2009-02-26

Alchemy: Transmuting Base Specifications into Implementations

Daniel Yoo
Worcester Polytechnic Institute

Alchemy: Transmuting Base Alloy Specifications into Implementations

by

Danny Yoo

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

December 2008

APPROVED:

Professor Kathi Fisler, Major Thesis Advisor

Professor George Heineman, Reader

Professor Michael A. Gennert, Head of Department

Abstract

Alloy specifications are used to define lightweight models of systems. We present Alchemy, which compiles Alloy specifications into implementations that execute against persistent databases. Alchemy translates a subset of Alloy predicates into imperative update operations, and it converts facts into database integrity constraints that it maintains automatically in the face of these imperative actions.

In addition to presenting the semantics and an algorithm for this compilation, we present the tool and outline its application to a non-trivial specification. We also discuss lessons learned about the relationship between Alloy specifications and imperative implementations.

Acknowledgements

I'm grateful to Yu Feng, Paul Freitas, Theo Giannakopolos, Chris King, and Tim Nelson for their good advice, fresh baked goods, and warm company. I'm especially thankful to all my advisors: Dan Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Their patience, humor, and constructive criticism mean the world to me. This work is partially supported by the NSF and a GAANN Fellowship.

Contents

List of Figures

Chapter 1

Introduction

Software engineering wisdom encourages developers to explore models of their systems before they commit to implementation details.[1] An especially powerful idea, *lightweight formal methods* [JW96] to prototype ideas and identify errors before realization, has gained substantial traction with the growth of corresponding tools. A leading modeling tool that supports this philosophy is Alloy [Jac00], which enables designers to author and explore petite descriptions of systems using a first-order relational specification language. Indeed, Alloy has become sufficiently popular that its specification language is becoming the focus of an ecosystem of tools, such as theorem provers to analyze specifications and test generators to construct test suites. We provide a brief tutorial on the use of Alloy as a data modeling language in Chapter 2, and cover the semantics of implementations and what such implementations may guarantee.

Having written an Alloy specification, however, its author is no closer to a working implementation. Our work is an attempt to bridge this gulf. Concretely, the elements of an Alloy specification suggest natural implementation counterparts. The

[1] A conference paper version of this work appears in FSE 2008. [KFDY08]

signatures lay out relations that translate directly into persistent database schemas. The facts—those properties that are meant to hold of all models constructed by Alloy—correspond to the database's integrity constraints; maintaining these *automatically* is one of our contributions. Finally, a subset of the predicates in an Alloy specification connote state changes; these (and related helper utilities) become the functions exported by an API. The heart of our synthesis work (Chapter 3) translates these predicates into a library of imperative functions. This work is presented not only formally but also through a working tool, Alchemy, whose implementation we discuss in Chapter 4.

In harmony with the lightweight formal methods philosophy of *partiality*, we focus on the generation of APIs rather than whole programs. Automatic invariant maintenance further supports this philosophy. This scope is of tremendous value because it enables us to generate, for instance, back-ends for Web applications. Chapter 5 relates our work to other program synthesis efforts. To evaluate the feasibility of Alchemy, we apply it on a non-trivial specification and discuss our results (Chapter 6).

From another viewpoint, this work enables the prototyping of application-specific database interfaces. Whereas most database engines exports a "one-size-fits-all" interface, we enable authors to define their desired interface in Alloy. Alchemy translates their specification into an API, hiding the database scaffolding and automatically maintaining integrity. This frees developers to focus on more challenging matters, and reduces vulnerability to some security attacks.

Besides concrete deliverables, we believe the value of this work resides as much in what we have learned from the process of designing Alchemy. In particular, we find a potential mismatch between a stateless, relational semantics and the expected behavior of an imperative implementation; this relationship needs further investi-

gation. We discuss our design decisions, constraints, and lessons at various points, and elaborate on the mismatch in Section 3.3, and conclude by looking into future avenues for improvement in Chapter 7.

Chapter 2

Alloy

The Alloy project [Jac00] consists of an analysis tool, called the *Alloy Analyzer*, and a specification language called *Alloy*. The Alloy Analyzer provides tools to visualize models and perform bounded model checking, working on specifications written in the Alloy language. In this section, we concentrate on the language and use two examples to give an overview of Alloy syntax and semantics. Readers familiar with Alloy should anyway peruse the examples, especially the second in Section 2.2, as we refer to it extensively in the rest of the thesis. Our work supports a subset of Alloy that includes this example; details on the subset are given in Section 2.3.

2.1 Alloy as a Modeling Language

Alloy as a specification language lets us describe a data model. A data model essentially describes three things:

- A description of the structure of the data
- Invariants and constraints on the structure
- Operations that act on that structure

```
class Book {
    Map<Person, Addr> addrs;
}

class Person {
    String firstName;
    String lastName;
}

class Addr {
    String street;
    String zip;
}
```

Figure 2.1: Address book class definitions.

As an example, we can consider a simple address book that maps names to addresses. In a typical object-oriented programming language, we might model an address book's structure as a class with attributes, as shown in Figure 2.1.

In the Java class definition, the `Book` is defined to contain an `addrs` attribute to hold a mapping between `Person` and `Addr`. In Alloy, the *signature* language construct describes structure in a way reminiscent of the class definitions; Figure 2.2 shows how the classes in Figure 2.1 might be modeled as signatures.

Alloy uses signatures to capture the sets and relations that comprise a data model. When we use a specification for design, we suppress details that may not be important to the model. In the Alloy signatures above, note that Person and Addr signatures aren't fully described: we may not necessarily care that a Person has a first and last name, as long as we can distinguish between two different people. Furthermore, we don't describe the mechanism of how *addrs* relates a *Person* and an *Addr*, as that too is an algorithmic detail that's unimportant for our modeling purposes.

```
sig Book {
    addrs: Person → one Addr
}

sig Person {}
sig Addr {}
```

Figure 2.2: Address book signature definitions.

Book
b1

Person
p1
p2

Addr
a1
a2

addrs		
b1	p1	a1
b1	p2	a2

Figure 2.3: A model of the address book.

Although the Alloy signatures look like class definitions, their semantics are not as classes, but as relations. The elements of these relations are called *atoms*; the type of each atom is its containing relation. *Book*, *Person* and *Addr* map to unary relations of their respective atoms. Fields of signatures define additional relations. *addrs* is a 3-arity relation of ($Book \times Person \times Addr$) tuples. A model of a book containing two people with addresses has the relations shown in Figure 2.3.

The use of **one** *Addr* in the definition of the *addrs* relation expresses a multiplicity constraint: for any particular *Book* and *Person*, there's exactly one *Addr* for that *Person*. This multiplicity constraint is an example of an invariant on this structure, which can be expressed with the signature definition. Alternatively, these invariants can be expressed explicitly as *facts* about the system, as shown in Figure 2.4. As these facts are written in first-order relational logic, they can have striking expressive power, which we'll explore in the gradebook example of Section 2.2.

One of the operations in Alloy's supported language is the operator [], which is a pun: although the square brackets look like associative array access, in reality it is a relational join operation. In the expression in Figure 2.4, $b.addrs[p]$ is desugared

```
fact {
    all b: Book, p: Person | one b.addrs[p]
}
```

Figure 2.4: An example of a constraint.

```
pred add(b, b': Book, p: Person, a: Addr) {
    b'.addrs = b.addrs + (p → a)
}
```

Figure 2.5: A example of a stateful predicate.

to the equivalent expression $(p.(b.addrs))$, where the dot notation is the natural relational join.

We finally turn to how to model the effect a stateful operation has on a system. One way to do this is with predicates that relate a pre-state to a post-state. For example, the *add* predicate in Figure 2.5 defines how to add a new entry. b and b' represent pre- and post-states, and the body of *add* captures the effect of the *add* operation.

A model that represents adding *Person p3* to *Book b1*, with *Addr a2* is shown in Figure 2.6. Note that *b1.addrs* and *b2.addrs* both evaluate to a 2-arity relations that can be viewed as pre-state and post-state, respectively. By setting aside one of the signatures, such as *Book*, as a *state* signature, we can project another relation R with the stateful one to get stateful views of R. The post-state variable has a prime in its name by convention. This stateful-primed idiom allows us to express stateful operations in Alloy. Each stateful predicate has parameters for the pre- and post-states of the operation (such as b and b') and constrains the latter to reflect the change applied to the former.[1]

[1]Alloy models of stateful systems often employ the ordering module to sequence states; we currently do not exploit this.

Book
b1
b2

Person
p1
p2
p3

Addr
a1
a2

addrs		
b1	p1	a1
b1	p2	a2
b2	p1	a1
b2	p2	a2
b2	p3	a2

Figure 2.6: A model of the address book satisfying `add(b1,b2,p3,a2)`.

2.2 The Gradebook Example

This example is a homework submission and grading system, shown in Figure 2.7. In this system, students submit work in pairs. The gradebook stores the grade for each student on each submission. Students may be added to or deleted from the system at any time, as they enroll in or drop the course. This example is adapted from a deployed system that my advisor, Kathi Fisler, developed for our department. She prototyped data models for the system in Alloy early in the design phase before manually porting the models to a Web-based implementation.

The system's data model centers around a course, which has three subfields: a roster (set of students), submitted work (relation from enrolled students to submissions), and a gradebook. As in the earlier example, each **sig** (*Submission*, etc.) defines a unary relation, and each field defines an n-arity relation. The **sig** for *Course* declares *roster* to be a relation on $Course \times Student$. Similarly, the relation *work* is of type $Course \times Student \times Submission$, but with the projection on *Course* and *Student* restricted to pairs in the *roster* relation. The **lone** annotation on *gradebook* allows at most one grade per submission.

The stateful predicates (*Enroll*, etc.) capture the actions supported in the system, following the same standard Alloy idiom for stateful operations. Facts (such as *SameGradeForPair*) capture invariants on the models. This particular fact states that students who submit joint work get the same grade.

```
sig Submission {}
sig Grade {}
sig Student {}

sig Course {
   roster : set Student,
   work : roster → Submission,
   gradebook : work → lone Grade }

pred Enroll (c, c' : Course, sNew : Student) {
    c'.roster = c.roster + sNew and
    no c'.work[sNew] }

pred Drop (c, c' : Course, s: Student) {
   s not in c'.roster }

pred SubmitForPair (c, c' : Course, s1, s2 : Student,
                    bNew : Submission) {
   // pre-condition
   s1 in c.roster and s2 in c.roster and
   // update
   c'.work = c.work + (s1 → bNew) + (s2 → bNew) and
   // frame condition
   c'.gradebook = c.gradebook }

pred AssignGrade (c, c' : Course, s : Student,
                  b : Submission, g : Grade) {
   c'.gradebook in c.gradebook + (s → b → g) and
   c'.roster = c.roster }

fact SameGradeForPair {
   all c : Course, s1, s2 : Student, b : Submission |
       b in (c.work[s1] & c.work[s2]) implies
           c.gradebook[s1][b] = c.gradebook[s2][b] }
```

Figure 2.7: Alloy specification of a gradebook.

