided beyond first-order logic and transitive closure, making the semantics of the language easily expressible, understandable, and extendable.

Alloy is described as a "lightweight formal method," since the Alloy Analyzer supports fully automatic analysis of Alloy models. While this analysis is bounded and thus not capable of producing proofs, it does allow for incremental, agile development of models; and the *small-scope hypothesis* [4]—which claims that most inconsistent models have counterexamples within small bounds—means that modellers may have high confidence in the results.

This thesis extends the Alloy language with the standard imperative programming constructs—field update, sequential composition, and loops—to aid in the specification of dynamic systems. The extension is designed to be minimal yet expressive, in keeping with the Alloy philosophy. In particular, we allow the programmer to mix declarative and imperative constructs.

The aim of this work is to provide a language and end-to-end tool support for the construction of correct programs. To that end, we have developed both an automatic translation from our extended Imperative Alloy language to standard Alloy and a compiler from Imperative Alloy to Prolog. The translation to Alloy allows the *verification* and *animation*, using the Alloy Analyzer, of specifications written in Imperative Alloy; the compiler to Prolog allows for the *execution* of those specifications.

Of course not every specification can be executed in an efficient manner. Unsurprisingly, specifications with more imperative constructs and fewer declarative ones are better suited to our execution strategy. We envision a process in which programmers begin by writing a declarative specification and then make incremental refinements to it, slowly replacing declarative constructs with imperative ones. At each step, the Alloy Analyzer can be used to automatically verify that no errors have been introduced during refinement; the programmer may terminate the process when the execution engine is able to run his or her specification *fast enough* for the application at hand.

1.1 Dynamic Systems

Dynamic systems are those systems whose purpose is to change the state of the world. The declarative logics of most specification languages—including Alloy—do not include the notion of side effects, so some standardized technique is needed to model them.

The typical approach to modeling dynamic systems, and the one taken by the specification languages Z [36], VDM [25], and DynAlloy [12, 11], is to model state changes using pre- and post-conditions on each transition. Existing idioms for modeling dynamic systems in Alloy—as well as our approach—support this technique.

While this existing technique often works well, there are some operations—especially ones comprising multiple steps—that are difficult to express. Navigation through a filesystem based on a pathname is one example: the obvious solution is to define a single step of the navigation process and then to iterate. Instead, an Alloy user must define a recursive "flattening" relation representing all *possible* navigations, and then define a one-step operation that queries the relation. This is a poor solution in two ways: first, it is clumsy and produces visualizations that are not useful, as they portray every possible navigation; second, it destroys the separation of concerns, as implementing the navigation *operation* requires the addition of the navigation *relation*.

1.2 Language Extension

This thesis extends Alloy with the standard imperative constructs: field update, sequential composition, and loops; we give these operators the expected, *operational*, semantics.

Moreover, our language extension allows for the separation of the static and dynamic elements of a model. Our extension allows dynamic operations to be added to a static model: it makes updates to mutable state explicit and separates imperative operations from static properties. This separation of concerns is important to the design of a system, and is not well-supported by the Alloy idioms currently in use.

Finally, the use of imperative operators in specifications simplifies the process of implementation. Using our language extension, modelers have the option of refining specifications (in the style of Morgan [32]) until they are essentially imperative implementations. Each refinement step is automatically checked by the Alloy Analyzer to ensure that no errors have been made.

These advantages come at no loss of expressive power. We place no restrictions on the existing language, and also allow actions to be defined declaratively, using pre- and post-conditions; our framework and composition operators also apply to these declarative actions.

We have developed an automatic translator from our Imperative Alloy extension to standard Alloy, so that specifications written using the extension can be analyzed using the existing Alloy Analyzer. Our translation makes use of a standard Alloy idiom, allowing for efficient analysis and good defaults for visualization.

1.3 Execution using Prolog

To complete our end-to-end tool support, we have also developed a compiler from Imperative Alloy to Prolog. Prolog represents an appropriate target language, since it supports nondeterminism and provides a database for storing global relations; our compiler uses these features to simulate Alloy's relational operators, quantifiers, and classical negation.

The existing Alloy Analyzer is designed for the *verification* and *animation* of specifications. Our compiler is intended to complement the Analyzer by *executing* specifications. Animators perform their analyses within a fixed universe of predetermined size, while execution engines allow the creation of new objects. In practice, animators typically deal with models containing tens of objects, while execution engines must handle hundreds or thousands. In our case, this increased scalability comes at the cost of analysis: the Alloy Analyzer can check millions of cases against a property, while our compiler runs only a single, concrete test case. In exchange, the compiler provides efficiency: most specifications can be executed fast enough to serve as prototype implementations.

The two tools are designed to complement one another: in combination, they provide for both verification and execution of Imperative Alloy specifications, yielding end-to-end support for the construction of correct programs.

Chapter 2

The Alloy Language

Alloy [21] is a modeling language based on first-order relational logic with transitive closure. It is designed to be simple but expressive, and to be amenable to automatic analysis. As such, few features are provided beyond the quantifiers of first-order logic and the operators comprising the relational calculus, making the semantics of the language easily expressible, understandable, and extendable.

The Alloy Analyzer is a tool for automatic analysis of Alloy models. While this analysis is bounded and thus not capable of producing proofs, it does allow for incremental, agile development of models; and the *small-scope hypothesis* [4]—which claims that most inconsistent models have counterexamples within small bounds—means that modellers may have high confidence in the results. This sacrifice of completeness in favor of automation is in line with the *lightweight formal methods* philosophy [22].

2.1 Relational Logic

Alloy's universe is made up of uninterpreted atoms, each of which belongs to one of the disjoint sets defined using *signatures*. For example [21], the following signatures define sets of names and addresses:

```
sig Name, Addr {}
```

Signature definitions may also contain field declarations. Fields are meant to follow the intuition behind the fields of objects in object-oriented programming, but actually represent global relations. The definition below, for example, defines a signature for an address book with a field "addr" that maps names to addresses:

```
one sig Book {
  addr: Name →lone Addr
}
```

The "Book" signature defines a global relation "addr" that contains 3-tuples of books, names, and addresses; we say that it has type Book →Name →Addr. The "**lone**" keyword specifies that each book and name pair maps to at most one address, and is an example of a multiplicity constraint on the "addr" relation. Others include

$$
\begin{array}{rclcl}
M & :: & \text{formula} \rightarrow \text{binding} \rightarrow \text{boolean} \\
E & :: & \text{expression} \rightarrow \text{binding} \rightarrow \text{relation} \\
\text{relation} & :: & \{(\text{atom}, ..., \text{atom})\} \\
\text{binding} & :: & \{\text{variable} \mapsto \text{relation}\} \\
\hline
M(!f, b) & \hat{=} & \neg M(f, b) \\
M(f \;\&\&\; g, b) & \hat{=} & M(f, b) \wedge M(g, b) \\
M(f \;||\; g, b) & \hat{=} & M(f, b) \vee M(g, b) \\
M(f \Rightarrow g, b) & \hat{=} & M(f, b) \Rightarrow M(g, b) \\
M(\textbf{all } x\text{: } e \mid f, b) & \hat{=} & \bigwedge\{M(f, [x \mapsto v]b) | v \in E(e, b) \wedge |v| = 1\} \\
M(\textbf{some } x\text{: } e \mid f, b) & \hat{=} & \bigvee\{M(f, [x \mapsto v]b) | v \in E(e, b) \wedge |v| = 1\} \\
M(p \textbf{ in } q, b) & \hat{=} & E(p, b) \in E(q, b) \\
M(p = q, b) & \hat{=} & E(p, b) = E(q, b) \\
\\
E(p + q, b) & \hat{=} & E(p, b) \cup E(q, b) \\
E(p \;\&\; q, b) & \hat{=} & E(p, b) \cap E(q, b) \\
E(p - q, b) & \hat{=} & E(p, b) \backslash E(q, b) \\
E(p{+}{+}q, b) & \hat{=} & E(q, b) \cup \\
 & & \{(p_1, ..., p_n) | (p_1, ..., p_n) \in E(p, b) \wedge (p_1, q_2, ..., q_n) \notin E(q, b)\}) \\
E(p.q, b) & \hat{=} & \{(p_1, ..., p_{n-1}, q_2, ..., q_m) | \\
 & & (p_1, ..., p_n) \in E(p, b) \wedge (q_1, ..., q_m) \in E(q, b) \wedge p_n = q_1\} \\
E(p{\rightarrow}q, b) & \hat{=} & \{(p_1, ..., p_n, q_1, ..., q_m) | \\
 & & (p_1, ..., p_n) \in E(p, b) \wedge (q_1, ..., q_m) \in E(q, b)\} \\
E(\tilde{}p, b) & \hat{=} & \{(p_m, ..., p_1) | (p_1, ..., p_m) \in E(p, b)\} \\
E(\hat{}p, b) & \hat{=} & \{(x, y) | \exists p_1, ..., p_n | (x, p_1), (p_1, p_2), ..., (p_n, y) \in E(p, b)\} \\
E(x, b) & \hat{=} & b(x) \qquad \text{(variables)} \\
E(r, b) & \hat{=} & \bigcup\{b(r_i) | r_i \text{ has name } r\} \qquad \text{(relations)}
\end{array}
$$

Figure 2-1: Alloy Language Semantics

"**one**," which specifies exactly one mapping, and "**set**," which explicitly allows many mappings.

To interact with the atoms and relations defined by signatures, we may write top-level predicates whose bodies are logical formulas parameterized by a set of arguments. To look up a name in the address book, for example, we can use the following predicate:

```
pred lookup[b:Book, n:Name, a:Addr] {
    b.addr[n] = a
}
```

This predicate also illustrates the use of Alloy's two kinds of relational join. The first, written using the dot (.), is the standard relational join generalized over relations of arbitrary arity. The expression "b.addr" thus represents the relation of type Name →Addr obtained by joining the particular book "b" with the "addr" relation.

The second, written using square brackets ($a[b]$) represents the join of b on the *left* of a. That is, $a[b]$ is equivalent to $b.a$. The two types of notation allow the "lookup" predicate to look very much like the corresponding method in an object-oriented language; the same square-bracket notation can be used for predicate application.

The "lookup" predicate also illustrates another principle of Alloy's design: every expression has a relation as its value. The argument "b" is meant to represent a single book—a scalar—but is represented in Alloy as a relation of arity one and cardinality one. Sets are similarly represented as relations of arity one, but arbitrary cardinality. This conflation of scalars and sets with relations allows the standard relational operators (join, union, difference, *etc.*) to be used on sets and scalars, and allows intuitive operations like the join of a scalar with a relation in the "lookup" predicate. The semantics of these generalized operations is given in Figure 2-1.

