An Improved Algorithm for Generating Database Transactions from Relational Algebra Specifications

Daniel J. Dougherty
Worcester Polytechnic Institute
Worcester, MA, USA, 01609
dd@wpi.edu

Alloy is a lightweight modeling formalism based on relational algebra. In prior work with Fisler, Giannakopoulos, Krishnamurthi, and Yoo, we have presented a tool, Alchemy, that compiles Alloy specifications into implementations that execute against persistent databases. The foundation of Alchemy is an algorithm for rewriting relational algebra formulas into code for database transactions. In this paper we report on recent progress in improving the robustness and efficiency of this transformation.

1 Introduction

Alloy [5] is a popular modeling language that implements the lightweight formal methods philosophy [6]. Its expressive power is that of first-order logic extended with transitive closure, and its syntax, based on relational algebra, is strongly influenced by object modeling notations. The language is accompanied by the Alloy Analyzer: the analyzer builds models (or "instances") for a specification using SAT-solving techniques. Users can employ a graphical browser to explore instances and counter-examples to claims.

Having written an Alloy specification, the user must then write the corresponding code by hand; consequently there are no formal guarantees that the resulting code has any relationship to the specification. The *Alchemy* project addresses this issue. Alchemy is a tool under active development [7, 4] at Worcester Polytechnic Institute and Brown University, by Kathi Fisler, Shriram Krishnamurthi, and the author, with our students Theo Giannakopoulos and Daniel Yoo, that compiles Alloy specifications into libraries of database operations. This is not a straightforward enterprise since, in contrast to Z [8] and B [1], where a notion of state machine is built into the language, Alloy does not have a native machine model.

Alchemy opens up a new way of working with Alloy specifications: as declarative notations for imperative programs. In this way Alloy models support a novel kind of rule-based programming, in which underspecification is a central aspect of program design.

In this note we report on recent progress in improving the process of generating imperative code for declarative specifications in a language like Alloy. This paper is a companion to [4], which developed a better semantic foundation for interpreting Alloy predicates as operations. With this better foundation we are able to generate code for a wider class of predicates than that treated in [7] and also prove a more robust correctness theorem relating the imperative code to the original specification.

2 Alloy and Alchemy

Some of the material in this expository section is taken from [7].

I. Mackie and A. Martins Moreira (Eds.): Tenth International
Workshop on Rule-Based Programming (RULE 2009)
EPTCS 21, 2010, pp. 77–89, doi:10.4204/EPTCS.21.7

2.1 An overview of Alloy

An excellent introduction to Alloy is Daniel Jackson's book [5]. Here we start with an informal introduction to Alloy syntax and semantics via an example. The example is a homework submission and grading system, shown in Figure 1. In this system, students may 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.

The system's data model centers around a course, which has three fields: a roster (set of students), submitted work (relation from enrolled students to submissions), and a gradebook. Alloy uses **sig**natures 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 $\mathit{Course}\times\mathit{Student}$. Similarly, the relation *work* is of type $\mathit{Course}\times\mathit{Student}\times\mathit{Submission}$, 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 **pred**icates (*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), with the intended interpretation that latter reflects a change applied to the former. Alloy **fact**s (such as *SameGradeForPair*) capture invariants on the models. This particular fact states that students who submit joint work get the same grade.

An important aspect of Alloy is that *everything is a relation.* In particular sets are viewed as unary relations, and individual atoms are viewed as singleton unary relations. As a consequence the **in** operator does double-duty: it is interpreted formally as subset, but also stands in for the "element-of" relation, in the sense that if—intuitively—*a* is an atom that is an element of a set *r*, this is expressed in Alloy as *a* **in** *r*, since *a* is formally a (singleton) set.

The Alloy semantics defines a set of models for the signatures and facts. Operators over sets and relations have their usual semantics: $\cup$ (union), $\cap$ (intersection), $\langle\ ,\ \rangle$ (tupling), and . (join).[1] As noted above, **in** denotes subset and is also used to encode membership. Square brackets provide a convenient syntactic sugar for certain joins: $e2[e1]$ is equivalent to $e1.e2$. The following relations constitute a model under the Alloy semantics.