$Student = \{Harry, Meg\}$
$Submission = \{hwk1\}$
$Grade = \{A, A-, B+, B\}$
$Course = \{c0, c1\}$
$roster = (\langle c0, Harry\rangle, \langle c1, Harry\rangle, \langle c1, Meg\rangle)$
$work = \{\langle c1, Harry, hwk1\rangle\}$
$gradebook = \{\langle c1, Harry, hwk1, A\text{-}\rangle\}$

Figure 2.8: A valid model of a specification.

The Alloy semantics defines a set of models for the signatures and facts. Operators over sets and relations have their usual semantics: + (union), & (intersection), **in** (subset), $\rightarrow$ (tupling), and . (join). The relations in Figure 2.8 constitute a valid model under the Alloy semantics.[2] All models of a specification are, by definition, consistent with its signatures and facts. A model of a predicate also associates each predicate parameter with an atom in the model such that the predicate body holds. The above set of relations models the *Enroll* predicate under bindings

$$c = c0,\ c' = c1, \text{ and } sNew = Meg.$$

A model may include tuples beyond those required to satisfy a predicate: the *Enroll* predicate does not constrain the *work* relation for pre-existing students, so the appearance of tuple $\langle c1, Harry, hwk1\rangle$ in the *work* relation is semantically acceptable.

The relations shown do not model *SubmitForPair*. For example, under bindings

$$c = c0 \text{ and } c' = c1$$

the requirement

$$c'.gradebook = c.gradebook$$

fails because the gradebook starting from c' has one tuple while that starting from c has none. The requirement on *work* also fails. Similar inconsistencies contradict other possible bindings for c and c'.

[2]For readability, we use concrete atom names rather than Alloy's abstract ones.

2.3 Alchemy's Supported Alloy Subset

Alchemy supports most of the Alloy language, including all of our running example. We omit integers and integer operations, as well as Alloy's built-in support for ordinals (via the ordering module).

The bodies of predicates and facts are terms in the Alloy Kernel logic (the core forms of Alloy [Jac06, page 291]). We support the full Kernel restricted to universally-quantified formulas in portions of the theory. The following grammar reproduces the Kernel language from Jackson's book [Jac06] sans the expr = expr form in elemFormula:

expr ::= rel | var | none | expr binop expr | unop expr

binop ::= + | & | − | . | →

unop ::= ~ | ^

formula ::= elemFormula | compFormula | quantFormula

elemFormula ::= expr **in** expr

compFormula ::= **not** formula | formula **and** formula

quantFormula ::= **all** var : expr | formula

We assume

$$expr1 = expr2$$

has been rewritten into

$$(expr1 \textbf{ in } expr2) \textbf{ and } (expr2 \textbf{ in } expr1).$$

This is sound in Alloy (which exploits explicit = in its analysis framework [Jac06, page 292]).

The rest of the thesis uses the term *basic formula* for elemFormulas or their negations. A *universal formula* is one in which all quantifiers are universal once

the formula is converted to Prenex Normal Form (i.e., all quantifiers grouped at the uppermost level of the formula).

The signatures, predicates, and facts in Alloy specifications are relevant to our work; assertions (properties to verify) are not relevant as they have no semantic content from the perspective of execution. Alchemy targets Alloy specifications that model stateful software systems. We recognize such specifications through Alloy's standard idiom, which uses some signature to represent the "state" of the system; predicates modeling stateful operations consume atoms representing the current and next state. In our running example, *Course* is the state; each operation takes *Course*s c and c' as inputs. We assume a *designated state signature* (herein denoted *state*) from which all other signatures are reachable. We view facts as integrity constraints on system states, requiring each to quantify over at most one *state* variable.

Our formal model of an Alloy specification is as follows:

Definition 1 An *Alloy specification* is a tuple $\langle S, P, F, state \rangle$, where:

- S is a set of *signatures*. A signature specifies its type name T_S, a set of fields, and an optional cardinality constraint. Each field has a name, an optional cardinality constraint, and a type specification $T_1 \times \ldots \times T_k$, where each T_i is the type name associated with some signature. The valid cardinalities are **lone**, **some**, and **one**. Our running example defines type names *Submission*, *Course*, etc.; the fields are *roster*, etc.

- *state* is the type name of some signature in S.

- P is a set of *predicates*. A predicate has a header and a body. The header declares a set of variable names, each with an associated signature type name. The body is a quantFormula in which the only free variables are defined in the

header. Our model limits the types of variables in the headers to names of signatures rather than arbitrary expressions on signatures (as in full Alloy). We call a predicate *stateful* if its header has exactly two variables of type *state* that share the same name with and without a prime (e.g., c and c').

- F is a set of *facts*. A fact is a closed formula. We assume facts have at most one quantified variable of type *state* and that this variable is unprimed; this is consistent with our viewing them as state invariants.

There are other small restrictions (Section 3.2.1) in our supported syntax, but these do not impact expressive power. Our signature definition diverges slightly from Alloy's in not including signature constraints in the model of the signatures, but most signature constraints, such as one signature being a subset of another, can be represented as universal facts. The exceptions are the **some** and **one** constraints, which our model captures as explicit cardinality constraints. We can express subtyping relationships between signatures as facts. We also restrict the type specifications in predicate headers to names of signatures, rather than permit arbitrary relational expressions. Richer parameter types can, however, be expressed as pre-conditions within the predicate body.

2.4 An Imperative Semantics

To see the main difference between Alloy's semantics and an imperative one, consider the *roster* relation in an Alloy model of a predicate. In the Alloy model from Section 2, *roster* contains tuples for both *c0* and *c1*; intuitively, these resemble timestamps where *c0* occurs before *c1*. An imperative program implementing operations would instead maintain a single (current) *Course* as a set of database tables

and update the *roster* table over time. In other words, an imperative program for this specification might have a *Course* named *theory* and include a table

$$roster = (\langle theory,\ Harry \rangle)$$

which, after enrolling *Meg*, changes to:

$$roster = (\langle theory,\ Harry \rangle,$$
$$\langle theory,\ Meg \rangle)$$

Our semantics represents imperative programs as transition systems over database instances. Instances of a given Alloy specification are over a database schema derived from its signatures and relations. Our semantics differs from Alloy's in modifying a database over time, whereas Alloy co-mingles all these database instances in a single relation. This has important consequences, as we discuss in Section 3.3. The rest of this section derives a database schema from an Alloy specification, then shows how to interpret predicates and facts relative to transitions over instances.

2.4.1 Database Schemas

Database schemas arise naturally from Alloy specifications. Each signature defines a unary relation over atoms. Each signature field defines a relation from atoms in that signature to the remaining elements in the field's specification. Our schemas use the same mapping from specifications to relations as in the Alloy semantics.

Definition 2 Let $A = \langle S, P, F, state \rangle$ be an Alloy specification. The *database schema* for A contains the following relations for each signature s in S, where T_s is the type name for s:

- a unary relation named T_s
- for every field $\langle D, c, T_1 \times \ldots \times T_k \rangle$ in s, a relation $D \subseteq T_s \times T_1 \times \ldots \times T_k$ with cardinality c.

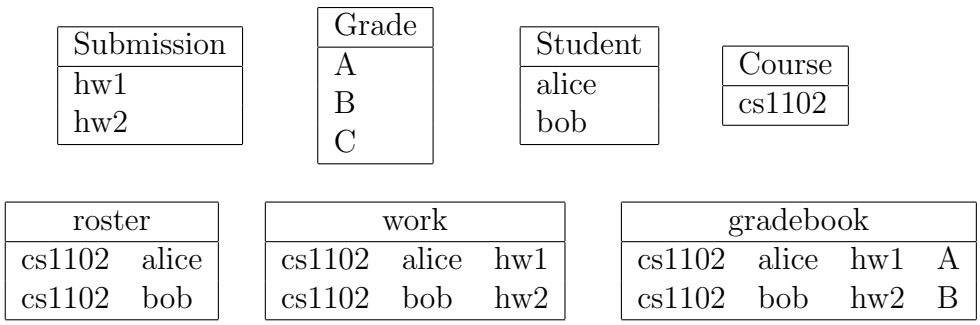

Figure 2.9: A schema instance I of the gradebook specification.

The distinguished *state* relation is restricted to only one atom (representing the current database state). An *instance* of the schema is any set of actual relations that conforms to the types in the schema. Instances must respect the cardinality constraints on signatures and fields: **one** allows only one tuple in a relation, **lone** allows at most one tuple in a relation, and **some** requires at least one tuple in a relation.

These cardinality interpretations are consistent with Alloy semantics. This definition differs from the Alloy semantics in only one detail: the restriction of the *state* relation to a single atom. This restriction lets us maintain only one active database instance while executing a specification, just as a programmer would expect.

Example 1 Given the gradebook specification in Figure 2.7, we can consider two example instances of the corresponding schema, shown in Figures 2.9 and 2.10. In both figures, *Course* is the distinguished state relation. In Figure 2.9, the two students have been assigned separate work and have been graded. In Figure 2.10, both students have been paired together to work on an assignment *hw3*, but haven't yet been graded.

Submission
hw1
hw2
hw3

Grade
A
B
C

Student
alice
bob

Course
cs1102

roster	
cs1102	alice
cs1102	bob

work		
cs1102	alice	hw1
cs1102	bob	hw2
cs1102	alice	hw3
cs1102	bob	hw3

gradebook			
cs1102	alice	hw1	A
cs1102	bob	hw2	B

Figure 2.10: Another schema instance I'.

2.4.2 Transition Systems over Instances

Each transition between instances in our imperative model arises from the execution of one stateful function corresponding to an Alloy predicate. Our semantics must therefore define when a predicate induces a transition from database instance I (the *pre-state*) to database instance I' (the *post-state*).

The key to this is deciding in which state to interpret a subexpression. Limiting individual identifiers to just the pre- or post-state is overly restrictive. For instance, *Enroll* contains

$$c'.roster = c.roster + sNew$$

A literal reading of primes would interpret c' in the post-state and both uses of *roster* in the pre-state. The *roster* relation in the pre-state, however, wouldn't include tuples that get introduced only in the post-state. It seems clear that the entire expression $c'.roster$ must be interpreted in the post-state. The right side of the equation, however, has one expr that appears to be from the pre-state ($c.roster$) and another from the post-state ($sNew$, the new student who should not be in the pre-state). This example shows that we must lift "priming" beyond individual variables, but without pulling expressions that are clearly in the pre-state into the

post-state.

Our semantics allocates expressions to the pre-state or post-state using a simple criterion: an expr is interpreted in the post-state iff it contains a primed variable. We call these *primed expressions*. In the body of *Enroll*, only $c'.roster$ is a (maximal) primed expression (and hence interpreted in the post-state). For each variable denoting a new atom (such as *sNew*), we augment the pre-state with a new atom; this lets us interpret $c.roster + sNew$ in the (extended) pre-state. We will use a naming convention (suffix *New*) to distinguish new variables (akin to using primes as a naming convention on next states). Treating new variables specially, rather than as post-state variables, yields a clean metric for determining whether a formula reflects an update versus a post-condition. We discuss this issue in more detail in Section 3.2.1.

Thus, our semantics distinguishes between three classes of identifiers: primed (such as c'), new (such as *sNew*), and unprimed (such as *s1* in *SubmitForPair*). Since both primed and unprimed expressions may include the *New* variables, we include these variables in each of the pre- and post-states when interpreting predicate bodies. We do not, however, include them in the pre-state when interpreting facts.

