Alchemy: Transmuting Base Alloy Specifications into Implementations

Shriram Krishnamurthi
Brown University

Daniel J. Dougherty
WPI

Kathi Fisler
WPI

Daniel Yoo
WPI

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.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; H.2.3 [**Database Management**]: Languages

General Terms

Design, Languages

Keywords

Alloy, relational specification, program synthesis

1. INTRODUCTION

Software engineering wisdom encourages developers to explore models of their systems before they commit to implementation details. An especially powerful idea, *lightweight formal methods* [18] 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 [16], 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.

Original paper appears in
ACM SIGSOFT Foundations of Software Engineering (FSE), 2008.

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 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 (Section 5.3). 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 (Section 5) translates these predicates into a library of imperative functions. This work is presented not only formally but also through a working tool, Alchemy, which we have evaluated on a non-trivial specification (Section 6).

In harmony with the lightweight formal methods philosophy of *partiality*, we focus on the generation of APIs rather than whole programs. Automatic repair further supports this philosophy. This scope is of tremendous value because it enables us to generate, for instance, back-ends for Web applications. Section 8 relates our work to other program synthesis efforts.

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 investigation. We discuss our design decisions, constraints, and lessons at various points, and elaborate on the mismatch in Section 7.

This paper uses three terms that are sometimes confused—"semantics", "algorithm", and "implementation"—to mean three different things. The *semantics* (Section 4) tells us what the implementation guarantees, but nothing about how to achieve it. The *algorithm* (Section 5) is one particular set of rules for realizing the semantics. Finally, the *implementation* (Section 6) briefly discusses some of the practical issues that arise in realizing the algorithm.

```
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 1: Alloy specification of a gradebook.

2. AN OVERVIEW OF ALLOY

In this section, we use an example to give an overview of Alloy syntax and semantics. Readers familiar with Alloy should anyway peruse the example, as we refer to it extensively in the rest of the paper. Our work supports a subset of Alloy that includes this example; details on the subset are given later in this section.

The example is a homework submission and grading system, shown in Figure 1. 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 the third author developed for her department. This author prototyped data models for the system in Alloy early in the design phase before manually porting the models to a Web-based implementation. Alchemy is designed to reduce the effort in this last, manual, step by automatically creating the database back-end. Furthermore, by construction, the back-end will automatically maintain integrity constraints that are encoded as system invariants in the Alloy model.

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. Alloy uses *signatures* to capture the sets and relations that comprise a data model. Each **sig** (*Submission*, etc.) defines a unary relation. The elements of these relations are called *atoms*; the type of each atom is its containing relation.

Fields of signatures define additional relations. The **sig** for *Course*, for example, 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 predicates (*Enroll*, etc.) capture the actions supported in the system. The predicates follow a standard Alloy idiom for stateful operations: each has parameters for the pre- and post-states of the operation (c and c', respectively) and constrains the latter to reflect the change applied to the former.[1] Facts (such as *SameGradeForPair*) capture invariants on the models. This particular fact states that students who submit joint work get the same grade.

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). Square brackets also represent join (. versus [] is used based on the location of the common type for the join in the respective tuples). The following relations constitute a valid model under the Alloy semantics.[2]

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

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 $c = c0$, $c' = c1$ 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*. Under bindings $c = c0$ and $c' = c1$, for example, 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'.

The Supported Alloy Language

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).

[1] Alloy models of stateful systems often employ the ordering module to sequence states; we currently do not exploit this.
[2] For readability, we use concrete atom names rather than Alloy's abstract ones.

The bodies of predicates and facts are terms in the Alloy Kernel logic (the core forms of Alloy [17, 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 [17] sans the expr = expr form in elemFormula:

expr ::= rel | var | none | expr binop expr | unop expr
binop ::= + | & | − | . | →
unop ::= ˜ | ˆ

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* **in** *expr2* **and** *expr2* **in** *expr1*. This is sound in Alloy (which exploits explicit = in its analysis framework [17, page 292]).

The rest of the paper 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 for such systems: some signature is effectively declared to represent the "state" of the system and 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 $\langle S, P, F, state\rangle$:

- 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 5.1.1) 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.

3. AN OVERVIEW OF ALCHEMY

Given the gradebook specification from Figure 1, Alchemy creates 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 Alchemy's features through a sample interaction using these generated functions. We create a course with two students using the following command sequence:

cs311 = *CreateCourse*("cs311");
pete = *CreateStudent*("Pete");
caitlin = *CreateStudent*("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 maintains 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 adds the pairs ⟨"cs311", "Pete"⟩ and ⟨"cs311", "Caitlin"⟩ to the *roster* table. The second clause of the *Enroll* specification guarantees 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 Alchemy does not add such tuples, the clause is unnecessary; instead, Alchemy enforces the constraint by removing any tuples that fail this condition.

Next, we submit a new homework for "Pete" and "Caitlin":

hwk1 = *CreateSubmission*("hwk1");
SubmitForPair(*cs311*, *pete*, *caitlin*, *hwk1*)

The implementation of *SubmitForPair* is straightforward relative to the specification. It treats the first clause in the specification as a pre-condition by terminating the computation with an error if the clause is false in the database at the start of the function execution. Next, it adds the *work* tuples required in the second (update) clause. Finally, it checks that the *gradebook* table is unchanged, as required by the third clause.

Assigning a grade illustrates Alchemy repair feature:

gradeA = *CreateGrade*("A");
AssignGrade(*cs311*, *pete*, *hwk1*, *gradeA*)

AssignGrade inserts a tuple into the *gradebook* relation according to the first clause, and checks 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 attempts to repair the database to satisfy both the predicate body and the fact. It determines that adding the tuple ⟨"cs311","Caitlin","hwk1", "A"⟩ to *gradebook* achieves this, and executes this command automatically. The fact therefore holds 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 will 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 provably preserves all facts as database invariants when its functions terminate without exceptions.

4. INTERPRETING ALLOY IMPERATIVELY: 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 *the_c* and include a table

$roster = (\langle the\_c, Harry\rangle)$

which, after enrolling *Meg*, changes to:

$$roster = (\langle the\_c, Harry\rangle,\\ \langle the\_c, Meg\rangle)$$

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 comingles all these database instances in a single relation. This has important consequences, as we discuss in Section 7. 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.

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.

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.

4.2 Transition Systems over Instances

Each transition 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 5.1.

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.

5. INTERPRETING ALLOY IMPERATIVELY: ALGORITHM

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. The semantics, however, admits many possible functions for each predicate. Our compilation algorithm must choose one that implements a predicate body without violating the facts (which constrain program states).

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.

5.1 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.

5.1.1 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 1, 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.[3] 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:

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.

[3]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.

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.

5.1.2 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.

5.1.3 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 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 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 7).

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 2. 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.