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

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 $\langle \mathit{c1}, \mathit{Harry}, \mathit{hwk1}\rangle$ in the *work* relation is semantically acceptable.

[1] For consistency with the presentation and analysis of the algorithms below, we use standard mathematical notation in two places where Alloy uses ASCII notation: $\cup$ is "+" in Alloy, $\cap$ is "&".

```
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
     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.

The reader may want to check that the relations shown do not happen to model the predicate *SubmitForPair*, in the sense that no bindings for $c, c', s1, s2, andbNew$ make the body of *SubmitForPair* true. Under *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*'.

2.2 An overview of Alchemy

We illustrate Alchemy in the context of the gradebook specification from Figure 1. Alchemy creates a database table for each relation (e.g., *Submission*, *roster*), a procedure for each predicate (e.g., *Enroll*), and a function for creating new elements of each atomic signature (e.g., *CreateSubmission*). A sample session using Alchemy might proceed as follows. 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-argument, in contrast to the two in 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.

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. It ensures that the *gradebook* table is unchanged, as required by the third clause.

Assigning a grade illustrates the way that Alloy facts constrain Alchemy's updates:

> *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 determines that adding the tuple ⟨*cs311*, *Caitlin*, *hwk1*, *A*⟩ to *gradebook* will satisfy both the predicate body and the *SameGradeForPair* fact, and executes this command automatically. If there is no way to update 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 predicate body).

Maintaining invariants Alloy's use of *facts* to constrain possibly-underspecified predicates offers a powerful lightweight modeling tool. The facts in an Alloy specification are axioms in the sense that they hold in any instance for the specification. We may view the facts as integrity constraints: they capture the fundamental invariants to be maintained across all transactions. Alchemy will guarantee preservation of all facts as database invariants. This is akin to the notion of *repair* of database transactions.

2.3 Formalities

Alloy specifications Formally, the *Alloy specifications* we treat in this paper are tuples of *signatures*, *predicates*, and *facts*. In practice Alloy specifications may also include *assertions* to be checked by the analyzer, but they do not play a direct role in Alchemy's code generation so we omit them here.

- A signature specifies its type name and a set of fields. Each field has a name and a type specification $A_0 \to A_1 \to \ldots \to A_n$, where each A_i is the type name of some signature.
- 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 formula in which the only free variables are defined in the header.
- A fact is a closed formula, having the force of an axiom: models of a specification are required to satisfy these facts. Alloy permits the user to specify certain constraints on the signatures and fields when they are declared, such as "relation *r* may have at most one tuple." These can be alternatively expressed as facts and, for simplicity of presentation, we assume this is always done.

The following language for expressions and formulas is essentially equivalent to the Kernel language of Alloy [5] (modulo the lexical differences between standard mathematical notation used here and Alloy's ASCII).

$$\begin{aligned}
&\mathsf{expr} ::= \mathsf{rel} \mid \mathsf{var} \mid \mathsf{none} \mid \mathsf{expr\ binop\ expr} \mid \mathsf{unop\ expr}\\
&\mathsf{binop} ::= \cup \mid \cap \mid - \mid . \mid \langle,\rangle\\
&\mathsf{unop} ::= \sim \mid *
\end{aligned}$$

$$\begin{aligned}
&\mathsf{formula} ::= \mathsf{elemFormula} \mid \mathsf{compFormula} \mid \mathsf{quantFormula}\\
&\mathsf{elemFormula} ::= \mathsf{expr}\ \ \mathbf{in}\ \ \mathsf{expr} \mid \mathsf{expr} = \mathsf{expr}\\
&\mathsf{compFormula} ::= \mathbf{not}\ \mathsf{formula} \mid \mathsf{formula} \wedge \mathsf{formula} \mid \mathsf{formula} \vee \mathsf{formula}\\
&\mathsf{quantFormula} ::= \forall\ \mathit{var}\!: \mathsf{expr}\ \{\ \mathsf{formula}\ \} \mid \exists\ \mathit{var}\!: \mathsf{expr}\ \{\ \mathsf{formula}\ \}
\end{aligned}$$

State-based specifications 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—function as database integrity constraints. Finally, under a commonly idiom, certain predicates in an Alloy specification connote state changes. It is these state-based specifications that Alchemy (currently) treats.

The state-transition idiom is a commonly understood convention rather than a formal notion in Alloy. To precisely define the class of specifications that Alchemy treats, we first require some terminology. Fix a distinguished signature, which we will call `State`. An *immutable type* is one with no occurrences of the `State` signature.

