Consistent ASM Updates from Atomic Composition

Colin Gordon, Leo Meyerovich, Joel Weinberger, Shriram Krishnamurthi
Brown University Computer Science Department
115 Waterman St.
Providence, RI, USA
{colin, lmeyerov, joel, sk}@cs.brown.edu

ABSTRACT

We propose an approach to the consistent update problem of Abstract State Machines through a correctness preserving composition operator. Inconsistent updates are transparently isolated and cause local failure rather systemic failure. This is achieved by a source-to-source translation rather than changing the semantics of Abstract State Machines, thus preserving findings of previous studies on Abstract State Machines and allowing independently verified modules to be composed. Our approach is motivated by experiences with our complementary case study in describing dynamic security policies with Abstract State Machines in which we found the problem pervasive, stemming from a composition abstraction barrier. Previous efforts for supporting ASM composition are susceptible to the inconsistent update problem and may benefit from our approach.

General Terms

abstract state machines

Keywords

atomicity, transactions, modules, composition, concurrency

1. INTRODUCTION

Abstract State Machines provide a modeling language rich enough to describe many common programs, but coarse enough that sufficiently strong properties can be proved about them. Security policy modeling, an active area of research, may benefit from the results achieved by the ASM community. In recent unpublished work[5], we began to analyze the relational and temporal requirements for our inhouse research conference paper and schedule management web program in use by several conferences [9], seeing a richer modeling environment was necessary relative to some of our previous work[4]. The intuitive specification of security policies for our web programs using typical ASM constructs leads to the classic inconsistent update problem, as to be discussed. We describe a source-to-source translation that achieves a novel notion of atomicity akin to optimistic concurrent transactions in order to prevent this problem and prove the translation's correctness. We also show the same basic idea can be used as a building block for alternative types of composition. Furthermore, by permitting concurrent batch updates except in the exact case of inconsistent updates, we create a basis of comparison for coarser concurrency controls like explicit locking that might be used by programs realizing our policies.

Security models are difficult to express accurately in software. While a certain set of security properties may be intended, it is generally difficult to extract a security model implicitly embedded in software and then verify it. Thus, it might be desirable to define the security model and verify it. This allows one to separate the problems of verifying the accuracy of the security model and the correspondence of a model to the program. Part of the challenge then becomes finding a useful class of models and corresponding representations and verification systems.

Our interest is motivated by development of the Continue Conference Manager web software which provides tools for research conference organization and paper reviewing. The software has an underlying role-based access control policy such that authorization is a function of the history of the conference in terms of items like users and papers being added and removed from the system . We are motivated by a desire to be able to verify the security features of programs following such policies, requiring both completeness and correctness in this domain. The relation-based nature of the Continue Conference Manager security policy fits the Abstract State Machine (ASM) model particularly well. By modeling the Continue security policy as an ASM, we can leverage findings on their verifiability.

However, while our specifications written with ASMs were fairly concise, they were also incorrect - the potential for an inconsistent update manifested itself frequently in our policy's original implementation: we found 20 such bugs in the first hundred lines [5] written before we were aware of the issue. We discuss why special casing such bugs leads to an undesirable specification and is thus not an option. To avoid this difficulty, we phrase the problem as one of composition, define the properties we would like to guarantee, and present an operator that satisfies these properties. We also see potential extensions of this basic technique of transforming programs with respect to inconsistent program updates.

In the following sections, we describe the basic structure of ASMs and then our basic usage of them to model the

Continue security policy, followed by why our naive usage was prone to inconsistent update bugs, as other highly concurrent programs' specifications would be. After the background and motivation, we introduce our basic notion of module composition and the corresponding source-to-source translation sufficient to prevent inconsistent updates. Finally, we show richer composition operators using the original one as a basis and discuss possible future directions as well as related work.

2. BACKGROUND

2.1 ASMs

Abstract State Machines are generalizations over Kripke structures. Instead of having a transition graph with a function labeling nodes with satisfied propositions, each node is a world in which first order logic sentences are true or false. Transitions between nodes thus represent valid updates of relations, not propositions.

Under this interpretation, programs are concise specifications of the transition function. For example, consider the rule

IF $p(\alpha)$ THEN $q(\beta)$

which specifies, for any transition, if p is satisfied for α in a given step, during the next transition, q will be true of β. More general logical statements over the relational data representing the program at that time may be written in the guard or the update body position. Programs are simply the union of rules like the above. With such a representation, a simulation of what we more commonly consider a program would correspond to iterative application of the corresponding ASM program.

To achieve certain complexity results, it is convenient to partition the various relations involved. A transition is intuitively from input relations and state relations to the next set of state relations. Input transitions may only appear in the guard of an update rule. The state relations may be partitioned into static relations, known as database relations, and dynamic relations. Database relations may only appear in guards. The dynamic relations are themselves partitioned into memory relations that can be added and removed from, and output relations that can only be added to. Memory relations may appear on either side of conditionals, while output relations can only appear on the right hand side. The transition function can thus be reduced to a mapping between input relations and memory relations to changes to the memory relations, known as the update set. According to the above rule, for states in which $p(\alpha)$ is satisfied, $q(\beta)$ must be in the update set of any transition out of it. The program thus specifies the updates to relations (addition or removal) after evaluation for the next state. More sophisticated rules are common, such as a nondeterministic choose operator, but for our problem domain of dynamic access control policies, conditionals and our introduced module operator were sufficient, and the richer rules (e.g. the nondeterministic choose operator) are omitted to keep our system's specification in a class of ASMs with a particular complexity[13].

For a more formal treatment relevant to our usage of them, Abstract State Machines, formerly known as Evolving or Dynamic Algebras, are introduced by Gottlob [6], advanced by Gurevich[7] and analyzed by Spielmann[11, 12].

