ASM Relational Transducer Security Policies

Leo A. Meyerovich, Joel H.W. Weinberger,
Colin S. Gordon and Shriram Krishnamurthi

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-06-12
November 2006

ASM Relational Transducer Security Policies

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

We present a model of the security policy for the Web-based Continue [10] conference management tool. The policy model and properties are written as ASM Relational Transducers [14], which we extend with a module system in order to simplify the handling of conflicting updates. We assume prior familiarity with the security policy concerns surrounding Continue. First, we review the ASM Relational Transducer modeling and property language. Then we describe the basic structure of our policy implementation and demonstrate the ability to model useful properties in the original core ASM [7] language. We exploring the use of the unmodified modeling language in a security policy context and describe typical ASM Relational Transducer complexity concerns [14] and how these minimally impact our implementation. Next, we discuss difficulties encountered in representing our policy and properties in the standard ASM language, including our implementation in the appendices. Following the description of adapting ASMs for use in security modeling, we introduce policy modules and a composition operator to overcome the difficulty of programming in the original language known as the consistent update problem. Finally, we describe a reduction from our extended language to the original language, and prove it satisfies our required correctness property.

1 Background: ASM Relational Transducer Model

ASM Relational Transducers [14] are implementations of relational transducers using Abstract State Machines (formerly known as Evolving and Dynamic Algebras), introduced by Gottlob [6], advanced by Gurevich [7] and extended by Spielmann [13, 12].

Relational transducers have been shown to be useful in encoding and verifying transactional business models [1]. Such transducers can be written as ASMs to alleviate some limitations of the earlier SPOCUS relational transducers [1] while maintaining PSPACE complexity [14].

An ASM can be viewed as an abstraction over nondeterministic finite state machines in which every state is labeled with atomic propositions that are true, also known as Kripke structures. Instead of every state being labeled by atomic propositions, every state is a model in which sentences in first order logic are either true or false. A transition function specifies how to generate a new state for the model based on the current state and a given set of input relations (as opposed to input symbols). Instead of specifying the transition function, which can be tedious, the transition function is extracted from a program consisting of model update rules that dictate how the values of relations and functions change during one step of the program. For example, the program to transition from a state in which a $user_a$ is a Reviewer to a state in which $user_a$ is an Admin can be written as a single rule as follows:

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

Note that $user_a$ is implicitly universally quantified in the guard of the conditional on the first line.

A program is a set of rules. The program can be repeatedly run, with the new start state being the the result of transitioning from the previous start state on some input[1].

At any given state, there is a danger of rules specifying conflicting updates. For example, it is possible that a user is added and deleted in the same state. While the core semantics of ASMs define such a conflicting update, which we call a no-op, to not occur and thus leave the relevent relation unmodified from the previous state, in practice the occurence of such a conflict is defined as a failed run. Even when following the core semantics, a no-op can cause undesirable behavior in the modelled system due to the lack of atomicity for sets of update rules.

Due to the varying uses and properties of relations, ASM relations are partitioned into the following classes:

- Input
- Output
- Memory
- Database

1.1 Input

Input relations drive our transitions. These occur in the guards of conditionals as we generally act upon received input. We define the names of our input relations at the beginning of our policy, just like any other relation in our model. For example:

[1]It should be noted that the core ASM program language includes another construct, *choose* : ϕ, Π, which is nondeterministic choice, and is not used for complexity reasons. We do not address nor use this construct as nondeterminism is an undesirable property in most security policies.

Input:
  ChangeReviewerToAdmin(x)
Memory:
  Admin(a)

UpdateRules : IF **ChangeReviewerToAdmin**($\mathbf{user_a}$) THEN
  Admin($user_a$)

A subtlety of the model is that all input relations and the possibility of multiple values must be considered. In the above example, ChangeReviewerToAdmin($user_a$) may hold for multiple users. This would be reasonable in the case that the user is using a form in which he/she may use a checkbox to specify that multiple reviewers should be converted to administrators at the same time. Similarly, care must be taken in circumstances in which multiple relations may have values that are intertwined, such as when ChangeJobToAdmin($user_a$) and ChangeJobToReviewer($user_a$) both hold for the same user. This will be a large motivation for Section 7 where we discuss policy composition.