2.2 Commands & Analysis

A command is an instruction to the Alloy Analyzer to perform an analysis. Commands specify whether a property should be *run* or *checked*; a *run* analysis involves a search for an example, while a *check* involves a search for a counterexample. A command also specifies bounds on the size of the universe to be considered in the analysis. For example, we might run a command to see an instance of an address book, or check the property that "lookup" has the intended meaning:

```
pred showInstance {
  some b: Book, n: Name, a: Addr |
    lookup[b, n, a]
}
run showInstance for 3 but 1 Book

assert lookupWorks {
  all b: Book, n: Name, a: Addr |
    lookup[b, n, a] ⇔n→a in b.addr
}
check lookupWorks for 4
```

This example illustrates several syntactic points. First, "**run**" goes with predicates, and "**check**" goes with assertions. Second, the user must provide a set of bounds on the size of the universe for each command; in the case of our "**run**," we have specified a maximum of three atoms of each type, but only a single address book. In addition, the example demonstrates the use of the "**some**" and "**all**" quantifiers, which correspond to the existential and universal quantifiers of first-order logic.

Internally, the Alloy Analyzer converts the model and the logical formula in the body of the command into a boolean formula. If the command is to be run, the Analyzer forms the conjunction of the model with the body of the command and instructs a SAT solver to find a satisfying instance. The Analyzer then maps this instance back to a set of atoms and a valuation for the model's global relations that

satisfy the constraints in the model and the command body. If the command is to be checked, the Analyzer forms the conjunction of the model with the *negation* of the command body. If the SAT solver finds a satisfying instance, it represents a counterexample to the desired property; if the boolean formula is unsatisfiable, then the property holds within the given bounds.

2.3 Dynamic Idioms

Since the Alloy language does not include built-in mechanisms for specifying stateful operations, modelers must model the notion of mutable state and the way in which that state changes over time. We briefly summarize the most popular idioms used to accomplish this.

The simplest way to define an operation that modifies an object o is as a predicate over its pre- and post-states o and o'. We can use this strategy to add stateful operations to the address book from the previous section, in an idiom we call *global state*:

```
pred add [b, b': Book, n: Name, a: Addr] {
  b'.addr = b.addr + n→a
}
```

This strategy is very simple; it doesn't require the addition of any dynamic information to the static model, and thus allows the separation of the dynamic and static concerns. It also makes checking some properties of the operations straightforward:

```
assert addLocal {
  all b, b': Book, n, n2: Name, a: Addr |
    add [b, b', n, a] && n != n2 ⇒ b.addr[n2] = b'.addr[n2]
}
```

While this strategy allows the expression of properties of single-step operations, it does not allow for properties of arbitrary executions of sets of actions. Moreover, it is difficult to see in the Analyzer's visualizer which object corresponds to which step in the operation. To solve these problems, we add execution traces to form the idiom we call *global state & traces*:

```
open util/ordering[Book]

fact Traces {
  all b: Book − last |
    some n: Name, a: Addr | add[b, b.next, n, a]
}
```

The additional fact organizes the address books in the universe into an ordered trace representing the progressively changing state of a single conceptual book. This strategy requires the possible actions to be enumerated in the fact (in this case, only one action is possible) and the trace is constrained so that one action is performed at every step in the trace. The instances produced by such models yield better

visualizations: the visualizer can be instructed to project over books, allowing the user to step through the book's states one at a time.

However, while the books in the trace represent the changing state of a single conceptual book, they are not the same object: the Alloy Analyzer represents each state using a unique atom. This means that the single conceptual book will not be equal to itself, for example; worse, it means that only a single conceptual book can be examined at once.

To resolve this issue, we introduce a third idiom, called *local state*, in which we add a new set of atoms to the universe representing the time-steps in the execution trace of the model. Then, we associate each tuple in the relations that change with the time-step at which that tuple is the "current" value of the relation. In our address book example, this means that there need be only a single book in the universe, and that book will change over time. Moreover, it means that actions are now defined in terms of the pre- and post-*times* they relate, instead of pre- and post-states:

```
open util/ordering[Time] as ord
sig Name, Addr, Time {}
sig Book {
  addr: Name →lone Addr →Time
}

pred add[b: Book, n: Name, a: Addr, t,t': Time] {
  b.addr.t' = b.addr.t + n→a
}
```

Since the time-steps are independent of the conceptual objects that change, many of these conceptual objects may change in a single time-step. The visualizer can be instructed to project over time-steps, and all of the changing objects can be examined simultaneously. Finally, no explicit enumeration of possible actions is required, as long as some action relates the beginning and ending time-steps, making it easier to combine single-step actions into larger ones.

For a more complete introduction to Alloy, the reader is referred to [21]. For completeness, we include the semantics (Figure 2-1) of the core Alloy language.

Chapter 3

Language Extension

A small extension to the Alloy language, summarized in this section, supports the modeling of dynamic systems.

3.1 Dynamic Fields

Immutable fields are declared in the traditional way:

sig Addr {}
sig Name {}

Mutable fields, whose values may vary with time, are defined using the "**dynamic**" keyword:

one sig Book { addr: **dynamic** (Name →**lone** Addr) }

This signature defines a single "Book" object with a mutable "addr" field mapping names to addresses.

3.2 Named Actions

Named actions can be defined at the top level, and can be invoked from within other actions. Adding an entry to the address book, for example, can be written as a named action that adds the appropriate tuple:

action add[n:Name, a:Addr] {
    Book.addr := Book.addr + (n →a) }

The deletion operation, on the other hand, removes all tuples containing a given name from the book:

action del[n:Name] {
    Book.addr := Book.addr − (n →Addr) }

3.3 Action Language

Our action language includes operators for imperative programming: field update, sequential composition, and loops. Pre- and post-conditions employ boolean-valued formulas (written φ) with the existing syntax and semantics of Alloy.

Act	::=	$o_1.f_1$, ..., $o_n.f_n$:= e_1, ..., e_n	(field updates)
	|	*Act* ; *Act*	(sequential composition)
	|	**loop** { *Act* }	(loop)
	|	*action*[a_1, ..., a_n]	(action invocation)
	|	**before** φ | **after** φ	(pre- and post-conditions)
	|	**some** v : τ| *Act*	(existential quantification)
	|	*Act* $\Rightarrow$ *Act* | *Act* $\wedge$ *Act* | *Act* $\vee$ *Act*	

A *field update* action changes the state of *exactly* those mutable fields mentioned, *simultaneously*. The action

b.addr := b.addr + (n→a)

for example, adds the mapping n→a to the address book b, while

a.addr, b.addr := b.addr, a.addr

swaps the entries of address books "a" and "b". *Sequential composition* composes two actions, executing one before the other:

add[n,a]; del[n,a]

performs the "add" operation and then the "del" operation.

A loop executes its body repeatedly, nondeterministically choosing when to terminate. The standard conditional loop may be obtained through the use of a post-condition; the action

loop {
  some name: b.addr.Addr | b.addr := b.addr − name→Addr
} && **after no** b.addr

for example, removes mappings from the address book "b" until the book is empty. Because they are nondeterministic, execution of these loops generally requires backtracking.

We view actions as relations between initial and final states. This view of actions allows for the lifting of the standard logical connectives and existential quantification into our action language, and for the mixing of declarative constraints with actions. The "**before**" and "**after**" actions, for example, introduce declarative pre- and post-conditions; these act as filters on other actions when combined using the logical connectives. The action

add[n,a] $\Rightarrow$ **after** n.Book.addr = a

for example, has executions that either end with the correct mappings in the address book or are not executions of "add."

3.4 Temporal Quantifiers

Actions have as free variables their beginning and ending states. Temporal quantifiers bind these variables: "**sometimes**," existentially; and "**always**," universally.

φ ::= < Alloy Formula >
  | **sometimes** | *Act*
  | **always** | *Act*

Given our view of actions as relations, a "**sometimes**" formula holds if and only if the action in its body relates *some* initial and final states; an "**always**" formula holds if and only if it relates *all* states. To visualize the result of adding the mapping n→a to the address book, for example, one executes the Alloy command:

run { **sometimes** | add[n,a] }

One can also check that "add" adds the mapping in all cases:

check { **always** | add[n,a] ⇒ **after** n→a **in** Book.addr }

Chapter 4

Examples

4.1 River Crossing

River crossing problems are a classic form of logic puzzle involving a number of items that must be transported across a river. Some items cannot be left alone with others: in our problem, the fox cannot be left with the chicken, or the chicken with the grain. A correct solution moves all items to the far side of the river without violating these constraints.

We begin by defining an abstract signature for objects, each of which eats a set of other objects and has a dynamic location. The objects of the puzzle are then defined as singleton subsets of the set of objects. Similarly, an abstract signature defines the set of locations, and two singleton sets partition it into the near and far sides of the river.

```
abstract sig Object { eats: set Object,
                      location: dynamic Location }
one sig Farmer, Fox, Chicken, Grain extends Object {}
abstract sig Location {}
one sig Near, Far extends Location {}
```

We define the "eats" relation to reflect the puzzle by constraining it to contain exactly the two appropriate tuples.

```
fact eating { eats = (Fox →Chicken) + (Chicken →Grain) }
```

The "cross" action picks an object *o* for the farmer to carry across the river, a new location *l* for the farmer, and a (possibly new) location *ol* for *o*, and moves the farmer and the object.

```
action cross {                                    −− pick an object & two locations
  some o: Object − Farmer, l: Location − Farmer.location, ol: Location |
  (Farmer.location := l, o.location := ol) && −− move the object and farmer;
    after (all o: Object |                        −− all objects end up with
             o.location = Farmer.location ||      −− the farmer, or not with
           (all o': (Object − o) |                −− objects they eat
             o'.location = o.location ⇒ o !in o'.eats)) }
```

