Towards a Transition System Semantics for Alloy

A Major Qualifying Project Report

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in

Computer Science

by

Theophilos John Giannakopoulos

March 16, 2009

APPROVED:

Professor Daniel Dougherty, MQP Advisor

Professor Michael Gennert, Head of Department

Abstract

Alloy is a language for modeling systems using first order logic and relational algebra. In this paper we examine the use of Alloy for creating models of stateful systems, and we explore semantics for Alloy that define transition systems over database instances based on Alloy specifications written in the state-signature idiom. One such semantics is fully adequate for the original semantics of Alloy. We prove an undecidability result concerning the automatic synthesis of programs from specifications under this semantics.

Acknowledgments

I would like to thank my advisor Professor Daniel Dougherty (WPI) for his help with this project, and Professor Kathi Fisler (WPI) and Professor Shriram Krishnamurthi (Brown) for starting me on this research along with Professor Dougherty during my fellowship over the summer.

I also would like to thank many times over Danny Yoo, Timothy Nelson and Yu Feng in the ALAS lab for accepting me in their research group and giving me the advice and moral support I needed to finish this project.

Contents

List of Figures

Chapter 1

Introduction

Alloy is a language for specifying properties of systems. The AlloyAnalyzer tool is used for reasoning about specifications written in the Alloy language. Because Alloy is based on first order logic and relational algebra, it is general enough to be used for modeling many different systems. One of the common uses of Alloy and the AlloyAnalyzer is to model operations of a stateful system and then check that the operations have examples corresponding to their execution or to check that the operations have some desired properties.

A main use of Alloy advocated by Daniel Jackson in *Software Abstractions* [9] is to write specifications such that instances of those specifications resemble traces (sequences of transitions) in a transition system. Assertions can then be written about the traces and checked against instances of the specification using the Alloy-Analyzer tool. Alloy is used in this manner to rapidly explore design choices before building complete prototypes.

The Alchemy project [10] is an attempt to address the gap between creating and reasoning about a specification written in Alloy and creating the implementation itself. In order to make Alloy more useful for software engineers, Alchemy creates

software libraries for prototyping implementations of Alloy specifications of stateful systems, where the state being modeled is a relational database. In the process of developing Alchemy it was discovered that formally defining implementations based on Alloy specifications in a manner consistent with Alloy's logical semantics was a non-trivial task. The problem was difficult enough that, as observed in [10], a naïve transition system semantics is not consistent with with Alloy's logical semantics.

In this paper we address the difficulty of connecting Alloy specifications with implementations–specifically, transition systems over databases. A first attempt to define an operational semantics for Alloy in a natural manner while maintaining the connection with Alloy as a logic will demonstrate the difficulties in the task. In order to explore the nature of the connections between logical semantics and transition system semantics, we invent an Alloy-like language named Alloy$^{\mathrm{II}}$ that has a natural connection to transition systems. We formalize the transition system and offer a pair of tools for analyzing Alloy$^{\mathrm{II}}$ specifications in the same manner as Alloy and for generating code from Alloy$^{\mathrm{II}}$ specifications in the same manner as Alchemy.

Using the experience from developing Alloy$^{\mathrm{II}}$, we examine the discrepancy between the naïve transition system semantics and the logical semantics of Alloy. We also propose an alternative transition system semantics for Alloy and discuss its benefits and drawbacks along with the benefits and drawbacks of the semantics described in [10].

From the insights gained in this work, we offer advice (which may be better characterized as warnings) to software engineers who use Alloy for modeling stateful systems.

Chapter 2

Alloy and Stateful Systems

2.1 Alloy

We assume that the reader is familiar with the Alloy language and the logical semantics of the language. For those readers who are not, a brief introduction is included in this section, and further information can be found in [9].

For the purposes of this paper, we will be considering an Alloy specification to be a collection of signatures, facts and predicates.

Definition 2.1 (Alloy specification)**.** An Alloy specification is a triple $\langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$ where

- Sigs is a set of Alloy signatures,
- Facts is a set of Alloy facts, and
- Preds is a set of Alloy predicates.

Figure 2.1 shows an example of an address book modeled in Alloy, based on an early example in Software Abstractions [9]. The address book is modeled as a dynamic system. Each element of the `Book` relation represents a state of the address

book being modeled. The predicates `add` and `del` describe the operations of adding a name-address pair to the address book and deleting a name-address pair from the address book. The command `run add` instructs the AlloyAnalyzer to find an example of adding a name-address pair to an address book.

The assertion `delUndoesAdd` claims that if a name-address pair is added to a book with no entries, and then the same pair is deleted from the resulting book, the final book will be identical to the first one. When issued the command `check delUndoesAdd` the AlloyAnalyzer will search for a counterexample to the claim (and not find one).

The example gives a general idea of the syntax of Alloy. Signatures declare the names of relations and their arities. The relations whose names appear after `sig` are called signature relations and those whose names appear in the bodies of a `sig` are called field relations.

Facts state constraints about the system using first order logic and relational algebra. Predicates and assertions also declare constraints about the system, but they are only used when the AlloyAnalyzer is instructed to `run` the predicate or `check` the assertion, or when the predicate is referenced in a fact or other predicate or assertion that is being used.

We will formally define the formulas that correspond to predicates with respect to each transition system semantics individually, since each semantics makes use of a slightly different formula. In general, each predicate and assertion corresponds to a formula in the kernel language of Alloy. The formula includes the body of the predicate with the variables from the header existentially quantified conjoined with the constraints in the bodies of the facts. When the AlloyAnalyzer is searching for an example for a predicate or a counterexample for a assertion, it is actually looking for an Alloy instance, a function from the names declared in the signatures

to relations, that satisfies or does not satisfy the formula according the semantics of the Alloy kernel language, which are essentially those of first order logic and relational algebra.

```
sig Name {}
sig Addr {}
sig Book { entries : Name -> Addr }

fact {
        entries in Book -> (Name -> lone Addr)
}

pred show {
        #Book.entries > 1
}

pred add[b, b' : Book, nNew : Name, aNew : Addr] {
        b'.entries = b.entries + (n -> a)
}
pred del[b, b' : Book, n : Name, a : Addr] {
        b'.entries = b.entries - (n -> a)
}

assert delUndoesAdd {
        all b, b', b'' : Book, n : Name, a : Addr |
                (no b.entries and add[b,b',n,a] and del[b',b'',n,a])
                implies
                (b.entries = b''.entries)
}
```

Figure 2.1: Modeling an address book in Alloy

2.2 Stateful Systems

The main focus of this paper is understanding specifications of stateful systems. For our purposes we will take the operational model of stateful systems to be transition systems.

Definition 2.2 (Labeled transition system)**.** A labeled transition system is a triple $\langle Q, A, \delta \rangle$, where

- Q is a set of states,
- Act is a set of actions, and
- $\delta \subseteq Q \times A \times Q$ is a set of transitions.

Henceforth we will refer to labeled transition systems as "transition systems." A transition system which is *trivial* is one in which the set of transitions is the empty set.

We define the notion of implementability for an Alloy specification with the understanding that transition systems are implementations.

Definition 2.3 (Implementability and non-trivial implementability)**.** A specification is implementable according to a semantics if that semantics assigns a transition system to that specification. A specification is non-trivially implementable according to a semantics if that semantics assigns a non-trivial transition system to that specification.

The transition system semantics that we will introduce in this paper differ in the mechanisms they use for assigning transition systems to specifications, but all share the same notion of implementability and non-trivial implementability.

Fix a countable set of atoms adom, of variable names, and of operation names, from a database schema $\mathcal{S}$ we can construct a general transition system $\mathcal{T}_{\mathcal{S}}$ such that every transition system with the same associated schema with which we will be concerned in this paper will be a subsystem. This system is essentially the complete directed graph on the set of database instances of $\mathcal{S}$, where each edge is labeled with every operation name and argument combination.

Definition 2.4 ($\mathcal{T}_\mathcal{S}$)**.** Define $Q_\mathcal{S}$ as the set of database instances on $\mathcal{S}$ (with atoms taken from adom), and Act as the set of possible operation name, argument pairs, where an argument is a map from a finite set of variable names to finite relations over adom. Then $\mathcal{T}_\mathcal{S} = \langle Q_\mathcal{S}, \mathsf{Act}, Q_\mathcal{S} \times \mathsf{Act} \times Q_\mathcal{S} \rangle$.

We also care about the transition system where the transitions are restricted by a bound on the post-state given the pre-state and arguments.

Definition 2.5 (Strict transition)**.** A transition $\langle q, \langle n, \mathsf{args} \rangle, q' \rangle \in Q_\mathcal{S} \times \mathsf{Act} \times Q_\mathcal{S}$ is strict if the set of all atoms in the relations in the range of q' is a subset of the set of those of q union those in the range of args.

With these definition, the operational interpretation of a specification associated with the schema $\mathcal{S}$ becomes a definition of which transitions in $\mathcal{T}_\mathcal{S}$ are acceptable and which transitions are associated with which operation names and arguments. In other words, in the transition system semantics that we will define in this paper, the meaning of a predicate in a specification is a set of transitions in a transition system.

Chapter 3

Naïve Transition System Semantics

The motivation for examining how Alloy specifications are interpreted as specifications of transition systems comes from a discrepancy discovered between the semantics defined in [10] and the logical semantics of Alloy. When creating the Alchemy tool for synthesizing Alloy specifications from Alloy specifications one of the desired properties of the code synthesis was that a specification would be satisfiable if and only if it were non-trivially implementable. However, the notion of "implementable" defined there did not possess this property with the implication going in either direction. In this section we will provide a formal counterexample to the consistency of those underlying semantics of Alchemy.