The assumptions Alchemy makes about the specifications it treats are:

- specifications are state-based, and
- facts have at most one variable of type `State`, and this variable is unprimed and universally quantified.

An operational semantics The static semantics of Alloy is based on the class of relational algebras. To give an operational semantics for state-based Alloy specifications, one that takes seriously the reading of predicates as state-transformers, we pass to the class of transition systems whose nodes are relational algebras. We also assume that each state has a single atom of type State. When individual relation algebras are read as database instances, transitions between states can be viewed as database update sequences transforming one state to another. We adopt a constant-domain assumption concerning our transition systems. Space consideration prohibit us from presenting the motivation and justification for this (including the explanation why it is not as great a restriction as it may appear); details are in [4].

Since predicates have parameters, the meaning of a predicate is relative to bindings from variables to values. It is technically convenient to assume that for a given specification we identify, for each type, a universe of possible values of this type. Then an *environment* η is a mapping from typed variables to values.

Definition 1 (**Operational semantics of predicates**)**.** Let p be a predicate with the property that p has among its parameters exactly two variables s and s$'$ of type State, and let η be an environment. The meaning $[\![p]\!]\eta$ of p under η is the set of pairs $\langle I, I'\rangle$ of instances such that

- η maps the parameters of p into the set of atoms of I (which equals the set of atoms of I'), mapping the unprimed State parameter to the State-atom of I and the primed State parameter to the State-atom of I';
- (I, I') makes the body of p true under the environment η: occurrences of the State variable s are interpreted in I, while occurrences of the State variable s' are interpreted in I'.

The meaning of a predicate p is a *set* of transitions because p can be applied to different nodes, with different bindings of the parameters of course, but also—and more interestingly—because predicates typically under-specify actions: different implementations of a predicate can yield different outcomes I' on the same input I. Any of these should be considered acceptable as long as the relation between pre- and post-states is described by the predicate.

3 Main Result

We observed that a predicate p determines a family of binary relations over instances, parametrized by environments. That is, for a given environment η:

$$[\![p]\!]\eta : Inst \to 2^{Inst}. \tag{1}$$

Now suppose t is a procedure defining a database transaction (so t is the sort of procedure that a predicate p specifies). Given an instance I and an environment η, t may return a new instance I', terminate with failure, or may diverge. None of the procedures we describe in this paper will diverge, so we are considering procedures t that (under an environment) determine a function over instances:

$$[\![t]\!]\eta : Inst \to (Inst + fail). \tag{2}$$

Alchemy's job is precisely the following: given predicate p, construct a procedure $t = \mathtt{code}(p)$ such that the semantics of $\mathtt{code}(p)$ as given in 2 refines the semantics of p as given in 1, in the following sense.

Theorem 2 (**Main theorem**)**.** *Let p be a predicate and let* $\mathtt{code}(p)$ *be any backtracking implementation of the algorithm $\mathbb{A}_p$, given in Definition 5 below. Then for each instance I and each environment η*

1. $[\![\mathsf{code}(p)]\!]\eta$ *terminates on* I*;*
2. *If there exists any instance* I' *such that* (I,I') *satisfies* p *under* η *then the result of* $[\![\mathsf{code}(p)]\!]\eta$ *is such an* I'*. In particular in this situation* $[\![\mathsf{code}(p)]\!]$ *does not return "failure" under* η *on* I*.*

Proof. The proof is given in Section 4.4. □

It is worth noting that the task of generating updates from specification submits to an uninteresting trivial solution, particularly if we are willing to tolerate partial functions. Given predicate p we could define $\mathsf{code}(p)$ by: *on input* I*, exhaustively generate all possible* I'*; for each one test whether* (I,I') *in* $[\![p]\!]$*. If and when such an* I' *is found, replace* I *by* I'*.* Obviously this is a silly algorithm, even though it is "correct" in a formal sense. Our goal with Alchemy is to write code that is intuitively reasonable, and still is correct in the sense of Theorem 2.

4 Code generation

Suppose we are given an Alloy predicate **p**. Alchemy generates code for a procedure with parameters corresponding to those of **p** (without the primed parameter).

As observed above, a crucial aspect of *Alloy* is that it encourages "lightweight" specifications of procedures: the designer is free to ignore details about the computation that she may consider inessential. As a consequence, *Alchemy* must be extremely flexible: different input instances may require quite different computations in order to satisfy a specification, yet Alchemy must generate code that works uniformly across all instances.