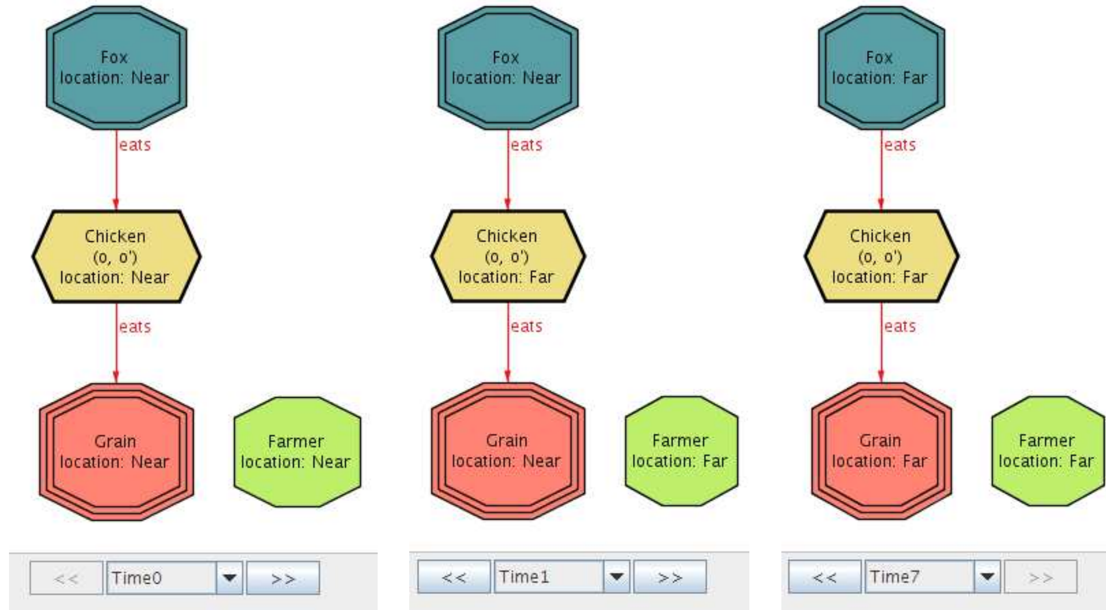

Figure 4-1: Alloy Analyzer Vizualization of the First, Second, and Final States of the River Crossing Solution

To obtain a solution, we find an execution that begins with all objects on the near side, calls "cross" repeatedly, and ends with all objects on the far side.

```
pred solvePuzzle {
  sometimes |                                       -- find some execution in which:
    before (all o: Object | o.location = Near) && -- objects start on near side,
    loop {
      cross []                                      -- cross runs repeatedly, and
    } && after (all o: Object | o.location = Far) } -- objects end on far side.
```

The "cross" action relies on the ability to mix declarative and imperative constructs: it chooses an object and a destination nondeterministically and then formulates the requirement that no object be eaten as a postcondition. In obtaining a solution, we have applied another imperative construct—**loop**—illustrating our ability to declaratively construct abstract actions and then compose them imperatively.

The Alloy Analyzer allows the vizualization of the puzzle's solution, and our implementation of the extension is designed to work well with the Analyzer's "magic layout" feature to allow the user to step through the evolving state in an execution. Figure 4-1 depicts the first two and last of the seven states of the correct solution to the river crossing puzzle.

4.2 Filesystem

As an example of the addition of dynamic operations to a static model, we present a simple filesystem. We begin with signatures for filenames and paths. File paths are

represented by linked lists of directories terminated by filenames.

```
sig Name {}
abstract sig FilePath { name: Name }
sig DirName extends FilePath { dnext: FilePath }
sig FileName extends FilePath {}
```

Next, we define the filesystem: an inode is either a directory node or a file node; a directory node maps names of files and directories to other inodes, and a file node contains some mutable data. The root node is a directory.

```
abstract sig INode {}
sig DirNode extends INode { files: Name →INode }
one sig RootNode extends DirNode {}
sig FileNode extends INode { data: dynamic Data }
sig Data {}
```

We now define operations over this static filesystem, beginning with navigation. We use a global MVar to hold the destination path, the current inode, and the data to be written to or read from the destination. One navigation step involves moving one step down the list representing the destination path and following the appropriate pointer to the corresponding inode.

```
one sig MVar { path: dynamic FilePath,
    current: dynamic INode, mdata: dynamic Data }

action navigate {                                   −− follow the path one step and then
    MVar.path := MVar.path.rest;                    −− update ‘‘current’’ to point to the
    MVar.current := (MVar.path.name).(MVar.current.files) −− corresponding inode
}
```

Reading from a file involves calling "navigate" until the destination inode has been reached and then reading its data into "MVar." Writing, similarly, involves navigation followed by a write.

```
action read {
    loop {
      navigate []                                   −− call navigate repeatedly
    } && after MVar.current in FileNode;            −− until we have reached the file inode
    MVar.mdata := MVar.current.data }               −− then read its data into MVar

action write {
    loop {
      navigate []                                   −− call navigate repeatedly
    } && after MVar.current in FileNode;            −− until we have reached the file inode
    let file = MVar.current |                       −− take the data from MVar
      file .data := MVar.mdata }                    −− and write it to the file inode
```

We would like a write to the filesystem followed by a read to yield the written data. We can verify this property by writing arbitrary data to an arbitrary file, reading it

back, and checking that the result is the original data. We use a global "Temp" to hold the original data.

```
one sig Temp { tdata: dynamic Data }

assert readMatchesPriorWrite {
  always |                                    -- if we begin at the root node,
    before (MVar.current = RootNode &&        -- and no file contains
            no f: FileNode | f.data = MVar.mdata) && -- MVar.mdata
    write [];                                 -- and we write MVar.mdata,
    Temp.tdata := MVar.mdata;                 -- store the original data,
    read [] ⇒                                 -- and read back the data
      after Temp.tdata = MVar.mdata }         -- then they're the same
```

This model illustrates the ability to build up multi-step actions using loops and sequential composition, and to verify properties of those actions.

4.3 Insertion Sort

Following Morgan [32], we present insertion sort as a refinement from a declarative specification to a deterministic, imperative implementation. We begin by defining mutable sequences of naturals and a declarative sortedness predicate.

```
sig Sequence { elts: dynamic seq Natural }

pred sorted[elts: seq Natural] {
  -- each element is less than the next
  all i: elts.inds - elts.lastIdx |
    let i' = i + 1 |
      i.elts <= i'.elts
}
```

Using this predicate, we can define a declarative sorting operation.

```
action declarativeSort [s: Sequence] {
  some s': Sequence |
    before (sorted[s'.elts] && s.elts = s'.elts) &&
    s.elts := s'.elts }
```

To bring this model closer to executable code, we define insertion sort as a series of swaps of elements of a sequence. We begin with a global counter and a declarative predicate to find the index of a sequence's smallest element, leaving the imperative definition of this predicate for later.

```
one sig Cnt { cur: dynamic Int }
pred minIdx [s: seq Natural, c,i: Int] {  -- i is the index greater than c whose
   i >= c && no i': s.inds | i' >= c && i'.s < i.s }     -- value in s is smallest
```

Next, we define the insertion step, in which the first element in the sequence is swapped, using relational override (++), with the smallest one.

```
action insertionStep [s: Sequence] {
  some i: s.elts.inds |                          -- nondeterministically pick an index
    (before minIdx[s.elts, Cnt.cur, i]) &&  -- whose element is smallest
    Cnt.cur := Cnt.cur + 1,                 -- and swap it with the first element
     s.elts := s.elts ++((Cnt.cur)→i.(s.elts)) ++(i→Cnt.cur.(s.elts)) }
```

The sorting action simply sets the counter to zero and runs the insertion step to the end of the sequence.

```
action insertionSort [s: Sequence] {
    Cnt.cur := 0;
    loop {
      insertionStep [s]
    } && after Cnt.cur = s.elts.lastIdx   }
```

Next, we show that the sort is correct by verifying that an arbitrary sequence is sorted when the sort completes.

```
assert sortWorks {
  all s: Sequence |
    always | insertionSort [s] ⇒ after sorted[s.elts] }
```

We now return to the problem of finding the minimum unsorted element in the sequence. We begin with a bit of global state to hold the current index in the search and the value and index of the minimal element found so far.

```
one sig Temp {idx: dynamic Int, min: dynamic Natural, minIdx: dynamic Int}
```

Next, we define an action to iterate over the subsequence s, checking each element against the minimal one found so far.

```
action findMin[s: Sequence] {
    Temp.idx := Temp.idx + 1;          -- increment the current index
    -- if the current value is less than the previous minimum, remember it
    (before Temp.idx.(s.elts) < Temp.min ⇒
      (Temp.min := Temp.idx.(s.elts), Temp.minIdx := Temp.idx)) &&
    (before Temp.idx.(s.elts) >= Temp.min ⇒ skip) } -- else nothing
```

Finally, we redefine insertionStep to use our new action.

```
action insertionStep [s: Sequence] {   -- start at the current index,
  Temp.idx := Cnt.cur, Temp.min := Cnt.cur.(s.elts), Temp.minIdx := Cnt.cur;
  loop {   -- run findMin over the suffix of the sequence,
    findMin[s]
  } && after Temp.idx = s.elts.lastIdx;
  (Cnt.cur := Cnt.cur + 1,   -- and swap minimum element with the current one
   s.elts := s.elts ++((Temp.minIdx)→Cnt.cur.(s.elts))
                    ++(Cnt.cur→Temp.minIdx.(s.elts))) }
```

Since our change was only incremental, we can show that the new sort refines the old one by verifying that repeating findMin yields the same element as our declarative minIdx.

```
assert findMinWorks {
  all s: Sequence |                                          -- for all sequences...
    always |
      (before (Temp.idx = Cnt.cur &&
               Temp.min = Cnt.cur.(s.elts) &&
               Temp.minIdx = Cnt.cur) &&
      loop {                 -- running findMin over the suffix of the sequence...
        findMin[s]
      } && after Temp.idx = s.elts.lastIdx) ⇒            -- finds the same element
           after minIdx[s.elts , Cnt.cur, Temp.minIdx] } -- as minIdx
```

Thus we can use the automated analysis our language extension affords us to support the stepwise refinement of a specification to executable, imperative code: our final version of insertionSort could easily be translated into an imperative programming language. Moreover, we have kept the analysis of our refinements tractable by performing it in a modular fashion, refining declarative specifications one at a time and analyzing the implementation of each separately.

Chapter 5

Translation to Alloy

We now present the translation (Figure 5-1) of our action language and associated operators into the first-order logic supported by the Alloy Analyzer.

5.1 Dynamic Idiom

Our translation uses two idioms that are common in the Alloy community for modelling dynamic systems. The first involves the addition of a "Time" column to each relation that represents local mutable state; the second involves the creation of a global execution trace using a total ordering on "Time" atoms.