The rest of this section simply formalizes the prose above. Our definition covers the introduction of new variables, the allocation of exprs to the pre- and post-states, and the handling of facts. As the latter are intended to capture state invariants, we expect them to hold in every state. We assume that the database is initialized with atoms and relations that satisfy the facts.

Definition 3 Let $A = \langle S, P, F, state \rangle$ be an Alloy specification and let I and I' be instances of the database schema for A. Let $p = \langle H, B \rangle$ be a stateful predicate in A (where H is the header and B the body). Let H^- be the subset of H that excludes the variables of type *state*. Let E (the parameter environment) bind every non-

new variable in H^- to some atom in I of the corresponding type for that variable. E also binds all variables of type *state* to the unique atom in the *state* relation. $(I, I') \models_E \langle p, F \rangle$ iff the following conditions hold:

1. There exists a mapping E_{new} from every new variable new_v of type T_v in H^- to an atom new_{v_m} in the relation for T_v in I' but not in relation T_v in I. With the exception of the atoms in the co-domain of E_{new}, all relations corresponding to signatures have the same atoms in I and I'.

2. Let I^+ and I'^+ extend I and I', respectively, with the new atoms in E_{new}. B evaluates to true (under the standard semantics for boolean, relational, and set-theoretic operators) when every maximal non-primed expr is interpreted in I^+, every maximal primed expr is interpreted in I'^+, and every identifier takes its value from $E \cup E_{new}$.

3. The facts F are true in both I and I'.

The definition of E ensures that there is only one state atom, no matter how many *state* variables alias it. Condition 3 uses our assumption that facts are invariants on individual states. If facts were allowed to have more than one *state* variable, they would end up bound to the same atom as there is only one atom for the *state* in the imperative model.

Example 2 We briefly consider a transition motivated by:

$$\mathit{SubmitForPair}(c,\ c',\ \mathit{alice},\ \mathit{bob},\ \mathit{hw3}).$$

We refer back to the instances I and I' from Figures 2.9 and 2.10. We formally define the header H for our *SubmitForPair* predicate to be:

$$H = \{c : \mathit{Course},\ c' : \mathit{Course},\ \mathit{s1} : \mathit{Student},\ \mathit{s2} : \mathit{Student},\ \mathit{bNew} : \mathit{Submission}\}$$

and, after stripping out the state variables, we define H^-:

$$H^- = \{s1 : Student,\ s2 : Student,\ bNew : Submission\}$$

Then the parameter environments E and E_{new}:

$$E = \{\ s1 : Alice,\ s2 : Bob\ \}$$

$$E_{new} = \{\ bNew : hw3\ \}$$

give us enough to satisfy the definition. One thing to note is that E doesn't need to explicitly bind for the stateful variables c or c', since E will, by definition, map those identifiers to the unique atom *cs1102*.

Given those mappings, the necessary conditions hold and $(I, I') \models_E \langle p, F \rangle$.

Chapter 3

Alchemy: Interpreting Alloy Imperatively

Alchemy is designed to reduce the effort in automatically creating the database back-end from an Alloy specification. Alchemy compiles stateful predicates into functions that implement those predicates according to our imperative semantics. These functions insert and delete tuples into tables corresponding to the relations in the specification's database schema. We first illustrate Alchemy's desired behavior, then present the algorithm underlying Alchemy.

3.1 Desired Alchemy Behavior

Given the gradebook specification from Figure 2.7, Alchemy should create a database table for each relation (e.g., *Submission*, *roster*), a function for each predicate (e.g., *Enroll*), and a function for creating new elements of each atomic signature (e.g., *CreateSubmission*).

We illustrate our expectations of Alchemy's features through a sample interaction

using these generated functions. We should be able to create a course with two students using the following command sequence:

$$cs311 = CreateCourse(\texttt{"cs311"});$$
$$pete = CreateStudent(\texttt{"Pete"});$$
$$caitlin = CreateStudent(\texttt{"Caitlin"});$$
$$Enroll(cs311, pete);$$
$$Enroll(cs311, caitlin)$$

Note that the *Enroll* function takes only one course, not two (unlike the original Alloy predicate), since the implementation must maintain only a single set of tables over time. The second course parameter in the predicate corresponds to the resulting updated table. Executing the *Enroll* function must add the pairs ⟨"cs311", "Pete"⟩ and ⟨"cs311", "Caitlin"⟩ to the *roster* table. The second clause of the *Enroll* specification should guarantee that the *work* table will not have entries for either student. This clause is necessary in Alloy, which is free to add arbitrary tuples that don't violate stated constraints. Because we want Alchemy not to add such tuples, the clause is unnecessary; instead, Alchemy should enforce the constraint by removing any tuples that fail this condition.

Next, we'd like to be able to submit a new homework for "Pete" and "Caitlin":

$$hwk1 = CreateSubmission(\texttt{"hwk1"});$$
$$SubmitForPair(cs311, pete, caitlin, hwk1)$$

The implementation of *SubmitForPair* should be straightforward relative to the specification. We expect it to treat the first clause in the specification as a precondition by terminating the computation with an error if the clause is false in the database at the start of the function execution. Next, it should add the *work* tuples required in the second (update) clause. Finally, it must check that the *gradebook* table is unchanged, as required by the third clause.

Assigning a grade illustrates our expectations of Alchemy repair feature:

$$gradeA = CreateGrade(\texttt{"A"});$$

$$AssignGrade(cs311, pete, hwk1, gradeA)$$

AssignGrade should insert a tuple into the *gradebook* relation according to the first clause, and must check that the roster is unchanged according to the second. If execution were to stop here, however, the resulting tables would contradict the *SameGradeForPair* invariant (which requires "Caitlin" to receive the same grade on the joint assignment). Alchemy thus must attempt to repair the database to satisfy both the predicate body and the fact. It should determine that adding the tuple ⟨"cs311", "Caitlin", "hwk1", "A"⟩ to *gradebook* achieves this, and execute this command automatically. The fact therefore should hold of the database when the *SubmitForPair* function returns. If there is no way to repair the database to respect both the predicate and the fact, Alchemy should raise an exception. This could happen, for example, if the first clause in *AssignGrade* used = instead of **in** (in this case, adding the repairing tuple would violate the =).

Automatic repair supports the lightweight formal methods philosophy. One could require that all predicate specifications were written to preserve all facts (in this case, by augmenting *AssignGrade* to add database tuples for all students on the same assignment). Such fully-specified predicates can get rather complicated, however, sometimes to the point of obscuring the essence of a predicate. Alloy's use of facts to constrain possibly-underspecified predicates offer a powerful lightweight modelling tool. Database repair is fundamental for carrying that power into synthesized implementations. Alchemy must preserve all facts as database invariants when its functions terminate without exceptions.

3.2 The Algorithm

The semantics admits many possible functions for each predicate. For example, the *Drop* predicate from Figure 2.7 has a natural implementation: delete student s as well as all associated information about that student. However, there's an alternative implementation, one that's allowed by the specification but is probably not what was intended: wipe out the entire gradebook altogether.

Our compilation algorithm must choose a function that implements a predicate body. Furthermore, the back-end should automatically maintain integrity constraints that are encoded as system invariants in the Alloy model, so as to not violate the facts.

Rather than attempt to both implement predicates and preserve facts simultaneously, we employ a two-phase algorithm. The first phase generates insert and delete commands to implement the body of the predicate. The second phase generates additional commands that *repair* the database to restore facts violated during the first phase. The algorithm backtracks to find repairs or, in the worst case, even fresh implementations that satisfy both the predicate and the facts. This separation into phases has proven extremely valuable. It supports a method for ensuring non-interference between repair and implementation (which in turn guarantees termination). In addition, each phase can exploit a different normal form for formulas. We explain these details after presenting the algorithm.

3.2.1 Generating Commands

Generating commands to implement predicate bodies requires several key design decisions, such as which formulas should yield commands at all, whether to implement a formula using insertion or deletion, and which database tables to edit. The

decisions affect not only Alchemy's theoretical foundations, but also its usability. Alloy users employ certain idioms and make certain assumptions about what specifications entail. The models that Alloy generates for specifications can surprise even seasoned Alloy users. While this is acceptable from a model-exploration tool, such surprises are generally undesirable in imperative code. Our design decisions try to strike a balance between making sense to Alloy users and resting on sound design principles.

Which Formulas Yield Commands

Alloy captures different sorts of requirements on the pre- and post-states using the same set of operators. In the *AssignGrade* predicate in Figure 2.7, for example, the first expression specifies an update to the *gradebook* relation, whereas the second is a constraint to not change the *roster* relation. The latter is a *framing condition*, which limits the scope of changes. Other expressions capture *pre-conditions* (the first clause of *SubmitForPair*) or *post-conditions* (the second clause of *Enroll*). We distinguish among updates, framing conditions, pre-conditions, and post-conditions using syntactic criteria.[1] Only updates are compiled into commands. The rest become guards that abort predicate execution and roll back to the pre-state if violated.

Our criteria classify basic formulas (outermost terms that encompass the set-theoretic and relational operators). Given a formula (e_1 **in** e_2) or (e_1 **not in** e_2), we classify based on patterns of primes and similarity between e_1 and e_2:

[1] We could distinguish these by other means, such as adding explicit annotations to Alloy. Different techniques would change some of the details of how we generate commands. The high-level algorithms for predicate execution and repair, however, would not be adversely affected.

neither e_1 nor e_2 primed	pre-condition
e_1 and e_2 both primed	post-condition
e_1 identical to e_2 sans priming	framing condition
else	update

The first two align Alloy idioms with the theory: if primes denote the post-state, then prime-free formulas should not explore the post-state (an analogous argument covers the pre-state). The characterization of framing conditions prevents these formulas from becoming no-ops (as they otherwise suggest an update involving no change). The remaining formulas become updates that must be decomposed into specific insertions and deletions.

Consequently, only formulas that use the primed variable for the next state are recognized as updates. Imagine that we extended our example system to store the date of enrollment in each student object. The *Enroll* predicate might require a statement like $sNew.date = today$. Our criteria would mark this as a pre-condition rather than an update. The equivalent statement $(c'.roster \ \& \ sNew).date = today$ captures the intent within our criteria.

The chart also justifies our *New* naming idiom. If we had reused the priming idiom for new atoms (calling the new student s'), then the expression $c.roster + s'$ would become a primed expression. This in turn would obscure that $c'.roster = c.roster + s'$ is an update rather than a post-condition. Altering the scope of prime lifting is an option, but finding a coherent definition that also supports set operations nested within tupling and joins has proven difficult.

Whether to Insert or Delete

Updates have one of four forms: (e **in** f'), (e' **in** f), (e **not in** f'), and (e' **not in** f), where e and f are each exprs. Following the convention that primes denote the post-

state, our algorithm chooses to insert or delete as needed to have the change affect the primed side. Consider (e **in** f'): we could make this true by deleting from e or inserting into f. We choose the latter since f bears the prime. By similar reasoning, (e' **not in** f) also yields insertions, while the other two forms yield deletions.