The top-level view of how Alchemy generates code for a procedure is as follows.

4.1 Outline

- In Definition 5 below we present a construction that, based on predicate p, builds a *non-deterministic procedure* $\mathbb{A}_p$.
- The code generated by Alchemy, $\mathsf{code}(p)$, is a backtracking implementation of $\mathbb{A}_p$. Computation paths that do not succeed are recognized as such and abandoned, and $\mathbb{A}_p$ is finite-branching, so $\mathsf{code}(p)$ will always terminate.
- If there exists any instance I' such that (I,I') satisfies p under η then some branch of $\mathbb{A}_p$ is guaranteed to compute such some such instance.

Coping with inconsistent predicates It is possible for the code for a predicate p to *fail* on a given database instance I, either because the predicate is internally inconsistent or because no update of I can implement p without violating the facts. Alchemy is guaranteed to detect such situations; we treat predicates as *transactions* that rollback if they cannot be executed without violating their bodies or a fact.

4.2 A normal form for predicates

The general form of an Alloy predicate that specifies an operation and that Alchemy treats is

$$\mathsf{pred}\ p(s,s' : \mathsf{State},\ a_1 : A_1,\dots,a_n : A_n)\{\vec{Q}x\ .\ \beta(\vec{a},\vec{x})\}$$

where $\vec{Q}$ is a sequence of quantified atoms and β is a quantifier free formula of relational algebra. Before giving an imperative interpretation of a predicate it is convenient to massage it into a convenient form.

Skolemization By the classical technique of Skolemization any formula $\vec{Q}x \,.\, \beta(\vec{a},\vec{x})$ can be converted into a universal formula which is satisfiable if and only if $\vec{Q}x \,.\, \beta(\vec{a},\vec{x})$ is satisfiable. We exploit this trick in Alchemy as follows. Given a predicate p we convert it to a predicate $p^{\forall}$ whose body is in universal form; this involves expanding the specification language to include the appropriate Skolem functions. Suppose we generate code for $p^{\forall}$ (over the expanded language). Then given an original instance I we may view it as an instance I_{+} over the enlarged schema, and apply the generated code to obtain an instance I'_{+}. We ultimately return the instance I' that is the reduct of I'_{+} to the original schema. So in what follows we restrict attention to predicates whose body is a universal formula.

Incorporating the facts Intuitively the facts in a specification comprise a separate set of constraints on how a predicate may build new instances from old ones. But by the following simple trick we can avoid treating the facts separately. When compiling a predicate to code we take each fact, prime every occurrence of the State sig, and add the fact to the body of the predicate. The use of primed State names means that the fact acts as a post-condition on the predicate. (Strictly speaking this is only true under an assumption of "state-boundedness" on the form of the facts, defined in [4]. The specifics of this syntactic assumption are irrelevant to the current paper so we omit details.) This in turn guarantees that any post-instance defined by the predicate will satisfy the facts.

The following is a convenient form for formulas.

Definition 3 (**Special formulas**)**.** A *special* formula is a formula in either of the two forms

$$(e_1 \cap \ldots \cap e_k) = \emptyset \qquad \text{or} \qquad (e_1 \cap \ldots \cap e_k) \neq \emptyset$$

for $k \geq 1$, with each e_i not containing $\cup$ or $\emptyset$ and with converse applied only to variables and relation names.

Lemma 4. *Any quantifier-free formula can be transformed into an equivalent Boolean combination of special formulas.*

Proof. It is easy to see that every expression is equivalent to one in which the converse operator $\sim$ applies only to relation names or variables. It is easy to see that every expression other than $\emptyset$ itself is equivalent to one in which the constant $\emptyset$ never appears. Because union distributes over the other connectives every expression is equivalent to one of the form $e_1 \cup \ldots \cup e_n$ ($n \geq 1$) with each e_i being $\cup$-free.

We may take any equation $e = f$ and replace it with $(e \textbf{ in } f) \wedge (f \textbf{ in } e)$. We do this as long as neither e nor f is the term $\emptyset$.

Now each basic formula is in one of the forms

$$(d_1 \cup \ldots \cup d_m) \textbf{ in } (f_1 \cup \ldots \cup f_n) \qquad \text{or} \qquad (d_1 \cup \ldots \cup d_m) \textbf{ not in } (f_1 \cup \ldots \cup f_n)$$

with $n, m \geq 0$, where the d_i and the f_i are $\cup$-free. We may transform the basic formulas above into the corresponding forms

$$(d_1 \cup \ldots \cup d_m) - (f_1 \cup \ldots \cup f_n) = \emptyset, \qquad \text{respectively,} \qquad (d_1 \cup \ldots \cup d_m) - (f_1 \cup \ldots \cup f_n) \neq \emptyset \tag{3}$$