2.2 Continue

The Continue conference management server [9] has been used to orchestrate over 30 conferences and workshops. It supports progress through multiple phases, including paper submission, bidding for paper review assignments, actual paper review assignments, paper reviewing, the PC meeting to determine which papers to include, and then the notification and publication stages. The permissions of any given individual in relation to a particular paper may change over time with respect to occurrences such as conflicts of interest or status changes. Rules can become intricate. For example, in some conferences, reviewers may only be able to view ratings given by other reviewers only if they are on the programming committee, or have already submitted their review and there is no conflict of interest. Additionally, changes to the role of a user must be consistent with security rules; many meetings have reviewers who are also submitters, so this is a common issue.

To use ASMs to model our policies[5], we maintain a memory relation to describe the current conference phases, as well as to describe papers, reviewers, and various derivatives such as conflicts of interests. Input relations are used to model requests such as those to add a paper or advance the conference to the next phase. Our overall policy had some global rules, but was mostly partitioned into large nested rule sets predicated by the phase. This exaggerated the problem of consistent updates to be shortly discussed. Essentially, as we support many actions at each step that can be performed concurrently by different users, we do want an implicitly parallel specification, but need a way of handling conflicting actions which may violate expected behavior of a composed system based on that of its individual parts.

2.3 Relational Transducers

Relational Transducers were introduced to provide a formal system for specifying business models in a transaction-based system. ASM Relational Transducers [13] are implementations of Relational Transducers using Abstract State Machines. ASMs provide a logical, natural basis for an implementation of Relational Transducers as both are input and state based modeling systems. They have been shown to be useful in encoding and verifying transactional business models [1]. Implementing Transducers as ASMs alleviates some limitations of the earlier SPOCUS Relational Transducers [1] while maintaining PSPACE complexity [13]. Specifically, Spielmann presents complexity results showing that verifying properties of programs written in the formalism he describes (Relational Transducers implemented as a restricted ASM) are decidable in PSPACE [12, 13]. The following constraints are introduced to achieve this result, which are acceptable with respect to our case study of the Continue web server:

1. **The maximal arity of relations is bounded**
   This is a reasonable assumption, as the maximal arity in our policy was 3. We see no reason to have unbounded arity.

2. **The maximal input flow is bounded: for any input relation, the number of tuples in that relation is bounded**
   This again is a reasonable assumption. Continue runs

on a web server so we can justify bounding the input an individual client can produce and bounding the number of total clients. An individual client will be making requests from web pages, and we generally do not see forms on-line which have an unbounded number of inputs, so bounding the number of requests per client during one transition is reasonable. Finally, as Continue runs on a physical server, it can only serve a fixed number of concurrent requests before it must start queuing. Given these physical constraints in our case, assuming the maximal input flow is bounded is acceptable.

By putting these restrictions on ASMs, Spielmann shows that nontrivial relations can be used within the ASM framework without an increase in complexity [12]. This allows for an increase in expressiveness without complexity costs in the specification of Continue.

ASM Relational Transducers act as a natural way to express an interactive web application such as the Continue Conference manager. By treating the application as a state-based database system, the specification of Continue appears quite natural.

Take, for example, expressing the rule for a request to change a user from a reviewer role to an administrator:

```
IF      ChangeJobToAdmin($user_a$) AND Reviewer($user_a$)
$THEN$  NOT Reviewer($user_a$)
        Admin($user_a$)
```

Note the use of user relations. This allows us, for example, to represent multiple users in our program and add more dynamically by having them simply input a string for their name. A full policy is provided by our corresponding technical report [5].

2.4 Motivation: (In)Consistent Updates

Our initial security policy, written in a basic ASM specification language, suffered repeatedly from the same type of bug. For every input relation that represented server requests from a subject, such as a request to submit a paper, we included a position in the relation to represent the subject. This meant that in every transition, the update set might be changed due to the result of multiple user actions being handled concurrently. This is a useful property when specifying the idealized behavior of a web program with many clients - especially during the minutes leading up to the paper submission deadline.

However, as the update set for a given program is defined as the union of update sets from isolated evaluation of its individual rules, an inconsistency arises if one rule set asserts some relation, adding to it, and another negates it, removing from it. In such an event, the semantics of an ASM is to fail entirely.

For example, with Continue, consider the situation of an author updating an already-submitted paper at the same time as a system administrator removing the author's account (perhaps an extreme example, but demonstrative). This would simultaneously attempt to add an author and paper to a relation such as SubmittedPapers because of the new submission, while removing all members of the relation containing that author. In a system where inconsistent updates are defined as errors, it causes problems in an otherwise reasonable input case. In a system where they are defined as silent failures with no effect, the system may reach an inconsistent state - possibly with the old paper still attached to an author who should no longer exist as far as the system is concerned. Both semantics leads to problems, suggesting a linguistic issue.

Defining an inconsistent update as yielding an undefined or halting transition leads to simple semantics, but has limited utility for the policy writer. However, there is still some utility for the policy writer: the verification problem becomes detecting such states and appropriately handling them. The conflicting segments of specification can be guarded against each other, but specifying such a guard by hand is time consuming and tedious, not to mention prone to error. Additionally, the dependency between the conflicting sets of rules is explicitly between them only because, at that time, those are the only rule sets involved with the conflicting relations. If the specification is later modified, by adding or removing potentially conflicting relation updates, new less obvious problems are introduced. If new relation updates are added, the whole policy will have to be revisited to check that any newly-introduced conflicting updates are guarded against. If existing relation updates are removed, the whole policy will have to be checked over again or there will be dead specification checking for conflicts which will now never occur.

A more universal statement is desired for any potentially conflicting rule set. When modeling systems such as Continue where many updates may occur at the same time, and thus update sets may stem from syntactically separate specifications, conflicting transitions can easily be introduced into a specification, and it becomes difficult to reason about different parts of a policy due to global failure inducing interdependencies which are not visible when the parts are examined individually. We explore what is desired in the next section.

3. COMPOSITION OPERATOR

We partition our programs, or sets of rules, using the term module to describe any given partition, and can now phrase our problem as one composition: we want to concurrently run modules, but avoid any updates that yield inconsistencies. The consistent update problem, in terms of modules, is to create a composition function over modules that guarantees several properties:

1. **ASM+MODULES has the same complexity as ASM.**

   This suggests a simple source-to-source translation for our operator.