What and Where to Insert or Delete

The most subtle decisions lie in determining which relations to edit when executing a command. Given the expression $c'.roster = c.roster + sNew$, we chose (in Section 2.4.2) to insert into $c'.roster$. The inserted tuples therefore should be in the set computed by expression $c'.roster$ in the post-state. We could do this by editing c', *roster*, or both.

A naïve reading of the primed-variable idiom suggests editing only c'. The imperative semantics, however, cannot realistically implement this strict reading. The Alloy semantics maps c and c' to atoms; the portion of the model reachable from each atom captures the overall pre- and post-states. Relations (such as *roster*) *appear* to change because different portions of them are reachable from the two atoms. The imperative version, however, doesn't define atoms for each possible state. Even if it did (which would require an a priori finite bound on the number of invocations of API functions or a garbage collection mechanism), storing each possible state in the database would be grossly inefficient. A more practical imperative approach would have a single *Course* object and modify the *roster* table to implement the predicate (as described at the start of Section 2.4). This approach is consistent with interpreting join like object navigation: the relation modified is a component of the primed state object. This pun between relational- and object-notation is a design feature of Alloy, yet one that has interesting consequences in the context of this project (see Section 3.3).

void *GenCommands*(*fmla*, *unprimed-db*, *primed-db*)
E = if e primed then *primed-db*(e) else *unprimed-db*(e)
F = if f primed then *primed-db*(f) else *unprimed-db*(f)

fmla	Commands
e **in** f'	insert all tuples in $E - F$ into F
e' **in** f	delete all tuples in $E - F$ from E
e **not in** f'	if $E \subseteq F$ delete some tuple in E from F
e' **not in** f	if $E \subseteq F$ insert some tuple not in F into E

Figure 3.1: Command generation.

In general, our algorithm may modify any relation mentioned in a primed expr when performing an update. It first computes the tuples that achieve an update, then decomposes commands on those tuples into commands on specific relations. The tuples and high-level commands are computed according to the chart in Figure 3.1. Because multiple parts of our algorithm use this table, we parameterize it over the formula and databases in which to compute the unprimed and primed expressions.

Command generation fails if there is no tuple to insert or delete in the third and fourth rows of Figure 3.1. Many commands could implement each formula. Removing *all* tuples from f' satisfies (e **not in** f'), for example, but is almost certainly not what the API user intended. The *Drop* predicate in Figure 2.7 would ideally remove only the indicated student. The chart attempts to minimize the changes made during an update. Repair may, however, add or remove other tuples; Section 3.2.3 discusses this in detail.

The third and fourth rows introduce non-determinism in the choice of tuples. In practice, framing conditions, post-conditions, and facts may constrain these cases to deterministic choices. Our current algorithm accounts for these constraints in the second (repair) phase. In the fourth row, when choosing tuples to insert, we use only atoms that already exist in the database. Our algorithm only creates new atoms

void *insertTuple*(t: $T_1 \times \cdots \times T_n$, e: expr) {
  match e
    [atom a: if $a \neq t$ then FAIL]
    [relation r: poststate$_r$:= poststate$_r$ + t]
    [none: FAIL]
    [e_1 + e_2: choose some e_i ; *insertTuple*(t, e_i)]
    [e_1 & e_2: *insertTuple*(t, e_1) ; *insertTuple*(t, e_2)]
    [~e: *insertTuple*(~t, e)]
    [$e_1 \rightarrow e_2$:
      let $t = t_1 \rightarrow t_2$ where t_i matches type of e_i
        insertTuple(t_1, e_1) ; *insertTuple*(t_2, e_2)]
    [$e_1 - e_2$: if t is not in e_2
          then *insertTuple*(t, e_1) else FAIL]
    [e_1 . e_2:
      let T be the common sig-type that joins e_1 and e_2
        if T is the type of e_1 then
          for some a in e_1, *insertTuple*($a \rightarrow t$, e_2)
        elseif T is the type of e_2 then
          for some a in e_2, *insertTuple*($t \rightarrow a$, e_1)
        else let a be any element of T
                $t_1 = s_1 \rightarrow a$
                $t_2 = a \rightarrow s_2$ such that t_1 . $t_2 = t$
          insertTuple(t_1, e_1) ; *insertTuple*(t_2, e_2)]
    [(e_1)^: *insertTuple*(t, e_1)]

Figure 3.2: Inserting a tuple into an expression.

when executing predicates with parameters that follow the *New* naming convention.

Figures 3.2 and 3.3 decompose insertions and deletions on relational expressions into similar commands on individual relations. Some operations have multiple valid implementations, owing to a choice of relations to manipulate. We choose between these non-deterministically, backtracking as needed if a choice does not lead to a valid implementation that can be repaired to satisfy the facts. The algorithms use the notation poststate$_r$ to denote relation r in the post-state.

The decision to FAIL in the $e_1 - e_2$ case of Figures 3.2 and 3.3 reflects a design decision on our part. We could handle the case where t is in e_2 by adding t to e_1

```
void deleteTuple(t : T_1 × ··· × T_n, e: expr) {
  match e
    [atom a: if a = t then FAIL]
    [relation r: poststate_r := poststate_r − t]
    [none: FAIL]
    [e_1 + e_2: deleteTuple(t, e_1) ; deleteTuple(t, e_2)]
    [e_1 & e_2: choose some e_i ; deleteTuple(t, e_i) ]
    [~e: deleteTuple(~t, e) ]
    [e_1 → e_2:
       let t = t_1 → t_2 where t_i matches type of e_i
         choose some e_i ; deleteTuple (t_i, e_i)
    [e_1 − e_2: if t is not in e_2
                then deleteTuple(t,e_1) else FAIL]
    [e_1 . e_2:
       let T be the common sig-type that joins e_1 and e_2
         if T is the type of e_1 then
              foreach a in e_1, deleteTuple(a → t, e_2)
         elseif T is the type of e_2 then
              foreach a in e_2, deleteTuple(t → a, e_1)
         else foreach a in T such that for some s_1, s_2
                          t_1 = s_1 → a in e_1 and
                          t_2 = a → s_2 in e_2 and t = t_1 . t_2
                    choose some e_i ; deleteTuple(t_i, e_i)]
    [(e_1)^:  foreach (x, y_1), (y_1, y_2), ..., (y_n, y) such that
              t = (x, y) and each pair is in e_1
                choose some pair (y_i, y_{i+1})
                deleteTuple(y_i → y_{i+1}, e_1)]
```

Figure 3.3: Deleting a tuple from an expression.

and removing t from e_2. Implementing abstract insertion operations with concrete deletions, however, has implications for termination, as we discuss in Section 3.2.3. As a general rule, we prefer to use only insertion operations to implement insertions, and analogously for deletions.

Given:

pred $p(s, s' : \mathit{state}, v_1 : T_1, \ldots, v_j : T_j,$
$\mathit{new\text{-}vk} : T_k, \ldots, \mathit{new\text{-}vn} : T_n)$
$\{\ \forall \overline{X}\ .\ (\varphi_1 \vee \ldots \vee \varphi_m)\ \}$

Generate:

list(atom) $p(v_1 : T_1$,..., $v_j : T_j)$
  let $\mathit{newv}_i = \mathit{new\_atom}$ (*pre-state*, T_i) for $i \in k, \ldots, n$
  post-state = *pre-state*
  let V map params to args and *new*-vars to *new*-atoms
    iterate until fixpoint on *post-state*
      foreach binding B to identifiers in $\overline{X}$
        choose a disjunct φ_i
        if some pre-condition in $\varphi_i[V \cup B]$ false then FAIL
        else foreach update in $\varphi_i[V \cup B]$
          GenCommands(*update*, *pre-state*, *post-state*)
          // Figure 3.1
        if some post-condition in $\varphi_i[V \cup B]$ false then FAIL
        add framing conditions in $\varphi_i[V \cup B]$ to *Guards*
    repair-facts() // Figure 3.5
    if some formula in Guards false then FAIL
    if some cardinality constraint false then FAIL
    pre-state = *post-state*; return newv_k, ..., newv_n

Figure 3.4: Pseudocode for a compiled predicate. $\varphi[Z]$ denotes φ substituted with all bindings in Z.

3.2.2 Compiling Predicates

Figure 3.4 shows the pseudocode that implements a predicate. We treat predicates as transactions that rollback if they cannot be executed without violating their bodies or a fact. If a cardinality constraint fails, or if backtracking fails to produce a set of commands that satisfy both the predicate and the facts, then predicate execution fails. This induces rollback of the database state to the pre-state. The pseudocode assumes two variables: *pre-state* (for the database contents at the start of the transaction) and *Guards* (for the set of formulas to check before committing the transaction).

Our model of Alloy specifications assumed that the body of every predicate is a universal formula. In generating code, we convert each of these formulas into disjunctive normal form. Each predicate body therefore has the form

$$\forall(x_1 : r_1) \ldots \forall(x_n : r_n) \; . \; (\varphi_1 \vee ... \vee \varphi_k)$$

where each φ_i is a conjunction of formulas of the form (e **in** f) or (e **not in** f). Each φ_i may reference variables declared in the header of the predicate.

The API function produced for a predicate takes arguments for the predicate parameters other than the state variables (implicit in the implementation) and the *New* variables. The function creates atoms for the *New* variables before attempting any updates. The bindings of new atoms to *New* variables are added to the parameter bindings.

The algorithm chooses a disjunct to implement to satisfy the predicate. If any pre-condition in the disjunct is false, the choice fails and the algorithm backtracks to select another disjunct. The function then generates commands for each update (using Section 3.2.1) and applies them to the post-state. A failed post-condition

causes the algorithm to backtrack to other command choices or to another disjunct selection (if necessary). Framing conditions are accumulated as guards to check after the database has been repaired to account for the facts. This ordering allows framing conditions to cover the entire predicate implementation, as expected. A failure when checking a framing condition would backtrack into the repair algorithm. Cardinality constraints (arising from **one**, **lone** and **some** constraints) are also checked at the end by comparing the size of the relation in the post-state to the size required by the constraint.

Disjunctive-normal form is natural for implementing predicates because it keeps all the related pre-conditions, post-conditions, and updates together. This can be useful in terminating a search path early, as any conjunct with a failed pre-condition can be rejected in its entirety.

The algorithm reveals a subtlety regarding *New* variables. Intuitively, these variables should appear only in the post-state. The algorithm, however, uses the pre-state as the *unprimed-db* argument to *GenCommands*. There must be an atom for *sNew* in the *unprimed-db* in order to add the new student to the roster in the *Enroll* predicate. We therefore add the new atoms to both pre- and post-state.

Two questions arise about the algorithm's correctness relative to our imperative semantics. First, we have sequentialized the processing of updates. This suggests that the edits from implementing one command might affect the edits required for another. Our algorithm applies edits to the post-state but computes tuples in the pre-state, so such leakage does not occur. Iterating the computation until a fixpoint on the post-state ensures that the chosen edits are valid regardless of ordering. Second, our algorithm seems to assume that repair cannot violate the body of the predicate (since repair is not within the code to iterate until fixpoint). While this could be a problem in general, our repair algorithm operates under a restriction that

eliminates this issue; the next section addresses this in more detail.