The troublesome specification identified in [10] was the one appearing in Figure 3.2. We will use this specification as the basis for our counterexample, but first we must give the formal definition of our naïve transition system semantics (or NTSS).

The Alloy specifications for which NTSS defines transition systems have the following properties:

- There is a distinguished state-relation, which without loss of generality we will call `State`.

- Each predicate in Preds has in its header exactly two variables (which we will call `s` and `s'`) of the type of `State`, which are identical except for an additional terminating prime on one of the variables. No primes appear in the specification except for on uses of that variable.

- Each argument in the header of the predicate is an element of a signature relation.

From an Alloy specification $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$ we can define a database schema $\mathcal{S}_\mathcal{A}$ that is associated with that specification. The schema $\mathcal{S}_\mathcal{A}$ is exactly the schema of Alloy instances of the specification. This schema is based entirely on the signatures of the specification, and not on the facts or predicates. From the headers of the predicates in the specification we can also determine a set of operation names and arities that are associated with the specification. It is worth noting that this information is independent of the constraints (the facts and bodies of predicates) in a specification.

Define

- $Q_\mathcal{A}$ the set of database instances over $\mathcal{S}_\mathcal{A}$, and

- $\mathsf{Act}_\mathcal{A}$ is a set of name-argument pairs, where the names are those of the predicates in Preds and the argument is a function mapping each name in the header of the named predicate to a relation of the specified arity and sort. Note that $\mathsf{Act}_\mathcal{A}$ is a subset of the action space of $\mathcal{T}_{\mathcal{S}_\mathcal{A}}$.

The definition of the transitions $\delta_\mathcal{A}$ is somewhat more involved. First we define a *maximal expression* occurrence as an expression p or q in the form p `in` q. That is,

maximal expression occurrences are those expressions that are the left- and right-hand-side subexpressions of atomic `in` formulas.

Informally, there is a transition from a database instance q to a database instance q' when both q and q' satisfy the facts, and when the body of the predicate holds when maximal primed expressions are interpreted in q' and all other expressions are interpreted in q. Also, the universe of q' is a subset of the universe of q union the atoms bound to variables annotated `New` in the header of the predicate. The `New` atoms may not appear in the pre-state.

More formally, we first define $\vDash$ as follows.

Definition 3.1. Let $\mathcal{A} = \langle \mathsf{Specs}, \mathsf{Preds}, \mathsf{Facts} \rangle$ be an Alloy specification and let $q, q' \in Q_{\mathcal{A}}$. Let $p = \langle H, B \rangle \in \mathsf{Preds}$, where H is an association of the variable names in the header of p to the signature types and B is the body of p. Let the parameter environment E be bind every non-new, non-state variable in H to some atom in I of the corresponding type for that variable. E also binds the variables of the type `State` to the unique atom in the state relation. E_{new} maps every new variable in H to an atom of the corresponding type q that does not appear in q'.

We say $q, q' \vDash_{E, E_{\mathsf{new}}} p, \mathsf{Facts}$ if the following conditions hold:

1. With the exception of the atoms in the co-domain of E_{new}, the universe of q and of q' have the same atoms.

2. B evaluates to true under the semantics of Alloy when every maximal non-primed expression is evaluated under q, every maximal primed expression is evaluated under q', and $E \cup E_{\mathsf{new}}$ is used as the environment.

3. The facts are true in both q and q'.

Given this definition of $\vDash$, we define when a specification determines that a transition is acceptable.

Definition 3.2 (NTSS acceptable transition)**.** Let $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$ be an Alloy specification. A transition $\langle q, \langle pname, E \cup E_{\mathsf{new}} \rangle, q' \rangle$ is $\mathcal{A}$-acceptable if for some $p \in \mathsf{Preds}$ named $pname$ we have that $q, q' \vDash_{E, E_{\mathsf{new}}} p, \mathsf{Facts}$.

Definition 3.3 (NTSS)**.** Let $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$ be an Alloy specification. From $\mathcal{A}$, NTSS defines the transition system $\langle Q_{\mathcal{A}}, \mathsf{Act}_{\mathcal{A}}, \delta_{\mathcal{A}} \rangle$ where $\delta_{\mathcal{A}}$ is the set of $\mathcal{A}$-acceptable transitions for $\mathcal{A}$.

Remark. NTSS is a semantics that assigns transition systems to specifications, so we may discuss implementability with respect to NTSS. Since NTSS does not assign transition systems to all Alloy specifications, there are some specifications which are not implementable according to NTSS. The distinction between those those specifications which are implementable and those which are not implementable is an explicit part of the definition of NTSS.

We are interested in the distinction between specifications which are trivially implementable and those which are non-trivially implementable. We are also interested in whether that distinction is manifested in the logical semantics of Alloy. In the case of NTSS, we shall find that it is not.

For example, the signatures, fact, and the `add` predicate from the address book example in Figure 2.1 in the Appendix form a specification for which NTSS defines a transition system. The instances in Figure 3.1 show an example of a pre-state, action, and post-state for the specification under NTSS.

Despite the niceness of the example in Figure 3.1, NTSS does lack the aforementioned correspondence between satisfiability and non-trivial implementability. This discrepancy between NTSS and the logical semantics of Alloy puts a damper on the usefulness of Alloy. The point of modeling systems in Alloy was so that we could explore or guarantee some properties of the corresponding implementations. If our

Pre-state q, such that

$$\begin{aligned} q(\texttt{Name}) &= \{\text{Bob}\} \\ q(\texttt{Addr}) &= \{30\text{ School Rd}\} \\ q(\texttt{Book}) &= \{b\} \\ q(\texttt{entries}) &= \{\langle b, \text{Bob}, 30\text{ School Rd}\rangle\} \end{aligned}$$

Action

$$\langle add, \{\texttt{n} \mapsto \text{Sue}, \texttt{a} \mapsto 5\text{ College Ln}\}\rangle$$

Post-state q', such that

$$\begin{aligned} q'(\texttt{Name}) &= \{\text{Bob}, \text{Sue}\} \\ q'(\texttt{Addr}) &= \{30\text{ School Rd}, 5\text{ College Ln}\} \\ q'(\texttt{Book}) &= \{b\} \\ q'(\texttt{entries}) &= \{\langle b, \text{Bob}, 30\text{ School Rd}\rangle, \langle b, \text{Sue}, 5\text{ College Ln}\rangle\} \end{aligned}$$

Figure 3.1: An transition of the address book example in NTSS

```
sig State { r : B }
sig B {}
fact { one r }

// this is non-trivially implementable, but not satisfiable
pred change_r1[s, s' : State] {
        s.r != s'.r
}

// this is satisfiable, but not non-trivially implementable
pred change_r2[s, s' : State, bNew: B] {
        s'.r = s.r + bNew
}
```

Figure 3.2: Specification showing discrepancy between Alloy and NTSS

intuitive or formal notion of the transition system associated with a specification does not adhere somehow to the semantics of Alloy, then whatever properties of the system that were verified in Alloy's logical semantics may not be true of the implementation under consideration. This lack of a correspondence extends even to checking for the existence or non-existence of a transition corresponding to an execution of the operation modeled by a predicate.

3.1 Satisfiability does not imply non-trivial implementability

In the case of `change_r1`, because NTSS interprets `s.r` and `s'.r` in different states and require that `one r` holds in each state individually, the predicate is easily implementable by anything changes the value of r from the pre-state to the post state.

However, in Alloy's semantics, there is only one instance. Since `r` is declared as a function from `State` to `B`, the constraint that `r` only have one row causes `State` to only have one row as well. Thus, the same atom is witness to both `s` and `s'`, so `s.r` must always equal `s'.r`, making the predicate unsatisfiable.

Proposition 3.1. *There exists an Alloy specification* $\mathcal{A} = \langle \mathsf{Specs}, \mathsf{Preds}, \mathsf{Facts} \rangle$ *with corresponding transition system* $T = \langle Q_{\mathcal{A}}, \mathsf{Act}_{\mathcal{A}}, \delta_{\mathcal{A}} \rangle$ *according the semantics of NTSS such that* $\delta_{\mathcal{A}}$ *is non-empty, yet there is no satisfiable predicate in* Preds.

Proof. Consider the specification in Figure 3.2 $\langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$, excluding `change_r2`. The transition in Figure 3.3 is a transition in the transition system for the specification according to NTSS.

The predicate `change_r1` is the only predicate in the specification, so we only need to show that it is unsatisfiable to show that the conjecture is false. Assume the

predicate to be satisfied by some instance I. $I(\texttt{s})$ and $I(\texttt{s'})$ must be different atoms, since otherwise the expressions `s.r` and `s'.r` are equivalent. However, because `r` is a function from `State` to `B` and `r` is constrained to be a singleton, *State* is also constrained to be a singleton relation. Therefore, $I(\texttt{s})$ must be the same as $I(\texttt{s'})$. This is a contradiction. Therefore, the predicate must not be satisfiable. □

3.2 Non-trivial implementability does not imply satisfiability

In the case of `change_r2`, the predicate is satisfiable in Alloy's semantics, because the witness to `bNew` may be the same as `a.r`, so when that the union is taken, there is no change, and the one row of `r` may serve as both pre-state and post-state without issue.

In NTSS, however, variables annotated as `New` may not appear in the pre-state. Therefore, `bNew` may not be the same as `s.r`, and so there is no transition that will satisfy the constraint.

Proposition 3.2. *There exists an Alloy specification* $\mathcal{A} = \langle \mathsf{Specs}, \mathsf{Preds}, \mathsf{Facts} \rangle$ *with corresponding transition system* $T = \langle Q_{\mathcal{A}}, \mathsf{Act}_{\mathcal{A}}, \delta_{\mathcal{A}} \rangle$ *according the semantics of NTSS such that there is a predicate in* Preds *that is satisfiable but* $\delta_{\mathcal{A}}$ *is empty.*

Proof. Consider the specification in Figure 3.2 $\langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$, excluding `change_r1`. The instance in Figure 3.4 is a satisfying instance of `change_r2` according to the semantics of Alloy.