1.2 Output

We may want to report the status of our model during certain transitions. Output relations cannot occur in the guard of a conditional as they are the result of our transition. They are never negated, meaning values are never removed from them, which can be explained by viewing them as a result. Thus, there are no output relation values until a rule explictly asserts the existance of one.

IF RequestAccount($user_a$) THEN
  notifySuccess($\mathbf{user_a}$)

1.3 Memory

Memory represents mutable state. A program can reference the current state and use it to determine the next state. Relations in memory start out empty, so initialization concerns arise that will be addressed in section 5.

IF ChangeJobToAdmin($user_a$) AND **Reviewer**($\mathbf{user_a}$) THEN
  NOTReviewer($\mathbf{user_a}$)
  Admin($\mathbf{user_a}$)

1.4 Database

Database relations are also known as Static relations in ASM literature. In our model, some things are always true and thus some relations should never change. Relations in the database only occur in rule guards as they are immutable. These relations generally do not start empty, as that would cause them to remain empty permanently.

```
IF   CurrentPhase = **InitialPhase** AND AdvancePhase THEN
     CurrentPhase = **SecondPhase**
```

1.5 Log

We want to verify properties of sequences of states achievable by program transitions. A common task may be to check that there is a valid sequence of transitions that would produce a certain sequence of input and output relations. Relations in the Log class are therefore also either in Input or Output. We are not exclusively interested in relations in the Log, nor limit property reasoning to them, but they are a useful abstraction.

Given the partitioning of relations into Input, Output, Memory, and Database, we must be more specific about how we author rules for updating the state through transitions. Only Output and Memory relations can be modified, and only the Output relations may recieve positive (additive) updates only. Spielmann therefore separates update rules into separate Output rules and Memory rule sets, though we find the majority of our Output and Memory rules share the same guards and thus it is counterproductive keeping the rules separate in terms of code length. Our policies combine the two for the sake of brevity. Formal treatise can be found in related work [13, 1].

2 Background: ASM Relational Transducer Property Language

We are interested in verifying properties of possible sequences of states. Classes of verification problems exist that are concerned with validity and reachability of **Log** relations [14], but we may also be interested in the properties of other relations that occur in states along valid sequences, such as whether it is ever possible to have no administrators in memory. A temporal logic would be a natural choice, and our description of states motivates the use of first order logic for state formulas. Spielmann uses a first order temporal logic, FTL, allowing first order state formulas and path formulas that are of the form of either $\mathrm{X}a$ or $a\mathrm{U}b$, but excluding quantification over paths.

The property language consists of a full first order logic on states and a limited temporal logic on paths. We lack for-all and there-exists over paths, which are not necessary because for any given state, the only path we are concerned with is that resulting from the running the update program with the current input. We can express properties about the entirety of the path we care about, and can express any first-order property within a given state through normal first order logic. The latter includes checking for the presence of a certain tuple in a relation, or quantifying over the members of a relation within a given state. Thus we can express most properties relevant to our policy in our property language. For example, we can express the property that authors can only submit papers during the Submission phase:

$$\begin{aligned}\text{G(FORALL } \mathit{paper}, \mathit{user}\text{:}\quad & \text{NOT CurrentPhase} = \text{Submission} \rightarrow \\ & ((\text{Papers}(\mathit{user}, \mathit{paper}) \rightarrow \text{X Papers}(\mathit{user}, \mathit{paper})) \\ & \ \wedge (\text{NOT Papers}(\mathit{user}, \mathit{paper}) \rightarrow \text{X NOT Papers}(\mathit{user}, \mathit{paper}))))\end{aligned}$$

Generally, the expressiveness of state formulae and the malleability of memory are useful when encoding intricate properties. For example, we can encode any need to reference all possible inputs by simply referencing contents of the memory.