2. **Any relation update in a transition of the composed system also occurs in an update of one of the modules being composed when applied to the same state and input in isolation.**

   More informally, this property states that any update in the composed system is the result of an update from a particular module. Even more specifically, if you take that module in isolation without any of the other modules, and apply the same state and input, the update will still occur. Essentially, we view inconsistent updates as possible decisions that are distinct from just adding or removing to a relation. This means that modules are independently verifiable for certain properties and these properties are guaranteed to hold in the composition.

Imagine having two modules for making an administrator of a reviewer and deleting an administrator:

Module 1:

```
IF  ChangeJobToAdmin($u$) AND Reviewer($u$)
    NOT Reviewer($u$)
    Admin($u$)
```

Module 2:

```
IF  RemoveAdmin($u$) AND Admin($u$)
    NOT Admin($u$)
```

If we see a transition in the composition from a specific state S and input I that updates the Admin relation for a specific user *user*, we know that a specific module can be isolated as the source of the update. In this case, if we were to create a new ASM with *just* Module 1 and not Module 2 and apply the state S and the input I to this new ASM, we would again see the update of the Admin relation for the user *user*.

3. **An inconsistent update will occur if and only if that inconsistent update would have occurred in one of the modules on its own.**

This property is subtly different from property 2. It would be violated if one were to naively compose modules by simply unioning their rules, unlike property 2. Property 2 is only violated if a positive or negative update occurs in the composition which would not have occurred for one of the modules in isolation. This property is violated if the composition can result in a new inconsistent update. This naive union would allow modules to still have inconsistent updates with each other even when they do not individually have inconsistent update sets. The desired outcome is that inconsistent updates should only occur if they occur within a specific module and that module alone such that if you were to remove all other modules, the inconsistent update would still occur. Reasoning in terms of inconsistent updates is at the core of our paper. Conflicting updates between modules should not mean systemic failure but local (module) failure, and motivates our final desired property.

One consequence of this property is that it helps to localize bugs in the specification. If an inconsistent update occurs, in most cases, this implies a bug in the specification. Since we know that all inconsistent updates will be from single modules, and not across modules, this property helps to localize bugs to single, specific sections of code.

Imagine a system that requires a GlobalAdmin, a super user above all other administrators of which their only exists one at any single time. Take the following erroneous specification of three modules whose rules deal with GlobalAdmins:

Module 1:

```
IF  StartSystem($user$)
    GlobalAdmin($user$)
```

Module 2:

```
IF  CloseSystem($user$)
    NOT GlobalAdmin($user$)
```

Module 3:

```
IF  ChangeJobToGlobalAdmin($user, current\_global$)
    GlobalAdmin($user$)
    NOT GlobalAdmin($current\_global$)
```

In this example, the property assures us that there cannot be an inconsistent update from any relation updates in Modules 1 and 2, but there can be an inconsistent update from relation updates in Module 3. This is because Modules 1 and 2 do not contain both the positive and negative update for any relation, while Module 3 does.

While it appears that Module 1 and Module 2 have the potential for the inconsistent update problem, this property guarantees that they will not. The property assures us that in the final, composed system, if an inconsistent update occurs, it cannot happen because of updates in two separate modules.

However, the composition of these modules can still cause an inconsistent update. In fact, any time there is an input relation of ChangeJobToGlobalAdmin there will be an inconsistent update. Because of a bug in the model (namely, we intended to update "NOT GlobalAdmin" of *current_global*, not *user*, in Module 3), we will get an inconsistent update. The property allows this, and it assures us that we only have to look within a module for inconsistent updates, potentially simplifying our search for such bugs.

4. **For any given update in a transition of the composed system, not only can that update be attributed to at least one module transition update set, but the entire update set of the module to which the update is attributed is a subset of the overall update set for that transition in the composed system.**

Intuitively, if progress is made, it must come from entire modules, or else it would be hard to reason about effects of modules. No module should have a strict subset of its update set applied, unless it is really the exact update set from another module. In such a case, we can attribute the updates to this other module. Effectively, a module is an atomic transaction of a set of update relations.

Looking at the example for property 2, this property guarantees that if Module 1 and Module 2 were to both run, none of the updates would occur. This includes the "NOT Reviewer(u)" update of Module 1. Without this property, we could leave our system in an inconsistent state in so far as we would have a user who was not a reviewer, but also not an administrator and is thus just a dangling user. This property prevents such state inconsistencies from occurring by guaranteeing an "all or nothing" principle for modules such that either all of the updates in a module occur or none of them do.

The following sections introduce a basic operator that achieves these properties. The operator is fairly intuitive. First, every module is reduced to a normal form consisting of a sequence of IF-THEN statements. Modules are then compared piecewise, iteratively creating a module guard for each pair, which should be false when there are conflicting rules from the modules of interest such that their guards can be simultaneously satisfied but their update sets in such a case could conflict. Thus, our operator is a function from a set of abstract state machines to one new abstract state machine, and can be nested. We build the formalism as needed until we can prove our properties. We actually prove a condensed property that entails the properties presented.

3.1 General Description

A module is an ASM program with an additional guard which controls the firing of the module. The first step of the composition is to normalize each module into a set of rules, each rule consisting of the following form:

$$\text{IF } mg \wedge ug \text{ THEN } update$$

where mg is the module guard, and ug is the logical *and* of all conditionals surrounding the original update rule, *update*, within the module.