Our translation adds a "Time" column to each **dynamic** field, and actions become predicates representing transitions from one time step to the next. We do not, however, enforce a global total ordering on time steps; instead, time steps are only partially ordered, allowing many traces to exist simultaneously.

In avoiding the single global trace, we gain the ability to compare executions, to run executions from within executions, and to run concurrent executions. The global trace does have performance and visualization benefits, however; fortunately, it is not difficult to infer that a particular analysis requires only a single trace, and then to enforce a total ordering on time steps. Our implementation performs this optimization, improving the performance and visualizability of many analyses considerably.

5.2 Translation

To translate our action language into a declarative specification following the trace-based idiom, we add a "Time" column to dynamic fields and thread a pair of variables through the action execution to represent the starting and ending time steps of that execution. We define a partial ordering on times using a field named "next:"

sig Time { next: **lone** Time }

We write the translation of action c into first-order logic in a *translation context* as $[\![c]\!](t, t')$ (or $[\![c]\!]C$ when the parts of C are not needed separately) where the context contains start and end time steps t and t'. We also assume a global set $sigs$ representing signatures with dynamic fields, and a global set of dynamic relations $fields$.

$$
\begin{aligned}
[\![o.f := e]\!](t,t') \quad &\hat{=} \quad o.f.t' = e[.]t \;\wedge \\
&\forall f' : (fields - f) \mid o.f'.t = o.f'.t' \;\wedge \\
&\forall o' : (sigs - o),\; f' : fields \mid o'.f'.t = o'.f'.t' \;\wedge \\
&t' = t.next \wedge t'.pc = fresh\ pc
\end{aligned}
$$

$$
\begin{aligned}
[\![\ c_1\ ;\ c_2\ ]\!](t,t') \quad &\hat{=} \quad \exists t_1 : Time \mid [\![c_1]\!](t,t_1) \wedge \\
&\qquad\qquad [\![c_2]\!](t_1,t')
\end{aligned}
$$

$$
\begin{aligned}
[\![\textbf{loop}\ \{c\}]\!](t,t') \quad &\hat{=} \\
&\exists\ begin, end : t.^*next - t'.\hat{}next \mid \\
&\quad [\![c]\!](t, begin) \wedge [\![c]\!](end, t') \wedge \\
&\quad \forall\ mid, mid' : t.^*next - end.\hat{}next \mid \\
&\qquad [\![c]\!](mid, mid') \Rightarrow \exists\ mid'' : mid'.\hat{}next \mid \\
&\qquad\qquad [\![c]\!](mid', mid'')
\end{aligned}
$$

$$
\begin{aligned}
[\![act[a_1, ..., a_n]]\!](t,t') \quad &\hat{=} \quad act[a_1, ..., a_n, t, t'] \\
[\![\textbf{before}\ \varphi\ ]\!](t,t') \quad &\hat{=} \quad \varphi[.]t \\
[\![\textbf{after}\ \varphi\ ]\!](t,t') \quad &\hat{=} \quad \varphi[.]t' \\
[\![\textbf{some}\ \ v : \tau \mid c]\!]C \quad &\hat{=} \quad \exists v : \tau \mid [\![c]\!]C \\
[\![c_1 \wedge c_2]\!]C \quad &\hat{=} \quad [\![c_1]\!]C \wedge [\![c_2]\!]C \\
[\![c_1 \vee c_2]\!]C \quad &\hat{=} \quad [\![c_1]\!]C \vee [\![c_2]\!]C \\
[\![c_1 \Rightarrow c_2]\!]C \quad &\hat{=} \quad [\![c_1]\!]C \Rightarrow [\![c_2]\!]C
\end{aligned}
$$

$$
\begin{aligned}
&[\![\textbf{action}\ name[\text{a1, ..., an}]\ \{\ Act\ \}]\!]\ \hat{=} \\
&\qquad \textbf{pred}\ name[\text{a1, ..., an, t, t'}]\ \{\ [\![Act]\!](t,t')\ \}
\end{aligned}
$$

$$
\begin{aligned}
[\![\textbf{sometimes} \mid Act\ ]\!] \quad &\hat{=} \quad \exists t, t' : Time \mid [\![Act]\!](t,t') \\
[\![\textbf{always} \mid Act\ ]\!] \quad &\hat{=} \quad \forall t, t' : Time \mid [\![Act]\!](t,t')
\end{aligned}
$$

Figure 5-1: Rules for Translating the Action Language to Alloy

We write $e[.]t$ to denote the replacement of every reference to a dynamic relation $f \in fields$ in e by the relational join $f.t$; this operation represents the evaluation of e at time t. We give the complete translation in Figure 5-1, and an example translation in Figure 5-2.

Assignment simulates the process of updating an implicit store. The first generated conjunct updates the field $o.f$ with the value of e at time t. The second and third represent the *frame condition* that the transition updates only f at o: the second ensures that the other fields of o do not change, while the third ensures the same for objects other than o. The fourth conjunct specifies that an update takes exactly one time step, and the fifth constrains the final time step's program counter.

Sequential composition is accomplished by existentially quantifying the time step connecting its two actions; loops are defined in terms of sequential composition. Action invocation passes the current time interval to the called action.

Named action definitions are translated into Alloy predicates with two extra arguments: the action's starting and ending times. The action representing the body is translated in the context of those times. A definition of an action is translated to a standard Alloy predicate, with the before and after times made explicit.

The translation of a "**sometimes**" formula existentially quantifies the beginning and ending states related by the result of translating the action in the body of the formula, while an "**always**" formula universally quantifies these states.

5.3 Semantic Implications

Our translation gives the language's imperative constructs the same relational semantics given by Nelson [33] to Dijkstra's original language of guarded commands [10]; these semantics also correspond to the standard operational semantics [34]. In addition, the relational semantics implies the existence of a corresponding semantics in terms of the weakest liberal precondition (namely, the *wlp*-semantics of Dijkstra's guarded commands, also given by Nelson [33]). Our translation does not, however, correspond to a semantics in terms of weakest preconditions. The use of *wp*-semantics allows termination to be expressed; our language can only express partial correctness properties.

The property that an abstract action of only one step is refined by another action is directly expressible. The same property for actions of more than one step, however, is not expressible due to the known problem of unbounded universal quantifiers in Alloy [28].

5.4 Semantic Correspondence

In this section, we show that the semantics of our assignment and sequential composition operators correspond to the standard semantics for those operators. We use Pierce's structural operational semantics for assignment and sequencing, reproduced in Figure 5-3.

To represent an update to the explicit store μ, Pierce writes $[l \mapsto v]\mu$, meaning "the store that maps l to v and maps each other location l' to $\mu(l')$." More formally:

Definition An *update* to the store μ is written $[l \mapsto v]\mu$. The set of mappings comprising the updated store $\mu' = [l \mapsto v]\mu$ is defined to be:

$$\{l \mapsto v\} \cup \{l' \mapsto v' \mid l' \neq l, l' \mapsto v' \in \mu\}$$

To show the correspondence, we view the strategy of adding a time column to existing relations as an alternative way of representing the stores of Pierce's semantics. We formalize the meaning of these implicit stores in terms of Pierce's stores and define equivalence between the two types of stores; we show that if two such stores are equivalent, then corresponding value lookups yield the same value.

Definition Let the tuple $< t, s_d, f_d >$ be an alternative notation for defining a store over the fixed set of locations $\{s.f \mid s \in s_d, f \in f_d\}$. The tuple $< t, s_d, f_d >$ represents the store containing the mappings $\{s.f \mapsto s.f.t \mid s \in s_d, f \in f_d\}$.

Definition Two stores $\mu_1 =< t, s_d, f_d >$ and $\mu_2 = [l_1 \mapsto v_1, ..., l_k \mapsto v_k]$ are *equivalent*, and we write $\mu_1 \equiv \mu_2$, if and only if:

1. μ_1 and μ_2 map the same locations:
   $\{s.f \mid s \in s_d, f \in f_d\} = \{l \mid l \mapsto v_n \in \mu_2\}$
2. Those locations map to the same values:
   $\forall\ s \in s_d, f \in f_d \mid s.f \mapsto s.f.t \in \mu_2$

Lemma 5.4.1 (Lookups are Identical) *If* $< t, s_d, f_d >\equiv \mu$ *then* $E[._{f_d}]t = E\mu$.

Proof $E[._{f_d}]t$ represents an expression in which every subexpression $o.f$ of E where $o \in s_d$ and $f \in f_d$ is replaced by $o.f.t$. By definition, if $< t, s_d, f_d >\equiv \mu$, then $o.f.t = \mu(o.f)$, so the corresponding subexpressions have identical values. ∎

We are now prepared to show the two main results: that the translations of our assignment and sequential composition operators have the same semantics as the operators presented operationally by Pierce. We do not consider evaluation of subterms; we assume that this will be delegated to the underlying set-theoretic semantics of Alloy.

Theorem 5.4.2 (Update Has Intended Semantics) *If*

1. $< t, s_d, f_d >\equiv \mu$
2. $(t, t', s_d, f_d)[\![o.f := e]\!]$ *holds*
3. $o.f := < e, \mu > \rightarrow < v, \mu' >$

then $< t', s_d, f_d > \equiv \mu'$.

Proof By item 3 and E-ASSIGN, we have that $\mu' = [o.f \mapsto e\mu]\mu$, which by our definition is the set of mappings $\{o.f \mapsto e\mu\} \cup \{l' \mapsto v' \mid l' \neq o.f, l' \mapsto v' \in \mu\}$. By lemma 5.4.1, we have that $e[._{f_d}]t = e\mu$ and that $\forall\ s \in s_d, f \in f_d \mid s.f.t = s.f\mu$. It follows by the definition of Figure 5-1 that:

1. $o.f \mapsto e\mu \in < t', s_d, f_d >$ (conjunct 1)
2. $\forall f' \in (f_d - f) \mid o.f' \mapsto o.f'\mu \in < t', s_d, f_d >$ (conjunct 2)
3. $\forall o' \in (s_d - o),\ f' \in f_d \mid o'.f' \mapsto o'.f'\mu \in < t', s_d, f_d >$ (conjunct 3)

Since the set of locations mapped by μ (and μ') is precisely:

$$\{o.f\} \cup \{o.f' \mid f' \in f_d, f' \neq f\} \cup \{o'.f' \mid o' \in s_d, o' \neq o, f' \in f_d\}$$

then by items 2 and 3 we have that $\{o'.f' \mapsto o'.f'\mu \mid o'.f' \neq o.f\} \in < t', s_d, f_d >$; it follows that $< t', s_d, f_d > \equiv \mu'$. ∎

Theorem 5.4.3 (Sequence Has Intended Semantics) *If*

1. $< t, s_d, f_d > \equiv \mu$,
2. $(t, t', s_d, f_d)[\![c_1\ ;\ c_2]\!]$ *holds*
3. $< c_1\ ;\ c_2, \mu > \to < v, \mu' >$

then $< t', s_d, f_d > \equiv \mu'$.