The first equation in 3 is equivalent, via distributivity of $\cup$ over $\cap$, to the *conjunction* of the equations

$$d_i - (f_1 \cup \ldots \cup f_n) = \emptyset \qquad 1 \leq i \leq m$$

In turn, *each* of these is equivalent to the special formula

$$(d_i - f_1) \cap \ldots \cap (d_i - f_n) = \emptyset$$

Similar reasoning shows that each dis-equation as in 3 is equivalent to a disjunction of special formulas

$$(\ldots((d_i - f_1) - f_2) - \ldots - f_n) \neq \emptyset$$

$\square$

4.3 Algorithms

Bridging the declarative/imperative gap The main procedure $\mathbb{A}_p$ below is generated by an induction that walks the structure of the formula that is the body of p. There is a natural correspondence between the logical operators in the predicate and control-flow operators in the generated procedure. The disjunctive (logical $\vee$ and $\exists$) constructors in predicates naturally suggest imperative nondeterminism; this of course results in *backtracking* in generated code. Likewise, conjunctive (logical $\wedge$ and $\forall$) constructors lead naturally to *sequencing*. This is natural enough, but a difficulty arises due to the fact that the logical operators are commutative but command-sequencing certainly is not. Indeed, implementing one part of a predicate can undo the effect achieved by an earlier part. The solution is to iterate computation until a fixed-point is reached on the post-state. So we must be careful to ensure that such an iteration will always halt.

Compiling special formulas to code Consider for example the body of the *Drop* predicate in Figure 1. There are certainly many ways to update the data to make this true; for example we could delete all the tuples in the roster table! This is not what the specifier had in mind. But even this silly example points out the need for a principled approach to update. We start with the following goal: we attempt to make a *minimal* set of updates (measured by the number of tuples inserted or deleted into tables) to the system to satisfy the predicate.

The virtue of special formulas is that they facilitate identifying minimal updates to make a formula true. For example the formula a **in** $s'.r$, which, when a is an atom, is to say that a is in the relation $s'.r$ is equivalent to the formula $a - (s'.r) = \emptyset$. So suppose $a - (s'.r) = \emptyset$ is part of the body of a predicate. We evaluate the expression $a - (s'.r)$ in the pre-state and the current post-state: if the value of this expression is indeed empty then there is nothing to do. If it is not empty then a is not in $s'.r$, and it is clear what action to take: add a to $s'.r$.

More generally, when confronted with a special formula $e = \emptyset$ we may view any tuples in the current value of e as *obstacles to the truth of the formula*. Then the action suggested by the formula is clear: make whatever insertions or deletions we can to ensure the formula becomes true. (The presence of the difference operator means that making an expression empty may involve insertions.) The important thing to note is that, obviously, we may focus exclusively on tuples that are already in the value of e in attempting to make $e = \emptyset$ in the updated state. This is our strategy for doing minimal updates for a predicate.

Inserting and deleting tuples We have seen that compiling a special formula amounts to orchestrating the insertion or deletion of individual tuples from the relations denoted by expressions. These expressions correspond to database *views*, and indeed the task of inserting or deleting a tuple from a view is an instance of the well-known *view update* problem [2, 3]. Our code proceeds by a structural induction over the expression: see the procedures insertTuple and deleteTuple below.

Putting it all together After the preceding discussion the pseudocode for the Alchemy's translation algorithm should be largely self-explanatory. For simplicity in notation we adopt the following conventions. There are global variables pre-state and post-state ranging over instances, and a global variable Updates which keeps a record of the insertions and deletions done as the algorithm progresses.

We make use of the following function $\mathbb{E}\text{val}(e : \text{expression},\ J, J' : \text{database instances})$ that returns the set of tuples denoted by expression e under the convention that immutable relation-name occurrences are interpreted in J and mutable relation-name occurrences are interpreted in J'. The pseudocode given here for procedures $\mathbb{A}_p$, $\mathbb{B}_p$, insertTuple, and deleteTuple is directly based on the discussion in the previous paragraphs.

Definition 5 (Algorithm $\mathbb{A}_p$)**.** Let p be a Alloy predicate of the form

$$\mathsf{pred}\ \ p(s, s' : \mathsf{State},\ a_1 : A_1, \ldots, a_n : A_n)\ .\ \{\forall \vec{x}\ .\ \bigwedge_i \bigvee_j \sigma_{i,j}\}$$

where each $\sigma_{i,j}$ is a special formula. The procedure $\mathbb{A}_p$ determined by p is as follows. Each of $\mathbb{A}_p$ and $\mathbb{B}_p$ reads the instance I globally and reads and writes I' and Updates globally.