Due to the nature of ASMs, some properties which make sense to check and verify in other models written in other langauges do not need to be checked for ASMs. Specifically, any property regarding a given operation (e.g. changing a user's password) leaving the rest of the state unchanged makes no sense because multiple operations can occur simultaneously in our use of ASMs. Extraneous checks such as these occured in an alternative Continue modeling attempt utilizing Alloy.

3 Policy Implementation Strategy

Our ability to describe states using first order logic made our translation from our Alloy model relatively intuitive. The Continue policy is a rich example of a security policy, characterizing common requirements for expressibility. First, we briefly discuss our superficial choice to deviate from the seperation of Memory and Output update rules. Next, we examine our representation of conference phases to show how to describe general transitions and paper phases to show initialization techniques. We also present how to allow the changing of the jobs of users, capturing the basic technique of adding and removing from relations. Allowing the addition and removal of reviews, conflicts, and other tasks common to the Continue policy can be similarly represented so we leave the rest to our implementation in the appendices.

3.1 Merging memory, output rules

While writing our model, we deviated from the standard ASM program notation. The standard notation separates the memory rules from the output rules, yet there is no inherent reason for this distinction; it seems to be done to provide a syntax close to that found in related relational transducer literature.

However, this is a very inconvenient notation as it naturally leads to much repetition of general structure and specifically the usage of guards. In a rule that both records a decision in memory and outputs a decision when given an input relation PaperDecision for a *paper*, the written description would require repeating the input guard twice:

```
Memory rules:
    ...
    IF AddConflict(curuser, user, paper) THEN                //input
        IF Admin(curuser) OR (curuser = user) THEN           //memory
            Conflicts(user,paper)                            //memory
...

Output rules:
...
    IF AddConflict(curuser, user, paper) THEN                //input
        IF Admin(curuser) OR (curuser = user) THEN           //memory
            ConflictAdded(user, paper)                       //output
        ELSE
            ActionFailed(curuser)                            //output
...
```

Thus, in our model, because no relation is both a memory relation and an output relation, we combine memory rules and output rules into one set of rules, modifying memory and generating output at the same time:

```
Memory and Output rules:
        ...
    IF AddConflict(curuser, user, paper) THEN           //input
        IF Admin(curuser) OR (curuser = user) THEN      //memory
            Conflicts(user,paper)                       //memory
            ConflictAdded(user, paper)                  //output
        ELSE
            ActionFailed(curuser)                       //output
...
```

Policy composition, dicussed in section 7, is not meaningfully impacted by this seperation.

3.2 Conference Phases

A large portion of Continue depends on advancement of phases in a conference. Certain actions are permitted in some phases but not in others. We demonstrate how we perform and authorize phase transitions:

```
    Memory and output rules:
        ...
       IF AdvancePhase(curuser) AND Admin(curuser) THEN    //input, memory
           IF CurrentPhase = Initialization THEN     //memory,db
                CurrentPhase = PreSubmission    //memory.db
                PhaseAdvanced(PreSubmission)    //output,db
           ELSE IF CurrentPhase = PreSubmission THEN     //memory,db
                CurrentPhase = Submission     //memory,db
                PhaseAdvanced(Submission)             //output,db
        ...
```

We check to see whether a user wants to advance the conference phase, and that the user is an administrator and, if so, update the CurrentPhase memory relation and generate the appropriate output relation. CurrentPhase is a nullary function so it can only have one value, namely the phase we transition to. A subtlety arises in guaranteeing that phase values are distinct, which we discuss in section 5.

3.3 Paper phases

Memory relations start empty and database relations are initialized with values that never change, yet memory relations sometimes should be initialized to values not dependent upon input. Our approach is to store initialization values in database relations, and when needing to initialize a memory relation, we just use the appropriate database relation.