Proof By structural induction on terms. By the definition of Figure 5-1 and item 2, there is a time step t_1 such that $(t, t_1, s_d, f_d)[\![c_1]\!]$ and $(t_1, t', s_d, f_d)[\![c_2]\!]$. By inversion of E-SEQ and item 3, there is a store μ'' such that $< c_1, \mu > \to < v', \mu'' >$ and $< c_2, \mu'' > \to < v, \mu' >$. By the inductive hypothesis $< t_1, s_d, f_d > \equiv \mu''$; it follows that $< t', s_d, f_d > \equiv \mu'$. ∎

```
one sig Book {
  addr: dynamic (Name→lone Addr)}


action add[n:Name, a:Addr] {
  Book.addr := Book.addr + (n→a)}


assert addAdds {
  all n: Name, a: Addr |
    always | add[n,a] ⇒
      after n.Book.addr = a}
```

```
one sig Book {
  addr: Name→lone Addr→Time}

pred add[n:Name, a:Addr, t, t':Time] {
  t' = t.next && t'.pc = pc0 &&
  all o:Book−Book | o.addr.t = o.addr.t' &&
  Book.addr.t' = Book.addr.t' + (n→a) }

assert addAdds {
  all n: Name, a: Addr |
    all t, t': Time |
      add[n, a, t, t'] ⇒
        n.Book.addr.t' = a}
```

Figure 5-2: Address Book Example (Left) and its Translation (Right)

$$\frac{\mu(l) = v}{< l, \mu > \rightarrow < v, \mu >} \qquad \text{(E-LOOKUP)}$$

$$< l := v_2, \mu > \rightarrow < \text{unit}, [l \mapsto v_2]\mu > \qquad \text{(E-ASSIGN)}$$

$$\frac{< c_1, \mu > \rightarrow < v_1, \mu' > \qquad < c_2, \mu' > \rightarrow < v_2, \mu'' >}{< c_1 \; ; \; c_2, \mu > \rightarrow < v_2, \mu'' >} \qquad \text{(E-SEQ)}$$

Figure 5-3: Structural Operational Semantics for Lookup, Assignment, and Sequencing

Chapter 6

Executing Alloy Specifications in Prolog

Operations specified using actions in our language extensions look suspiciously like executable code, and we have shown that the control flow operators we use have the standard operational semantics, meaning that our specifications should be executable. The existing declarative parts of Alloy, however, present a problem: since our language allows arbitrary mixing of declarative and imperative constructs, a single declarative element might spoil our ability to execute a specification.

In order to execute the largest possible number of specifications, we have chosen to use Prolog as the execution engine. Like our extended specification language, large parts of the Prolog language have operational semantics, but Prolog also supports nondeterminism as a fundamental language construct. This support means that we retain the ability to execute many declarative elements of the Alloy language.

To compile Alloy to Prolog, we must construct representations for relations as first-class values, for imperative constructs, and for the relational operators. Fortunately, signature fields in Alloy can map to relations in Prolog's global database; Prolog execution engines typically handle nondeterminism well; and Prolog's unification allows for the expression of some types of quantification.

6.1 Relational Expressions

Expressions in Alloy evaluate to relations; these relations may be combined using the standard relational operators, and like Alloy's signature fields, may have arbitrary arity.

Relational values in Alloy may be thought of as sets of tuples. In compiling Alloy into Prolog, we can represent each such tuple using a term; to represent the set of tuples in a relation, we can yield multiple instantiations of the term—one for each tuple in the Alloy relation. For example, the Alloy relation $R = \{(a,b),(c,d)\}$ can be represented using the Prolog expression `R=(a,b) ; R=(c,d)`.

The other relational operators (intersection, difference, and so on) can similarly be compiled into expressions involving only Prolog's logical connectives. Formulas

involving relations can be compiled to comparisons between their possible instantiations: $r_1 \subseteq r_2$, for example, assuming that r_1 and r_2 are binary relations, becomes `forall(r1(A,B), r2(A,B))`.

We chose this style of compiling relational expressions to Prolog expressions with many instantiations to maximize laziness: it makes constructing relations very cheap. An alternative would have been to enumerate the values of each expression explicitly (*e.g.* in a list or in the global database); this strategy may make lookup faster, but it would force each relational operator to enumerate its results, taking time (and space) linear in the size of its argument relations. The relational operators encourage a style in which relation construction is at least as common as lookup, making our lazy style of evaluation more efficient.

6.2 Imperative Constructs

Imperative Alloy also differs from Prolog in its imperative constructs: assignment, sequential composition, and loops are notions built into the language. While these constructs are not built into Prolog, side effects *are* present in the form of `assert` and `retract`, conjunction can be used for sequencing, and loops can be defined in the language. Combined with our technique for compiling relational expressions, `assert` and `retract` allow the simulation of assignments. To retain the use of nondeterminism, we define `assertl` to assert a list of terms, and then retract them upon backtracking:

```
assertl(L) :- lassertaux(L); lretractaux(L), fail.
lassertaux([A|D]) :- assert(A), lassertaux(D).
lassertaux([]).
lretractaux([A|D]) :- retract(A), lretractaux(D).
lretractaux([]).
```

`assertl` implements backtracking updates, following a common Prolog idiom. The first clause of `assertl` asserts the list of updates using `lassertaux`. When backtracking occurs, the second clause is chosen; it retracts the erroneous updates and then fails, causing backtracking to continue. The two auxiliary predicates simply call `assert` or `retract` on each element of a list of rules.

Our current implementation keeps track of the entire history of each global relation. The compiler adds a single argument to each global relation representing a *time-step*; when compiling an expression that references one of these relations, we pass the current time-step to the relation and it instantiates the other arguments with its value at that time-step. The "addr" relation from the address book, for example, is given the type Time→Name→Addr, and a reference to "addr" relating name `N` to address `A` at time `T` would be compiled to `addr(T, N, A)`. We place the time argument in the first position because most Prolog systems index on that argument: relation histories may become long, so indexing on time is important for efficiency. The relation history is necessary because the value of a relation at one time is often dependent on its previous value; parts of the history upon which no "current" values depend may be eliminated in a process analagous to garbage collection.

This infrastructure makes compiling field assignments straightforward. An update of the form $o.f := e$ is translated by compiling e at the current time-step, then using `assertl` to update the global relation f. The first two arguments to f in the update are the next time-step and o; the remaining arguments are the free variables of the result of compiling e, and the body is the expression to which e compiles. The update x.f := y + 5, for example, compiled at time-step `T`, might become `assertl((f(T,X,Z) :- Z is Y + 5))`.

6.3 "New" vs. "Exists"

Besides decidability of language constructs, perhaps the most important difference between an executable language and a logical one built for analysis is that the former must allow for the construction of new objects, while the latter assumes an existing universe from which objects with desired properties may be selected. For this reason, specifications in logics such as Alloy's often use the existential quantifier to simulate object construction; inferring which quantifiers are intended to simulate construction is impossible in general. For example, a side-effecting "cons" action might be written in Alloy as follows:

```
action cons[e: Elem, l: List] {
  some c: ConsCell |
    before c.elem = e && l.ls := c
}
```

There is no way for a compiler to determine whether the programmer intends for the existential quantifier to *create* a cons cell or to find an *existing* one. To the Alloy Analyzer, this distinction does not exist, but in Prolog, it is important.

Fortunately, specifications of behavior in high-level languages like Alloy typically construct very few objects, and instead specify *manipulations* of existing objects. It is not a great burden on the programmer, then, to require the explicit specification of object construction, allowing our execution engine to construct new objects in the standard way during execution, and the Analyzer to simulate that construction using existential quantification during analysis. In our "cons" action, for example, the programmer would replace "**some**" with "**new**:"

```
action cons[e: Elem, l: List] {
  new c: ConsCell |
    before c.elem = e && l.ls := c
}
```

Chapter 7

Executing the Examples

7.1 Address Book

The address book specification from Section 2 can be directly translated into Prolog using our technique. The signature definitions, originally written as:

sig Addr, Name {}
one sig Book {
    addr: **dynamic** (Name →Addr)
}

Can be translated as follows:

```
:- dynamic sigAddr/1, sigName/1, sigBook/1, addr/4.
sigBook(gensym30).
```

This code defines the global mutable relation "addr," as well as the relations representing membership in the three signatures defined in the specification. For example, `sigAddr(A)` succeeds if and only if `A` is an address atom, and `addr(B, N, A, T)` succeeds if and only if the "addr" field of the address book `B` contains the mapping from name `N` to address `A` at time `T`. Since the "Book" signature is defined with the keyword "**one**," a fact is generated to declare that the atom `gensym30` is a book. Next, we examine the "add" action and its translation:

action add[n:Name, a:Addr] {
    Book.addr := Book.addr + (n→a)
}

```
add(N, A, T0, T1) :-
  T1 is T0 + 1, sigBook(B),
  assertl([((addr(T1, B, Np, Ap) :-
             (Np = N, Ap = A));
             (sigBook(B2), addr(T0, B2, Np, Ap)))]).
```

The action is compiled into a predicate that relates not only the original two arguments but also the starting and ending times of the action's execution. Since the body of the action contains only a single update, the compiled predicate begins by specifying that the action takes only a single time-step. Next, the predicate updates the relation "addr" at the object "Book" by constraining `B` to be a book and then asserting a new rule for `addr`. The time step for `addr`'s new value—in this case, `T1`—is its first argument; the second argument is `B`, the book whose "addr" field we wish to update.