```
procedure $\mathbb{A}_p$ ($I$: database instance) {
    initialize poststate $I$' to be $I$;
    initialize Updates to be empty;
    repeat $\mathbb{B}_p(a_1 : A_1, \ldots, a_n : A_n)$
    until no change in Updates
}
procedure $\mathbb{B}_p(a_1 : A_1, \ldots, a_n : A_n)$ {
    for each binding $\vec{b}$ of values in $I$ for the variables in $\vec{a}$:
    let $\bigwedge_i \bigvee_j \overline{\sigma}_{i,j}$ be the body of $p$ instantiated by $\vec{b}$:
        for each conjunct $\bigvee_j \overline{\sigma}_{i,j}$
                choose some $\overline{\sigma}_{i,j}$ and realize $\overline{\sigma}_{i,j}$ as follows:
                Case 1: $\overline{\sigma}_{i,j}$ is of the form $(e_1 \cap \ldots \cap e_k) = \emptyset$
                set $e \equiv (e_1 \cap \ldots \cap e_k)$
                for each tuple $t$ in $\mathbb{E}\text{val}(e, I, I')$:
                    call $deleteTuple(t, e, I, I')$;
                Case 2: $\overline{\sigma}_{i,j}$ is of the form $(e_1 \cap \ldots \cap e_k) \neq \emptyset$
                set $e \equiv (e_1 \cap \ldots \cap e_k)$
                choose some $t$ of the same type as $e$
                    call $insertTuple(t, e. I, I')$
            update Updates accordingly;
}