```
void $GenCommands(fmla, unprimed\text{-}db, primed\text{-}db)$
    $E$ = if $e$ primed then $primed\text{-}db(e)$ else $unprimed\text{-}db(e)$
    $F$ = if $f$ primed then $primed\text{-}db(f)$ else $unprimed\text{-}db(f)$
```

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 2: Command generation.

Command generation fails if there is no tuple to insert or delete in the third and fourth rows of Figure 2. 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 1 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 5.3 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 when executing predicates with parameters that follow the *New* naming convention.

Figures 3 and 4 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 and 4 reflects a design decision on our part. We could handle the case where t is in e_2 by adding t to e_1 and removing t from e_2. Implementing abstract insertion operations with concrete deletions, however, has implications for termination, as we discuss in Section 5.3. As a general rule, we prefer to use only insertion operations to implement insertions, and analogously for deletions.

5.2 Compiling Predicates

Figure 5 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

```
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: Inserting a tuple into an expression.

```
void $deleteTuple(t : T_1 \times \cdots \times T_n$, $e$: 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 $e_i$ ; $deleteTuple(t, e_i)$ ]
    [~$e$: $deleteTuple($~$t, e)$ ]
    [$e_1 \rightarrow e_2$:
        let $t = t_1 \rightarrow t_2$ 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 $e_1$, $deleteTuple(a \rightarrow t, e_2)$
          elseif $T$ is the type of $e_2$ then
              foreach $a$ in $e_2$, $deleteTuple(t \rightarrow a, e_1)$
          else foreach $a$ in $T$ such that for some $s_1$, $s_2$
                    $t_1 = s_1 \rightarrow a$ in $e_1$ and
                    $t_2 = a \rightarrow s_2$ 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 $(y_i, y_{i+1})$
                $deleteTuple(y_i \rightarrow y_{i+1}, e_1)$]
```

Figure 4: Deleting a tuple from an expression.