The next step is to prefix each update from each module with the negated module guard of any other module which has an update which would conflict with an update in that module, along with checks that they actually conflict for the current input. For each relation, in each module, consider the variety of update (additive, etc.) which that module results in for the tuple particular to that input. For every other module, if that module yields a different result for that relation and tuple, prefix the module guard for the current module with the negation of the other modules guard logically and-ed with checks to determine that the relevant unbound variables in the guard would result in an actual conflict. For example, assuming a Users relation, consider composing a module for adding users to a system with a module for deleting users:

Module 1:

```
IF   DeleteUser($u$)
     NOT Users($u$)
```

Module 2:

```
IF   AddUser($u$)
     Users($u$)
```

These two modules can lead to a transition with a conflicting update set. However, we do not simply want to prefix each module's guard with the negation of the others guard to only disallow simultaneously adding a user and deleting another user. Instead, to allow progress in the case of a potential conflict, the new form for Module 1 should be:

```
IF   DeleteUser($u$) AND NOT (AddUser($p$) AND $p = u$)
     NOT Users($u$)
```

The above is not in the normalized form, but demonstrates the intuitive transformation. Normally, the prefix would occur on each normalized update which originated in a particular module. Our full, formal algorithm and a formal main property follows. A correctness proof is available [5]. The following transformation fits within the ASMRT framework as presented by Spielmann[13] almost trivially due to our source-to-source approach. While our composition operator is applicable to ASMs that are not bound to the restricted set of ASMs presented in the ASMRT work, we verified our composition properties against such ASMs as that was all we were interested in our modeling task for Continue.

3.2 Informal Properties

In addition to complexity-related constraints, the properties the composition must satisfy appear to be similar to those describing concurrent transactions in that either all of the applicable updates from a module occur or none of them occur. We suggest several properties our composition operator must satisfy as rephrasings of the original ones, slowly approaching the formal property we desire:

1. **No newly introduced update decisions:** A decision of adding to or removing from a relation in the composition occurs only if that same decision occurs in one or several modules being composed. This is a refinement of the second property from the very beginning of this section. Note that this is a one way relationship: we do not currently provide any strong guarantees of progress. It may be an orthogonal issue for future work, which we also briefly discuss.

2. **No newly introduced update conflicts:** Under property 1, one module may add to an update relation, and another remove, so in the composition, there will be a conflict. We disallow this scenario, but if a module has an inconsistent update set, we allow it as we assume the module was individually verified and thus this was the desired action or a localized error. Additionally, if one module has an inconsistent update set, and another a consistent, neither should be allowed. This is a slightly more particular version of the third property from the beginning of this section.

3. **Module atomicity:** Properties 1 and 2 allow cases in which some updates from a module are used and others are not, which is exactly what we are trying to avoid as it has limited utility for a specification writer. Therefore, to say an applicable update in a module occurs, all of the applicable updates in the module must occur, and implicitly, no conflicting updates can be introduced strictly as a result of the composition. This strengthens the previous properties, and is a stronger version of the last of the more intuitive properties given at the start of this section.

We combine these properties into one stronger formal property stating that a decision occurs in the composition of modules if and only if it also occurs as a decision in one of the modules, and none of the decisions from that module conflict with those of any other module being composed when given the same input and state. A key result of this property is that any conflict that occurs is intentional, meaning it was entirely specified in a module being composed, and therefore not introduced by the composition operator.

Finally, to aid understanding of runs of a program written in our ASMRT language extended with modules, we describe an alternative translation to the original ASM language that creates additional output whenever conflicts occur in a transition. Such reflection can be useful when modules are prevented from updating, as the next transition can appropriately respond, such as by trying to update again. The composition property shows that these conflicts are intentional: they must have occurred due to updates within the body of some non-conflicting module. There may be value in also detailing from which paths of modules a conflict derives, but we do not pursue this issue. Our intuition is that a technique similar to that used in reporting original conflicts can be used.

3.3 Reporing Conflicting Updates

The occurrence of conflicting updates during the transitions of an ASM in an implementation where their effect is defined as a no-op can confuse the understanding of a policy because some updates written may not occur in the final transitioned state. In an implementation where conflicting updates are considered errors, it drastically reduces the system's usefulness. In Fig. 1, we describe an alternative translation from a module into an ASM, creating a new no-op relation $r_{NOOP,i}$ for everyregular output relation r_i and adding the additional output of which relations' updates no-op during a transition. The conditionals shown in this example are such as would be constructed by our actual composition operator, which will be explained later.

3.4 Definition of a Module

We define a module in terms of ASM update rules and assume an ASM program can be processed into a normal form consisting of a set of if-then statements.

Module m : a module m is a finite set of tuples, each consisting of a module guard and an update rule:

$$m \equiv \{(mg_k, u_k)\}$$

The module guard mg_k is any sentence (in which all variables are bound) that can be used within the test condition of an ASM IF-THEN update rule conditional. Note that we do not annotate a variable with its module identifier if it is apparent from context.

An update rule u_k is a tuple of the sign s_k of the relation to update (positive or negative, meaning adding to or removing from a relation), the name of the relation r_k to update, the applicability guard sentence ug_k for the update rule, the parameters of the update ${}_k\vec{p}$ with universally quantified variables replaced with a unique variable ub, and the set of indices UB_k of the inline universally bound parameters replaced with ub by a function o:

$$u_k \equiv (s_k, r_k, ug_k, {}_k\vec{p}, UB_k)$$

For example, consider the program consisting of the update rule u_k to remove reviewer r's conflicts on all papers:

$$\forall p. \neg Conflict(r, p)$$

To turn this into a module, we have:

$$u_1 = (\neg, Conflict, true, (r, ub), \{2\}) \quad (1)$$
$$m = \{(true, u_1)\} \quad (2)$$

The updates from the module are always accepted, as they're guarded by true: our algorithm will generate stronger guards. While it may be possible to combine update and module guards, keeping them distinct is clean: all of the variables of a relation being updated are bound within the update guard, and when composing modules, the update rules do not change.

Module $\mathbf{m_i}$: ASM program $\pi_\mathbf{i} \equiv [\![\mathbf{m_i}]\!]$
Intuitively, when a program consists of only module m_i, update $u_{i,k}$ is applied whenever $mg_{i,k} \wedge ug_{i,k}$ is satisfied. Thus, we define $[\![m_i]\!]$ in terms of ASM update rules as follows, knowing $UB_{i,k}$ is finite and totally ordered by o, where o is a function that given an index in the parameter list returns an integer in UB:

$$[\![m_i]\!] \equiv \begin{array}{l} IF\ mg_{i,1} \wedge ug_{i,1}\ THEN \\ \quad \forall ub_{i,1}, \cdots, \forall ub_{i,|UB_{i,1}|} s_{i,1} r_{i,1}(f({}_{i,1}\vec{p}, UB_{i,1})) \\ \quad \vdots \\ IF\ mg_{i,n} \wedge ug_{i,n}\ THEN \\ \quad \forall ub_{i,n}, \cdots, \forall ub_{i,|UB_{i,n}|} s_{i,n} r_{i,n}(f({}_{i,n}\vec{p}, UB_{i,n})) \end{array}$$

where $n = |m_i|$, and $f({}_{i,k}\vec{p}, UB_{i,k}) = {}_{i,k}\vec{q}$ such that

$${}_{i,k}\vec{q} = \begin{cases} {}_{i,k}\vec{p}_j & \text{if} j \notin UB_{i,k} \\ ub_{i,o(j)} & \text{if} j \in UB_{i,k} \end{cases}$$

Note that, as $|UB_{i,k}|$ and $|m_i|$ are bounded, this translation terminates. Additionally, for brevity, we will write $f(\vec{p}, UB)$ as $\vec{q}$ in future sections.

3.5 Desired Properties of Module Composition

We describe the main property that our composition operator $\otimes : \mathcal{P}(\{m_i\}) \rightarrow m_j$ (where $\mathcal{P}(S)$ is the power set of set S) must satisfy:

Given input module set $\{m_i\}$, the composed module $m_j = \otimes(\{m_i\})$ must translate into an ASM which makes update decisions as follows: an update decision for some relation in the composed module, whether it be addition to the relation, removal from the relation, a conflict leading to no operation on the relation for a given tuple, or no applicable operation, occurs if and only if the same decision is made by one of the modules being composed when considered independently, and none of the decisions from other modules being composed, when also considered independently, conflict with any decisions of the first module being considered independently.

A formal version of the decision function is presented in Fig. 3. The decision of a module m_j for a particular relation r and bindings $\vec{q}$ is split into four possible outcomes. The decision is not applicable (NA) if none of the updates in the module refer to that relation or perform an update to it with that set of bindings. The decision is positive if the module performs a positive update to that relation using those bindings, and has no negative update with the same conditions. The case for a negative decision is analogous. The decision is a no-op ($NOOP$) if the module contains updates to the

$$[\![m]\!]_{nreport} \equiv \begin{cases} IF\ mg_1\ \wedge\ ug_1\ THEN \\ \quad \forall ub_1, \cdots, \forall ub_{|UB_1|}\ s_1r_1(f({}_1\vec{p}, UB_1)) \\ \quad IF\ (mg_{s,1,1}\ \wedge\ ug_{s,1,1}\ \wedge\ \vec{p}_1 = \vec{p}_{s,1,1})\vee \\ \qquad\qquad \cdots \\ \qquad \vee\,(mg_{s,1,|similar(r_1,m)|}\ \wedge\ ug_{s,1,|similar(r_1,m)|}\ \wedge\ \vec{p}_1 = \vec{p}_{s,1,|similar(r_1,m)|}) \\ \qquad THEN \\ \qquad\quad \forall ub_1, \cdots, \forall ub_{|UB_1|}\ s_1r_{NOOP,1}(f({}_1\vec{p}, UB_1)) \\ \quad \vdots \\ IF\ mg_n\ \wedge\ ug_n\ THEN \\ \quad \forall ub_n, \cdots, \forall ub_{|UB_n|}\ s_nr_n(f({}_n\vec{p}, UB_n)) \\ \quad IF\ (mg_{s,n,1}\ \wedge\ ug_{s,n,1}\ \wedge\ \vec{p}_n = \vec{p}_{s,n,1})\vee \\ \qquad\qquad \cdots \\ \qquad \vee\,(mg_{s,n,|similar(r_n,m)|}\ \wedge\ ug_{s,n,|similar(r_n,m)|}\ \wedge\ \vec{p}_n = \vec{p}_{s,n,|similar(r_n,m)|}) \\ \qquad THEN \\ \qquad\quad \forall ub_n, \cdots, \forall ub_{|UB_n|}\ s_nr_{NOOP,n}(f({}_n\vec{p}, UB_n)) \end{cases}$$

Figure 1: Alternative translation from modules to an ASM which reports conflicting updates

same relation, with the same bindings, and opposite signs (one or more positive and one or more negative updates).

A formal version of the property the composition must satisfy is presented in Fig. 2. For all decisions we care about (any time the module reaches a decision), for all relations and all bindings, the decision of the composition for those cases is such if and only if there is a module in the input set which reaches the same decision for those circumstances, and there is no module in the input set which reaches a different decision. The *conflict* function is defined as one would intuitively expect, but is also formally defined in Fig. 4. Also, if the decision of the composition is NA (that is, it reaches no decision and does not modify that relation) if and only if for every module in the input set which reaches a decision on that relation and input, there is another module in the input set which reaches a different decision.

3.6 Formal Algorithm

Figure 6 presents the formal module composition algorithm. For each relation, for each module, we build replacement module guards which guard against conflicting sets of modules executing simultaneously. We then union the resulting modified modules. The replacement module guards are put in place in the copy by mutating the module guard of every update rule in a given module with a guard which prevents it from executing under conditions when another update would conflict.

Figure 5 is the boolean statement which actually builds the modified guards. It is the logical OR of four clauses. The first clause deals with positive updates which might conflict with negative updates. The next clause deals with the reverse. The third clause deals with no-decision modules, and the final clause deals with guarding against modules which arrive at no-ops for the specified relation.

Consider the example of composing these two simple modules, not written in normal form for clarity's sake:

Module 1:

IF	MakeReviewer(u)
	Reviewers(u)

Module 2:

IF	DeleteReviewer(u)
	NOT Reviewers(u)
	NOT Users(u)

First the input set will be copied, so that this copy can be mutated without enlarging the input. For each relevant relation, for each module in the input, and for each rule in the module, we build a replacement module rule with a modified guard, and add this to the result. In this case, there is only one relation, Reviewers. So for each update in the first module (which has only one), we build a new guard for that update, using the $getPrime$ function, given positive, Reviewers, and the original input set. We logically AND the guard for each update in that module with the generated guard supplement from $getPrime$, which will check that DeleteReviewer is not called at the same time with the same user. Now repeat this task for the second module. The only update in Module 2 which could conflict is the first update. $getPrime$ will return a guard which checks that MakeReviewer is not simultaneously called with the same user. The resulting new update rules are then unioned, and the result reduces to what is shown in Fig. 7.

Consider a set of modules composed into one new module under our composition operator. During a transition, an update rule from one of the modules fires only if it comes from a module that does not have any update decisions that conflict with update decisions from other modules. Thus, if the decision of a module on any relation conflicts with that of any other module, none of the updates in the first module should fire (nor any in the second). Thus, we define a function that first detects, for any given relation, whether any modules conflict on its decision for that transition. Then, for any module that can make an update decision for that relation, we add to the guards of all update rules in that module a check for a potential conflict on that relation. The resultant guards are increased in length by a polynomial amount, so the final policy is still in O(PSPACE) according to Spielmann's work[13].

For a proof that this algorithm satisfies the correctness property described above, see [5]. A weak form of progress is clear but is not as strong as that provided by other composition techniques[10]. This is due to an assumption of

$$
\begin{array}{lcl}
 & & \otimes : \mathcal{P}(\{m_i\}) \rightarrow \{m_j\} \text{ st.} \\
\forall r \forall \vec{q} \forall d \in \{+, -, NOOP\} \ . & & \\
 & & \exists m_l \in \{m_i\}.decision(r, \vec{q}, m_l) = d \\
decision(r, \vec{q}, m_j) = d & \iff & \ \wedge\ \forall r\prime, \vec{q}\prime.\neg \exists m_k \in \{m_i\}. \\
 & & conflict(decision(r\prime, \vec{q}\prime, m_l), decision(r_k, \vec{q}\prime, m_k)) \\
\forall r \forall \vec{q} \ . & & \\
 & & \forall m_l \in \{m_i\}.decision(r, \vec{q}, m_l) \neq NA \\
decision(r, \vec{q}, m_j) = NA & \iff & \ \rightarrow\ \exists r\prime, \vec{q}\prime.\exists m_k \in m_i. \\
 & & conflict(decision(r\prime, \vec{q}\prime, m_l), decision(r\prime, \vec{q}\prime, m_k))
\end{array}
$$

Figure 2: Formal property which the composition satisfies

$$
decision(r, \vec{q}, m_j) = \begin{cases}
NA & \forall (mg_i, u_i) \in m_j.(mg_i \ \wedge\ ug_i \rightarrow r \neq r_i \vee \vec{q_i} \neq \vec{q}.) \\[2ex]
+ & \begin{array}{l} \exists (mg_i, u_i).(r = r_i \ \wedge\ mg_i \ \wedge\ ug_i \ \wedge\ \vec{q} = \vec{q_i} \ \wedge\ s_i = +) \\ \quad \wedge\ \forall (mg_i, u_i).(r = r_i \ \wedge\ mg_i \ \wedge\ ug_i \ \wedge\ \vec{q} = \vec{q_i}) \rightarrow s_i = + \end{array} \\[2ex]
- & \begin{array}{l} \exists (mg_i, u_i) \in m_j.(r = r_i \ \wedge\ mg_i \ \wedge\ ug_i \ \wedge\ \vec{q} = \vec{q_i} \ \wedge\ s_i = -) \\ \quad \wedge\ \forall (mg_i, u_i).(r = r_i \ \wedge\ mg_i \ \wedge\ ug_i \ \wedge\ \vec{q} = \vec{q_i}) \rightarrow s_i = - \end{array} \\[2ex]
NOOP & \begin{array}{l} \exists (mg_i, u_i), (mg_f, u_f) \in m_j.r = r_i = r_f \ \wedge\ mg_i \ \wedge\ ug_i \ \wedge \\ \quad mg_j \ \wedge\ ug_f \ \wedge\ \vec{q} = \vec{q_i} = \vec{q_f} \ \wedge\ s_i \neq s_f \end{array}
\end{cases} \tag{3}
$$

Figure 3: Definition of the decision function

only consistent updates in the composition by other work - the current algorithm reduces to Nicolosi-Asmundo and Riccobene's feature composition given such a strong condition.

4. PRIORITIZED COMPOSITION

Given a way to control composition of ASM components and their execution as atomic units, the next logical step is to find more flexible ways of using this than simply blocking any set of modules which would conflict if executed simultaneously on related inputs.

In our case, a motivating example may be a conflict between an author uploading a paper, and an author's account being deleted. While disallowing these if they were attempted simultaneously is useful, it would be preferable to be able to designate that the account deletion by an administrator should take precedence, and thus should execute in this example.

Atomicity is guaranteed by prefixing each conflicting module with the negation of the other modules' guards. For any two modules, only prefixing one of them with the other's module guard would still guarantee atomicity in the sense of preventing the introduction of additional conflicting updates. This suggests a useful and very straightforward modification of the composition algorithm to allow one of each group of conflicting modules to still execute:

Annotate each module with a priority. During the prefixing step of the algorithm, instead of unconditionally prefixing negated module guards from conflicting modules, prefix only when the conflicting module has a higher priority than the module whose rule is being prefixed (with the prefixing still occurring for both if the modules have equal priority).

Thus, the desired outcome from the situation described above could be achieved by giving the module dealing with account deletion a higher priority than that dealing with uploading papers. Furthermore, the outcome of the original operator is equivalent to assigning all modules the same priority.

Alternative operators can be described. The original composition operator, to provide a higher likelihood of progress, could be intuitively redefined in the above way without specification writers explicitly giving priorities, implicitly using rule order to give priority. Richer operators are possible, such as one that chooses the module with the maximal update set based on cardinality.

5. FUTURE WORK

There are clear improvements which could be done on the simple prioritized module composition idea. Consider the case of three modules, 1, 2, and 3, with high, medium, and low priorities respectively. If 1 and 3 conflict with 2, but not each other, consider the case when inputs are received which would individually trigger each of the modules. 1 has the highest priority, so it will execute. The inputs for 1 and 2 are both received, but 2 has lower priority, so it will not execute. The inputs triggering 2 and 3 are both received, and 3 has lower priority than 2, so it will not execute. However module 2 is not executing because it conflicts with module 1. So because 1 and 3 do not conflict, in this case all conflicts are resolved by disallowing execution of only module 2, yet the prioritized composition operator as described would still prevent module 3 from executing, even though it is safe to allow if module 2 does not execute.

We see that other operators may also be desirable. While we describe a basic way of achieving localized failure by disregarding the update sets of conflicting modules, we do not focus on progress guarantees. Prioritization, implicit or explicit, is one idea and we showed how to achieve it by a simple change to our algorithm, but others, such as maximal updates, may be desirable. We believe progress guarantees can be built upon our basic notion of localized failure.

In our technical report, we discuss other potential extensions, such as conflict reporting as opposed to failure. As

$$conflict(d, d\prime) = \left\{ \begin{array}{ll} false & d = d' \vee d = NA \vee d' = NA \\ true & else \end{array} \right. \tag{4}$$

Figure 4: Definition of the conflict function

$$\text{getPrime}\ (s, r, \{m_i\}) =$$

$$\bigvee \begin{array}{l}
\left(\wedge \begin{array}{l}
\bigvee_{\left\{\substack{(mg_k,u_k)\in\{m_i\}: \\ r_k=r, \\ s_k=+}\right\}} mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k \\
\bigwedge_{\left\{\substack{(mg_k,u_k)\in\{m_i\}: \\ r=r_k, \\ s=-}\right\}} \neg(mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k)
\end{array} \right) \\
\left(\wedge \begin{array}{l}
\bigvee_{\left\{\substack{(mg_k,u_k)\in\{m_i\}: \\ r_k=r, \\ s_k=-}\right\}} mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k \\
\bigwedge_{\left\{\substack{(mg_k,u_k)\in\{m_i\}: \\ r=r_k, \\ s=+}\right\}} \neg(mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k)
\end{array} \right) \\
\left(\bigwedge_{\left\{\substack{(mg_k,u_k)\in\{m_i\}: \\ r=r_k}\right\}} \neg(mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k) \right) \\
\left(\begin{array}{l}
\left(\left(\bigvee_{\left\{\substack{(mg_k,u_k): \\ (mg_k,u_k)\in(m_l\in\{m_i\}), \\ r=r_k, \\ s_k=+}\right\}} mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k \right) \wedge \left(\bigvee_{\left\{\substack{(mg_i,u_i): \\ (mg_i,u_i)\in m_l, \\ r=r_l, \\ s_l=-}\right\}} mg_l \wedge ug_l \wedge \vec{q}_1 = \vec{q}_l \right) \right) \\
\wedge \bigwedge_{\{m_l\in\{m_i\}\}} \left(\left(\bigvee_{\left\{\substack{(mg_k,u_k): \\ (mg_k,u_k)\in m_l, \\ r_k=r, \\ s_k=+}\right\}} mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k \right) \Leftrightarrow \left(\bigvee_{\left\{\substack{(mg_i,u_i): \\ (mg_i,u_i)\in m_l, \\ r_i=r, \\ s_i=-}\right\}} mg_i \wedge ug_i \wedge \vec{q}_1 = \vec{q}_i \right) \right)
\end{array} \right)
\end{array} \tag{5}$$

Figure 5: Definition of the getPrime function

```
$\otimes(\{m_i\})$
 1  $result \leftarrow \emptyset$
 2  $function$ GETPRIME$(s_k, r_k, \{m_i\}) = (Fig.5)$
 3  for $r \in Memory \cup Output$
 4  do $copy \leftarrow$ DEEPCOPY$(\{m_i\})$
 5      for $m_l \in copy$
 6      do for $(mg_k, u_k) \in m_l$
 7          do for $(mg_x, u_x) \in m_l$
 8              do $mg'_x \leftarrow$ GETPRIME$(s_k, r_k, \{m_i\}) \wedge mg_x$
 9              for $(mg_k, u_k) \in m_l$
10              do $result \leftarrow result \cup \{(mg'_k, u'_k)\}$
11  return $result$
```

Figure 6: Formal composition algorithm

```
IF      MakeReviewer($u$) AND NOT (DeleteReviewer($u2$) AND $u = u2$)
THEN    Reviewers($u$)
IF      DeleteReviewer($u$) AND NOT (MakeReviewer($u2$) AND $u = u2$)
THEN    NOT Reviewers($u$)
IF      DeleteReviewer($u$) AND NOT (MakeReviewer($u2$) AND $u = u2$)
THEN    NOT Users($u$)
```

Figure 7: The reduction of the result of composing modules for Module 1 and Module 2 from Sec. 3.6

the update set of a transition from composed modules may not include the update set of an individual module, additional specification of how that situation should be handled may be necessary, but has not been examined in our current work.

Prioritized atomic module composition is not the final goal. Ideally, we could parameterize the composition by a set of properties which we desire the result to satisfy, though it is not yet clear to us how to achieve this with sufficient generality. While it might not be tractable to compose based on a consistency property, useful results might be achieved in terms of choosing which modules to include in some otherwise conflicting update set.

6. RELATED WORK

Spielmann [11, 12, 13] provides a basis for verification of ASMs with polynomial time complexity. He proves ASMs useful in the general framework of verifying business-like transaction systems, similar to our dynamic security model.

Our complementary technical report [5] further establishes the usefulness of ASMs for modeling semantic systems, specifically the security policy for the Continue Conference Manager. In it, we also formalize our intuitive composition operator, desired correctness properties, and prove one satisfies the other.