Example 3 Let's apply the algorithm from Figure 3.4 on the *Enroll* predicate of the gradebook example. The use of equality in the first subformula is first desugared into two **in** forms, producing the body

$$c'.roster \textbf{ in } c.roster + sNew \textbf{ and}$$
$$c.roster + sNew \textbf{ in } c'.roster \textbf{ and}$$
$$\textbf{no } c'.work[sNew]$$

The first two subformulas are basic formulas that will be analyzed by the translation. However, the third primed subformula isn't basic, so is treated as a postcondition: at the end of the predicate's evaluation, all postconditions are checked, and if any are violated in the poststate, a FAIL occurs.

The first subformula is in the form e' **in** f, and so we delete all tuples in $c.roster - (c.roster + sNew)$ from $c'.roster$. Deleting a tuple t from $c.roster$ will recursively delete the tuple $c \rightarrow t$ from roster. However, since there are no tuples in $c.roster - (c.roster + sNew)$, the evaluation of this implementation is a no-op and will produce no changes to the poststate.

In contrast, the second subformula is in the form e **in** f'; we insert all tuples in $(c.roster + sNew) - c.roster$ into $c.roster$, and inserting a tuple t into $c.roster$ recursively reduces to an insertion of the tuple $c \rightarrow t$ into the *roster* relation. The imperative semantics of the predicate, then, is to insert the tuple $c \rightarrow sNew$ into *roster*.

3.2.3 Compiling Facts into Database Repairs

The repair phase takes a set of facts and a database instance and edits the database (if necessary and possible) so that it satisfies the facts. Figure 3.5 presents the

pseudocode. The algorithm only repairs universal formulas. Other facts are treated as guards that get checked after repair as shown in the predicate pseudocode in Figure 3.4.

The repair algorithm assumes that all universal facts are in conjunctive normal form. Distributing the quantifiers over the conjuncts yields a set of facts, each of the form

$$\forall \overline{X}.(\alpha_1 \wedge \cdots \wedge \alpha_k) \Rightarrow (\beta_1 \vee \cdots \vee \beta_h)$$

where each α_i and each β_j is an elemFormula $(e_1$ **in** $e_2)$. This form simply groups the positive and negative elemFormulas on either side of the implication operator. Either side of the implication could have no subterms, in which case the normal logical rules apply: if there are no α_i, the body is equivalent to $(\beta_1 \vee \cdots \vee \beta_h)$; if there are no β_j, it is equivalent to $\neg(\alpha_1 \wedge \cdots \wedge \alpha_k)$.

The algorithm repeatedly selects a fact and checks whether it is true in the post-state. If not, the algorithm must modify the database to nullify each witness to the failure. Falsifying any α_i or $\neg\beta_j$ nullifies a witness. Each α_i or β_j is of the same core form $(e$ **in** $f)$ used to generate commands for implementing predicates (Section 3.2.1). Once we decide whether to nullify using insertion or deletion (a decision discussed momentarily), we reuse the table in *GenCommands* (Figure 3.1) to generate the appropriate commands. Nullifying α_i via insertion uses row 1; nullifying α_i by deletion uses row 3. Nullifying β_j follows row 2 (insertion) or 4 (deletion). As when generating commands to execute predicates, command choices may induce backtracking should a choice lead to an inconsistent database.

This algorithm raises several potential concerns:

- **Termination:** Repairing one fact might break another. In theory, two facts could iteratively undo each others' repairs ad-infinitum.

void *repair-facts* ()
  iterate until fixpoint on *post-state*
    foreach fact $F_i = \forall \overline{X} \,.\, (a_1 \wedge \ldots \wedge a_k) \Rightarrow (b_1 \vee \ldots \vee b_h)$
      foreach solution S to $\exists \overline{X} \,.\, a_1 \wedge \ldots \wedge a_k \wedge \neg b_1 \wedge \ldots \wedge \neg b_h$
        instantiate fact body with bindings from S
        choose some a_i or b_j
        GenCommands($\neg$*choice*, *post-state*, *post-state*)
        // Figure 3.1
    insert all non-universal facts into *Guards*

Figure 3.5: Algorithm to repair the database.

- **Correctness:** Repairing a fact might undo the effect of the predicate we were attempting to execute.

- **Efficiency:** In the worst case, we could iterate over every possible combination of insertions and deletions over every combination of atoms in the database.

- **Predictability:** The repair algorithm might modify some relation that was not in a primed maximal expr within the predicate body, thus affecting the database in unexpected ways from the API user's perspective.

Predictability isn't a problem if the user considers the facts as well as the predicate body. Our algorithm modifies only those relations that are mentioned in the facts or predicate being executed. The absence of sufficient post-conditions and framing conditions could result in undesirable implementations, but this is inherent to underspecification, not an artifact of Alchemy. Our implementation ameliorates underspecification to a small extent, and intelligent heuristics for this are an interesting topic for future work.

Following a simple principle mitigates the first three problems: repair insertions with other insertions and deletions with other deletions. Consider the *AssignGrade* predicate from Figure 2.7. Executing this predicate assigns a grade to the given

student, but not to her partner. The *SameGradeForPair* fact is intended to assign the same grade to her partner as well. From a semantic perspective, however, we could restore the fact by removing the partner from the course (prevented by the framing condition) or by removing the submission from each students' work. Repairing insertions by insertions blocks the latter option and results in the desired repair that adds the grade to the other student.

This principle guarantees that repair will terminate, since there are at most a finite number of insertions involving existing database atoms. It improves the efficiency of repair by restricting the search space of commands to consider. This is an extremely useful consequence of our two-phase algorithm. It also identifies cases in which repair will not undo the effect of a predicate: commands can never undo the effect of commands of the same type (insertion or deletion).

Applying the principle at the level of an entire predicate, however, is often too restrictive. For termination, never inserting to and deleting from any individual relation suffices. We call a predicate execution *homogeneous* if it doesn't insert and delete from the same relation. In practice, predicates often have homogeneous executions even though *syntactically* they appear to always mix insertions and deletions. When we expand = into two **in** expressions, any expression using = yields one form that creates insertions and another that creates deletions (by the table in *GenCommands*). At run-time, however, one of these two forms often reduces to a no-op because one side is a subset of the other (as in the body of the *Enroll* predicate). A general *syntactic* characterization of homogeneity is left for future work.

Example 4 We consider the *SameGradeForPair* fact as used in the repair algorithm. The body is first translated into conjunctive normal form containing two conjuncts:

$$b \textbf{ in } (c.work[s1] \ \& \ c.work[s2]) \rightarrow$$
$$c.gradebook[s1][b] \textbf{ in } c.gradebook[s2][b]$$

and

$$b \textbf{ in } (c.work[s1] \ \& \ c.work[s2]) \rightarrow$$
$$c.gradebook[s2][b] \textbf{ in } c.gradebook[s1][b]$$

We consider the situation where assignment b has been graded for a pair of students *s1* and *s2*, but where an entry in the gradebook for *s1* has been inserted, but not a corresponding one for *s2*.

Each conjunct is checked, and although the second conjunct is vacuously true, the first is not, and remedial action needs to be taken. We have two options:

- Nullify the fact $\alpha_1 \equiv b \textbf{ in } (c.work[s1] \ \& \ c.work[s2])$
- Nullify the fact $\neg\beta_1 \equiv c.gradebook[s1][b] \textbf{ not in } c.gradebook[s2][b]$

By the homogeneity assumption, we only consider repairs that act through insertion. We tentatively commit to the first option. If we try to nullify α_1, then some tuple outside of $c.work[s1] \ \& \ c.work[s2]$ is inserted into b. However, because b is an atomic variable, this option will fail and trigger a backtracking.

We then consider the second alternative to nullify $\neg\beta_1$. We do so by inserting all the tuples in $c.gradebook[s1][b]$ into $c.gradebook[s2][b]$. This effectively forces the second student to have the new gradebook entry of the first. The repair procedure then iterates, finds that all facts are satisfied, and finally terminates.

3.3 Discussion: Relating the Alloy and Imperative Semantics

The astute reader will have noticed that we do not formally link the meaning of predicates in Alloy to their implementation in Alchemy. One possibility is to link statements about the implementation with assertions validated by the Alloy Analyzer. The Analyzer, however, validates these only over domains with bounds, which in turn are usually chosen for computational tractability. Thus, we must instead focus on meaning according to the Alloy semantics. There are two difficulties with establishing this link.

Mixing Models One basic property one might hope to preserve is satisfiability. Therefore, consider the following claim: every satisfiable predicate is implementable imperatively, and vice versa. If we could establish this property, we might be able to proceed to stronger statements that link the proof of satisfaction of a predicate to the behavior of the implementation.

Unfortunately, we cannot establish even this claim due to subtle yet significant differences between the relational and imperative semantics. The following fragment shows a satisfiable stateful predicate over *state* signature A that has no implementation in our semantics:

sig A { $r : B$ }

sig B {}

fact { **one** r }

// this is satisfiable

pred *change_r*(a, $a' : A$, *bNew:* B) {$a'.r = a.r + bNew$}

The fact on r cascades to allow only one element of signature A. Because of the fact, $a.r$ and $a'.r$ can't refer to different elements. Thus, *bNew* must be in $a.r$,

but our semantics introduces a new atom for *bNew*. This model therefore cannot be implemented under our semantics. (With larger examples we can remove this dependence on new variables, so that is not at the heart of the problem.)

How about in the other direction: if a specification is implementable, is it satisfiable in Alloy? Sadly, no, as another simple example demonstrates (relative to the same signatures and fact):

$$\text{// this is not satisfiable}$$
$$\textbf{pred } \mathit{change\_r}(a,\ a' : A)\ \{a'.r\ \mathrel{!}= a.r\}$$

Since $a.r$ and $a'.r$ can't refer to different elements, the predicate is not satisfiable. The implementation, however, is straightforward: the table for r has one row with different values in the pre- and post-states.

These tiny examples point to a general problem. As we have discussed in Section 2, an Alloy model includes *all* the "states" at once, whereas the imperative implementation examines only one state at a time. A predicate that fails to account for this difference—and, in particular, for the conflation of all states into a single model in Alloy—runs the risk of being satisfied by Alloy but not by the imperative semantics, or vice versa.

This does, however, raise a conjecture. It may be possible to impose a discipline on the use of predicates in assertions that demands they always account for the relationship between the Alloy states and what is reachable from them. It may even be possible to automatically augment predicates to impose this expectation. We believe that such augmented predicates are essential to the design of a lightweight Alloy-esque modeling language that is faithful to cross-state assertions with an imperative meaning, not just to invariants.

```
sig Person {
    friends : set Person
}

sig SocNetwork {
    members : set Person
}

pred befriend(s, s' : SocNetwork,
              p1, p2 : Person) {
    p1 != p2
    p2 not in s.members[p1 → p1].friends
    s'.members[p1 → p1].friends = s.members[p1 → p1].friends + p2
}
```

Figure 3.6: An unsatisfiable Alloy specification.

The Prime Suspect A related problem is the meaning of primes in conjunction with joins. As the multiple discussions about the *roster* relation in Section 2.4 illustrate, the . operator—a relational join that evokes object dereference—may be confusing in the presence of state. In Section 2.4, for instance, the Alloy user writes $c'.roster = c.roster + sNew$ to update the *roster* relation. Technically, however, *roster* is the same relation on both sides of the equation; it is c' that projects a different portion of *roster* than c. To an object-oriented programmer, however, c remains the same (due to object identity); it is the *roster* field that changes. A "stateful Alloy" must reconcile these readings.