procedure insertTuple($t$ : tuple,   $e$: expression) {
    match $e$:
    atom $a$: if $a \neq t$ then FAIL else RETURN
    immutable relation $r$: if $t \notin r$ then FAIL else RETURN
    mutable relation $r$: if $t$ has been previously deleted from $r$ then FAIL
            else add $t$ to the table $r$ in $J'$
    $e_1 \cup e_2$: choose some $e_i$ ; insertTuple($t, e_i$)
    $e_1 \cap e_2$: insertTuple($t, e_1$) ; insertTuple($t, e_2$)
    $\sim e$: insertTuple( $t, e$)
    $\langle e_1, e_2\rangle$: let $t = \langle t_1, t_2\rangle$ where $t_i$ matches type of $e_i$; insertTuple($t_1, e_1$) ; insertTuple($t_2, e_2$)
    $e_1 - e_2$: insertTuple($t, e_1$) ; deleteTuple($t, e_2$)
    $e_1.e_2$: 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 $\mathbb{E}\mathrm{val}(e_1, I, I')$, insertTuple($\langle a, t\rangle, e_2$)
            elseif $T$ is the type of $e_2$ then for some $a$ in $\mathbb{E}\mathrm{val}(e_2, I, I')$, insertTuple($\langle t, a\rangle, e_1$)
            else **choose** $a : T$ ; set $t_1 = \langle s_1, a\rangle$ and set $t_2 = \langle a, s_2\rangle$;
                insertTuple($t_1, e_1$) ; insertTuple($t_2, e_2$)
    $(e_1)^*$: insertTuple($t, e_1$)