The new rule for `addr` specifies that its value can come either from the old value of `addr` (in the first disjunct) or from the tuple n→a (in the second). The action for deleting a name from the book can be translated in a very similar way:

action del[n:Name] {
  Book.addr := Book.addr − (n→Addr)
}

```
del(N, T2, T3) :-
  T3 is T2 + 1, sigBook(B),
  assertl([((addr(T3, B, Np, Ap) :-
              sigBook(B2), addr(T2, B2, Np, Ap),
              \+ (Np = N, sigAddr(Ap))))]).
```

In this case, however, the new rule for `addr` specifies that values must come from the old value of `addr` and must *not* be present in the tuple n→Addr. To express this in Prolog, we use negation-as-failure (`\+`), which is known to be problematic. If the argument `N` were not instantiated, for example, this predicate could fail when it should not, since `N` could be chosen specifically to be `Np`. Fortunately, it is straightforward to generate mode restrictions on the compiled predicates that prevent unwanted behavior and ensure the safety of this style of negation.

We can also compile a multi-step action. The "replace" action first removes all addresses for a name, then adds the given address:

action replace[n: Name, a: Addr] {
  Book.addr := Book.addr − (n→Addr);
  Book.addr := Book.addr ++(n→a)
}

```
replace(N, A, T4, T6) :-
  T5 is T4 + 1, sigBook(B),
  assertl([((addr(T5, B, Np, Ap) :-
              sigBook(B2), addr(T4, B2, Np, Ap),
              \+ (Np = N, sigAddr(Ap))))]),
  T6 is T5 + 1, sigBook(B3),
  assertl([((addr(T6, B3, Np, Ap) :-
              (Np = N, Ap = A, !) ;
              (sigBook(Var24), addr(T6, Var24, Np, Ap))))]).
```

Since it contains two sequential updates, this action takes two time-steps. The right-hand side of the first update contains a relational difference, resulting in the use of negation-as-failure. The second update contains a relational override, the simulation of which in Prolog requires the use of the cut (`!`). In this case, the cut prevents values from coming from the old rule for `addr`.

7.2 River Crossing

While the river crossing puzzle can be compiled to Prolog, its style of execution is not a good match for this particular problem. Solving the river crossing puzzle is a search problem: the specification is inherently nondeterministic. The only way to find the solution is to try all possibilities. Alloy's SAT-based analysis engine is far better suited to this task than the rule-based system underlying Prolog. For completeness, we present the translation of the action that solves the puzzle; after running for more than an hour, however, this program ran out of memory. The Alloy Analyzer, by contrast, found the solution in less than one second.

```
abstract sig Object { eats: set Object,
                      location: dynamic Location }
one sig Farmer, Fox, Chicken, Grain extends Object {}
abstract sig Location {}
one sig Near, Far extends Location {}

action cross {                                        -- pick an object & two locations
  some o: Object − Farmer, fl: Location − Farmer.location, ol: Location |
  (Farmer.location := fl , o.location := ol) &&   -- move the object and farmer;
    after (all o: Object |                            -- all objects end up with
              o.location = Farmer.location ||         -- the farmer, or not with
           (all o': (Object − o) |                    -- objects they eat
              o'.location = o.location ⇒ o !in o'.eats)) }
```

→

```
cross(T0, T1) :-
  sigObject(O), \+ sigFarmer(Var1), sigLocation(Fl),
  \+ (sigFarmer(Var5), location(T0, Var5, Fl)),
  sigLocation(Ol), T1 is T0 + 1, sigFarmer(Var9),
  assertl([((location(T1, Var9, Fl) :- true)),
           ((location(T1, O, Ol) :- true))]),
  forall((sigObject(O1)),
         ((location(T1, O1, Var15), sigFarmer(Var14), location(T1, Var14, Var15));
          (forall((sigObject(Op), \+ (Op = O1)),
                  ((\+ (location(T1, Op, Var21), location(T1, O1, Var21)));
                   (\+ eats(Op, O1))))))).
```

Because the assignment that updates the current location of the chosen object is essentially declarative, it picks both the object to move and its destination nondeterministically. Only after the assignment is made does the program check that none of

the constraints is violated—and even if the chosen movement is allowed, it may not actually make progress towards a solution. The sheer number of possible solutions, and the lack of deterministic constraints on them, makes this a very difficult problem for a Prolog implementation.

7.3 Filesystem

A filesystem is a perfect candidate for development using our approach. As shown earlier, the specification can be defined and verified using the Alloy Analyzer; by executing precisely that specification in Prolog, we can be assured of the filesystem's correct operation. As in the other examples, we begin by translating the signatures of the original specification.

```
sig Data {}
abstract sig INode {}
sig DirNode extends INode { files: Name →INode }
sig FileNode extends INode { data: dynamic Data }
one sig RootNode extends DirNode {}

sig Name {}
abstract sig FilePath { name: Name }
sig DirName extends FilePath { dnext: FilePath }
sig FileName extends FilePath {}

one sig MVar {
   path: dynamic FilePath, current: dynamic INode, mdata: dynamic Data
}
```

→

```
:- dynamic sigName/1, sigFilePath/1, sigDirName/1, sigFileName/1,
           sigINode/1, sigDirNode/1, sigRootNode/1, sigFileNode/1,
           sigData/1, sigMVar/1, data/3, path/3, current/3,
           mdata/3, name/2, dnext/2, files/3.
sigRootNode(gensym62).
sigMVar(gensym63).
```

Next, we compile the action for navigation. Both steps of this action are simple updates, but since the filesystem has many more mutable relations than the previous examples, each call to `assertl` also contains several rules representing the frame condition. In the first call below, for example, a new rule for the `data` relation is produced that delegates its values to the old definition of the relation.

```
action navigate {
   MVar.path := MVar.path.dnext;
   MVar.current := (MVar.path.name).(MVar.current.files)
```

}

→

```
navigate(T0, T1) :-
  T2 is T0 + 1, sigMVar(Mv),
  assertl([((path(T2, Mv, Path) :- sigMVar(Mv2), path(T0, Mv2, Var3), dnext(Var3, Path))),
           ((data(T2, Var6, Var7) :- data(T0, Var6, Var7))),
           ((current(T2, Var8, Var9) :- current(T0, Var8, Var9))),
           ((mdata(T2, Var10, Var11) :- mdata(T0, Var10, Var11)))]),
  T1 is T2 + 1, sigMVar(Mv3),
  assertl([((current(T1, Mv3, Var22) :-
              sigMVar(Mv4), path(T2, Mv4, Var15), name(Var15, Var21),
              sigMVar(Mv5), current(T2, Mv5, Var20), files(Var20, Var21, Var22))),
           ((data(T1, Var24, Var25) :- data(T2, Var24, Var25))),
           ((path(T1, Var26, Var27) :- path(T2, Var26, Var27))),
           ((mdata(T1, Var28, Var29) :- mdata(T2, Var28, Var29)))]).
```

We compile the action for reading from the filesystem in a similar way, except for its loop and declarative post-condition. We can compile loops to nondeterministic repetition of an action, which we implement using the `loop` predicate. The post-condition checks that the current node is a file using the subset operator; in Prolog, this requires checking that the right-hand side of the "**in**" formula succeeds for every possible instantiation of the left-hand side.

action read {
  loop { navigate[] } && **after** MVar.current **in** FileNode;
  MVar.mdata := MVar.current.data
}

→

```
read(T3, T4) :-
  loop(T3, T5, navigate, []),
  forall((sigMVar(Mv), current(T5, Mv, Var33)), sigFileNode(Var33)),
  T4 is T5 + 1, sigMVar(Var39),
  assertl([((mdata(T4, Var39, Var38) :-
              sigMVar(Var35), current(T5, Var35, Var37), data(T5, Var37, Var38))),
           ((data(T4, Var40, Var41) :- data(T5, Var40, Var41))),
           ((path(T4, Var42, Var43) :- path(T5, Var42, Var43))),
           ((current(T4, Var44, Var45) :- current(T5, Var44, Var45)))]).
```

Finally, we compile the action for writing to the filesystem. Except for the "**let**" formula, it is nearly identical to that for reading.

action write {
  loop { navigate[] } && **after** MVar.current **in** FileNode;
  let file = MVar.current | file .data := MVar.mdata
}

→

```
write(T6, T7) :-
  loop(T6, T8, navigate, []),
```

```
forall((sigMVar(Var47), current(T8, Var47, Var49)), sigFileNode(Var49)),
sigMVar(Var51), current(T8, Var51, File), T7 is T8 + 1,
assertl([((data(T7, File, Var55) :- sigMVar(Var54), mdata(T8, Var54, Var55))),
         ((path(T7, Var56, Var57) :- path(T8, Var56, Var57))),
         ((current(T7, Var58, Var59) :- current(T8, Var58, Var59))),
         ((mdata(T7, Var60, Var61) :- mdata(T8, Var60, Var61)))]).
```

This collection of predicates represents a simplified model of a filesystem that can be executed by a Prolog system; combined with a tool like FUSE (Filesystem in User Space [14]), it can be used as a prototype implementation to store real data and be tested on an actual system. The user may add features slowly, using the Alloy Analyzer to verify their correctness.

7.4 Insertion Sort

Finally, we present an executable version of the insertion sort specification. The sort operates on a mutable sequence of natural numbers; the sequence is implemented as a relation from integer indices to these natural numbers.