The root of the issue, the mismatch between primed-state specification and deep internal structure, can be deceptively difficult to see. As an example, we can consider a small system that maintains a social network of friends as shown in Figure 3.6. This social network consists of a set of people, and each person holds a set of friends. A *befriend* predicate takes an old social network and produces a new social network where one person has befriended the other.

Although this specification may look like a natural way to express the update, it's incorrect: it can't be satisfied by any Alloy model, even though there is a natural implementation defined by our imperative semantics. The simple stateful-primed idiom is insufficient when state is distributed among several signatures. When we use that idiom, although we can express a pre and post state of the social network, we can't express the statefulness of a Person in that system.

Such statefulness, however, is prevalent in any nontrivial object-oriented system. This reveals a fundamental mismatch that can occur between a specification writer's intent and the semantics provided by Alloy. It highlights a need for a "stateful alloy" to support the expression of state changes on all the relations of a specification, and not just the relations that fall beneath the stateful signature.

Chapter 4

Implementation

The implementation of Alchemy is written in PLT Scheme, a dialect of Scheme that supports pattern matching, higher-order programming, and linguistic abstraction. Alchemy consists of a compiler and a runtime; the compiler consumes Alloy specifications and produces code objects that represent implementations, while the runtime primarily consumes code objects and applies their operations on a database instance. Our implementation uses the Alloy parser as a front-end and a Postgres database back-end. With a little additional work, Alchemy could automatically generate a Web Service interface as well.

The implementation and algorithms differ in a few places:

- Alchemy does not generate command options to insert or delete into an atom. For example, inserting into $c.gradebook[s1][b]$ (from Figure 2.7) generates options to modify each of c, *gradebook*, *s1*, and b. All but the option for *gradebook* will fail immediately in Figure 3.2. This optimization significantly reduced case-explosion on some of our examples.
- Because some cardinality constraints require existential quantification, they do not fall under the aegis of our current repair algorithm. As a result, Alchemy

can only *check* cardinality; it does not repair it. Therefore, failure of a cardinality constraint leads to transaction failure, rather than backtracking.

We have not yet implemented two features. First, predicate implementations don't generate atoms for *New* parameters automatically; the API user must do this manually before invoking the function (as in Section 3.1). Second, all references to predicates must be inlined. Since predicates are first-order formulas, this does not limit expressiveness.

4.1 Modules

Alchemy is designed in several modules:

- basic expressions and environment support
- the core algorithm
- compiler
- runtime
- host language support

4.1.1 Basic Expressions and Environment Support

The expression and formula forms in Alloy are represented as AST trees in Alchemy; to get the appropriate ASTs, we reuse the Alloy 4 parser from the Alloy project, which allows us to take advantage of its internal typechecking and well-formedness checks. A tree walker processes the Alloy data structures and dumps out an abstract syntax tree, which we then process. The signatures in the AST are serialized out

to the compiler's output, which leaves the predicates and facts to be analyzed by Alchemy's other modules.

Other helper functions include predicates for checking a formula for basic form and transformations to normalize formulas into CNF/DNF. The environment support allows us to reflect changes to relations back onto the corresponding database tables.

4.1.2 Core Algorithm

The core algorithm layer implements the algorithms described in Chapter 3, and is implemented as a standard pattern matching against a formula's structure. Given a stateful predicate, the predicate's body is normalized into disjunctive normal form; each disjunct is then processed to produce an implementation of that disjunct.

The implementations of all the disjuncts are collected into a single structure to be interpreted at runtime. During runtime, one of the implementations is chosen nondeterministically to implement the predicate, using a backtracking package (described in Subsection 4.1.5) to support the nondeterminism.

We perform a similar analysis on the global facts of the model; its implementation drives the repair process, reusing the basic infrastructure used by the predicate code. Additional support code drives the iterative fixed-point computation of repairs, which runs the predicate and repair code until the relations stabilize.

Not all predicates in an Alloy specification are stateful. We check a predicate for statefulness by a simple syntactic heuristic: if the first two variables of a predicate use the same type, and the second variable's name is a primed version of the first, we treat the predicate as stateful. Otherwise, Alchemy ignores the predicate, which is safe under our assumption about inlining.

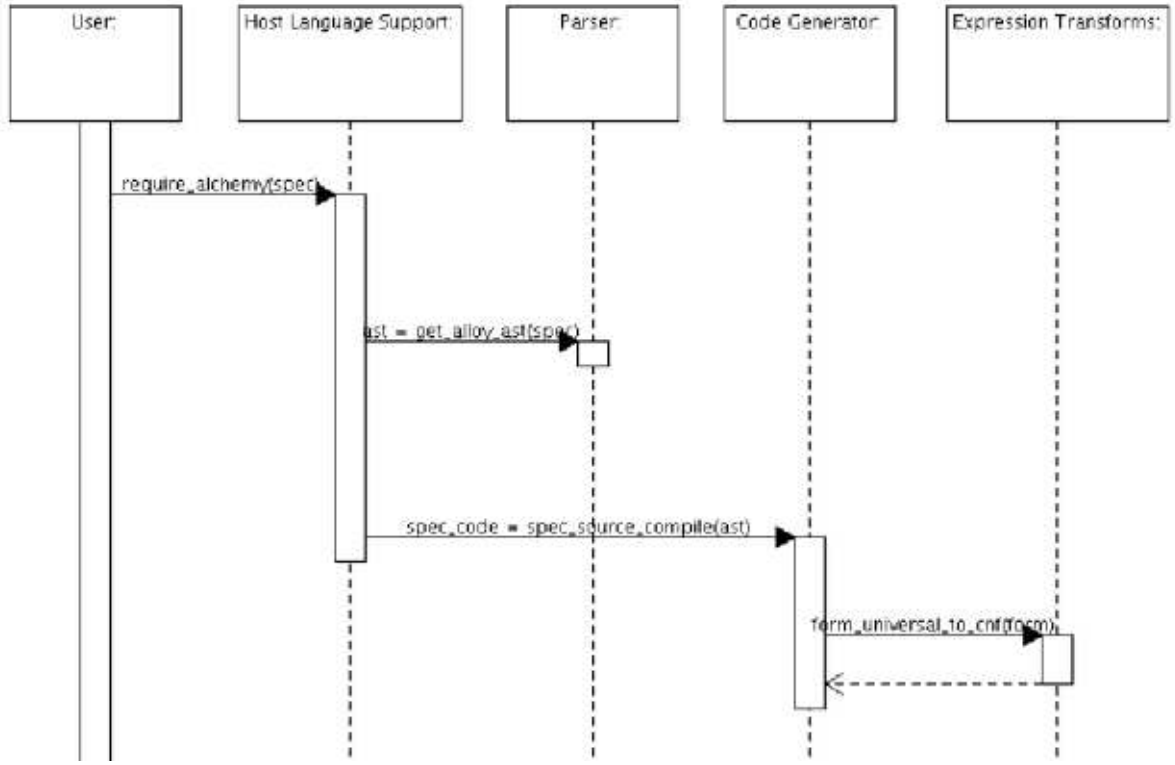

Figure 4.1: An example run of the compiler.

4.1.3 Compiler and Runtime

Alchemy consists of a compiler and runtime; the compiler preprocesses a specification, produces an implementation for each stateful predicate, and exposes that implementation for clients to use. These implementations depend on some additional library support that's provided by the Alchemy runtime; the runtime provides things like a simple database-access layer and a backtracking facility to handle the nondeterminism in a specification.

The sequence diagram in Figure 4.1 shows an example run of the Alchemy compiler. The user invokes the compiler by using the *require-alchemy* form. From there, an AST is extracted from the spec, and that AST is analyzed by the code generator. The code generator invokes all of the core algorithms mentioned in Section 4.1.2. Once the compiler processes the AST, it emits a specification code function, which

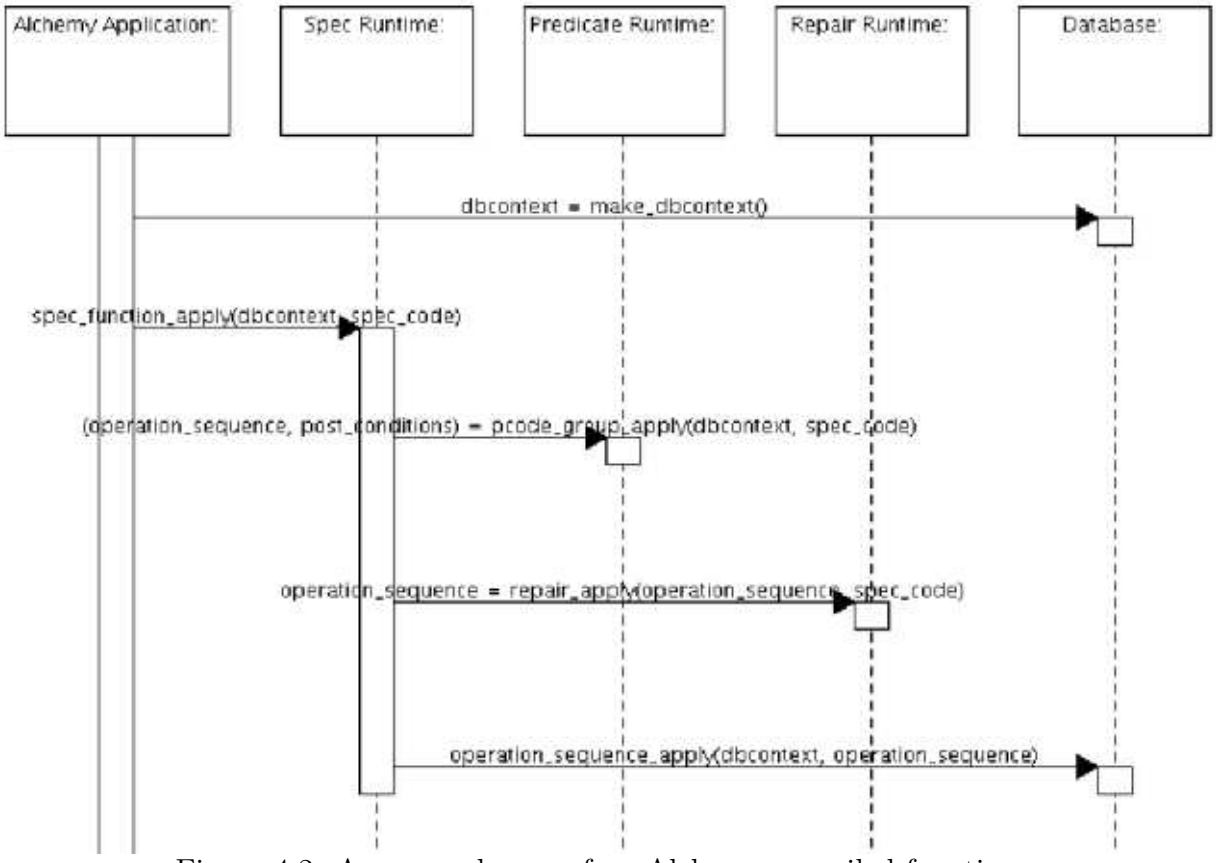

Figure 4.2: An example run of an Alchemy-compiled function.

is serialized into a user's source code to be later used during run-time.