Let $\langle Q_{\mathcal{A}}, \mathsf{Act}_{\mathcal{A}}, \delta_{\mathcal{A}} \rangle$ be the transition system corresponding to the specification according to NTSS. Assume $\delta_{\mathcal{A}}$ is non-empty. Then there is some q,q' and $E \cup E_{\mathsf{new}}$ such that $\langle q, \langle \texttt{change\_r2}, E \cup E_{\mathsf{new}} \rangle, q' \rangle$ is in $\delta_{\mathcal{A}}$. Since the transition is is in $\delta_{\mathcal{A}}$, it

must be that $q, q' \vDash_{E,E_{\mathsf{new}}} \texttt{change\_r2}, \mathsf{Facts}$. Since there is a variable annotated `New` in the header of `change_r2`, E_{new} binds that variable to an atom that appears in q' but not in q.

Note that as before, `r` must be a singleton due to the fact that declares `one r`. Also note that since the atom bound to `bNew` by E_{new} cannot appear in q, it does not appear in the evaluation of `s.r`, which is interpreted in q. Thus, according to the body of the predicate, `r` must have at least two tuples, which is impossible. Therefore $\delta_{\mathcal{A}}$ must be empty. □

Pre-state q such that

$$\begin{aligned} q(\texttt{State}) &= \{s0\} \\ q(\texttt{B}) &= \{b0, b1\} \\ q(\texttt{r}) &= \{\langle s0, b0\rangle\} \end{aligned}$$

Action

$$\langle \texttt{change\_r1}, \emptyset \rangle$$

Post-state q' such that

$$\begin{aligned} q'(\texttt{State}) &= \{s0\} \\ q'(\texttt{B}) &= \{b0, b1\} \\ q'(\texttt{r}) &= \{\langle s0, b1\rangle\} \end{aligned}$$

Figure 3.3: The transition for the counterexample

The instance I such that

$$
\begin{aligned}
I(\texttt{State}) &= \{s0\} \\
I(\texttt{B}) &= \{b0\} \\
I(\texttt{r}) &= \{\langle s0, b0\rangle\} \\
I(\texttt{s}) &= \{s0\} \\
I(\texttt{s'}) &= \{s0\} \\
I(\texttt{bNew}) &= \{b0\}
\end{aligned}
$$

Figure 3.4: The satisfying instance for the counterexample

Chapter 4

A New Specification Language: Alloy^{II}

In order to better understand why the transition systems defined by NTSS are not fully adequate for the logical semantics of Alloy, we chose to create a new Alloy-like language designed to make the transition system semantics of the language follow naturally from the logical semantics. The new language Alloy^{II} accomplishes this goal by using a pair of Alloy instances as an Alloy^{II} instance (hence the name, where the Roman numeral II has the appearance of a pair of instances).

The key insight in creating the syntax of Alloy^{II} was that the placement of the prime in expressions in Alloy was an accident of the idiom used to specify stateful systems. In Alloy^{II} the prime is placed on the relations whose post-state we are actually referring to: those that are declared in the signatures of the specification.

```
formula ::= formula fbinop formula
          | not formula
          | quant var : expr | formula
          | expr in expr

fbinop ::= and | or
quant ::= some | all

expr ::= expr binop expr
       | unop expr
       | univ maybeprime
       | reln maybeprime
       | iden
       | var

maybeprime ::= | '
binop ::= + | & | - | -> | .
unop ::= ~ | ^
```

Figure 4.1: The kernel syntax of Alloy$^{\text{II}}$

4.1 Kernel Syntax

The syntax of the kernel language Alloy$^{\text{II}}$ essentially the same as that of Alloy with the addition of the prime (') as a special character. The syntax for a formula is given by the BNF grammar in Figure 4.1.

4.2 Kernel Logical Semantics

The semantics of the kernel language for Alloy$^{\text{II}}$ are very much like those of Alloy (unsurprisingly, since both are based on relation algebra and first order logic). The key differences are between the two languages are the definition of an instance and how identifiers are interpreted. An instance in Alloy$^{\text{II}}$ is a triple $\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle$ that consists of two Alloy-like instances i_{pre} and i_{post} that map from relation names (declared in the signature of a specification) to relations, and an environment η

which maps from variable names to relations. Without loss of generality, the set of relation names and variable names can be considered disjoint.

When a variable name is interpreted, the binding in the environment is used. When an unprimed relation name is interpreted, the binding in the first instance is used, and which a primed relation name is interpreted, the binding in the second instance is used.

The semantics for Alloy$^{\text{II}}$ are given in Figure 4.2 in the same style as the semantics of Alloy are in *Software Abstractions* [9]. M is a function from formulas and instances to boolean values and E is a function from expressions and instances to relations. If $M[f]\,\langle i_{\text{pre}}, i_{\text{post}}, \eta\rangle$ is true, then the instance $\langle i_{\text{pre}}, i_{\text{post}}, \eta\rangle$ satisfies the formula f.

4.3 Full Language

The full language of Alloy$^{\text{II}}$ is similar to Alloy. Signatures may be written in the same way as in Alloy, but each name declared in a signature corresponds to two relations: one in the first part of an instance and one in the second part. Predicates may be written as in Alloy as well, but special conventions involving state-variables are not necessary for declaring operations. Facts may be written as in Alloy.

Just as an Alloy specification corresponds to a formula consisting of the conjunction of the facts, implicit facts, and predicate body, an Alloy$^{\text{II}}$ specification does as well. This means that constraints written without primes in facts only refer to the first part of an Alloy$^{\text{II}}$ instance Because facts are frequently intended to be written as state-invariants, facts may be written as `global`.[1] This may be viewed as syntactic sugar for replicating the body of the fact (which contains no primed names), priming all of the relation names in the replicated body, and conjoining the

[1]Calling the facts `global` is a reference to Linear Temporal Logic, in which the G operator means that a formula should hold globally over the rest of an execution.

$$\mathrm{M} : \mathsf{fmla} \times \mathsf{instance} \to \mathsf{boolean}$$
$$\mathrm{M} : \mathsf{expr} \times \mathsf{instance} \to \mathsf{relation}$$

$$\begin{aligned}
\mathrm{M}[f \texttt{ and } g]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \mathrm{M}[f]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \wedge \mathrm{M}[g]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \\
\mathrm{M}[f \texttt{ or } g]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \mathrm{M}[f]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \vee \mathrm{M}[g]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \\
\mathrm{M}[\texttt{not } f]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \neg\, \mathrm{M}[f]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \\
\mathrm{M}[\texttt{some } v : p \mid f] &= \bigvee \{\mathrm{M}[f]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta[v \mapsto t]\rangle \\
&\qquad\qquad |t \subseteq \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \wedge \#t = 1\} \\
\mathrm{M}[\texttt{all } v : p \mid f] &= \bigwedge \{\mathrm{M}[f]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta[v \mapsto t]\rangle \\
&\qquad\qquad |t \subseteq \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \wedge \#t = 1\} \\
\mathrm{M}[p \texttt{ in } q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \subseteq \mathrm{E}[q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}[p \texttt{ + } q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \cup \mathrm{E}[q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \\
\mathrm{E}[p \texttt{ \& } q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \cap \mathrm{E}[q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \\
\mathrm{E}[p \texttt{ - } q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \setminus \mathrm{E}[q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \\
\mathrm{E}[p \texttt{ -> } q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \{\langle p_1, \ldots, p_n, q_1, \ldots, q_m\rangle \mid \\
&\qquad \langle p_1, \ldots, p_n\rangle \in \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \\
&\qquad \wedge \langle q_1, \ldots, q_n\rangle \in \mathrm{E}[q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle\} \\
\mathrm{E}[p.q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \{\langle p_1, \ldots, p_{n-1}, q_2, \ldots, q_m\rangle \mid \\
&\qquad \exists t.\, \langle p_1, \ldots, p_{n-1}, t\rangle \in \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle \\
&\qquad \wedge \langle t, q_2, \ldots, q_n\rangle \in \mathrm{E}[q]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle\} \\
\mathrm{E}[\sim p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \{\langle y, x\rangle \mid \langle x, y\rangle \in \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle\} \\
\mathrm{E}[\hat{}p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \{\langle x, y\rangle \mid \\
&\qquad \exists x_1, \ldots, x_n. \\
&\qquad \langle x, x_1\rangle, \langle x_1, x_2\rangle, \ldots, \langle x_n, y\rangle \in \mathrm{E}[p]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle\} \\
\mathrm{E}[r]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= i_{\mathrm{pre}}(r) \\
\mathrm{E}[r']\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= i_{\mathrm{post}}(r) \\
\mathrm{E}[v]\,\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta\rangle &= \eta(r)
\end{aligned}$$

Figure 4.2: Semantics of Alloy$^{\mathrm{II}}$

replicated body with the original as a normal fact.

Alloy$^{\text{II}}$ currently has no notion of assertions, but the meaning of *running* a predicate in Alloy$^{\text{II}}$ is the same as in Alloy: find a satisfying instance. Theorem 4.1 demonstrates that this is a decidable problem, as expected.

Theorem 4.1. *The following problem is decidable:*

> INPUT*: An Alloy*$^{\text{II}}$ *specification* $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$
>
> QUESITON*: Is there some instance* I *such that* I *satisfies the formula corresponding to some* $p \in \mathsf{Preds}$*?*

Proof. Given a set of atoms that acts as an upper bound on the universe of an Alloy$^{\text{II}}$ instance, we may conduct a brute force search through instances within that bound to determine if a formula is satisfiable. □

4.4 Alloy$^{\text{II}}$ Examples

In order to clarify the Alloy$^{\text{II}}$ syntax and semantics, we will translate the address book example into Alloy$^{\text{II}}$ and present a second example in which we give the signatures and facts for a model of a media library.