Current implementations of abstract state machines (ASM-Workbench [3], XASM [2], ASML [8]) provide methods of composing ASMs, but none of these address the consistent update problem. ASML does permit some degree of atomicity through exceptions, but this requires more more work than simply defining an atomic unit, and it is unclear how to reflect upon conflicts in this system.

Nicolosi-Asmundo and Riccobene [10] present two forms of composition for building ASMs from independently written components. The first, feature composition, is the naive approach to composition existent in most current ASM systems: it simply runs all programs in parallel. The second, component composition, adds explicit communication channels for isolated components to communicate, similar to message passing. Some validity checking is involved in both compositions to detect (but not correct) potential inconsistent updates: no shared access to per-component state is allowed. The check is done statically but is not complete, eventually appealing to a theorem prover, while our approach is dynamic and complete, but at a quadratic runtime cost per level of module nesting. Both cited composition techniques are useful in some situations; frequently, feature composition is more than sufficient, and component composition is well suited to systems with components which have independent state which need not be accessible by other parts of a system. Both fail (the composition algorithm rejects the input) in cases of inconsistent updates, but this allows the composition to maintain a strict safety property, that any run of an individual component can be found in the runs of the composed system. We do not maintain this because we are more interested in system consistency than strict correctness (which is not possible to satisfy when intentionally disabling portions of the system that would introduce inconsistencies in the composition but not in isolation). Our original motivation was dissatisfaction with the feature composition typical of ASM systems, and component composition is too coarse of a restriction, moving away from the more declarative nature of abstract state machine specifications. In the absence of conflicts, our operator reduces to feature composition, but, in practice, this assumption may not be valid, as seen in our usage of ASMs.

These composition techniques work well for modeling systems more independently defined than ours, such as hardware systems, where all actors are known in advance and have explicit interfaces. However, our notion of module composition offers an easier way to specify method than component composition, while providing consistency guarantees not available with feature composition. Partially ordered runs [14] are a way to guarantee atomicity, but limit the extent of implicit concurrency found with traditional ASMs: we want concurrency except in the case of conflicts, not restricting specifications to concurrency except in the cases of *potential* but not guaranteed conflicts. Instead of changing the base semantics, we see the problem as one of composition abstractions, and provide a way to maintain implicit concurrency with what, in our case, is the intended semantics, and reuse the base language. Module composition requires that slightly modified behavior be acceptable, but we believe this behavior is closer to what is desired by specification authors: implicitly concurrent specifications with fine-grained, controlled, localized transactional failure permitting the separate verification of individual modules.

7. CONCLUSION

We have described a composition technique that allows for automatic joining of separately-written system components. The composition preserves system consistency, introducing no new state changes or conflicting updates, and guarantees atomicity. We have also described a prioritized version which gives preference to a module deemed by the system designer as more important, allowing it to execute when it does not conflict more important modules, by disallowing execution of lower-priority modules. Additionally, as another approach to handling inconsistent updates, we have shown how to support reflection. Inconsistent updates are inherent to programs in the style of ASM specifications, but we have shown ways of enabling specification authors to reason about them and limit their scope while maintaining implicit concurrency.

Thanks go to Dan Dougherty, Kathi Fisler, and the entire cs296-1 class for helping us develop our ideas. Egon Boerger, Marianna Nicolosi-Asmundo, Wolfram Schulte, Wolfgang Schoenfeld, and Yuri Gurevich helped point us towards useful resources. Thanks also to Don Batory for his helpful insight.

8. REFERENCES

[1] Serge Abiteboul, Victor Vianu, Brad Fordham, and Yelena Yesha. “Relational Transducers for Electronic Commerce”. In *Symposium on Principles of Database Systems*, pages 179–187, 1998.

[2] Matthias Anlauff. “XASM - An Extensible, Component-Based ASM Language”. In *ASM ’00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, pages 69–90, London, UK, 2000. Springer-Verlag.

[3] G. Del Castillo. “The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines”. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

[4] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. “Verification and Change-Impact Analysis of Access-Control Policies”. In *International Conference on Software Engineering*, 2005.

[5] Colin Gordon, Leo Meyerovich, and Joel Weinberger. “ASM Relational Transducer Security Policies”. Brown University Department of Computer Science Technical Report, 2006.

[6] G. Gottlob, G. Kappel, and M. Schrefl. “Semantics of object-oriented data models—The evolving algebra approach”. In J. Schmidt and A. Stogny, editors, *Next Generation Information System Technology, First International East/West Database Workshop*, volume 504, pages 144–160, Kiev, USSR, October 1990. Springer-Verlag.

[7] Yuri Gurevich. “Evolving Algebras: An Attempt to Discover Semantics”. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, River Edge, NJ, 1993.

[8] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. “Semantic Essence of AsmL”. In *Theoretical Computer Science*, pages 360–412, 2005.

[9] Shriram Krishnamurthi. “The CONTINUE Server (or, How I Administered PADL 2002 and 2003)”. In *PADL ’03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 2–16, London, UK, 2003. Springer-Verlag.

[10] Marianna Nicolosi-Asmundo and Elvinia Riccobene. “Consistent Integration for Sequential Abstract State Machines”. In *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003. Proceedings*, pages 324–340. Springer Berlin / Heidelberg, 2003.

[11] M. Spielmann. “Automatic Verification of Abstract State Machines”. In *Proceedings of 11th International Conference on Computer-Aided Verification (CAV ‘99)*, volume 1633 of *LNCS*, pages 431–442. Springer-Verlag, 1999.

[12] M. Spielmann. “Model Checking Abstract State Machines and Beyond”. In *International Workshop on Abstract State Machines ASM 2000*, LNCS, pages 323–340. Springer-Verlag, 2000.

[13] M. Spielmann. “Verification of Relational Transducers for Electronic Commerce”. In *19th ACM Symposium on Principles of Database Systems PODS 2000, Dallas*, pages 92–93. ACM Press, 2000.

[14] Y. Gurevich and D. Rosenzweig. Partially Ordered Runs: A Case Study. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912, pages 131–150. Springer-Verlag, 2000.