The runtime libraries provide support to evaluate the implementations of the compiled stateful predicates. As shown in Figure 4.2, an implementation's evaluation will generate sequences of primitive operations, where a primitive operation is either an *insertion* or *deletion* of a table tuple. This operation sequence (*opseq*) is accumulated during a code evaluation but not immediately applied to the database: in the case of conflicts, such as when the homogeneity assumption is violated, we must backtrack. Once a fixed-point is reached without violating any of the predicate or repair facts, then the code object is fully evaluated, and the operations are then applied and committed to the database. Other helper functions in the runtime provide support for creating new atoms for the relations, automatically constructing

the tables that reflect the relations used in a specification, and providing functions that can attach additional data to an atom.

4.1.4 Host Language Support

The host language support provides syntactic sugar that exposes each Alchemy-compiled predicate as a function that can be called by the end user. We provide this kind of support within PLT Scheme by using a macro called *require-alchemy*, which invokes the Alchemy compiler as a pre-processor.

Since each implementation of a stateful predicate is represented as a data structure, that structure can be marshalled into source code: at compile time, the macro injects the Alchemy compiler's result into a client's source code, so that much of the analysis is done at pre-processor time. The macro automatically generates a function wrapper for each predicate; each wrapper's body calls out to the Alchemy runtime, setting up a nondeterministic context for evaluating the implementation.

Let's consider the gradebook specification in Figure 2.7 and see how we can write a PLT Scheme program that uses the specification as a library. Figure 4.3 shows an example program. The line

(*require-alchemy* "alchemy.ss")

is pre-processed at compile-time to invoke the Alchemy compiler, do the analysis of the given specification, and to include the compiler's output into the client program. It introduces wrapper functions for Enroll, SubmitForPair, and AssignGrade, each of which are exercised later in the program.

We then use the Alchemy-provided helper functions for creating atoms; the current version of Alchemy does not automatically create new atoms on its own. Finally, we use the API to assign grades, and look at the results of AssignGrade. The repair

```
#lang scheme
(require "alchemy.ss")

;; Compile the Alloy specification, get function
;; and relation bindings.
(require-alchemy "gradebook.als")

;; Create a new database connection.
(define db (new-dbcontext #:database "gradebook" #:user "dyoo"))

;; Create a few atoms.
(define student-1 (make-Student-Atom db "Andrew"))
(define student-2 (make-Student-Atom db "Bean"))
(define course (make-Course-Atom db "Classroom 1"))
(define hw (make-Asgmt-Atom db "HW"))
(define A-grade (make-Grade-Atom db "A"))
(define B-grade (make-Grade-Atom db "B"))

;; Enroll both students, make them partners, and give a grade to one of them.
(Enroll db course student-1)
(Enroll db course student-2)
(SubmitForPair db course student-1 student-2 hw)
(AssignGrade db course student-1 hw A-grade)

;; Finally, inspect the student roster.
(printf "Current course roster: ~s~n" (Course-roster db c))
;; Print out the grades
(for ([row (Course-gradebook db course)])
  (let-values ([(a-student a-work a-grade) (apply values row)])
    (printf "~s ~s ~s~n" a-student a-work a-grade)))
```

Figure 4.3: Use of an Alchemy-compiled library.

that's performed after each run of a stateful predicate will ensure that *student-2* gets the same grade as *student-1*.

4.1.5 Miscellaneous: `amb` Backtracking

Alchemy uses nondeterminism widely, both at compile and run-time, to explore the search space for an implementation. We use the `amb` library module [Far] to give us simple, depth-first backtracking through this search space.

One example of a use of compile-time nondeterminism is the treatment of an insertion into a union, $e1 + e2$, where either *e1* or *e2* may be the target for insertion. An example of nondeterminism at runtime is the command generation for the core formula e **not in** f', where we choose a tuple from the database instance.

In both cases, we cannot commit to a choice until run-time. Code generation choices are represented explicitly as a tree within a code object. During runtime, at any point of nondeterminism, `amb` records a choice among the possibilities, and in the event that we have to backtrack, `amb` reconsiders the most recent nondeterministic choice.

`amb` is particularly simple to write with, but provides no control to steer the search toward promising avenues; it is future work to consider a nondeterminism mechanism that navigates the search space more intelligently.

4.2 Database Mapping

Alchemy, for the most part, uses a fairly straightfoward mapping between relations and database tables. However, although atoms are relational values, we have chosen not to create a named unary table for each individual atom, given that the Alloy language provides no way to access an individual atom without going through a

signature. For that reason, atoms are handled as a special case in our mapping. Many of our design decisions, such as the representation of atomic values and the signature hierarchy, leaned toward ease of implementation rather than performance; future work may involve revisiting these database mapping decisions.

4.2.1 Atoms

There are two strategies to representing uninterpreted atom values:

1. Atoms can be represented as primary key values in the table that represents the base signature type of that atom.

2. Atoms can reside in a separate systemwide table, and signatures hold references to the atoms in that system table.

Although the first option is likely to perform better because it avoids the table joins that the second option may require, we've chosen the second option in our initial implementation because it was easier to implement. A system table called *alchemy:atom* holds all the atoms used in an Alchemy system, and an additional column allows auxiliary information to be attached to an atom.

One case that complicates the use of the first option is the presence of subset signatures. Subset signatures may include the atoms of other base signatures, in which case the atom may be redundantly duplicated in several places. Using references in the signature tables allows the proper sharing of atoms.

4.2.2 Signatures and Fields

Given an Alloy specification containing signature and field structure, the signatures are reflected as single-arity database tables. Fields are represented as multi-arity tables whose columns, too, are references to atoms.

In the following example,

sig A {}

sig B {

c : **set** A

}

we generate the schema:

create table A (*id integer* **not** *null primary key*);

create table B (*id integer* **not** *null primary key*);

create table c (*id0 integer* **not** *null*, *id1 integer* **not** *null*);

where the id columns are references to the atoms in the *alchemy:atom* system table.

Multiplicity constraints are currently treated outside of the database schema by becoming facts that are maintained as part of the repair algorithm. It is future work to incorporate those constraints directly into the database schema.

4.2.3 Signature Hierarchies

Alloy signatures can be arranged in an inheritance hierarchy. There are three mapping strategies listed in Ullman [GMUW01] from hierarchies to database tables:

1. Use a single table to hold all the instances of the subclasses.

2. Use multiple tables, one table per subclass, with information spread across several tables.

3. Use multiple tables, one table per subclass, with an entity's information consolidated in a single table, where duplicate attributes may be present across all subclasses.

We choose the second option, mapping a table to every relation in an Alloy specification. This fits closely with Alloy's model of the signature hierarchy. The

second and third options are equivalent in our context, because there is no auxiliary information associated with the fields of an Alloy signature. The first option could have also been used. Alchemy extracts the hierarchical constraints as facts, and the repair algorithm maintains the subset/superset relationship between parent and child signatures.

Chapter 5

Related Work

5.1 Synthesis

Software synthesis is an elusive goal, as Rich and Waters summarize [RW88]. Both Green [Gre69] and Waldinger and Lee [WL69] are generally credited with initiating this effort. Bates and Constable [BC85] discuss the relationship between constructive proofs and programs; this connection continues to be exploited in modern theorem provers that extract programs from proofs. Burstall and Darlington [BD77] instead define rules to transform specifications into programs, which Manna and Waldinger [MW80] combine with theorem-proving and induction. Some authors such as Smith [Smi85] have instead focused on the synthesis of particular types of *algorithms* rather than programs. Unlike our work, most of these approaches usually involve considerable human interaction, and have tended to be applied to pure functions that generally avoid any reference to state and mutation.

Executable UML [MB02] and other model-driven approaches attempt to proceed from specifications to programs, but with a significant difference in philosophy from ours: they tend to start with large, multi-modal specifications, which are rather

unlike lightweight specifications in the style of Alloy. This philosophical difference has practical consequences: Executable UML tends to be used to produce entire working systems, while Alchemy focuses on translating partial specifications into partial programs (specifically, libraries). In addition, Executable UML is based on an object rather than relational language of specification.

SPECWARE [MA01], the current incarnation of a series of innovative tools, is a synthesis engine that has been successfully applied to build several systems. It uses a refinement-based approach to obtain programs from specifications. In general, this involves the creation of proof-obligations that the user must eventually discharge. In contrast, our work attempts to simply find an interpretation for operations, using various heuristics to narrow the search space.

The B-method [Abr96] has been used to develop several significant systems. The B approach is to convert specifications into programs through a process of applying refinements. In particular, a specification is refined until it is deterministic, at which point it can be translated directly into code. Alchemy sits at a very different point in the design space of synthesizers, trying to relieve developers the burden of proceeding from a partial, non-deterministic specification to a rapid (and hopefully usable) prototype; to instead build large, industrial systems, Alchemy would probably have to adopt techniques such as refinement.

Numerous tools "animate" specifications in Z and similar languages (e.g., [HST97, MFMU05]), B [WB98], and the Java Modeling Language [BDLU05]. These tools typically refine a given specification gradually into first-order logic or a language such as Prolog. The goal of animation is to detect errors and improve comprehension. Unlike Alchemy, these tools use animation as one more tool in the design and specification process (e.g., in this methodology one is typically not targeting Prolog code as a final product) rather than produce code suitable for deployment on real

databases.

DynAlloy [FLB+05] is an extension to Alloy to express state change in specifications. The authors make the same observations as we do about the intentional reading of predicates, but choose to alter the language to reflect this explicitly. DynAlloy supports only analysis, not code-generation.

Gheyi, Massoni, and Borba [GMB07] recognize the difficulty in correctly expressing framing conditions for state transitions. They present a set of refactoring rules to translate between the global state idiom used in our work into a local state idiom. These rules may be of use in refining Alchemy.

5.2 Databases

We can view Alchemy as realizing a form of the Semantic Data Model (SDM) [HM78], which is an early and important framework for describing hierarchical data models. Like Alloy, the SDM supports features such as object hierarchies, data constraints, aggregation of entities, and definitions for derived data, but it does so through separate semantic concepts, which can result in more unwieldy descriptions than those obtained thanks to Alloy's uniformity.

Hammer and Berkowitz's DIAL system [HB80] describes a database programming language based on the SDM. The dynamics of a system are described by procedure definitions analogous to the Alloy predicates relating pre- and post-states. Unlike Alchemy, however, DIAL does not automatically guarantee that the actions of these procedures will agree with the static constraints of an SDM model; instead, it triggers "entry procedures" that must the programmer must manually implement to perform repair.

The Galileo [ACO85] programming language features a rich type system and

supports certain integrity constraints. While Alchemy's types are weaker, Galileo does not address our main goal of bridging the gap between declarative specification and implementation, and Alchemy enforces a richer class of semantic invariants.

Stemple, Mazumdar, and Sheard [SMS87] choose first-order logic as their constraint language, as in Alchemy. Their strategy, however, is very different: they use a theorem prover to search for a proof of satisfiability; unsafe operations leave a residue of unsatisfiable subgoals that are rewritten as operations to repair a transaction.