Example:

```
IF AdvancePhase(curuser) AND Admin(curuser) THEN    //input, memory
    ...
    ELSE IF CurrentPhase = Bidding THEN    //memory,db
        CurrentPhase = Assignment    //memory,db
        PhaseAdvanced(Assignment)    //output,db
        forall u, p | Papers(u, p):    //memory
        PaperPhase(p, Passignment)  //memory,db
```

In the above example, we initialize paper phases to the assignment phase as soon as we move from the conference bidding phase to the conference paper assignment phase. Initialization of the entire system, for example Continue's conference phase, requires similar initialization. See section 5 for more.

3.4 Changing jobs

A common action is to add an element to a relation or remove an element from a relation, such as changing a user's job. An example of this was shown in the introduction.

3.5 Sequential Programs

For a given state and its input relations, the next state and resultant output relations are deterministic.[2] However, concurrency is possible. For example, when describing transitions, we actually allow multiple users to perform actions in parallel. This is achieved by specifying the current user attempting to perform an action as a member of every **Input** relation:

```
IF EditConferenceInfo (curuser, conference-info) THEN    //input
    IF Admin (curuser) THEN     //memory
            ConferenceInfo = conference-info    //memory
            InfoChanged     // output
        ELSE
            ActionFailed(curuser)             // output

IF ModifyUserInfo (curuser, user, name) THEN     //input
    IF Admin (curuser) OR (curuser = user) THEN    //memory
        (forall names : NOT User (user, names))    //memory
        User (user, name)            //memory
        UserModified(user)            //output
    ELSE
        ActionFailed(curuser)            //output
```

These two update rules (each of the outer IF blocks) are applied independently at the top level, so the pair of rules allows different users to modify user information and conference information at the same time. If a conflicting update occurs, such as two different users updating ConferenceInfo to two different values in the same step, neither update occurs, which we call a no-op.

Since separate rules for the same relation have their update decisions combined, if a conflict of these decisions creates an inconsistent state for the model, we must address the possibility explicitly. Mutable relations in our policy often include a user parameter, as seen in the second rule, so we can typically avoid instances of multiple users making conflicting updates as the updates are parameterized by the user. We can write a precondition that throws out many multiple actions by the same user, effectively making input relations nullary functions parameterized by the user

[2]This results from the omission of the *choose* construct as mentioned earlier.

and preventing many conflicts from the same user, such as trying to change one's password to two different values in the same step.

Embracing parallelism is better than explicitly specifying when we do want to allow updating of relations to occur simultaneously because parallelism is arguably the more general case for a web application. As discussed, this default assumption of parallelism will generally not cause problems because most distinct rules do not affect the same relations and most rules that can be performed simultaneously by different users do not affect the same relations in conflicting ways and, more importantly, captures our desired intent; servers should support concurrent use. However, this is not a guarantee that conflicting updates will not occur. Care must still be taken in this area and is the motivation for our module composition operator, which gives control over the scope of conflicting updates.

The above shows that we can naturally express fundamental notions of the Continue policy.

4 Complexity Invariants

Spielmann presents complexity results showing that verifying properties of programs written in the formalism he describes are decidable in PSPACE [13, 14]. To achieve this result, Spielmann introduces the following two constraints:

4.1 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.

4.2 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 unbounded input forms on-line, 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 has to start queuing. Given these physical realities, assuming the maximal input flow is bounded is acceptable.

We see that Spielmann's assumptions, which are sufficient to achieve PSPACE complexity results, are reasonable in the case of Continue.

5 Cons of Modeling Continue as an ASM Relational Transducer

The specification and verification languages of ASM relational transducers present a number of difficulties. The language for specification tends to lack some desired expressiveness, forcing us to write facts and properties in a counterintuitive manner. Among the more prominent issues are the inability to separate the program from the access control policy, difficulty with initializing the state of the transducer, and difficulty in resolving conflicts among rule interpretations (i.e. the appearance of no-ops).

Perhaps the largest issue is the inability to separate the access control policy from the actual program rules. For example, when writing the rules for adding a conflict for a particular user and paper, it is necessary to check whether the user is a reviewer or an administrator in the rule itself. This mixes the business logic level allowance of adding conflicts with the access control policy restriction of who may add a conflict. We also see this problem in other proposed models: we have an underlying model of business logic, and want to express authorization rules further restricting transitions of the business model, yet are forced to describe both together, conflating concepts.

A possible approach would be to separately specify a policy and provide a transformation operator to make the ASM conform to the policy. Recent related work by Pucella [11] explores modifying an FSM, focusing on what paths would change if the policy were to change mid-execution or have a schedule of changes.

Forcing the policy to be combined with the program is a poor formalization strategy. Besides making the rules of the program much harder to read, this is confusing two fundamentally different tasks. If a certain property fails verification, it is extremely beneficial to know whether the failure was due to a mistake in the rules of our program or in the access policy. Thus, the failure to cleanly separate the two is a weakness of our model. On the other hand, there are very few systems that fully, if at all, separate policy from program. Given this common practice, the ASM Relational Transducer model is no worse than the rest.

Initialization of state in a transducer is not obvious nor is establishing invariants. The general problem is encountered when we want to verify the continual existence of at least one administrator. While it is straightforward to maintain the invariant by never permitting the removal of an administrator when there is only one administrator, it is not directly apparent how to establish that, from the beginning, there must be at least one administrator. In order to create initialization invariants, it is sufficient to place an extra "AND" clause on every rule stating "AND ModelIsInitialized" in our policy. This is implicitly applied to all rules in our model. Initialization can be dealt with easily by making the ASM finitely initialized, so all memory relations begin empty [13].

This method of initialization requires wrapping a large IF-THEN clause around the main program. If ModelIsInitialized is empty, then initialize memory. If ModelIsInitialized is non-empty, then the main program runs. The body of the rule for the former case initializes memory as needed and, finally, sets the ModelIsInitialized relation. Thus, all the other memory and output rules may run only after the first transition in which the memory is initialized. While this at first seems like a strange side effect of using ASM relational transducers, it is actually natural:

In almost any system that can be modeled, there is some conceptual point where the system is set up before beginning the first run of the system. For example, when starting a computer, the BIOS or firmware initializes peripherals before allowing the operating system to run. When starting a desktop application, the operating system maps the executable into a new process's address space before allowing the program to execute. In any case, before this type of event, all other input to the system is invalid, ineffectual, or has unexpected consequences. By triggering or waiting for this event, the system is able to initialize its own state, which had previously been empty. Once it is ready, the program starts accepting a wider variety of user input. Our IF-THEN wrapper maps well to this initialization step. The body of the THEN clause will not execute more than once, but initializes several memory relations of the transducer to prepare it for future runs.

A related issue is that an initialized state must satisfy certain invariants. In an Alloy program, some invariants are facts which are enforced while generating the model, not necessarily properties verified by the model after generation [9]. In ASMs most initialization values come from constant Database relations, yet these may map to the same points even when that is exactly what we do not want. Specifically, if we describe our phases as constants in the Database relation set, how can we guarantee that the constants do not have the same value?

There is no way to specify initialization invariants in the database within the model, though we can check them in our properties. Thus we simply assure that such constants are not equal by effectively stopping the run of the transducer if such invalid information is in the database. In the initialization rules, we create guards checking that the constants are not equal. If they are equal, the guard fails, and thus the remainder of the program nested inside the conditional is not run. Despite preventing invalid models from transitioning, this approach is not satisfying as it still allows invalid models to exist in the space of models we will verify our properties against. Further vigor must be used to ensure that we consistently only consider the intended subset of possible models.

The final major problem is the counter-intuitive semantics of no-ops. It should be noted that we see merits of not defining the occurence of no-ops to be program failures, but determining whether they impact a policy fragment being written is extremely tedious. For example, if two inputs, applied independently, cause conflicting changes to memory regarding the review of paper, a no-op would occur if both inputs were presented together, and thus the relevant part of the state would remain unchanged. However, determining that a no-op occurred and changing the output and memory based on a no-op occurring is usually difficult. It is possible to detect input that causes no-ops, but these cases must be explicitly dealt with, requiring the construction of large guards.