**procedure** deleteTuple($t$ : tuple,   $e$: expression) {
    match $e$:
    atom $a$: if $a = t$ then FAIL else RETURN
    immutable relation $r$: if $t \in r$ then FAIL else RETURN
    mutable relation $r$: if $t$ has been previously inserted into $r$ then FAIL
            else delete $t$ from the table $r$ in $J'$
    $e_1 \cup e_2$: deleteTuple($t, e_1$) ; deleteTuple($t, e_2$)
    $e_1 \cap e_2$: **choose** some $e_i$ ; deleteTuple($t, e_i$)
    $\sim e$: deleteTuple( $t, e$)
    $\langle e_1, e_2\rangle$: let $t = \langle t_1, t_2\rangle$ where $t_i$ matches type of $e_i$; **choose** some $e_i$; deleteTuple($t_i, e_i$)
    $e_1 - e_2$: **choose:** deleteTuple($t, e_1$) or insertTuple($t, e_2$)
    $e_1.e_2$: let $T$ be the common sig-type that joins $e_1$ and $e_2$;
            if $T$ is the type of $e_1$ then for each $a$ in $\mathbb{E}\mathrm{val}(e_1, I, I')$, deleteTuple($\langle a, t\rangle, e_2$)
            elseif $T$ is the type of $e_2$ then for each $a$ in $\mathbb{E}\mathrm{val}(e_2, I, I')$, deleteTuple($\langle t, a\rangle, e_1$)
            else for each $a : T$ such that for some $s_1, s_2$,
                    $\langle s_1, a\rangle = t_1$ is in $e_1$ and $\langle a, s_2\rangle = t_2$ is in $e_2$ and $t_1.t_2 = t$;
                    **choose** $e_i$ then deleteTuple($t_i, e_i$)
    $(e_1)^*$: for each $(x, y_1), (y_1, y_2), \ldots, (y_n, y)$ such that $t = (x, y)$ and each pair is in $e_1$
            **choose** some pair $(y_i, y_{i+1})$; deleteTuple($\langle y_i, y_{i+1}\rangle, e_1$)
```

4.4 Proof of correctness

Proof of Theorem 2 Theorem 2 follows from the following lemma about $\mathbb{A}_p$.

Lemma 6. *Let p be a predicate; let $\mathbb{A}_p$ be the non-deterministic procedure constructed from p by Definition 5. Then for every instance I and binding η for the parameters of p:*

1. *Every computation of $\mathbb{A}_p$ terminates on I under η, and if $\mathbb{A}_p$ returns an instance I', we have $(I, I') \in [\![p]\!]_\eta$;*
2. *If there is an instance I' such that $(I, I') \in [\![p]\!](\eta)$ then $\mathbb{A}_p$ will not fail.*

Proof of the lemma. For the first claim, first note that algorithm $\mathbb{B}_p$ proceeds by primitive recursion over the body of the predicates and algorithms insertTuple and deleteTuple proceed by primitive recursion over the body of expressions. So it suffices to argue that the iteration until fixed point in algorithm $\mathbb{A}_p$ always terminates. But this follows from the fact that we never add or delete the same tuple from a given relation and the total size of the domain we work with never changes. It is easy to see that when $\mathbb{A}_p$ halts without failure it is the case that the body of the predicate has been satisfied.

To establish the second claim we start with a definition. Given instances I and I' let us say that instance J is an (I,I')-*approximation* if $I - J \subseteq I - I'$ and $J - I \subseteq I' - I$. We abuse notation slightly here: these calculations are done on a per-relation basis. Intuitively J is an (I,I')-approximation if J can be obtained from I by making *some* of the inserts and deletes that transform I into I'. Note that I is an (I,I')-approximation, as is I'. Now the second claim follows from the fact that, for initial instance I and chosen I' with $(I,I') \in [\![p]\!](\eta)$, whenever algorithm $\mathbb{B}_p$ is called (by $\mathbb{A}_p$) when the current value of the poststate is an (I,I')-approximation then there is a computation of $\mathbb{B}_p$ that (i) does not fail, and (ii) updates the poststate so that it still is an (I,I')-approximation. In particular $\mathbb{A}_p$ will never fail. $\square$

Complexity There is nothing interesting that can be said about the run-time complexity of $\texttt{code}(p)$ since it depends on the nature of the predicate p, and p can be an arbitrary predicate. On the other hand it is natural to ask about the complexity of $\texttt{code}()$ itself. In other words, what is the running time *of Alchemy's code generation algorithm?* Since $\texttt{code}(p)$ comprises a backtracking wrapper around the algorithm $\mathbb{A}_p$ the question is essentially the same as asking: what is the complexity of building the text of algorithm $\mathbb{A}_p$ from the text of predicate p? It is easy to see that this is linear in p. Note in particular that the procedures insertTuple and deleteTuple do not depend on p at all.

5 Related Work

For an extensive discussion of previous research relevant to the Alchemy project itself we refer the reader to the related work section in [7]. The relationship of the present paper to the previous work on Alchemy is as follows. In [7] we did not handle the relational difference operator, we did not treat Skolemization, and our correctness result was only for a subset of Alloy predicates (those admitting "homogeneous" implementations as defined there). But most importantly, the treatment of when relation names were evaluated in the pre-state and when in the post-state was ad-hoc: in the current paper this important semantic decision rests on the secure foundations of the work in [4]. This allows us to prove a true soundness and completeness theorem (Theorem 2) for our code-generation algorithm.

References

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

[2] José A. Blakeley, Per-Åke Larson & Frank Wm. Tompa (1986): *Efficiently Updating Materialized Views*. In: Carlo Zaniolo, editor: *SIGMOD Conference*. ACM Press, pp. 61–71. Available at `http://doi.acm.org/10.1145/16894.16861,db/conf/sigmod/BlakeleyLT86.html`.

[3] Vanessa P. Braganholo, Susan B. Davidson & Carlos A. Heuser (2004): *From XML View Updates to Relational View Updates: old solutions to a new problem*. In: Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley & K. Bernhard Schiefer, editors: *VLDB*. Morgan Kaufmann, pp. 276–287. Available at `http://www.vldb.org/conf/2004/RS7P3.PDF`.

[4] Theophilos Giannakopoulos, Daniel J. Dougherty, Kathi Fisler & Shriram Krishnamurthi (2009): *Towards an Operational Semantics for Alloy*. In: *Proc. 16th International Symposium on Formal Methods*. To appear.

[5] Daniel Jackson (2006): *Software Abstractions*. MIT Press.

[6] Daniel Jackson & Jeanette Wing (1996): *Lightweight Formal Methods*. *IEEE Computer* .

[7] Shriram Krishnamurthi, Daniel J. Dougherty, Kathi Fisler & Daniel Yoo (2008): *Alchemy: Transmuting Base Alloy Specifications into Implementations*. In: *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. pp. 158–169.

[8] J. Michael Spivey (1992): *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition.