Towards an Operational Semantics for Alloy

Theophilos Giannakopoulos,[1] Daniel J. Dougherty,[1]
Kathi Fisler,[1] Shriram Krishnamurthi[2]

[1] Department of Computer Science, WPI
[2] Computer Science Department, Brown University

Abstract. The Alloy modeling language has a mathematically rigorous denotational semantics based on relational algebra. Alloy specifications often represent operations on a state, suggesting a transition-system semantics. Because Alloy does not intrinsically provide a notion of state, however, this interpretation is only implicit in the relational-algebra semantics underlying the Alloy Analyzer. In this paper we demonstrate the subtlety of representing state in Alloy specifications. We formalize a natural notion of transition semantics for state-based specifications and show examples of specifications in this class for which analysis based on relational algebra can induce false confidence in designs. We characterize the class of facts that guarantees that Alloy's analysis is sound for state-transition systems, and offer a sufficient syntactic condition for membership in this class. We offer some practical evaluation of the utility of this syntactic discipline and show how it provides a foundation for program synthesis from Alloy.

1 Introduction

Alloy [1], a popular relational modeling language, provides a syntax reminiscent of class-based programming languages, and its semantics is essentially equivalent to first-order logic with transitive closure. The language is accompanied by an Analyzer; this explores whether a specification has models through compilation into SAT problems and checking for satisfiability. Users can employ a graphical browser to explore instances of models and counter-examples to claims.

Though Alloy relations are powerful enough to encompass many common modeling techniques, Alloy does not have a native executable or machine model. For instance, the Alloy book says:

> *Typically an instance represents a state, or a pair of states (corresponding to execution of an operation), or a execution trace. The language has no built-in notion of state machines, however, ...*
> —*Software Abstractions* [1, page 258]

This is in contrast to B [2] and Z [3], for each of which a notion of state machine is built into the language. Alloy's flexibility is one of its main selling points: it supports a variety of idioms. However, this means the user of Alloy must always be vigilant: they must first choose an idiom and then ensure that they are constantly faithful to it. The language itself does not provide any special support for encoding or checking conformance to specific idioms. Furthermore, failure to adhere is punished not explicitly

but implicitly: in the best case through unexpected outcomes, and in the worst case by incorrect decisions based on the Analyzer's output.

Our first contribution is to show that representing state in Alloy specifications is more subtle than it appears at first glance. We present what might seem to be the obvious operational semantics, the one that a designer would intuit based on the descriptions in, for instance, the Alloy book. But we show that this fails: there are specifications in this class that are very naturally viewed as representing executions whose logical (Alloy) semantics is not faithful to the operational semantics. The consequences of this misalignment are drastic: there are situations in which the Alloy Analyzer will unavoidably fail to report invalid assertions about the code and situations in which the Analyzer will give the designer spurious simulations of specified operations that cannot in fact be implemented.[3]

Based on this analysis, we offer a proposal to rectify the situation. Concretely, we give a characterization of the class of facts for which we can guarantee that Alloy's analysis is sound for state-transition systems, and we offer a sufficient syntactic condition on the form of facts that guarantees that they are in this class.[4]

Experienced Alloy users might argue that they would not be stumped by these examples (though in our experience, even expert Alloy users do not immediately spot the problems). One shouldn't, however, have to be an expert to use a tool safely. We identify the difficulties and explain why things go wrong, and most importantly prescribe a discipline which, if followed, ensures that specifications will not go wrong. We give a precise definition of a state-based modeling idiom with accompanying guarantees, obeying the discipline "satisfiability iff implementability".

Specifications of stateful systems are useful in their own right, and they would be especially useful if they can support not only analysis but also synthesis of executable code. A synthesizer must, however, maintain a sound relationship between transition system specifications and the executable code it produces. This is especially interesting to us due to our prior work on Alchemy [4], a synthesizer that generates executable libraries over databases from Alloy specifications. Our observations while designing Alchemy about the difficulties of pinning down the meaning of stateful specifications inspired this work. But it should be stressed that the problem of reconciling the denotational and operational semantics of a language like Alloy is of fundamental importance to analysis itself, and is independent of any attempt at automatic code generation.

Contributions To summarize:

- we formalize a natural way to extract transition-system executions from relational-algebra instances;
- we show examples of specifications in this class for which analysis based on relational algebra can induce false confidence in designs;

[3] It is important to note that these are mismatches relative to the *semantics* of Alloy [1, Appendix C] and independent of the bounded-scope used by the Analyzer.

[4] *Facts* are statements used to eliminate invalid models—and hence always true in the resulting models—whereas *assertions* are statements that may be true or false.

– we characterize of the class of facts that guarantees that Alloy's analysis is sound for state-transition systems and offer a sufficient syntactic condition for ensuring this behavior; and
– we offer some practical evaluation of the utility of this syntactic discipline.

A by-product of these contributions is a firm foundation for establishing correctness of a synthesizer for state-based specifications [4, 5].

2 Examples

We use a series of examples to illustrate the potential pitfalls in analysis and modelling of specifications with both relational and stateful interpretations. Figure 1 shows a sample Alloy specification. Signatures define domains and relations over domains: this example defines two domains (*State* and *Data*) and a relation *lastUsed* that maps each element of *State* to an element of *Data* (the domains are treated as unary relations): such a collection of domains and relations determines an *instance*, or for emphasis, a *relational algebra instance*. Facts capture closed formulas that must hold of every instance of the domains and relations specified through signatures. A common idiom for stateful specifications uses predicates to model operations over pre- and post-instances of some state object (a prime conventionally connotes the post-state): this example contains an operation *updateLastUsed* that caches the last datum accessed.

We are interested here in examples in which the Alloy Analyzer (which enforces the relational semantics) yields results that contradict stateful interpretations of the example. The Analyzer supports two kinds of analysis: simulation (running a predicate to obtain a satisfying instance) and checking (verifying that an assertion is valid of all instances). Both are important: as Jackson notes [6, page 4], simulation catches errors of overconstraint, while checking detects underconstraint. The soundness of both forms is essential to Alloy's contributions: quoting Jackson [op cit., page 16], "The analysis is guaranteed to be sound, in the sense that a model returned will indeed be a model. There are therefore no false alarms, and samples are always legitimate (and demonstrate consistency of the invariant or operation)".

In the context of stateful interpretations, simulating a predicate (such as *updateLastUsed* from Figure 1) should correspond to the execution of some code that induces the effect of the predicate (updating the cache). Notions of satisfiability and implementability for predicates are therefore at the heart of our explorations. While formal definitions are given later (Section 3), for now we rely on the following informal characterizations. Let p be a predicate (for example *updateLastUsed* in Figure 1) in a specification $\mathcal{A}$; p has a set of parameters (for example s, s', and d in *updateLastUsed*) and a *body* (the remainder of the predicate text). We say that p is *satisfiable* if there is a relational algebra model of the facts of the specification and a binding of the parameters to values such that the body of p holds. We say that p is *implementable* if, when viewed as a procedure, it can be realized as a transition—between nodes bound to s and s'—in a transition system in which each node is an instance satisfying the facts.

Suppose we ask the Alloy Analyzer to check the *newStamp* assertion of Figure 1. This assertion is not valid: there is nothing in the specification as written that requires

```
sig State {lastUsed : Data}
sig Data {stamp : Clock}
sig Clock {}

// remembering a new most recently used value
pred updateLastUsed [s, s' : State, d : Data] {
   s'.lastUsed = d and s.lastUsed != d}

// statically inconsistent with updateLastUsed
fact storeOne {#lastUsed = 1}

// not valid
assert newStamp { all s, s' : State| all d : Data |
   updateLastUsed [s,s',d] implies s'.d.stamp != s.d.stamp}
```

Fig. 1. An Alloy specification that is implementable but not satisfiable.

stamps to be fresh. But rather than generate a countermodel to the assertion, the Analyzer will report that *newStamp* "may be valid." Since the Analyzer always works with a bounded domain size it is properly modest in suggesting validity. But in fact the Analyzer cannot find a countermodel for *newStamp* even in principle. The problem is that the *updateLastUsed* predicate is unsatisfiable. Thus, since no instance satisfies the antecedent of the implication in *newStamp*, the assertion is in fact valid.

Why is *updateLastUsed* not satisfiable? At first glance, it seems to be an entirely reasonable predicate specification. And indeed the natural implementation of this specification seems to obey the predicate body as well as the *storeOne* fact, which expresses the constraint that exactly one item should be cached via *lastUsed*: each call to *updateLastUsed* replaces the value of *lastUsed* in the current state. Unfortunately, the specification as written is not satisfiable because the *storeOne* fact captures more than the author intended. Under the Alloy semantics, the fact constrains instances to a total of one *lastUsed* value across *all* states, not *per* state. Indeed, the effect of writing #*lastUsed* = 1 is to constrain Alloy models of this specification to conflate what are really two distinct states (pre and post), whereas in an imperative implementation only one is ever active at a time. (If the author had written *#lastUsed* = 1 as a "signature fact", that is, within the paragraph declaring *State*, then under the Alloy semantics this constraint would be treated as syntactic sugar for the constraint that for all States *s*, *s.lastUsed* has one item. The above scenario would arise if an author mistakenly moved a signature fact into a standalone fact.) This highlights the first pitfall to using the Analyzer to reason about a stateful system:

> **False confidence in assertion-checking:** the Analyzer cannot generate countermodels for invalid assertions about implementable predicates that are unsatisfiable under the facts.

Figure 2 shows a richer model of caches, in which each state contains a cache that maps keys to data. Keys are unique within each state. Adding a cache entry with a new

```
sig State {cache : set Key → Data}
sig Key {}
sig Data {}

fact cacheKeysUnique {
   all s : State | no k : Key | #s.cache[k] > 1}

// cache d under a key that is not used in s
pred addEntryNewKey [s, s' : State, d : Data] {
   some k : Key | no s.cache[k] and
   s'.cache = s.cache + k→d}

fact oddCached {#cache = 1 or #cache = 3 or #cache = 5}
```

Fig. 2. An Alloy specification that is satisfiable but not implementable.

key inserts a datum into the cache using a key that was unused in the previous state. To limit the cache size, the specification author includes a fact that the number of cache lines must always be a small odd number (we use concrete numbers in light of Alloy's domain-size restrictions). Under Alloy's semantics, this predicate is satisfiable. It is not, however, implementable: using the *addEntryNewKey* operation, the number of cache lines will alternate between being odd and even in successive states. The fact, then, is not an invariant in the implementation. This illustrates another pitfall when reasoning about stateful specifications:

> **False confidence in simulation:** a design can include a predicate that cannot—in the context of the stated facts—correspond to any transition at all, yet this impossibility will go undetected by the analysis, in the sense that the Analyzer will build a satisfying instance without complaint.

These two examples exploit a similar problem: the Alloy specification includes a fact on the full model, rather than just facts on individual states. If the specification happens to talk about multiple points in time, special care must be taken to separate them. Imperative interpretations, in contrast, view only a single state at a time. In effect, the implementation views facts at a different level of granularity than the specification.

The lack of alignment between implementability and satisfiability under conventional relational algebra semantics exposes potentially serious problems for lightweight formal methods. Implementability without satisfiability implies that designers cannot reason about their designs through their specifications (once a model is unsatisfiable, the designer does not get useful feedback about its other properties). Satisfiability without implementability implies that assertions verified about the model might not hold of an actual implementation, so the verification effort has been wasted.

3 Transition Semantics

In order to formalize (and address) the problems with assertion checking over unimplementable predicates, we need a transition-system semantics for relational specifications, as well as characterizations of relational specifications for which those semantics yield meaningful results.

An Alloy *specification* $\mathcal{A} = (Sigs, Facts, Preds)$ is given by a set of signatures, facts, and predicates. It will be convenient to assume that all constraints on signatures are expressed as elements of *Facts* (this is without loss of generality).

The signature and facts in a specification provide the setting and constraints under which predicates and assertions are explored.

3.1 State-Based Frameworks

In a state-based modeling setting the most typical use of facts is to express state invariants, and this will be reflected in the semantics we define. But facts are not *necessarily* state-invariants: a naturally-occurring example is the use of trace constraints. For example one might impose the constraint that certain properties hold in the initial state of a system (such a property is not an invariant) or the constraint that all transitions must be an operation specified by one of the predicates (such a property is not a property of individual states).

So a transition system must obey two different kinds of constraints: local constraints on the states, and global constraints across states. We recognize this distinction in the following definition.

Definition 1. *An Alloy* framework $\mathcal{F} = (Sigs, Facts, Inv)$ *is given by a set of signatures, a set of facts, and a distinguished subset of the facts, the "state invariant" facts.*

This designation of certain facts as state invariants is not part of the Alloy language definition. So for each Alloy specification the semantics we develop in this paper is parametrized by the author's intentions as to which constraints in the set *Facts* are to be treated as invariants.

Our work on Alchemy [4] shows that identifying the updates required by relational specifications is the key challenge to interpreting Alloy specifications statefully. In particular, the relational semantics of arbitrary terms over the pre- and post-state atoms in predicates allow substantial leeway in how to perform an update. This work aligns relational and stateful interpretations using some restrictions on signatures and facts. These require some terminology:

Fix a distinguished signature, which we will call *State*. We call an Alloy relational type *immutable* if it has no occurrences of the *State* signature.

Definition 2. *An Alloy framework is a* state-based framework *if the type of each declared relation name is either immutable or is a sum of types of the form* $State \to A_1 \to \cdots \to A_n$ *where each* A_i *is immutable.*

The restriction that the *State* signature be the leftmost sig occurring is a matter of notational convenience; the essential requirement is that no relation name have more than

one occurrence of *State*. For a formal treatment of the notion of type of a relation name see Edwards et al. [7].

Trace-based reasoning over states is typically done in the context of the Alloy *util/ordering* module: if the specification orders the *State* with this module then the functions **first**, **next**, and **last** are available. In this case the types of these functions violate the conditions in Definition 2. But such specifications are still considered "state-based" since **first**, **last**, and **next** are not "declared" in the specification. Indeed the semantics of these functions will be hard-wired into the transition semantics below.

For the rest of the paper we assume that all specifications are state-based.

3.2 Transition Systems

We base our operational semantics on transition systems. In anticipation of the use of the *util/ordering* module, we define ordered transition systems.

To avoid subtleties having to do with underlying data models we take the states in our transition systems to be relational algebras precisely of the sort that the Alloy Analyzer constructs; these can be viewed as database instances. In this case the transitions between states are the obvious database updates transforming one state to another.

If we are to think of the individual instances as each representing one state of the application we should certainly expect that each of the instances has a unique atom in the extension of the *State* signature name. And if we take seriously the notion that the *State* signature is supposed to capture the data that changes, we should require that the extensions of the immutable relation names should be the same in each state. This motivates the notion of a *coherent* set of instances, a set of instances comprising the set of nodes in a transition system.

Definition 3. *A set $\mathcal{Q}$ of instances is said to be* coherent *if*

- *each immutable relation name r has the same interpretation in each instance: $\forall I, I' \in \mathcal{Q} \,.\, I(r) = I'(r)$,*
- *each instance has a unique atom in the State signature: $\forall I \in \mathcal{Q}. |I(State)| = 1$, and*
- *no two instances have the same state atom: if $I \neq I'$ then $I(State) \neq I'(State)$.*

We do not assume that the set $\mathcal{Q}$ is finite in this definition.

Definition 4. *Let $\mathcal{F} = (Sigs, Facts, Inv)$ be a framework. A* transition system *$\mathcal{T}$ over the signatures of $\mathcal{F}$ is a pair $\langle \mathcal{Q}, \delta \rangle$, where $\mathcal{Q}$ is a coherent set of relational algebra instances whose signature is given by Sigs, and $\delta \subseteq \mathcal{Q} \times \mathcal{Q}$ is a transition relation.*

$\mathcal{T}$ is an ordered transition system *if it has a designated linear ordering **next** on states and distinguished **first** and **last** states.*

Note that in the definition above we have not insisted that $\mathcal{T}$ obey the constraints imposed by the facts of $\mathcal{F}$. In fact we need to do some work to make sense of that notion, since transition systems are not themselves relational algebras, and so do not come equipped with a way to evaluate relational algebra expressions and formulas. The result of this work will be Definition 7.

We turn to the task of defining how to interpret expressions and formulas over $\mathcal{F}$ in a transition system. To do so we use a natural construction that allows us to treat a finite transition system as a single relational-algebra instance.

Definition 5 (Merging). *Let $\mathcal{Q}$ be a finite coherent set of instances. The instance $\sqcup\mathcal{Q}$ is given by setting, for each relation name r,*

$$\sqcup\mathcal{Q}(r) = \bigcup\{I(r) \mid I \in \mathcal{Q}\}$$

When $\mathcal{T} = (\mathcal{Q}, \delta)$ is a transition system it will be convenient to write $\sqcup\mathcal{T}$ for $\sqcup\mathcal{Q}$.

Observe that the notion of merging is well-defined only by virtue of our assumption that the instances in question are coherent. (Indeed, we note that the "$\bigcup$" in Definition 5 is somewhat of a red herring for immutable relations, since they have the same value in each instance of $\mathcal{Q}$.)

Definition 6. *Let $\mathcal{T} = (\mathcal{Q}, \delta)$ be a transition system.*

The value $\mathcal{T}(e)$ of an expression e is the set of tuples that is the value of e in the relational algebra $\sqcup\mathcal{T}$.

Say that sentence σ is true in $\mathcal{T}$, written $\mathcal{T} \models_{TS} \sigma$ if σ is true in $\sqcup\mathcal{T}$ in the ordinary relational algebra sense, that is, if $\mathcal{T} \models_{RA} \sigma$

We are now ready for the key definition for the transition-system semantics of a state-based framework, the notion of a *transition system for framework $\mathcal{F}$*.

Definition 7. *Let $\mathcal{F} = (Sigs, Facts, Inv)$ be a framework. A* transition system for *$\mathcal{F}$ is a transition system $\mathcal{T}$ over the signatures of $\mathcal{F}$ such that*

- *each node I satisfies each fact in Inv, and*
- *$\sqcup\mathcal{T}$ satisfies each fact not in Inv.*

If $\mathcal{F}$ includes an ordering on State then we require that $\mathcal{T}$ be an ordered transition system.

Definition 7 highlights the distinction between the facts that are intended to be viewed as state invariants and those that play the role of global constraints on the system. Assertions and the bodies of predicates that define operations must obviously be able to make reference to more than one state and so must be evaluated globally, that is, over the merge of the nodes as described in Definition 5.

The Transition Semantics of Predicates For those predicates written in order to define operations we may define their transition-system semantics as follows.

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 because predicates typically underspecify actions: different implementations of a predicate can yield different outcomes I' on the same input I. These should all be considered acceptable as long as the relation between pre- and post-states is described by the predicate.

Definition 8. *Fix an Alloy framework $\mathcal{F}$, and let p be a predicate over $\mathcal{F}$ with the property that p has among its parameters exactly two variables `s` and `s'` of type State. Let $\mathcal{T}$ be a transition system for $\mathcal{F}$. The meaning $[\![p]\!]^{\mathcal{T}}$ of p in $\mathcal{T}$ is the set of triples $\langle I, I', \eta \rangle$ 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';*
- $\sqcup\{I, I'\}$ *makes the body of p true under the environment* η.

We say that predicate p is implementable *if there exists a transition system $\mathcal{T}$ for $\mathcal{F}$ such that $[\![p]\!]^{TS} \neq \emptyset$.*

Our definition of "implementable" might appear odd at first glance. One might initially expect that an implementable predicate be defined as one for which there exists code that carries any I to an I' such that (I, I') makes the body of p true. Further consideration suggests that that is too much to ask: we should only insist that our code behave properly on nodes I that satisfy the pre-conditions of the predicate. But that won't work either, since there is no well-defined notion of "pre-condition" in an Alloy specification: in the rich language of Alloy predicates primed and unprimed elements mix freely within expressions and formulas. In this light the definition of "implementable" above seems to be the most restrictive reading that encompasses the intuitively implementable operation specifications.

3.3 Transition Systems from Instances

Having developed an "abstract" general notion of transition system for a framework the obvious question presents itself: what is the relationship of this class of structures to the relational algebra instances that are the foundation of Alloy?

The relationship is straightforward. In a natural way we can extract transition systems from relational algebra instances, formalizing the mental construction that Alloy users do whenever they are confronted with an instance for an analysis constraint in a state-based framework.

An instance that is intended to capture a transition typically has two atoms in the extension of the *State* signature and we read off the pre- and post-instances by projecting over these two atoms. Similarly for an instance modeling a trace: we think of each state atom in the instance as being an index into the part of the instance relevant to a particular transition-system node. (This is exactly what the standard Alloy visualization does, if one were to select a projection on *State*.) The next definition formalizes this intuition. It is convenient for our purposes to do this operation while retaining the state-atom, so it corresponds to an ordinary database join.

Definition 9 (Localizing). *Let I be an instance for a state-based specification and let $a \in I(State)$. The instance I_a is defined by*

- $I_a(r) = I(r)$ *when r is an immutable relation name;*
- $I_a(r) = a \bowtie I(r)$ *when r is a mutable relation name.*

Here $\bowtie$ is standard database join, so that $a \bowtie I(r)$ is the set of tuples in $I(r)$ whose entry in the State-column is a.

So any instance yields a transition system. What about the converse? We have seen in Definition 5 how to merge a transition system to obtain an instance; it remains to observe that merging and localization interact smoothly.

Lemma 10. *Merging and localizing are mutual inverses. That is,*

- *merging undoes localization: if I is an instance with $I(State) = \{a_j \mid j \in J\}$ then $\sqcup\{I_{a_j} \mid j \in J\} = I$;*
- *localization undoes merging: If $\mathcal{Q}$ is a finite coherent set of instances, then the set of instances obtained by localizing $\sqcup\mathcal{Q}$ is $\mathcal{Q}$: $\{(\sqcup\mathcal{Q})_a \mid a \in (\sqcup\mathcal{Q})(State)\} = \mathcal{Q}$.*

It would, however, be a mistake to conclude from Lemma 10 that transition systems can be identified with relational-algebra instances. The central point is that *there is no reason to expect facts to be preserved* by merging or by localizing. And the facts that are viewed by the designer as state invariants are in consequence treated specially by our semantics: see Definition 7.

Example Consider the relations in Figure 1. An Alloy instance would have this form:

State	= {s0, s1, ...}
Data	= {d0, d1, ...}
Clock	= {t0, t1, ...}
lastUsed	= {(s0, d0), (s1, d1), ...}
stamp	= {(d0, t0), (d1, t1), ...}

When this instance is systematically localized at the values in *State* we get a family of instances—

State	= {s0}
Data	= {d0, d1, ...}
Clock	= {t0, t1, ...}
lastUsed	= {(s0, d0)}
stamp	= {(d0, t0), (d1, t1), ...}

State	= {s1}
Data	= {d0, d1, ...}
Clock	= {t0, t1, ...}
lastUsed	= {(s1, d1)}
stamp	= {(d0, t0), (d1, t1), ...}

...

—which form the nodes in a transition system.

4 Achieving Confidence in Analysis

The notions of localization and merging shed light on the examples from Section 2. Consider the specification in Figure 1. We observed that the predicate *updateLastUsed* was intuitively implementable; it is not hard to see that it is indeed implementable in the sense of Definition 8. But the predicate is not satisfiable. We understood intuitively that the source of the difficulty is the fact *storeOne*; now we can make the precise observation that *the fact storeOne is not preserved under merging*.

Next consider the specification in Figure 2. We observed that the predicate *addEntryNewKey* was (intuitively) not implementable; indeed it is not implementable in the sense of Definition 8. But the predicate is not satisfiable. This time the reason is the fact *oddCached*; this fact precludes implementation. Now we note that *the fact oddCached is not preserved under localization*.

These phenomena are perfectly general, as we summarize here.

Theorem 11. *Let $\mathcal{F} = (Sigs, Facts, Inv)$ be a framework.*

1. *The following are equivalent:*
   - *The sentences in Inv are preserved by arbitrary merging;*
   - *Every implementable predicate over $\mathcal{F}$ is satisfiable.*
2. *The following are equivalent:*
   - *The sentences in Inv are preserved by arbitrary localization.*
   - *Every satisfiable predicate over $\mathcal{F}$ is implementable.*

We have noted that the mismatch between satisfiability and implementability manifests itself in practical terms as an obstacle to having confidence in constraint-solving analyses. Specifically, confidence in assertions-checking arises precisely when countermodels to assertions in the relational algebra semantics encode countermodels in the transition semantics. This in turn means that validity in the transition system semantics implies validity in the relational algebra semantics. Dually, confidence in simulation (of predicates) arises from a guarantee that a relational algebra instance of a predicate does indeed correspond to a transition.

The next definition and result formalize these remarks.

Definition 12. *Let $\mathcal{F}$ be a framework and σ a sentence.*

- *We write $\mathcal{F} \models_{RA} \sigma$ to mean that σ holds in every relational algebra instance for $\mathcal{F}$.*
- *We write $\mathcal{F} \models_{TS} \sigma$ to mean that σ holds in every transition system over $\mathcal{F}$, in the sense of Definition 6.*

Then to say we can have confidence in assertions-checking in a framework $\mathcal{F}$ is to say that for any σ, $\mathcal{F} \models_{TS} \sigma$ implies $\mathcal{F} \models_{RA} \sigma$. To say we can have confidence in simulation in a framework $\mathcal{F}$ is to say that for any σ, $\mathcal{F} \models_{RA} \sigma$ implies $\mathcal{F} \models_{TS} \sigma$.

Proposition 13. *Let $\mathcal{F} = (Sigs, Facts, Inv)$ be a framework.*

1. *The following are equivalent:*
   - *the sentences in Facts are preserved by arbitrary merging.*
   - *for any σ, $\mathcal{F} \models_{TS} \sigma$ implies $\mathcal{F} \models_{RA} \sigma$. (We can have confidence in assertions-checking.)*
2. *The following are equivalent:*
   - *the sentences in Facts are preserved by arbitrary localization.*
   - *for any σ, $\mathcal{F} \models_{RA} \sigma$ implies $\mathcal{F} \models_{TS} \sigma$. (We can have confidence in predicate simulation.)*

A Sufficient Condition for Reliable Analysis

It may be illuminating to identify the preservation of properties under localization and merging as being at the heart of sound analysis, but since they are described in semantic terms they do not in themselves provide much guidance to the specification author. We next present a simple syntactic criterion that ensures that analysis can be trusted.

The difficulties explored in this paper all arise from the following dichotomy: certain expressions and formulas are naturally interpreted *in individual states* from the point of view of the implementer yet are interpreted *globally* by Alloy. The latter condition

occurs because all states relevant to a formula being modeled are encoded into each individual Alloy instance.

Observe that for an immutable relation name *r*, the meanings of *r* in the various nodes of $\mathcal{T}$ are identical since Q is coherent. On the other hand, the interpretation mutable relations will of course vary across nodes. As a consequence, if *e* is an expression involving mutable relations, the value of *e* computed at a particular node *I* in $\mathcal{T}$ will in general be different from the "global" value $\mathcal{T}(e)$, and similarly for formulas. There is no surprise here, but this points to the need for care in defining the semantics of predicates and assertions since these typically involve formulas explicitly referring to more than one state. Indeed, it might suggest that our device of defining semantics in $\mathcal{T}$ in terms of standard semantics in $\sqcup\mathcal{T}$ does not capture intended usage.

These considerations motivate the next definition.

Definition 14 (Absoluteness). *Let e be an expression with at most a single State variable s occurring (more than one occurrence of s is permitted). Say that e is* absolute *if the following holds for every transition system $\mathcal{T}$. Let I be the unique node of $\mathcal{T}$ such that $I(State) = \mathcal{T}(s)$; then*

$$\mathcal{T}(e) = I(e)$$

So the meaning of an absolute expression survives the pun of viewing a relational algebra instance as representing a fragment of a transition system. Next we give a sufficient condition for expressions to be absolute, and a sufficient condition for facts to be preserved and reflected by the passage from instances to transition systems.

Definition 15 (State-bound expressions). *A* state-bound expression *is one for which every occurrence of a mutable relation name r is within the scope of some state variable s: that is, for each occurrence of r there is a subterm of the form $s.f$ such that r is a subterm occurrence of f.*

A sentence σ *is a* state-bound sentence *if*

- *every expression occurring in* σ *is a state-bound expression, and*
- *either* σ *has no occurrence of State variables, or is of the form* ***all*** *s: State . B with s the only State variable possibly occurring in B.*

For example, in Figure 2, the occurrence of *s.cache* in the fact *cacheKeysUnique* is state-bound; but the occurrence of *cache* in the fact *oddCached* is not state-bound. Of course, an expression involving only immutable relations is automatically state-bound.

Theorem 16. *State-bound expressions with at most one state-variable are absolute.*

State-bound facts are preserved by localizing and by merging.

As an immediate consequence of Proposition 13 and Theorem 16 we obtain the following sufficient condition for achieving confidence in both assertions-checking and simulation.

Corollary 17. *Let $\mathcal{F}$ be a framework whose associated set of facts is state-bound. Then a predicate is satisfiable if and only if it is implementable.*

Constraints in Predicate Bodies Our results have so far constrained the form of facts, but not of predicates. This is perhaps surprising, as stateful predicate specifications often contain clauses that seem similar to facts (such as those capturing pre-conditions, post-conditions, or framing conditions). If is therefore natural to ask what happens if a predicate body violates our state-bound discipline. The answer is interesting, and sheds some additional light on the nature of the transition system semantics we have defined.

As a concrete example, let us revisit the *updateLastUsed* predicate from Figure 1, but with the problematic fact "inlined" into the body of the predicate.

```
// remembering a new most recently used value
pred updateLastUsed [s, s' : State, d : Data] {
  #lastUsed = 1 and
  s'.lastUsed = d and s.lastUsed != d}
```

This predicate poses no problems in assertions-checking or in the relationship between satisfiability and implementability. With the constraint that *lastUsed* is a singleton, the *updateLastUsed* predicate becomes unimplementable as well as unsatisfiable. This is a consequence of interpreting the predicate body using relational semantics over the merge of the individual states. As the inlined fact will never be true in any merged instance, the predicate is not satisfiable in any transition system. Although this may seem odd, it is consistent with the observations made in the discussion prior to Definition 6.

A similar analysis applies to the situation where a non-state-bound sentence that is not preserved under localization is used in a predicate body.

5 Advice to Alloy Users

Our results identify a subset of (or idiom over) Alloy specifications that capture transition systems without sacrificing accuracy of analysis in the relational semantics. Alloy users who wish to write such specifications should adopt two concrete guidelines:

1. Facts intended to capture state invariants must be preserved under localization and merging. Writing such facts either as signature constraints on the *State* signature or as state-bound sentences (Definition 15) ensures this.
2. Relations that are intended to be mutable (in an implementation) must be declared within the *State* signature.

Violating these rules can yield unreliable results from simulation or assertions-checking relative to the transition semantics defined in this paper.

As an example of these guidelines, imagine a designer trying to model a simple social networking application. The model captures each person's friends, as well as the members of the social network using two signatures:

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

The designer proposes the following predicate to capture making one person (*p2*) a new friend of another (*p1*) (where & denotes intersection and + denotes union):

pred *makefriends* (*s*, *s*' : *SocNetwork*, *p1*, *p2* : *Person*) {
p2 **not in** *p1.friends* **and**
(*s*'.*members*) & *p1.friends* =
(*s.members*) & *p1.friends* + *p2*}

This predicate violates guideline 2: the predicate is trying to update the *friends* relation, but that relation is not a component of the *SocNetwork* signature (which provides the *State* signature for this model). Instead, the designer should write the model as

sig *Person* {}
sig *SocNetwork* {*members* : **set** *Person*,
friends : *Person* → *Person*}

pred *makefriends* (*s*, *s*' : *SocNetwork*, *p1*, *p2* : *Person*) {
p2 **not in** *s.friends.p1* **and**
s'.*friends*[*p1*] = *s.friends*[*p1*] + *p2*}

This example also helps illustrates a subtlety in guideline 1. Imagine adding the constraint that all friends are also members. The guideline (specifically, Definition 15) suggests writing this fact as **forall** *s: State* | *s.friends* **in** *s.members*, rather than the logically equivalent, and admittedly simpler, *friends* **in** *members*. Given the equivalence, the latter form is also preserved under localization and merging, despite the syntactic mismatch. This reflects the syntactic nature of guideline 1. In practice, we have not found this difference to be a problem, as discussed in the next section.

6 Validation: From Semantics to Synthesis

Until now, we have presented an idiomatic sub-language of Alloy for which we can define a coherent operational semantics. We now discuss two practical issues: usability in the sense of expressiveness for specification, and the potential for synthesis.

Usability While usability can be hard to evaluate in an unbiased manner, we can at least ask whether existing specifications fall within the idiom defined here. In addition to several small and synthetic specifications, we are aware of at least two large specifications that fall within this language. The first is a specification of the access-control and execution behavior of Continue [8], a conference management application in use by several actual conferences (`continue2.cs.brown.edu`). Though Continue is co-authored by the fourth author, the specification was written by students unrelated to the project several years before the present research. Despite this, their specification nicely falls entirely within our subset (with all facts treated as state invariants).

The second such specification is for a new collaborative, Web-based programming environment that is under construction. That specification also has several diverse elements: operations for content creation, sharing, hiding, rating, commenting, and so forth. Again, the author of the specification was working entirely independently of this research and was unaware of it. That specification has one fact, of the form **forall** *x:X* **exists** *s:State* ..., that falls outside our subset. We interviewed the author to learn that this fact was included *only* to constrain the space of models to improve performance of

the Analyzer; it does not capture a constraint of the logical model (and thus would not be required for code synthesis).

Synthesis These specifications can also be processed by the Alchemy synthesizer. It cannot be re-used as a black-box, however, because the operational behavior of Alchemy is overly broad; for instance, given the specification

sig *State* $\{r : A\}$
sig B $\{t : A\}$
pred p $\{s.r$ **in** $s'.\mathrm{r} + B.t\}$

Alchemy would be free to modify t (as we discuss in section 7). By restricting the operations Alchemy can generate, we can therefore obtain a synthesizer for specifications in the language of this paper whose generated code behaves consistently with the semantics defined here. Furthermore, Alchemy already produces systems with reasonable performance, at least for prototyping purposes [4]; by restricting the space of synthesized operations, we would be further improving its performance.

7 Related Work

We can view our work as providing an *adequate* semantics for Alloy. The notion of adequacy is usually credited to Plotkin's seminal work on the treatment of LCF as a programming language [9]. In our case, adequacy is a relationship between the denotational world of models and analysis, and the operational world of the implementation.

DynAlloy [10] originates, as does our work, from the observation that Alloy has only an "implicit" notion of operational semantics. Their response is different: they add another primitive notion, that of *actions,* to the language, together with a way of making partial correctness assertions. The emphasis in the DynAlloy work is on expressiveness of, and analysis of specifications in, their expanded language. In contrast, our focus is on the semantics of the common state-based idiom as expressed in pure Alloy.

Massoni, Gheyi and Borba [11] address the question of "conformance" between object-models and programs. They define a notion of "syntactic coupling" (defined in the PVS language) that relates object models with representations of run-time heaps. The main goal is to define and reason about the correctness of refactorings; the emphasis is on preservation of data properties expressed in the specification. They do not analyze the way that Alloy predicates induce operations on data.

Three of the present authors, with Yoo, introduced the Alchemy [4] program synthesizer for Alloy. Due to the lack of a crisp operational interpretation of Alloy, Alchemy relies on ad hoc syntactic criteria to determine the specification author's intent with respect to state changes. In contrast, this paper presents a precise operational characterization, providing a more rigorous formal footing for Alchemy.

Several efforts have tried to relate proofs to running programs. Bates and Constable [12] initiated a significant research program on the extraction of computational context, in the form of programs, from constructive proofs. This effort continues in popular proof assistants such as Coq [13]. Of course Alloy has no notion of proof structure. Nevertheless, we share their desire to have the executable code behave consistently with the outcome of any static analysis.

Our work can be seen as a result toward software synthesis, an effort initiated by Green [14] and Waldinger and Lee [15] and summarized by Rich and Waters [16]. Our prior work [4] discusses in detail the relationship between our approach and others.

Acknowledgments We are grateful to Daniel Jackson for several helpful and detailed conversations about this work, including comments on several drafts of this paper. Michael Butler provided helpful information about the B method. This research is partially supported by the NSF.

References

1. Jackson, D.: Software Abstractions. MIT Press (2006)
2. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
3. Spivey, J.M.: The Z Notation: A Reference Manual. 2nd edn. Prentice Hall (1992)
4. Krishnamurthi, S., Dougherty, D.J., Fisler, K., Yoo, D.: Alchemy: Transmuting base alloy specifications into implementations. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering. (2008)
5. Dougherty, D.J.: An improved algorithm for generating database transactions from relational algebra specifications. In: International Workshop on Rule-Based Programming. (2009)
6. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology **11**(2) (2002) 256–290
7. Edwards, J., Jackson, D., Torlak, E.: A type system for object models. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering. (2004)
8. Krishnamurthi, S., Hopkins, P.W., McCarthy, J.A., Graunke, P.T., Pettyjohn, G., Felleisen, M.: Implementation and use of the PLT Scheme web server. Higher-Order and Symbolic Computation **20**(4) (2007) 431–460
9. Plotkin, G.D.: LCF considered as a programming language. Theoretical Computer Science (1977) 223–255
10. Frias, M.F., López Pombo, C.G., Galeotti, J.P., Aguirre, N.M.: Efficient analysis of DynAlloy specifications. ACM Transactions on Software Engineering and Methodology **17**(1) (December 2007)
11. Massoni, T., Gheyi, R., Borba, P.: A framework for establishing formal conformance between object models and object-oriented programs. Electronic Notes in Theoretical Computer Science **195** (2008) 189–209
12. Bates, J.L., Constable, R.L.: Proofs as programs. ACM Transactions on Programming Languages and Systems **7**(1) (1985) 113–136
13. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2004) Version 8.0.
14. Green, C.C.: Application of theorem proving to problem solving. In: International Joint Conference on Artificial Intelligence. (1969)
15. Waldinger, R.J., Lee, R.C.T.: PROW: A step toward automatic program writing. In: International Joint Conference on Artificial Intelligence. (1969)
16. Rich, C., Waters, R.C.: Automatic programming: Myths and prospects. IEEE Computer **21**(8) (1988) 40–51