The address book example as written in Alloy$^{\text{II}}$ appears in Figure 4.3. Note that the explicit support for modeling stateful programs in Alloy$^{\text{II}}$ alloys for a more "object-oriented" syntactic style of specifying the relations in a model. That is, instead of having to bundle the relations that make up the state under a single signature, relations can be organized as if they were fields of classes.

Also, we do not need any special syntax or semantics for enforcing the "newness" of values. All we do is specify that each value is not in the signature relation in the pre-state. The inclusion of the value in the post-state is ensured by the declaration

```
sig Name {
        addr : lone Address
}
sig Address {}

pred add[n : Name', a : Address'] {
        n not in Name // n is new
        a not in Address // a is new
        addr' = addr + (n -> a)
}

pred del[n : Name, a : Address] {
        addr' = addr - (n -> a)
}
```

Figure 4.3: The address book example in Alloy^{II}

of the variables in the header of the predicate and by the inclusion of the values in `addr`. As with many Alloy models, this Alloy^{II} model is underspecified. There is no constraint on the contents of `Name` and `Address` aside from what appears in `addr`. That is, some atoms may be added to or removed from each of these relations from the pre-state to the post-state portion of an instance, so long as their addition or removal does not impact whether the instance satisfies the predicate under consideration. The ability to underspecify a model is presented as an advantage of Alloy [9], and so we kept that advantage in Alloy^{II}.

The address book can be more completely specified by stating in the predicate how the `Name` and `Address` relations should be constrained from the pre-state to the post-state. Most likely one would add framing conditions like `Name' = Name + n` to the `add` predicate and `Name' = Name - n` to the `del` predicate.

The portion of a media library specification in Figure 4.4 demonstrates how `global` facts act as state invariants, as NTSS wanted to treat facts.

In the media library example, the cardinality constraint on `CurrentCatalog`

```
sig Catalog {
        assets: set Asset,
        // 'disj' is not part of the kernel, but its meaning is the
        // same as in Alloy
        disj hidden, showing: set assets,
        selection: set assets
}

global fact {
        all c : Catalog | c.hidden+c.showing = c.assets
}

sig CurrentCatalog in Catalog {}

global fact {
        one CurrentCatalog
}

sig Asset {}
sig Buffer in Asset {}


pred showSelected [] {
        // changes
        CurrentCatalog.selection != none
        CurrentCatalog.showing' = CurrentCatalog.selection

        // 'framing' conditions: things that stay the same
        selection' = selection
        assets' = assets
        Catalog' = Catalog
        CurrentCatalog' = CurrentCatalog
        Asset' = Asset
        Buffer' = Buffer
        // only part of showing stays the same
        all c : Catalog - CurrentCatalog |
                c.showing' = c.showing
}
```

Figure 4.4: A partial model of a media library in Alloy^{II}

does act as a state invariant, so that for any satisfying instance I of some predicate in the specification there is one atom in $I($`CurrentCatalog`$)$ for each of the pre-state and post-state. As with Alloy, any fact that was not marked as `global` would be treated as written, constraining relations in the pre- or post-state as the presence of primes indicated.

Chapter 5

A Transition System Semantics for Alloy$^{\mathrm{II}}$

The definition of the logical semantics of Alloy$^{\mathrm{II}}$ allows for a natural mapping from instances to transitions. In contrast to Alloy, where each instance contained a transition or a trace, in Alloy$^{\mathrm{II}}$ each instance in essence *is* a transition. This makes the definition of the transition system semantics trivial: those transitions that satisfy the specification appear in the transition system.

For each specification $\mathcal{A} = \{\mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds}\}$, let $\mathcal{I}_{\mathcal{A}}$ be the set of Alloy$^{\mathrm{II}}$ instances of $\mathcal{A}$. First define a database schema $\mathcal{S}_{\mathcal{A}}$ associated with the specification. The schema is exactly the schema of each of the Alloy instances that make up each Alloy$^{\mathrm{II}}$ instance of the specification. As with NTSS and Alloy, from the headers of the predicates in the specification, we can determine the a set of operations names and arities that are associated with the specification. Again, this information is independent of the constraints in the specification.

Define

- $Q_{\mathcal{A}}$ the set of database instances over $\mathcal{S}_{\mathcal{A}}$,

- $\mathsf{Act}_{\mathcal{A}}$ the set of name-argument pairs, where the names are those of the predicates in Preds and the argument is a function from each name in the header of the named predicate to a relation of the specified arity and sort.

Definition 5.1. Given a specification $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$, for each predicate $p \in \mathsf{Preds}$, define a formula $\phi_{\mathcal{A},p}$ that is the conjunction of the bodies of the facts, the implicit constraints present in the signatures, the body of the predicate and the implicit constraints in the header of the predicate. This formula is the translation of the predicate into the kernel language of Alloy$^{\mathrm{II}}$ with the existentially quantified variables from the header of the predicate removed.

Denote the universe of i as $|i|$ and the co-domain of η as $|eta|$. An instance $\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta \rangle$ *strictly* satisfies $\phi_{\mathcal{A},p}$ if $\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta \rangle$ satisfies $\phi_{\mathcal{A},p}$ and $|\eta| - |i_{\mathrm{pre}}| \subseteq |i_{\mathrm{post}}| \subseteq |\eta| \cup |i_{\mathrm{pre}}|$.

Instead of inventing an independent way of determining if a transition is in the transition system, we take seriously the notion of a transition being embedded in an Alloy$^{\mathrm{II}}$ instance. For each specification and predicate under consideration we can use the natural method of extracting a transition from an Alloy$^{\mathrm{II}}$ instance[1]. The method is the function $\tau_{\mathcal{A}} : \mathsf{Preds} \times \mathcal{I}_{\mathcal{A}} \to Q_{\mathcal{A}} \times A_{\mathcal{A}} \times Q_{\mathcal{A}}$, such that for each predicate $p \in \mathsf{Preds}$ and instance $\langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta \rangle \in \mathcal{I}_{\mathcal{A}}$, $\tau_{\mathcal{A}}(p, \langle i_{\mathrm{pre}}, i_{\mathrm{post}}, \eta \rangle) = \langle i_{\mathrm{pre}}, a, i_{\mathrm{post}} \rangle$ where $a = (pname, \mathsf{args})$ where args is such that for each variable name v declared in the header of the predicate $p \in \mathsf{Preds}$ named $pname$, $\mathsf{args}(v) = \eta(v)$.

Definition 5.2 (Acceptable Alloy$^{\mathrm{II}}$ transition)**.** Fix a specification $\mathcal{A} = \{\mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds}\}$. A transition $\langle q, a, q' \rangle$ is $\mathcal{A}$-acceptable if there is some instance $I \in \mathcal{I}_{\mathcal{A}}$ such that there is some $p \in \mathsf{Preds}$ named $pname$ such that $\tau_{\mathcal{A}}(p, I) = \langle q, a, q' \rangle$ where

[1]The idea of extracting transitions from instances has the feel of hitting the instance with just the right hammer, so that a transition falls out. The function τ is that hammer.

$a = \langle pname, \mathsf{args} \rangle$, args is a function from the variables in the header of p to relations of the corresponding types, and I satisfies $\phi_{\mathcal{A},p}$.

Definition 5.3 (Alloy$^{\mathrm{II}}$ transition system semantics)**.** The transition system semantics for Alloy$^{\mathrm{II}}$ are defined such that for each Alloy specification $\mathcal{A}$ there is an associated transition system $(Q_{\mathcal{A}}, A_{\mathcal{A}}, \delta_{\mathcal{A}})$ where if $\delta_{\mathcal{A}}$ is the set of acceptable transitions in $Q_{\mathcal{A}} \times A_{\mathcal{A}} \times Q_{\mathcal{A}}$.

Definition 5.4 (Strict Alloy$^{\mathrm{II}}$ transition system semantics)**.** The strict transition system semantics for Alloy$^{\mathrm{II}}$ is defined such that for each Alloy specification $\mathcal{A}$ there is an associated transition system $(Q_{\mathcal{A}}, A_{\mathcal{A}}, \delta_{\mathcal{A}})$ where if $\delta_{\mathcal{A}}$ is the set of strict acceptable transitions in $Q_{\mathcal{A}} \times A_{\mathcal{A}} \times Q_{\mathcal{A}}$.

Once again we have semantics that assigns transition systems to specifications. Therefore, we may discuss the notions of trivial and non-trivial implementability with respect to these semantics.

5.1 Properties of the Transition System Semantics of Alloy$^{\mathrm{II}}$

The purpose of creating the logical and transition system semantics of Alloy$^{\mathrm{II}}$ was to aid in understanding how to create the transition system semantics so that the transition systems defined for a specification has a connection to the interpretation of the specification in the logical semantics. Theorem 5.2 verifies that the transition system semantics defined for Alloy$^{\mathrm{II}}$ achieve that goal. Theorem 5.3 in conjunction with Theorem 5.4 demonstrate that it decidability is not a barrier to creating an Alchemy-like tool for the semantics such that the generated implementations are connected to the logical semantics of Alloy$^{\mathrm{II}}$.

Before we prove these theorems, we will show the connection between the satisfiability of an Alloy$^{\text{II}}$ predicate p and the satisfiability of $\phi_{\mathcal{A},p}$ from Definition 5.1.

Lemma 5.1. *Given an Alloy$^{\text{II}}$ specification $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$, a predicate $p \in \mathsf{Preds}$ is satisfiable if and only if $\phi_{\mathcal{A},p}$ is satisfiable.*

Proof. Assume p is satisfiable. This means that the corresponding formula ψ in the kernel semantics of Alloy$^{\text{II}}$ is satisfiable. Because $\phi_{\mathcal{A},p}$ is identical to ψ except for the initial existential quantifiers, by the definition of satisfiability for Alloy$^{\text{II}}$, $\phi_{\mathcal{A},p}$ is also satisfiable.

Assume $\phi_{\mathcal{A},p}$ is satisfiable. An instance which satisfies $\phi_{\mathcal{A},p}$ trivally satisfies the kernel language formula ψ corresponding to p. Therefore, p is satisfiable. □

Theorem 5.2. *Given an Alloy$^{\text{II}}$ specification $\mathcal{A} = \{\mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds}\}$, there is some $p \in \mathsf{Preds}$ such that $\phi_{\mathcal{A},p}$ is satisfiable if and only if $\mathcal{A}$ is non-trivially implementable in the Alloy$^{\text{II}}$ transition system semantics. Moreover, for each $p \in \mathsf{Preds}$ such that $\phi_{\mathcal{A},p}$ is satisfiable, there is some transition $\langle q, a, q' \rangle \in \delta_{\mathcal{A}}$ where p is the first component of* Act.

Proof. This follows directly from the definition of the Alloy$^{\text{II}}$ transition system semantics.

Let $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$. Assume that $p \in \mathsf{Preds}$ is such that $\phi_{\mathcal{A},p}$ is satisfiable. Then there is some instance $\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle$ such that $\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle \vDash \phi_{\mathcal{A},p}$. Therefore, $\tau(\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle) = \langle i_{\text{pre}}, a, i_{\text{post}} \rangle$ is a transition in $\delta_{\mathcal{A}}$. Also, $a = \langle pname, \mathsf{args} \rangle$ where *pname* is the name of p.

Assume that there is some transition $\langle i_{\text{pre}}, a, i_{\text{post}} \rangle$ in $\delta_{\mathcal{A}}$ where $a = \langle pname, \mathsf{args} \rangle$. Then there is a corresponding instance $\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle$ such that $\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle \vDash \phi_{\mathcal{A},p}$ where p is the predicate in Preds named *pname*. □

Remark. Lemma 5.1 demonstrates that Theorem 5.2 is a proof of our desired result about the connection between satisfiability in Alloy$^{\text{II}}$'s logical semantics and non-trivial implementability in Alloy$^{\text{II}}$'s transition system semantics.

Theorem 5.3. *Given an Alloy$^{\text{II}}$ specification $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$, there is some $p \in \mathsf{Preds}$ such that $\phi_{\mathcal{A},p}$ is strictly satisfiable if and only if $\mathcal{A}$ is non-trivially implementable in the strict Alloy$^{\text{II}}$ transition system semantics. Moreover, for each $p \in \mathsf{Preds}$ such that $\phi_{\mathcal{A},p}$ is strictly satisfiable, there is some transition $\langle q, a, q' \rangle \in \delta_{\mathcal{A}}$ where p is the first component of* Act.

Proof. This follows directly from the definition of the strict Alloy$^{\text{II}}$ transition system semantics.

Let $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$. Assume that $p \in \mathsf{Preds}$ such that $\phi_{\mathcal{A},p}$ is strictly satisfiable. Then there is some instance $\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle$ such that $\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle$ strictly satisfies $\phi_{\mathcal{A},p}$. Therefore, $\tau(\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle) = \langle i_{\text{pre}}, a, i_{\text{post}} \rangle$ (where the universe of i_{post} is a subset of the universe of i_{pre} and the atoms bound to the arguments in the action) is a transition in $\delta_{\mathcal{A}}$. Also, $a = \langle \mathit{pname}, \mathsf{args} \rangle$ where *pname* is the name of p.

Assume that there is some transition $\langle i_{\text{pre}}, a, i_{\text{post}} \rangle$ in $\delta_{\mathcal{A}}$ where $a = \langle \mathit{pname}, \mathsf{args} \rangle$ and the transition is strict. Then there is a corresponding instance $\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle$ such that $\langle i_{\text{pre}}, i_{\text{post}}, \eta \rangle$ strictly satisfies $\phi_{\mathcal{A},p}$ where p is the predicate in Preds named *pname*. □

Remark. Lemma 5.1 in conjunction with Theorem 5.3 does not yield a relationship between satisfiability and non-trivial implementability in the strict Alloy$^{\text{II}}$ transition system semantics. This differs from the results drawn from Theorem 5.2 about the Alloy$^{\text{II}}$ transition system semantics because Lemma 5.1 only concerns satisfiability of $\phi_{\mathcal{A},p}$ not *strict* satisfiability of $\phi_{\mathcal{A},p}$.

Theorem 5.4. *The following problem is decidable:*

> INPUT*: An Alloy*$^{\text{II}}$ *specification* $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$*, a state* $q \in Q_{\mathcal{A}}$*, an action* $a \in A_{\mathcal{A}}$
>
> QUESTION*: Is there some* $q' \in Q_{\mathcal{A}}$ *such that the transition* $\langle q, a, q' \rangle \in \delta_{\mathcal{A}}$ *as defined by the strict Alloy*$^{\text{II}}$ *transition system semantics?*

Moreover, there is a procedure that yields q' *if it exists.*

Proof. The bound imposed on the post-state of the transition implies that there are a finite number of potentially acceptable transitions. Each of these transitions may be mapped to several instances such that applying $\tau_{\mathcal{A}}$ to the instance-predicate pair will yield the original transition. However, all of the variables which are free in $\phi_{\mathcal{A},p}$ (where p is the predicate named in the action of the transition) are necessarily the same in each of those instances. So, if any one of those instances satisfies $\phi_{\mathcal{A},p}$, every one of them does.

Therefore, there are a finite number of instances which to test to determine if there is some post-state that forms an acceptable transition. Also, once a satisfying instance has been found, the post-state can be determined from that instance using $\tau_{\mathcal{A}}$. □

Chapter 6

A Second Transition System Semantics for Alloy

Using the experience of creating a transition system semantics for Alloy^{II} we can now identify what caused the naïve transition system semantics to not match the logical semantics of Alloy. Once we have identified what caused the mismatch, we can create a new transition system semantics that does correspond to the logical semantics.

6.1 Two vs. One

The most obvious candidate for causing the mismatch is that in NTSS is that there are two Alloy instances under consideration–one for each of the pre- and post-state–and in Alloy's semantics there is one instance under consideration at a time. When modeling stateful operations, the multiple states are embedded in the instance using the atoms in the state-relation bound to the state variables in a predicate header to identify different states.

This becomes apparent when we consider what it was about the counter examples that was the source of the semantic mismatch. When we interpret `one r` in an instance that is representing multiple states, the interaction between the `one r` constraint, which means the relation `r` has only one tuple, and the declaration of `r` as a function from `State` to `B` imposes a constraint that makes `State` a singleton relation. Thus, the same atom from state will be assigned to each of `s` and `s'` in the predicate, essentially ensuring that the predicate is only satisfiable if it is when the state does not change from pre-state to post-state. Since `change_r1` explicitly says that the pre-state and post-state should be different, the predicate becomes unsatisfiable.

NTSS doesn't reflect this problematic interaction because it keeps two separate relations for `r` in the pre-state and `r` in the post-state and applies the constraint to each state individually. This is the fundamental mismatch between NTSS and the logical semantics of Alloy.

6.2 New Things in the Universe

The second place where issues occur involves the special meaning in NTSS for the variables annotated as `New`. In a sense this break is more fundamental than the last. The semantics of the New variables involve adjoining atoms to signature relations. This is something that simply cannot be done in Alloy, and unlike the previous break that corresponded to a technique used within the semantics of Alloy to model statefulness in other relations, there is no corresponding technique for changing signature relations.

6.3 Fixing the Problems with NTSS

Small adjustments to NTSS give us a transition system semantics for Alloy specifications that have the satisfiability-implementability correspondence property that we would like. These adjustments primarily involve repairing the two above-mentioned problems with Alchemy's semantics.

To fix the first problem, we use a single instance, instead of two, to represent both states of a transition. So, instead of the two separate instances that appear in Figure 3.1, when discussing the address book example, we have the single instance in Figure 6.1.

Pre-state and post-state in one instance I, such that

$$\begin{aligned} I(\texttt{Name}) &= \{\text{Bob}, \text{Sue}\} \\ I(\texttt{Addr}) &= \{30 \text{ School Rd}, 5 \text{ College Ln}\} \\ I(\texttt{Book}) &= \{b1, b2\} \\ I'(\texttt{entries}) &= \{\langle b1, \text{Bob}, 30 \text{ School Rd}\rangle, \langle b2, \text{Bob}, 30 \text{ School Rd}\rangle, \\ &\qquad \langle b2, \text{Sue}, 5 \text{ College Ln}\rangle\} \end{aligned}$$

The corresponding action

$$\langle \texttt{add}, \{\langle \texttt{n}, \text{Sue}\rangle, \langle \texttt{a}, 5 \text{ College Ln}\rangle\}\rangle$$

Figure 6.1: Two states merged into one instance

Whether or not a transition exists between the two states represented here can now be determined by whether this instance satisfies the specification. However, we now have to figure out how to interpret this as two distinct states for the purposes of the transition system semantics. Separating the single instance into the two instances we had originally considered is simple enough, but because they must come from one instance, we have complicated the notion of new variables.

Instead of using an entire instance as the state in the transition system, we will

use only the part of the instance that can be modeled as changing: the fields of the state signature. With this we can formally define a new imperative semantics for Alloy.

6.4 State-Signature Transition System Semantics

The key to defining the transition system semantics for Alloy$^{\text{II}}$ was in defining the connection between instances and transitions explicitly and then using satisfiability of the instance to determine acceptability of the transition. We repeat this construction with the state-signature transition system semantics (SSTSS) so that it will correspond to the logical semantics of Alloy.

Given a specification $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$, let $\mathcal{I}_{\mathcal{A}}$ be the set of instances corresponding to the relation names defined in $\mathcal{A}$. First we will define a schema $\mathcal{S}_{\mathcal{A}}$ associated with the specification. This schema is not the same as the schema for the Alloy instances of $\mathcal{A}$. In Alloy$^{\text{II}}$ we were able to use the same schema as for the instances because the mechanism for identifying which state to which each relation belongs was made part of the semantics of the language. Alloy has no such mechanism, so we have an extra relation whose purpose is solely to identify the state of each tuple in each relation. Moreover, those relations that are not tagged with atoms from this state relation are not modeled to change over time, and so can be considered independent of the state being modeled.

With this in mind, the schema $\mathcal{S}_{\mathcal{A}}$ includes the names of the relations that are the fields of `State` and associates them with the same sorts as the corresponding schema for the Alloy instances, but with the initial `State` sort removed.

Now we can define

- $Q_{\mathcal{A}}$ the set of database instances over $\mathcal{S}_{\mathcal{A}}$, and

- $\mathsf{Act}_{\mathcal{A}}$ the set of name-argument pairs, where the names are those of the predicates in Preds and the argument is a function from each name in the header of the named predicate other than `s` and `s'` to a relation of the specified arity and sort.

Definition 6.1. Given an Alloy specification $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$, define for each predicate $p \in \mathsf{Preds}$, a formula $\phi_{\mathcal{A},p}$ that is the conjunction of the bodies of the facts, the implicit constraints present in the signatures, the body of the predicate and the implicit constraints in the header of the predicate. This formula is the translation of the predicate into the kernel language of Alloy with the existentially quantified variables from the header of the predicate removed.

An instance I *strictly* satisfies $\phi_{\mathcal{A},p}$ if I satisfies $\phi_{\mathcal{A},p}$ and the set of atoms in the image of I under the state-signature fields joined to $I(\mathtt{s'})$ is a subset of the set of atoms in the image of I under the state-signature fields joined to $I(\mathtt{s})$ and the atoms in the image of I under the variable names declared in the header of p.

As with the transition system semantics for Alloy$^{\mathrm{II}}$, we take seriously the notion of a transition being embedded in an Alloy instance. Because the embedding is more complicated in Alloy, the extraction is more involved[1]. However, the definition of the function is still quite natural. Define $\tau_{\mathcal{A}} : \mathsf{Preds} \times \mathcal{I}_{\mathcal{A}} \to Q_{\mathcal{A}} \times A_{\mathcal{A}} \times Q_{\mathcal{A}}$, such that for each predicate $p \in \mathsf{Preds}$ and instance $I \in \mathcal{I}_{\mathcal{A}}$, $\tau_{\mathcal{A}}(p, I) = \langle q, a, q' \rangle$ where for each relation name `r` in the state-signature $q(\mathtt{r}) = I(\mathtt{s}).I(\mathtt{r})$ and $q'(\mathtt{r}) = I(\mathtt{s'}).I(\mathtt{r})$, and $a = (p, \mathsf{args})$ where args is such that for each variable name `v` declared in the header of p, $\mathsf{args}(\mathtt{v}) = I(\mathtt{v})$.

This definition fixes the atom bound to `s` as the marker for the pre-state and the atom bound to `s'` as the marker for the post-state. The assumption that specifications intend this interpretation is part of the idiom of using Alloy to write

[1] We must use a more subtle hammer, and there are several choices of such hammers.

specifications of stateful systems in this style.

Definition 6.2 (SSTSS acceptable transition)**.** Fix a specification $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$. A transition $\langle q, a, q' \rangle$ is $\mathcal{A}$-acceptable if there is some instance $I \in \mathcal{I}_{\mathcal{A}}$ such that there is some $p \in \mathsf{Preds}$ such that $\tau_{\mathcal{A}}(p, I) = \langle q, a, q' \rangle$ and I satisfies $\phi_{\mathcal{A},p}$. When the intended specification is clear, we will simply write acceptable instead of $\mathcal{A}$-acceptable.

Definition 6.3 (SSTSS)**.** The state-signature semantics are defined such that for each Alloy specification $\mathcal{A}$, we have the transition system $(Q_{\mathcal{A}}, A_{\mathcal{A}}, \delta_{\mathcal{A}})$ where $\delta_{\mathcal{A}}$ is the set of acceptable transitions in $Q_{\mathcal{A}} \times A_{\mathcal{A}} \times Q_{\mathcal{A}}$.

Definition 6.4 (Strict SSTSS)**.** The strict SSTSS are defined such that for each Alloy specification $\mathcal{A}$ we have a transition system $(Q_{\mathcal{A}}, A_{\mathcal{A}}, \delta_{\mathcal{A}})$ where $\delta_{\mathcal{A}}$ is the set of strict acceptable transitions in $Q_{\mathcal{A}} \times A_{\mathcal{A}} \times Q_{\mathcal{A}}$.

Examples

As an example of applying these semantics, consider the address book example in Figure 2.1, treating the name `Book` as `State` and `b` and `b'` as `s` and `s'` in the semantics. As it currently is written, the state space would consist of different values for the `entries` relation. Since we would like to have access of the names and addresses in the book that may not be in `entries`, it make sense to modify the specification as in Figure 6.2.

Note that we no longer need special annotation for adding new atoms to a state. Figure 6.3 gives an example of a transition in this specification according SSTSS, and Figure 6.4 gives an instance that satisfies the specification and corresponds to the transition.

```
sig Name {}
sig Addr {}
sig Book {
        names : Name, // so it will be included in the state
        addrs : Addr, // so it will be included in the state
        // using names and addr instead of Names and Addr
        // enforces that entries can only have names and address
        // from names and addr at the correct timestamps
        entries : names -> lone addr
}

pred show {
        #Book.entries > 1
}

pred add[b, b' : Book, n : Name, a : Addr] {
        // a is new -- we could not say this before
        a not in b.addrs
        a in b'.addrs
        // n is new -- we could not say this before
        n not in b.names
        n in b'.names

        b'.entries = b.entries + (n -> a)
}

pred del[b, b' : Book, n : Name, a : Addr] {
        b'.entries = b.entries - (n -> a)
}

assert delUndoesAdd {
        all b, b', b'' : Book, n : Name, a : Addr |
                (no b.entries and add[b,b',n,a] and del[b',b'',n,a])
                implies
                (b.entries = b''.entries)
}
```

Figure 6.2: Specification for an address book that works better with SSTSS

Pre-state q

$$
\begin{aligned}
q(\texttt{names}) &= \{\text{Bob}\} \\
q(\texttt{addrs}) &= \{\text{30 School Rd}\} \\
q(\texttt{entries}) &= \{\langle \text{Bob}, \text{30 School Rd}\rangle\}
\end{aligned}
$$

Action

$$\langle \texttt{add}, \{\langle \texttt{n}, \text{Sue}\rangle, \langle \texttt{a}, \text{5 College Ln}\rangle\}\rangle$$

Post-state q'

$$
\begin{aligned}
q'(\texttt{names}) &= \{\text{Bob}, \text{Sue}\} \\
q'(\texttt{addrs}) &= \{\text{30 School Rd}, \text{5 College Ln}\} \\
q'(\texttt{entries}) &= \{\langle \text{Bob}, \text{30 School Rd}\rangle, \langle \text{Sue}, \text{5 College Ln}\rangle\}
\end{aligned}
$$

Figure 6.3: A transition in the state-signature semantics

$$
\begin{aligned}
I(\texttt{Name}) &= \{\text{Bob}, \text{Sue}\} \\
I(\texttt{Addr}) &= \{\text{30 School Rd}, \text{5 College Ln}\} \\
I(\texttt{Book}) &= \{\text{b1, b2}\} \\
I(\texttt{names}) &= \{\langle b1, \text{Bob}\rangle, \langle b2, \text{Bob}\rangle, \langle b2, \text{Sue}\rangle\} \\
I(\texttt{addrs}) &= \{\langle b1, \text{30 School Rd}\rangle, \langle b2, \text{30 School Rd}\rangle, \langle b2, \text{5 College Ln}\rangle\} \\
I(\texttt{entries}) &= \{\langle b1, \text{Bob}, \text{30 School Rd}\rangle, \langle b2, , \text{Bob30 School Rd}\rangle, \\
&\qquad \langle b2, \text{Sue}, \text{5 College Ln}\rangle\} \\
I(\texttt{b}) &= \{b1\} \\
I(\texttt{b'}) &= \{b2\} \\
I(\texttt{n}) &= \{\text{Sue}\} \\
I(\texttt{a}) &= \{\text{5 College Ln}\}
\end{aligned}
$$

Figure 6.4: Instance corresponding to the transition in Figure 6.3

6.5 Properties of SSTSS

The definitions of SSTSS and strict SSTSS again merit discussion about their properties regarding trivial and non-trivial implementability. Theorem 6.2 proves that SSTSS has the desired correspondence with the logical semantics of Alloy. Theorem 6.3 gives us the same result for the strict case, as we saw was useful for Alloy$^{\text{II}}$, however Theorem 6.4 shows that we cannot create an Alchemy-like tool that uses the SSTSS as its underlying semantics.

As we did with the Alloy$^{\text{II}}$ transition system semantics, we will first show the connection between the satisfiability of an Alloy predicate p and the satisfiability of $\phi_{\mathcal{A},p}$ from Definition 6.1.

Lemma 6.1. *Given an Alloy specification* $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$*, a predicate* $p \in \mathsf{Preds}$ *is satisfiable if and only if* $\phi_{\mathcal{A},p}$ *is satisfiable.*

Proof. Assume p is satisfiable. This means that the corresponding formula ψ in the kernel semantics of Alloy is satisfiable. Because $\phi_{\mathcal{A},p}$ is identical to ψ except for the initial existential quantifiers, by the definition of satisfiability for Alloy, $\phi_{\mathcal{A},p}$ is also satisfiable.

Assume $\phi_{\mathcal{A},p}$ is satisfiable. An instance which satisfies $\phi_{\mathcal{A},p}$ trivally satisfies the kernel language formula ψ corresponding to p. Therefore, p is satisfiable. □

Theorem 6.2. *Given an Alloy specification* $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$*, there is some* $p \in \mathsf{Preds}$ *such that* $\phi_{\mathcal{A},p}$ *is satisfiable if and only if* $\mathcal{A}$ *is non-trivially implementable in the state-signature semantics. Moreover, for each* $p \in \mathsf{Preds}$ *such that* $\phi_{\mathcal{A},p}$ *is satisfiable, there is some transition* $\langle q, a, q' \rangle$ *in the implementation where the name of* p *is the first component of* a*.*

Proof. Assume there is some $p \in \mathsf{Preds}$ such that $\phi_{\mathcal{A},p}$ is satisfiable. Let I be a satisfying instance for $\phi_{\mathcal{A},p}$. Then, $\tau(p, I) = \langle q, a, q' \rangle$ is $\mathcal{A}$-acceptable, and so

$\langle q, a, q' \rangle \in \delta_{\mathcal{A}}$. Thus, the implementation is non-trivial. Also, the first component of a is the name of p.

Assume that the implementation of $\mathcal{A}$ is non-trivial. Then there is some transition $\langle q, a, q' \rangle \in \delta_{\mathcal{A}}$. That transition is $\mathcal{A}$-acceptable, so there is some $p \in \mathsf{Preds}$ and $I \in \mathcal{I}_{\mathcal{A}}$ such that I satisfies $\phi_{\mathcal{A},p}$. Also, the first component of a is the name of p. □

Remark. As with Theorem 5.2 for Alloy$^{\text{II}}$, Theorem 6.2 demonstrates in conjunction with Lemma 6.1 our desired result about the connection between satisfiability in Alloy's logical semantics and non-trivial implementability in SSTSS.

Theorem 6.3. *Given an Alloy specification* $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$*, there is some* $p \in \mathsf{Preds}$ *such that* $\phi_{\mathcal{A},p}$ *is strictly satisfiable if and only if* $\mathcal{A}$ *is non-trivially implementable in strict SSTSS. Moreover, for each* $p \in \mathsf{Preds}$ *such that* $\phi_{\mathcal{A},p}$ *is strictly satisfiable, there is some transition* $\langle q, a, q' \rangle$ *in the implementation of* $\mathcal{A}$ *where the name of* p *is the first component of* a*.*

Proof. This proof is identical to the proof of Theorem 6.2, with the addition that an instance strictly satisfies $\phi_{\mathcal{A},p}$ if and only if $\tau(p, I)$ is strict. □

Remark. As was the case with the strict Alloy$^{\text{II}}$ transition semantics, Lemma 6.1 in conjunction with Theorem 6.3 does not yield a relationship between satisfiability and non-trivial implementability in strict SSTSS. As before, this differs from the results drawn from Theorem 6.2 about the SSTSS because Lemma 5.1 only concerns satisfiability of $\phi_{\mathcal{A},p}$ not *strict* satisfiability of $\phi_{\mathcal{A},p}$.

In the spirit of Alchemy, we also wish to explore whether it is possible to automatically synthesize an implementation from an Alloy specification according to strict SSTSS. Theorem 6.4 shows that it is not possible to do so in general.

Theorem 6.4. *The following problem is undecidable:*

> INPUT*: An Alloy specification* $\mathcal{A} = \langle \mathsf{Sigs}, \mathsf{Facts}, \mathsf{Preds} \rangle$*, a state* $q \in Q_{\mathcal{A}}$*, an action* $a \in \mathsf{Act}_{\mathcal{A}}$
>
> QUESTION*: Is there some* $q' \in Q_{\mathcal{A}}$ *such that the transition* $\langle q, a, q' \rangle \in \delta_{\mathcal{A}}$ *for the implementation of* $\mathcal{A}$ *in strict SSTSS?*

Proof. The key to this proof is that the bounds on the post-state under these imperative semantics do not necessarily mean that the instance corresponding to the transition is bounded itself. To exploit this and show that the problem is undecidable, we will reduce finite satisfiability in predicate logic to this problem.

Consider the specification $\mathcal{A}$ in Figure 6.5. According to the state-signature semantics, this specification corresponds to a transition system where the state-space is the set containing the empty set and the action space is the set of pairs where the first element is `p1` and the second is a binding of `s` and `s'`. Thus, the only possible transition is one from the empty set to itself, with some action. As is, it is clear that $\delta = Q_{\mathcal{A}} \times A_{\mathcal{A}} \times Q_{\mathcal{A}}$.

Take any first order logic sentence σ written in Alloy and add a fact with that sentence as its body to $\mathcal{A}$ to form $\mathcal{A}^+$. Assume there is an algorithm that can determine whether there is an $\mathcal{A}^+$-acceptable transition whose pre-state is q and whose action is $\langle \mathtt{p1}, \{\langle s, s0 \rangle, \langle s', s0 \rangle\} \rangle$. Note that the bound on the post-state

If that algorithm determines that there is a post-state, then we know that the sentence σ is finitely satisfiable. If the algorithm determines that there is not, then we know that the sentence σ is not finitely satisfiable. Thus the algorithm would decide finite satisfiability in first order logic, which is impossible.

Therefore, the above problem is undecidable.

□

```
State { }
pred p1[s, s' : State] { }
```

Figure 6.5: A very simple specification

The significance of this result is that we cannot create automatically create implementations of Alloy specifications in accord with the state-signature semantics, because finding the post state, given a pre-state and action, is undecidable.

It is worth noting that the definition of strict satisfiability for use with the state-signature semantics hinted at this result, since the bound was only on part of the instance as opposed to the whole instance as with the naïve transition system semantics. This could be repaired by finding a way to extend the bound to the whole instance, probably by specifying the state-signature semantics using a function τ^{-1} (such that $\tau(\tau^{-1}(t)) = t$, where t is a transition) instead of τ itself, but this seemed to have great potential for creating semantics that were very unintuitive and it did not solve the usability issues with restricting the types of specifications that could be written to a subset of all Alloy specifications.

Chapter 7

Advice for Alloy Users

Our results suggest advice to Alloy users: that they should make note that, despite what may be suggested by the examples in *Software Abstractions* [9], facts in an Alloy specification are not state invariants. Facts hold over the relations that represent all states at once, not over each state individually. To write a fact that does act as a state invariant, Alloy users should either write the fact as a sig-fact on the `State` signature, or should begin their fact with `all s : State | ...` and preface each field of the `State` signature with `s.`. When using other methods for modeling state similar approaches should be used.

The reasons for this particular convention are exemplified by the specification that was problematic for NTSS. The problem is so nefarious because many state invariants can be written as facts without worrying about facts holding over the relation containing all states. In particular, formulas of the form `p in q` do not cause problems, because those formulas are equivalent to `all s : State | s.p in s.q`. Cardinality restrictions were probably the first examples found where treating facts as state invariants ceases to work because cardinality restrictions are a very easy way of restricting a relation across all states.

Since stateful systems are being modeled *within* Alloy, the techniques used to model the system must adhere to the semantics of Alloy. A more thorough understanding of the semantics of Alloy and of what transition system semantics are being used to interpret Alloy specifications as stateful systems is the best way to prevent mistakes like the one above. However, we cannot expect every software engineer to be a theorist as well, so the syntactic advice for how to avoid the particular misunderstanding of facts as state invariants must suffice.

An alternative would be to use a modeling language that was designed for the modeling of stateful systems, so that the intuitive interpretation of specifications yielded the correct understanding in both the logical semantics and the transition system semantics. Once candidate for such a language is Alloy$^{\mathrm{II}}$, which has the additional advantage of using a syntax familiar to Alloy users. The prototype implementation of an analyzer for Alloy$^{\mathrm{II}}$ which we created is discussed in Appendix A. Since the transition system semantics for Alloy$^{\mathrm{II}}$ supports the creation of a automatic code generator for specifications written in Alloy$^{\mathrm{II}}$, we have created a prototype Alchemy$^{\mathrm{II}}$ as well, which is discussed in Appendix B.

Chapter 8

Related work

Our work can be seen as a result toward software synthesis, an effort initiated by Green [6] and Waldinger and Lee [19] and summarized by Rich and Waters [16]. Subsequent projects, such as those by Burstall and Darlington [3] and Manna and Waldinger [11], have instead attempted program synthesis in a rule-based fashion by transformation and deduction. Some, like Smith [17], have focused on the programs obeying certain decomposition strategies. These efforts tend to require significant manual interaction, and also typically avoid state at least at the syntactic level.

The notion of adequacy is standard in relating two semantics for a given language. Its introduction is usually credited to Plotkins seminal work on the treatment of LCF as a programming language [15]. In our case, adequacy is a relationship between the denotational world of models and analysis, and the operational world of the implementation.

Several efforts have tried to relate proofs to running programs. Bates and Constable [2] initiated a significant research program on the extraction of computational context, in the form of programs, from constructive proofs. This effort continues in popular proof assistants such as Coq [12]. Unfortunately, the proof structures used

in Alloy do not lend themselves to such extraction.