The “minIdx” predicate declaratively finds the index in the sequence that holds the smallest value. The “insertionStep” action uses “minIdx” to find the index whose value should be swapped with that of the current index and performs that swap. The completed sort, then, is obtained by iterating the insertion step until the final index has been reached, meaning that the whole sequence is sorted.

```
one sig Sequence {
   a: dynamic (seq Natural) }

one sig Cnt { cur: dynamic Int }

pred minIdx [s: Sequence, c,i: Int] {
   -- i is the index greater than c whose value in s is smallest
   i >= c && (all i': s.a.inds | (! i' < c) || lt[i'.(s.a), i.(s.a)])  }

action insertionStep [s: Sequence] {
   some i: s.a.inds |
      (before minIdx[s, Cnt.cur, i]) && -- find the smallest element
         -- and swap it with the first element
         (Cnt.cur := Cnt.cur + 1,
          s.a := s.a ++((Cnt.cur)→i.(s.a)) ++(i→Cnt.cur.(s.a))) }

action insertionSort [s: Sequence] {
   -- run the insertion step until the end of the sequence
   loop { insertionStep[s] } && after Cnt.cur = lastIdx[s.a]  }

assert sortworks {
   -- sorting the whole sequence results in a sorted sequence
```

always | **before** Cnt.cur = 0 && insertionSort[Sequence] ⇒
after sorted[Sequence.a] }

The translation of this specification to Prolog raises several interesting issues. First, each field update is compiled to a larger piece of code, since the inclusion of multiple mutable relations in the specification makes frame conditions necessary. Second, the specification makes use of arithmetic and loops, which require power from the underlying Prolog implementation. Finally, the specification makes explicit use of the universal quantifier, which the compiler translates to the Prolog `forall`.

```
:- dynamic sigSequence/1, sigCnt/1, a/4, cur/3.

sigSequence(gensym125).
sigCnt(gensym126).

minIdx(S, C, I, T3) :-
  I >= C,
  forall((a(T3, S, Ip, _)),
         ((\+ (Ip < C)) ;
          (a(T3, S, Ip, Val), a(T3, S, I, Val2), lt(Val, Val2, T3)))).

insertionStep(S, T4, T5) :-
  a(T4, S, I, _),
  sigCnt(Cnt), cur(T4, Cnt, CurCnt), minIdx(S, CurCnt, I, T4),
  T5 is T4 + 1, sigCnt(Cnt2),
  assertl([((cur(T5, Cnt2, CurCnt2) :-
               sigCnt(Cnt3), cur(T4, Cnt3, CurCnt3),
               CurCnt2 is CurCnt3 + 1)),
           ((a(T5, S, Ip, Val) :-
              (Ip = I, sigCnt(Cnt4), cur(T4, Cnt4, CurCnt4),
               a(T4, S, CurCnt4, Val), !) ;
              (sigCnt(Cnt5), cur(T4, Cnt5, Ip), a(T4, S, I, Val), !) ;
              a(T4, S, Ip, Val)))]).

insertionSort(S, T6, T7) :-
  loop(T6, T7, insertionStep, [S]),
  sigCnt(Cnt), cur(T7, Cnt, CurCnt),
  findall(Idx, (a(T7, S, Idx, _)), IdxList),
  max(IdxList, CurCnt).
```

Because this example is nondeterministic and our translation has not yet been optimized, our sorting predicate is not as fast as it could be. Even so, it is certainly fast enough to function as a prototype implementation: it can sort a sequence of a hundred elements in just a few seconds, as shown in the comparison graph in Figure 7-1. The Alloy Analyzer, on the other hand, cannot sort more than ten elements in a reasonable time.

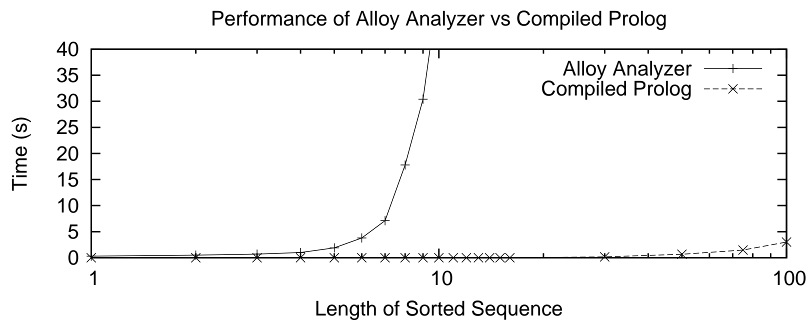

Figure 7-1: Scalability Comparison between Alloy Analyzer and Compiled Prolog: Sorting Concrete Sequences

Chapter 8

Compiling to Prolog

Our compiler transforms a complete Alloy specification into a Prolog program. The resulting program uses rules and facts in the global database to represent the fields associated with signatures, and also defines the set of atoms belonging to each signature using global facts. This approach is intended to closely parallel the Alloy universe of global relations over a finite set of atoms.

8.1 Expressions

Alloy expressions evaluate to relations. In Alloy, sets are represented as unary relations; scalars, then, are singleton sets. For an Alloy expression whose value is an n-ary relation, our compiler produces a Prolog expression with n free variables; each possible instantiation of those free variables represents one tuple of the original relation.

Our compiler therefore produces a 2-tuple (e, v) containing the compiled Prolog expression e and a list of free variables v. We present the complete set of compilation rules used in our implementation in Figure 8-1; r represents the set of global relations in the original Alloy model, while r_d is the set of dynamic relations.

In addition to the expression to compile, our translation requires a representation of the time-step at which the expression is being evaluated. Our implementation represents time-steps using integers; each global relation accepts one of these time-steps as its first argument and instantiates its other arguments to the values of the relation at that time-step.

Some relational operators (*e.g.* union, intersection) are trivial to express in Prolog. Others (*e.g.* difference, override), however, require the use of the cut or negation-as-failure. These impure elements restrict the contexts in which the compilation produces useful programs: the Alloy expression !(i < j), for example, produces the Prolog expression `\+ (I<J)`, which will not correctly instantiate `I` or `J`.

C_E	::	expression → time → (Prolog expression, [variable])
$C_E(a,t)$ $(a \in vars)$	$\hat{=}$	$(\emptyset, [A])$
$C_E(f,t)$ $(f \in r)$	$\hat{=}$	$(f(A_1, A_2, ..., A_n), [A_1, A_2, ..., A_n])$ where f has arity n; $A_1, ..., A_n$ are fresh variables
$C_E(f,t)$ $(f \in r_d)$	$\hat{=}$	$(f(A_1, A_2, ..., A_n, t), [A_1, A_2, ..., A_n])$ where f has arity n; $A_1, ..., A_n$ are fresh variables
$C_E(e_1 \rightarrow e_2, t)$	$\hat{=}$	$((E_1, E_2), [A_1, ..., A_n, B_1, ..., B_n])$ where $C_E(e_1, t) = (E_1, [A_1, ..., A_n])$ and $C_E(e_2, t) = (E_2, [B_1, ..., B_n])$
$C_E(e_1.e_2, t)$	$\hat{=}$	$((A_n = B_1, E_1, E_2), [A_1, ..., A_{n-1}, B_2, ..., B_n])$ where $C_E(e_1, t) = (E_1, [A_1, ..., A_n])$ and $C_E(e_2, t) = (E_2, [B_1, ..., B_n])$
$C_E(e_1 + e_2, t)$	$\hat{=}$	$((A_1 = B_1, ..., A_n = B_n, E_1; A_1 = C_1, ..., A_n = C_n, E_2), [A_1, ..., A_n])$ where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$ and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$
$C_E(e_1 - e_2, t)$	$\hat{=}$	$((A_1 = B_1, ..., A_n = B_n, E_1, A_1 = C_1, ..., A_n = C_n, \backslash\texttt{+}E_2), [A_1, ..., A_n])$ where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$ and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$
$C_E(e_1 \& e_2, t)$	$\hat{=}$	$((A_1 = B_1, ..., A_n = B_n, E_1, A_1 = C_1, ..., A_n = C_n, E_2), [A_1, ..., A_n])$ where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$ and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$
$C_E(e_1{+}{+}e_2, t)$	$\hat{=}$	$((A_1 = C_1, ..., A_n = C_n, E_2, \texttt{!}; A_1 = B_1, ..., A_n = B_n, E_1), [A_1, ..., A_n])$ where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$ and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$

Figure 8-1: Rules for Compiling Alloy Expressions into Prolog

C_M	::	formula → time → Prolog expression
$C_M(e_1 \in e_2, t)$	$\hat{=}$	`forall(`$(E_1), (B_1 = C_1, ..., B_n = C_n, E_2)$`)` where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$ and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$

$C_M(e_1 = e_2, t)$	$\hat{=}$	`forall(`$(E_1), (B_1 = C_1, ..., B_n = C_n, E_2)$`),`
		`forall(`$(E_2), (B_1 = C_1, ..., B_n = C_n, E_1)$`)`
		where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$
		and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$

$C_M(f_1 \&\& f_2, t)$	$\hat{=}$	$C_M(f_1, t)$ `,` $C_M(f_2, t)$
$C_M(f_1 \,\|\|\, f_2, t)$	$\hat{=}$	$C_M(f_1, t)$ `;` $C_M(f_2, t)$
$C_M(!f, t)$	$\hat{=}$	`\+` $C_M(f, t)$
$C_M($**all** x:ite $\|\, f, t)$	$\hat{=}$	`forall((`$x = A$`,`E`),` $C_M(f, t)$`)` where $C_E(e, t) = (E, [A])$
$C_M($**some** x:ite$\|f, t)$	$\hat{=}$	$x = A$`,`E`,`$C_M(f, t)$ where $C_E(e, t) = (E, [A])$

Figure 8-2: Rules for Compiling Alloy Formulas into Prolog

8.2 Formulas

Compiling Alloy formulas is straightforward, since Alloy's logical connectives map directly to those of Prolog. The equality and subset operators are the most interesting: since expressions evaluate to relations, both logical operators must examine *all* instantiations of the expressions' free variables generated by the resulting Prolog expressions. Figure 8-2 contains the rules for compiling formulas; again, the rules require the time at which the formula is being evaluated.

8.3 Actions

The time-steps required for expressions and formulas are manipulated in the compilation of commands. The rule for field assignment updates the global relation f at the object o and time step $t + 1$ with the results of the right-hand side expression e. Each field assignment, then, uses one time-step. The second two lines of the rule express the frame condition: first, that the values of the relation f at objects other than o do not change, and second, that the values of the relations not being updated do not change. We use the "meta" quantifier $\bar{\forall}$ to represent quantification over the

C_A	::	action → time → time → Prolog expression

$C_A(o.f{:=}e, t, t')$	$\hat{=}$	t' `is` $t+1$`,`
		`assertl((`f`(`$o, A_1, ..., A_n, t'$`) :- `E`)),`
		`assertl((`f`(`$O, B_1, ..., B_n, t'$`) :-`
		`dif(`O, o`), `f`(`$O, B_1, ..., B_n, t$`))),`
		$\bar{\forall} r : relations \mid$ `assertl((`r`(`$O', C_1, ..., C_k, t'$`) :-`
		r`(`$O', C_1, ..., C_k, t$`)).`
		where $C_E(e, t) = (E, [A_1, ..., A_n])$

$C_A(a_1;\ a_2, t, t')$	$\hat{=}$	$C_A(a_1, t, T'')$`,` $C_A(a_2, T'', t')$ where T'' is a fresh variable
$C_A(a_1 \&\& a_2, t, t')$	$\hat{=}$	$C_A(a_1, t, t')$ `,` $C_A(a_2, t, t')$
$C_A(a_1 \mid\mid a_2, t, t')$	$\hat{=}$	$C_A(a_1, t, t')$ `;` $C_A(a_2, t, t')$
$C_A(action[e_1, ..., e_n], t, t')$	$\hat{=}$	`action(`$e_1, ..., a_n, t, t'$`)`

$C_A($**loop** $\{act[e_1,...,e_n]\}, t, t')$	$\hat{=}$	E_1`,`$...$`,`E_n`, loop(`$t, t', act, [V_1, ..., V_n]$`)`
		where $C_E(e_i, t) = (E_i, [V_i])$ and
		`loop(T, T, F, Args).`
		`loop(T, Tp, F, Args) :-`
		`append(Args, [T, T1], A),`
		`apply(F, A), loop(T1, Tp, F, Args).`

Figure 8-3: Rules for Compiling Alloy Actions into Prolog

(static) set of relations in a given specification.