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 5.1) 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.

5.3 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 6 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 5.

The repair algorithm assumes that all universal facts are in conjunctive normal form. Distributing the quantifiers

```
Given:

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

Generate:

list(atom) $p(v_1 : T_1, \ldots, v_j : T_j)$
  let $newv_i = new\_atom\ (pre\text{-}state, T_i)$ for $i \in k, \ldots, n$
  $post\text{-}state = pre\text{-}state$
  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 $\varphi_i$
        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 2
        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 6
    if some formula in Guards false then FAIL
    if some cardinality constraint false then FAIL
    pre-state = post-state; return $newv_k, \ldots, newv_n$
```

Figure 5: Pseudocode for a compiled predicate. $\varphi[Z]$ denotes φ substituted with all bindings in Z.

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 5.1). Once we decide whether to nullify using insertion or deletion (a decision discussed momentarily), we reuse the table in *GenCommands* (Figure 2) 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.
- **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.

```
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 2
    insert all non-universal facts into Guards
```

Figure 6: Algorithm to repair 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 1. 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.

Finally, we state two theorems about the algorithm in this section. The first says that the algorithm is faithful to the semantics we have defined. One consequence of this theorem is that the facts are state invariants over the API functions. The second states conditions under which the function for a predicate can successfully execute the predicate. We omit both proofs for sake of space.

THEOREM 1. *Let $A = \langle S, P, F, state\rangle$ be an Alloy specification. Let I be an instance of the database schema for A and let p be a predicate in P. Let X_p be the executable function compiled for p and let E be a binding to the parameters of X_p from atoms in I. The execution of X_p on I will terminate. If it succeeds, it results in a new database instance I' such that $(I, I') \models_E \langle p, F\rangle$.*

THEOREM 2. *Let $A = \langle S, P, F, state\rangle$ be an Alloy specification that does not use the difference operator. Let I be an instance of the database schema for A and let p be a predicate in P. Let X_p be the executable function for p. Suppose there exists a binding E of the parameters of X_p to atoms in I such that X_p has a homogeneous execution on E from I. Then running X_p on I and E results in an instance I' such that $(I, I') \models_E \langle p, F\rangle$.*

The restriction regarding difference arises from our decision to FAIL in some cases when inserting into difference expressions in Figures 3 and 4. Relaxing that restriction would interfere with the homogeneity assumptions that guarantee termination of our executable functions. Other FAIL cases identify logical inconsistencies, rather than cut off potentially sound implementations.

6. IMPLEMENTATION & EVALUATION

Our implementation (in PLT Scheme [9]) uses the Alloy parser as a front-end and a Postgres database back-end. The compiler produces a stand-alone library that contains a function for each stateful predicate, as well as functions for initializing a database according to the schema. 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 1) generates options to modify each of *c*, *gradebook*, *s1*, and *b*. All but the option for *gradebook* will fail immediately in Figure 3. 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). Second, all references to predicates must be inlined. Since predicates are first-order formulas, this does not limit expressiveness.

We have run our implementation on several examples, including the running example from this paper and simple examples from the Alloy book [17]. More usefully, we have applied Alchemy to a model reflecting the features of Continue (`http://continue2.cs.brown.edu/`), a working system that manages papers for academic conferences.

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 (upto 40 papers and 24 reviewers). These workflows thoroughly exercised repairs. Even though our prototype implementation lacks numerous optimizations (Section 9), 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).

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.

7. DISCUSSION: RELATING THE ALLOY AND IMPERATIVE SEMANTICS

The astute reader will have noticed that we offer a guarantee about the preservation of facts, but 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):

// this is not satisfiable
pred *change_r*(a, a' : A) {a'.r != 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 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.