Executable UML [14] and other approaches to model-driven development are attempts to proceed from specifications to programs, with the expectation of applying rich analytic tools to the specifications. As a notable difference, their specifications tend to cover most of the system, rather than being lightweight; this has practical consequences in the synthesis of programs rather than libraries (as in our case). Furthermore, the range of analyses available is correspondingly greater due to the richness and variety of expression. Nevertheless, we are not aware of work in this area that reconciles the problems we discuss and produces implementations that demonstrate computational adequacy.

In contrast to Executable UML, two other efforts — SPECWARE [13] and the B-method [1] — offer much more formal approaches to software development. Both use refinements as a basis for converting specifications into programs. In SPECWARE, the refinement process creates proof-obligations that the user must discharge. In B, specifications are refined until they have been made deterministic, at which point code-generation is straight-forward. Our approach is far more automated, and our focus is not on synthesis but on relating verification to the resulting code. Nevertheless, both SPECWARE and B have been successfully applied to large, non-trivial systems, so there are numerous lessons to be learned from them as we scale our work.

DynAlloy [5] is an extension to Alloy to express state change in specifications. The authors of this work make observations we echo about the difficulties of reading predicates, and design a related language as a consequence. We believe, however, that their design decisions result in a more traditionally imperative language than Alloy$^{\mathrm{II}}$. More importantly for this paper, their work focuses on analysis, and does not tackle the generation of code or its relationship to the results of analysis.

Our work is also related to the Semantic Data Model (SDM) [7], an influential formalism for describing hierarchical data models. This model supports a rich set of features such as object hierarchies, data constraints, aggregation of entities, and definitions for derived data, but with less regularity than Alloy. Though there have been efforts to build programming languages atop this model (DIAL [8] is an early example), we are not aware of verification efforts for the SDM, so these efforts are not comparable to ours in relating these two contexts. Similarly, our implementation techniques bear a resemblance to efforts on active databases [4], but again our focus is on relating the verification and implementation realms.

Krishnamurthi, Dougherty, Fisler and Yoo's work [10] introduced the Alchemy project, which was where the difficulties in interpreting Alloy specifications as defining transition systems was first discovered.

Appendices

Appendix A

Implementation of a Prototype for an Alloy^{II} Analyzer

Since bounded satisfiability is decidable, as demonstrated in Theorem 4.1, it is possible to implement an analyzer tool for Alloy^{II} like the AlloyAnalyzer for Alloy. We have prototyped an analyzer that functions by translating Alloy^{II} source into Alloy source, which can then be analyzed using the AlloyAnalyzer.

Alloy^{II} differs syntactically from Alloy in reserving priming as an operator. The heart of our translation, then, is the implementation of this operator in Alloy. Replicating each Alloy^{II} signature in a primed version turns uses of the prime operator into valid Alloy syntax over relations. For example, given Alloy^{II} signature

```
sig Person  { friends : set Person }
```

the translator produces the Alloy signatures

```
abstract sig Person'A {}
sig Person { friends : set Person } in Person'A
sig Person' { friends' : set Person' } in Person'A
```

The abstract signature allows the pre- and post-state relations to share atoms (it is not needed for signatures already in another signature). The new signatures make any predicate specification from Alloy^{II} syntactically valid in Alloy. Once "global" facts have been de-sugared, no translation is necessary for facts. The current prototype for the implementation does not handle signature extension or signatures contained in unions of other signatures, simply because translating those constructs to Alloy becomes complicated and tedious.

A.1 Future work

An production quality analyzer for Alloy^{II} would probably translate directly into the language used by Kodkod [18], the engine used by Alloy for finding bounded instances that satisfy first order logic formulas. In addition, because of the similarity between Alloy^{II} and Alloy, it would be convenient for users if the GUI used by Alloy was adopted for use with Alloy^{II}.

A.1.1 Assertions

A more important area for future work on the semantics of Alloy^{II} is creating the syntax and semantics for assertions. In stateful specifications in Alloy, assertions are used to verify properties of a series of states linked by the predicates that represent operations. In this way, the assertions in Alloy bear resemblance to the linear temporal logic (LTL) formulas used to describe properties of Büchi automata. This this resemblance suggests that a sort of bounded LTL-like language might be appropriate for writing assertions about Alloy^{II} specifications.

Appendix B

Implementation of a Prototype for Alchemy^{II}

The simplest way to implement a code library generation tool for Alloy^{II} would be to use the Kodkod engine [18] to find an instance with bounds on the pre-state and post-state portions of an instance, as given by the pre-state and action, and to extract the post-state from that instance. Our Alchemy^{II} prototype was created before we realized the full power of the connection between Alloy^{II} instances and transitions, so it takes a slightly different approach to synthesizing the library code.

The Alchemy^{II} prototype functions by navigating the kernel-language formula associated with a specification for a predicate, modifying post-state relations so that the formula is satisfied, and backtracking when conflicts occur.

The code for inserting and deleting tuples from expressions is nearly the same as used in the Alchemy project [10], with the difference being that only relations annotated with a prime are considered modifiable. Unlike Alchemy, Alchemy^{II} operates on the formula in negation normal form with small modifications to the base formulas of the form `p in q` and `not p in q`, which are rewritten as `p - q in none`

and `not p - q in none`, respectively.

The algorithm for determining the post-state given a pre-state and action is given in Figure B.1. Termination of the algorithm is guaranteed by a homogeneity condition, indicating that any tuple cannot be both inserted and deleted from any relation. If the algorithm encounters a situation where this would occur, it backtracks and tries again. If no choices are left, the algorithm returns failure.

Safety for the algorithm in Figure B.1 is easily seen, as the iteration to a fixpoint will only halt and return an answer when the post-state yields a transition that corresponds to a satisfying instance. Completeness is also easily seen, as the algorithm will generate every change to the pre-state involving those relations that appear in the formula under consideration. Since relations that do not appear in the formula do not affect whether the formula is satisfied, this is sufficient to determine if there is any post-state corresponding to a satisfying instance.

The code generated by the implementation is usable through a PLT Scheme API. The code acts on a file that serves as a persistent store of the database, but as what the code generates is the changes that are to be made to the state in terms of the insertion and deletion of tuples, it is a simple matter to adjust the algorithm to produce SQL statements to operate on a relational database.

B.1 Future work

The primary work that remains to be done on Alchemy^{II} are to support a richer full Alloy^{II} language and to improve the user interface so that more complex interactions with the database and code library are possible. The runtime of Alchemy^{II} is also currently painfully slow, even for a tool for prototyping projects, so either improvements to the current algorithm or making use of the Kodkod engine may be

```
// η corresponds to the action
findPostState(f :formula, η : environment, i_pre) {
        i_post ← i_pre
        iterate until fixpoint on i_post
                i_post = makeTrue(f, η, i_pre, i_post)
        return i_post
}

makeTrue(f : formula, η : environment, i_pre : database,
                        i_post : database) {
        case f:
        f ≡ f1 and f2
                makeTrue(f1, η, i_pre, makeTrue(f2, η, i_pre, i_post))
        f ≡ f1 or f2
                choose(makeTrue(f1, η, i_pre, i_post),
                       makeTrue(f2, η, i_pre, i_post))
        f ≡ all x : A | f1
                fold((λ v i . makeTrue(f1, η⊕(x ↦ v),
                        i_pre, i)), i_pre, eval(A))
        f ≡ some x : A | f1
                choose some v in A
                makeTrue(f1, η⊕(x ↦ v), i_pre, i_post)
        f ≡ e in none
                fold((λ t i . deleteTuple(t, e, η, i_pre, i)),
                        i_post, eval(e))
        f ≡ not (e in none)
                choose some t of the type of e
                // if possible, choose tuples in e1,...,en first,
                // where e = e1 & ... & en
                insertTuple(t, e, η, i_pre, i_post)
}
```

Figure B.1: Algorithm for Alchemy^{II}

appropriate for decreasing the runtime of the execution of operations. Making use of Kodkod directly may prove the most fruitful since Kodkod is an already mature tool for satisfiability solving for first order logic and relational algebra.

Bibliography

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

[2] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, 1985.

[3] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1), January 1977.

[4] Stefano Ceri and Jennifer Widom. Deriving incremental production rules for deductive data information systems. *Information Systems*, 19(6):467–490, 1994.

[5] Marcelo F. Frias, Carlos G. López Pombo, Gabriel A. Baum, Nazareno M. Aguirre, and Thomas S. E. Maibaum. Reasoning about static and dynamic properties in Alloy: A purely relational approach. *ACM Transactions on Programming Languages and Systems*, 14(4):478–526, 2005.

[6] Cordell C. Green. Application of theorem proving to problem solving. In *International Joint Conference on Artificial Intelligence*, 1969.

[7] Michael Hammer and Brian Berkowitz. DIAL: A programming language for data intensive applications. In *ACM SIGMOD International Conference on Management of Data*, 1980.

[8] Michael Hammer and Dennis McLeod. The semantic data model: a modelling mechanism for data base applications. In *ACM SIGMOD International Conference on Management of Data*, 1978.

[9] Daniel Jackson. *Software Abstractions*. MIT Press, 2006.

[10] Shriram Krishnamurthi, Daniel J. Dougherty, Kathi Fisler, and Daniel Yoo. Alchemy: Transmuting base alloy specifications into implementations. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2008.

[11] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.

[12] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[13] James McDonald and John Anton. SPECWARE - producing software correct by construction. Technical Report KES.U.01.3, Kestrel Institute, March 2001.

[14] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[15] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, pages 223–255, 1977.

[16] Charles Rich and Richard C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, 21(8):40–51, 1988.

[17] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[18] Emina Torlak and Greg Dennis. Kodkod for Alloy users. In *Alloy Workshop*, 2006.

[19] R. J. Waldinger and R. C. T. Lee. PROW: A step toward automatic program writing. In *International Joint Conference on Artificial Intelligence*, 1969.