McCune and Henschen [MH89] perform queries that check the complete conditions for preserving database constraints across transactions to avoid rollback. They apply a theorem-proving search to establish that a transaction preserves a constraint and, if the search fails, use the counterexamples to generate runtime checks. In contrast to Stemple, et al., they concentrate on determining how to optimize away particular checks of constraints. They raise the possibility of computing transaction repair with their work, but instead focus on runtime checks for violation detection.

Ceri and Widom [CW94] describe a system for automatically maintaining the consistency of a data model. Their maintenance procedures change a set of derived tables based on Datalog-defined rules. Like Alchemy, these repair rules are automatically generated by the system. However, their system has access only to a set of base tables in computing repair, so they cannot handle recursive rules. Later work by Ceri, et al. [CFPT94] lifts these restrictions and allows fine-tuning by the designer.

Orman [Orm01] defines transaction repair for database updates, handling constraints written in non-recursive Datalog. The system treats a single constraint after a homogeneous update and does not attempt to manage the difficulties arising from the presence of multiple constraints and mixed insert/delete transactions. The Alloy

language presents additional challenges not faced there due to the rich structure of expressions in relational algebra.

Nentwich, Emmerich, and Finkelstein's document consistency manager [NEF03] defines a notion of repair specialized for XML data structures. While we treat atom creation separately from repair, their repair semantics allows the creation or destruction of domain elements. Their work permits user interaction with the repair algorithm.

Demsky and Rinard [DR03] describe a system for automatically repairing errors in program data structures from constraints. While their work interprets atomic data, such as numbers, it is limited to removing or deleting single tuples. Their work presents a cost function for directing repair search, which we lack.

Melnik, Adya, and Bernstein's work enables efficient representation and access to relations in a relational schema [MAB07]. Given a constraint mapping between a conceptual schema and a store relational schema, the thesis defines an algorithm for computing views that express one schema in terms of the other. Its result addresses the problem of how one can retrofit an Alloy data model on top of a pre-existing legacy database, or between the idealized Alloy data model and an optimized implementation.

Ruby on Rails [Rub] includes an implementation of the ActiveRecord data-mapping protocol [Fow02], where database access is exposed through classes. Like Alchemy, ActiveRecord supports the development of data models with hierarchies and object associations, supporting a limited set of multiplicity constraints. However, constraint maintenance is done through validation rather than automatic repair.

Chapter 6

Evaluation

We have run Alchemy on several examples, including the running example from this thesis and simple examples from the Alloy book [Jac06]. More usefully, we have applied Alchemy to a model reflecting the features of *Continue*[1], a working system that manages papers for academic conferences.

One of the dangers of our approach is that it might perform in an unreasonable amount of time. The Alloy Analyzer tool uses the notion of the small scope hypothesis to justify the use of very small models, using a handful of atoms to exercise a specification's assertions. But a more realistic instance of a system may have many more atoms, and under that situation, the Alloy Analyzer can take minutes for its analysis. In our system, we use nondeterminism and backtracking to search for a satisfying operation sequence; because we are doing an exhaustive search, there is a risk that Alchemy may get lost within the search space and perform in time exponential in the size of the input. To our relief, this hasn't happened for the common cases we've tried.

The *Continue* specification, a specification of nontrivial complexity, defines the

[1] `http://continue2.cs.brown.edu/`

behavior of a conference manager. *Continue* maintains authors, publications, and reviewers. Publications may be in a particular stage of processing and approval, and reviewers can bid for papers and be assigned to a set for review. Actions are managed by an access-control policy that considers the actors, the objects, and the overall system context. We observed Alchemy's behavior on working on substantial inputs, and measuring how long it takes a predicate to terminate.

The *Continue* specification's *state* object has 15 fields, several of which have sub-structure. There are 25 other signatures, of which 15 represent enumerated types. Most of these have signature constraints that turns into facts. The model has 22 stateful predicates, most of which have either an update or framing condition for each of the 15 *state* fields.

We tested typical workflows (submission, bidding, assignment, reviewing, etc.) representing small conferences (up to 40 papers and 24 reviewers). These workflows thoroughly exercised repairs. Even though our prototype implementation lacks numerous optimizations (Chapter 7), each procedure execution, including repair, took under a second (executed locally, to avoid network overhead) on a laptop (MacBook Pro, 2.33GHz Intel Core 2 Duo with 2Gb RAM).

The workflow we simulated performed the following:

- created a program committee (PC)
- submitted papers
- added bids for those papers by the PC
- assigned papers to the PC

with the measured times shown in Figure 6.1.

# of papers	size of PC	seconds	number of operations
		In *DrScheme*	
5	8	14.13	43
5	12	16.04	47
5	16	22.13	51
5	20	28.17	55
5	24	34.32	59
10	8	34.74	78
10	12	37.93	82
10	16	49.08	86
10	20	64.18	90
10	24	76.33	94
15	8	68.11	113
15	12	80.22	117
15	16	100.78	121
15	20	132.05	125
15	24	136.67	129
		In *MzScheme*	
5	8	10.83	43
10	8	28.61	78
15	8	61.62	113
25	8	178.62	183
30	8	286.20	218
40	8	575.00	288

Figure 6.1: Performance times for the simulated workflow. *DrScheme* is a graphical environment that introduces some runtime overhead for profiling and debugging support. We re-ran our measurements using *MzScheme* to remove this overhead: this allowed us to exercise Alchemy for the larger workflows.

Chapter 7

Conclusion and Future Work

The use of software tools like Alloy allows designers to explore designs of systems interactively without having to prematurely commit to implementation details. We've shown how Alloy can be used as a data modeling language by using the primed-state idiom to express state change. We want to squeeze more value out of such an appropriate specification and extract a working implementation. We presented a semantics of what such an implementation would guarantee, an algorithm for achieving those semantics, and the Alchemy compiler that provides this functionality for the implementer.

Because the Alchemy compiler itself executes in mere seconds, users can quickly obtain at least a prototype, and perhaps even a small deployment, of a persistent store. This frees them to focus on the rest of their system. Hopefully, the existence of Alchemy thus creates additional incentive to write lightweight formal specifications.

Naturally, a system of this scope offers numerous opportunities for future work.

- Our algorithms have much room for optimization. For instance, the truth of all facts in the pre-state may help identify what relations need and need not be searched for repairs. Syntactic characterizations of conditions such

as homogeneity (Section 3.2.3) would help Alchemy identify both errors and optimizations statically. In general, moving work from run-time to compile-time is an important area of future work.

- As noted in Section 3.3, the mismatch between primed-state notation for expressing state change vs. the stateless Alloy semantics poses a modeling challenge. One avenue for future work is to develop an Alloy semantics that treats state change formally, so that all signature and field relations may be viewed under a pre-and-post state. Such a semantics would be more compatible with Alchemy's imperative semantics and encourage the use of Alloy as an object-modeling language.

- We can lift our restriction on universal formulas (Section 2) by Skolemization, which has the cost of introducing additional **one** constraints.

- One part of the beast not used in this sausage is the set of assertions in an Alloy specification. It would be worthwhile to turn them into assertions about the program that are enforced via contracts and monitoring.

- Sophisticated software synthesis tools make considerable use of human guidance. Alchemy was an experiment in how far we can go with almost total automation, and the results have been positive. In a working system, however, users are likely to want much greater control over both the meaning and performance of generated code. The rich literature on synthesis and refinement (Chapter 5) will be inspirational in this regard.

- Finally, many synthesis tools employ proofs about the specification to guide program generation. Because the proofs from the Alloy Analyzer are both over bounded domains and usually have little constructive content, we have

not pursued this path. In future, however, it would be interesting to enrich our synthesizer to utilize proof information. The semantic mismatch described in Section 3.3 raises interesting challenges here.

Bibliography

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ACO85] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: a strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[BC85] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, 1985.

[BD77] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1), January 1977.

[BDLU05] Fabrice Bouquet, Frédéric Dadeau, Bruno Legeard, and Mark Utting. Symbolic animation of JML specifications. In *International Symposium of Formal Methods Europe*, 2005.

[CFPT94] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems*, 19(3):367–422, 1994.

[CW94] Stefano Ceri and Jennifer Widom. Deriving incremental production rules for deductive data information systems. *Information Systems*, 19(6):467–490, 1994.

[DR03] Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2003.

[Far] Will M. Farr. The classic ambiguous operator. `http://planet.plt-scheme.org/display.ss?package=amb.plt&owner=wmfarr`.

[FLB+05] 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.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[GMB07] Rohit Gheyi, Tiago Massoni, and Paulo Borba. Formally introducing Alloy idioms. In *Brazilian Symposium on Formal Methods*, 2007.

[GMUW01] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[Gre69] Cordell C. Green. Application of theorem proving to problem solving. In *International Joint Conference on Artificial Intelligence*, 1969.

[HB80] Michael Hammer and Brian Berkowitz. DIAL: A programming language for data intensive applications. In *ACM SIGMOD International Conference on Management of Data*, 1980.

[HM78] 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.

[HST97] Daniel Hazel, Paul Strooper, and Owen Traynor. Possum: An animator for the SUM specification language. In *Asia-Pacific Software Engineering and International Computer Science Conference*, 1997.

[Jac00] Daniel Jackson. Automating first-order relational logic. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2000.

[Jac06] Daniel Jackson. *Software Abstractions*. MIT Press, 2006.

[JW96] Daniel Jackson and Jeanette Wing. Lightweight formal methods. *IEEE Computer*, April 1996.

[KFDY08] Shriram Krishnamurthi, Kathi Fisler, Daniel J. Dougherty, and Daniel Yoo. Alchemy: transmuting base alloy specifications into implementations. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 158–169, New York, NY, USA, 2008. ACM.

[MA01] James McDonald and John Anton. SPECWARE - producing software correct by construction. Technical Report KES.U.01.3, Kestrel Institute, March 2001.

[MAB07] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In *ACM SIGMOD International Conference on Management of Data*, 2007.

[MB02] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[MFMU05] Tim Miller, Leo Freitas, Petra Malik, and Mark Utting. CZT support for Z extensions. In *International Conference on Integrated Formal Methods*, 2005.

[MH89] William W. McCune and Lawrence J. Henschen. Maintaining state constraints in relational databases: a proof theoretic basis. *Journal of the ACM*, 36(1):46–68, January 1989.

[MW80] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.

[NEF03] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *International Conference on Software Engineering*, 2003.

[Orm01] L. V. Orman. Transaction repair for integrity enforcement. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):996–1009, November 2001.

[Rub] Ruby On Rails. Ruby on Rails. `http://rubyonrails.org/`.

[RW88] Charles Rich and Richard C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, 21(8):40–51, 1988.

[Smi85] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[SMS87] David Stemple, Subhasish Mazumdar, and Tim Sheard. On the modes and meaning of feedback to transaction designers. *SIGMOD Record*, 16(3):374–386, December 1987.

[WB98] Helene Waeselynck and Salimeh Behnia. B model animation for external verification. *International Conference on Formal Engineering Methods*, 1998.

[WL69] R. J. Waldinger and R. C. T. Lee. PROW: A step toward automatic program writing. In *International Joint Conference on Artificial Intelligence*, 1969.