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 4 illustrate, the . operator—a relational join that evokes object dereference—may be confusing in the presence of state. In Section 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.

8. RELATED WORK

Synthesis.

Software synthesis is an elusive goal, as Rich and Waters summarize [28]. Green [12] and Waldinger and Lee [32] are generally credited with initiating this effort. Bates and Constable [3] 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 [5] instead define rules to transform specifications into programs, which Manna and Waldinger [19] combine with theorem-proving and induction. Some authors such as Smith [29] 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 [22] 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 [21], 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 [1] 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., [15, 24]), B [31], and the Java Modeling Language [4]. 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 [10] 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 [11] 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.

Databases.

We can view Alchemy as realizing a form of the Semantic Data Model (SDM) [14], 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 [13] 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 [2] 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 [30] 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 [20] 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 [7] 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. [6] lifts these restrictions and allows fine-tuning by the designer.

Orman [26] 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 [25] 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 [8] 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 [23]. Given a constraint mapping between a conceptual schema and a store relational schema, the paper 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.

9. CONCLUSION & FUTURE WORK

We have presented Alchemy, a program synthesizer for Alloy specifications. Alchemy consumes partial specifications of systems and translates these into implementations of libraries. Concretely, it (a) translates the signatures into persistent database tables; (b) converts predicates that follow a standard stateful convention of Alloy into imperative updates over these databases; and (c) compiles Alloy facts into state invariants, and automatically repairs the database after each update to restore these invariants. As a result, Alchemy is useful at least for prototyping persistent backends whose data models have been explored using the Alloy Analyzer and other tools.

In addition to our concrete offerings (the semantics, algorithms, and tool), we have made observations about the mismatch between the relational and imperative-object notation and semantics (especially as discussed in Section 7), and presented our design choices. Taken together, these imply several challenges for the design of languages and tools to capture partial specifications of stateful systems.

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 5.3) 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.

- 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 (Section 8) 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 7 raises interesting challenges here.

Still, we should remember the wisdom of Alan Perlis [27]: "When someone says 'I want a programming language in which I need only say what I wish done,' give him a lollipop".

Acknowledgments

We are grateful to William Cook and Daniel Jackson for extensive and helpful discussions. We thank Felix Chang and Theo Giannakopoulos for comments, and Matthias Felleisen for inspiration. Michael Butler, Uğur Çetintemel, Chuck Rich, Jim Woodcock, and Stan Zdonik discussed related work with us. The Spring 2006 Brown cs296-1 class, especially Jay McCarthy and Warren Schudy, helped model Continue. This work is partially supported by the NSF, Cisco, Google, and a GAANN Fellowship.

10. REFERENCES

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

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

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

[4] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. In *International Symposium of Formal Methods Europe*, 2005.

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

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

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

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

[9] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[10] M. F. Frias, C. G. López Pombo, G. A. Baum, N. M. Aguirre, and T. 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.

[11] R. Gheyi, T. Massoni, and P. Borba. Formally introducing Alloy idioms. In *Brazilian Symposium on Formal Methods*, 2007.

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

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

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

[15] D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the SUM specification language. In *Asia-Pacific Software Engineering and International Computer Science Conference*, 1997.

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

[17] D. Jackson. *Software Abstractions*. MIT Press, 2006.

[18] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, Apr. 1996.

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

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

[21] J. McDonald and J. Anton. SPECWARE - producing software correct by construction. Technical Report KES.U.01.3, Kestrel Institute, Mar. 2001.

[22] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[23] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. In *ACM SIGMOD International Conference on Management of Data*, 2007.

[24] T. Miller, L. Freitas, P. Malik, and M. Utting. CZT support for Z extensions. In *International Conference on Integrated Formal Methods*, 2005.

[25] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *International Conference on Software Engineering*, 2003.

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

[27] A. J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9):7–13, Sept. 1982.

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

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

[30] D. Stemple, S. Mazumdar, and T. Sheard. On the modes and meaning of feedback to transaction designers. *SIGMOD Record*, 16(3):374–386, Dec. 1987.

[31] H. Waeselynck and S. Behnia. B model animation for external verification. *International Conference on Formal Engineering Methods*, 1998.

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