The rule for sequential composition introduces a fresh variable to represent the time-step in between the two actions. The other rules are similar to those in the case of formulas, but work on a pair of time-steps.

Loops are compiled into calls to the `loop` predicate, which is written to execute the action it is given nondeterministically. Generally, a post-condition is given using **after** that allows `loop` to stop at the appropriate time. Figure 8-3 contains the complete set of rules for compiling actions.

Chapter 9

Related Work

Our approach to modeling dynamic systems is similar to Carroll Morgan's [31], the primary difference being that Morgan defines a programming language and then adds specification statements, while we begin with a specification language and extend it with commands. Like Morgan's language, however, our command language supports the practice of refinement-based program development [32]. Our language is also similar to Butler Lampson's system specification language Spec [29], which also provides both declarative and imperative constructs. The B Method [1] also provides the same imperative constructs that we present here, and gives them the same semantics. Abstract State Machines [7] represent another operational specification technique, but ASMs lack the declarative features of Alloy. None of these approaches currently support the Alloy Analyzer's style of analysis.

Other traditional methods (such as Z [36] and VDM [25]) for specifying dynamic systems and analyzing those specifications center around the definitions of single-step operations, and do not offer a command language. Z does provide sequential composition, but no looping construct.

DynAlloy [12, 11] has a very similar motivation to our work. It likewise extends Alloy, and offers operational constructs, but based on dynamic logic rather than relational commands. Unlike our extension, however, DynAlloy extends the semantics of Alloy, and translations are not intended to be human-readable.

Alchemy [27] also defines state transitions declaratively, but has the goal of compiling Alloy specifications into imperative implementations. Since it uses an idiom-based approach to state transitions, this work has prompted an exploration [16] of the properties that a declarative specification must have in order to correctly define a transition system. The specifications generated by our translation satisfy the necessary conditions by construction.

Some similar executable languages also exist: Crocopat [6] and RelView [5] both allow the definition and execution of relational programs. While these tools can execute commands over very large relations, they cannot perform the kind of exhaustive analysis that the Alloy Analyzer supports.

A survey of existing techniques for executing expressive specifications [17] reveals that fully-automated approaches such as ours have only rarely been attempted. Most similar to our own work is an approach that translates Z specifications into Prolog [9].

Because this translation works on fully-declarative specifications rather than our mix of declarative and imperative constructs, the programs it produces are slow enough that they are useful only for the kind of animation that the Alloy Analyzer already provides. Squander [35] animates Alloy specifications embedded in Java programs, but provides the same level of performance as the Alloy Analyzer. Our technique, on the other hand, produces programs that are fast enough to serve as prototype implementations.

Many non-automated approaches [18, 26, 13] have been proposed, but the possibility of introducing errors during manual translation makes these unattractive, and most still require further refinement—even after a manual translation effort—for efficient execution.

Animation of expressive specifications is a well-studied topic. The toolkit supporting the B method [2] and tools for JML [8] and Z [24] all support animating specifications. Such tools, however, do not provide end-to-end support: most do not allow animation of the most expressive parts of the language, require a concrete instantiation of the initial state, and use constraint-solving approaches that do not scale as well as Prolog's search. By separating verification and animation from execution, we provide both analysis and verification (using the Alloy Analyzer) and efficient execution (using our Prolog compiler).

Logic programming has long been considered an appropriate middle ground between expressive specifications and executable programs: Hoare, for example, points out [19] that the most obvious target for an initial executable version of a specification is a logic program. The addition, over the past decades, of a considerable amount of expressive power in the form of constraints [23], more expressive logics [30, 20], and more efficient execution strategies [3] has made this connection even stronger. The addition of classical negation to logic programming to produce answer-set programming [15] has brought the two worlds entirely together; answer-set programming is equivalent in expressive power to most specification languages. However, this equivalence also means that implementations of answer-set programming are *animators* rather than engines for *execution*.

Chapter 10

Conclusions and Future Work

This thesis has presented end-to-end language and tool support for constructing correct programs. By extending the Alloy specification language to support the definition of imperative programs, we have shown how to leverage existing tool support for verification and animation of these programs; to execute them, we have again leveraged existing tools by compiling to Prolog.

Our examples are indicative of our experience using this process: dynamic models can be built statically and the dynamic elements added after analysis has shown that the static model is correct. Moreover, the addition of sequential composition and looping constructs make models of dynamic systems more concise and easier to read. Our execution engine has been surprisingly adept at executing the resulting programs; we have had to make only minimal modifications to most of our specifications in order to obtain reasonable speed.

We have also experimented with a refinement-oriented approach to producing executable specifications. The similarity of our language extension to the programming language used by Morgan [32] makes this an obvious approach, and our ability to perform automated analysis on each refinement step makes it very attractive. Using refinement, we have been able to produce imperative programs from specifications in an incremental fashion, checking after each step that no errors have been introduced. The result is a program that is guaranteed to match the original declarative specification.

Our experience with our toolchain has identified three key areas for future work. First, the compiler does not yet identify parts of input specifications that may be troublesome to execute. Some Alloy constructs, such as negation and quantification, can make the resulting Prolog program impossible to execute or even incorrect. Specifications that make extensive use of these constructs do not seem to occur often in practice, but a warning message from the compiler would be useful to the user in case they do. We would like to formally define these "problem" constructs and modify the compiler to emit warnings when they are encountered.

Second, performance of the compiled specifications is not yet optimal. With better knowledge of the strengths and weaknesses of the particular Prolog implementation we plan to target, we should be able to generate more appropriate code. Moreover, the compiler itself may be able to detect code that will perform poorly and signal a

warning. We have already encountered cases requiring refinement of the specification in order to obtain efficient code; a profiling tool for a future version of the compiler might be able to suggest refinement in these cases.

Third, our language extension lacks several features that would make writing specifications more convenient. The most obvious of these is support for local variables; to work around this limitation, many of our models define pieces of global state that are actually used in a local fashion. These kinds of features are simple bits of syntactic sugar that should be easy to add to our tools.

Even without these improvements, our tools have proven useful. Our translation of Imperative Alloy specifications into Alloy has allowed us to verify specifications and refinements; our compiler produces efficient code for most specifications, and these compiled programs are useful as prototype implementations.

Bibliography

[1] J.R. Abrial. *The B-book: assigning programs to meanings*. Cambridge Univ Pr, 1996.

[2] J.R. Abrial, M.K.O. Lee, D. Neilson, PN Scharbach, and I. Sørensen. The B-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development*, volume 2, pages 398–405. Springer, 1991.

[3] H. Ait-Kaci. *Warren's abstract machine: a tutorial reconstruction*. Citeseer, 1991.

[4] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis". In *In Popl '02: Proceedings Of The 29th Acm Symposium On The Principles Of Programming Languages*, 2002.

[5] R. Behnke, R. Berghammer, E. Meyer, and P. Schneider. RELVIEW-A system for calculating with relations and relational programming. *Lecture Notes in Computer Science*, 1382:318–321, 1998.

[6] D. Beyer. Relational programming with CrocoPat. In *Proceedings of the 28th International Conference on Software engineering*, pages 807–810. ACM New York, NY, USA, 2006.

[7] E. Börger and R.F. Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Verlag, 2003.

[8] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. *Lecture notes in computer science*, 3582:75, 2005.

[9] AJJ Dick, PJ Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In *Z User Workshop: proceedings of the Fourth Annual Z User Meeting, Oxford, 15 December 1989*, page 71. Springer Verlag, 1990.

[10] E.W. Dijkstra. *A discipline of programming*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

[11] M.F. Frias, C.G.L. Pombo, J.P. Galeotti, and N.M. Aguirre. Efficient Analysis of DynAlloy Specifications. 2007.

[12] MR Frias, JP Galeotti, CGL Pombo, and NM Aguirre. DynAlloy: upgrading alloy with actions. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 442–450, 2005.

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

[14] FUSE: Filesystem in Userspace, 2010. (`http://fuse.sourceforge.net/`).

[15] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Logic programming*, page 597. MIT Press, 1990.

[16] T. Giannakopoulos, D.J. Dougherty, K. Fisler, and S. Krishnamurthi. Towards an Operational Semantics for Alloy. In *Proceedings of the 16th International Symposium on Formal Methods. To appear*, 2009.

[17] A. Gravell and P. Henderson. Executing formal specifications need not be harmful. *Software engineering journal*, 11(2):104–110, 1996.

[18] A. M. Gravell and P. Henderson. Why execute formal specifications? pages 165–184, 1991.

[19] CAR Hoare. An overview of some formal methods for program design. *Computer*, 20(9):85–91, 1987.

[20] J.S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[21] D. Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press, 2006.

[22] D. Jackson and J. Wing. Lightweight formal methods. *Lecture Notes in Computer Science*, 2021:1–1, 2001.

[23] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM New York, NY, USA, 1987.

[24] X. Jia. An approach to animating Z specifications. *COMPSAC-NEW YORK-*, pages 108–108, 1995.

[25] C.B. Jones. *Systematic software development using VDM*. Prentice Hall New York, 1990.

[26] A. Kans and C. Hayton. Using ABC to prototype VDM specifications. *ACM SigPLAN Notices*, 29(1):27–36, 1994.

[27] S. Krishnamurthi, K. Fisler, D.J. Dougherty, and D. Yoo. Alchemy: transmuting base alloy specifications into implementations. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 158–169. ACM New York, NY, USA, 2008.

[28] V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. *ACM SIGSOFT Software Engineering Notes*, 30(5):216, 2005.

[29] Butler Lampson. 6.826 class notes, 2009. (`http://web.mit.edu/6.826/www/notes/`).

[30] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1-2):125–157, 1991.

[31] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):403–419, 1988.

[32] C. Morgan. *Programming from specifications*. 1990.

[33] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):517–561, 1989.

[34] B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[35] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 999–1006. ACM, 2009.

[36] JM Spivey. *The Z notation: a reference manual*. 1992.