Language support to detect no-ops either within rule specification or property specification is invaluable. We can verify properties to track down no-ops, but this approach is not simple. For example, no-ops only occur for memory, not for output. If every memory update coincides with an output update, we must check that output updates imply memory updates. Thus, we can detect that a no-op occurs, but it is difficult to tell why it occurred or control the occurrence of no-ops in an easy manner.

No-ops provide a semantically simple way to resolve conflicts by default among relations: do nothing. However, their occurrence made authoring and modifying output and memory update rules extremely difficult. While writing the rules in the standard ASM language, the possibility of a no-op was always on the forefront of our mind. We both tried to avoid them and, when we recognized their possibility, had to modify our rules to cover them. This is a fundamental problem, despite the power of no-ops.

As a brief note, we emphasize that the ASM language has an inconsistent use of quantification. Namely, in the guard of a rule, no quantification is needed; the seemingly free variables are actually implicitly universally quantified. However, in the program after the guard, it is necessary to quantify free variables. This would often lead to confusion when writing rules over what was bound and what was not bound as requiring this dual-syntax for binding variables is syntactically inconsistent. In any case, we cannot determine any real significance to this other than a minor syntactic inconvenience when writing the rules.

6 Pros of Modeling Continue as an ASM Relational Transducer

There are many properties of ASM transducers that are useful for modeling Continue: automatic concurrency, straightforward semantics of the handling of contradictory decisions, ease of transition due to the use of update sets for the state transitions, and expressive power.

Because the number of relations serving as input to each state can be greater than one, each state transition can encompass multiple inputs. Thus, the system models concurrent requests well. This does open the door to contradictory input, but because contradictory updates are ignored (no-ops), the system will be in a predictable state if such a thing occurs - neither action succeeds.

The main issue with no-ops is detecting that a contradiction occurred when specifying rules, which is not intuitive in our policy; the state rules must simply be modified manually to check for the contradictory input explicitly, which has the potential to miss some possible conflicts. For example, by checking if there are two individually valid requests to change the same user's password to two different things, the only checks needed are to see whether the user in question for two password-change requests is the same, that the old-password is the same, and that the requested new passwords are different. An approach to verify the lack of no-ops is to check that whenever we have an output, memory is properly updated, as generally, every memory update is associated with an output. If an update of an output relation occurs but the corresponding memory update does not, we clearly have a no-op. Another issue with this approach is that it requires that one portion of a policy be written in a way which creates dependency on the way another portion of the model is written. This is not insurmountable, but is undesirable.

As any relations not explicitly modified remain the same after a transition, the memory rules can be much shorter than in other specification languages, such as Alloy, where anything which does not change must have some statement explicitly maintaining consistency between states. This allows multiple changes at once.

Finally, and perhaps most importantly, despite any awkwardness or shortcomings of using ASM relational transducers to model Continue, they are more than expressive enough. Writing slightly odd rules is a reasonable price to pay for a language powerful enough to express our system without difficult encodings. Overall, we found that ASM relational transducers were more than sufficient to express Continue while remaining fairly intuitive.

7 Composition of ASM Modules and ASM No-op Reporting

Sections 3 and 5 present intricacies of policy authoring using the basic ASM Relational Transducer language. When writing update rules, it is unclear whether all of the updates within a set of rules occur without another conflicting update occuring elsewhere in the policy. We describe how to partition a policy into modules that can be individually reasoned about about and can be composed.

Given a set of update rule modules, we describe a composition operator that satisfies certain properties relating to the atomicity of decisions made by a module and about the origin of any decision made. To do so, we define modules and a translation from ASM relational transducers extended with modules to the original ASM relational transducer language. Then, we formalize the notion of safety our composition operator must satisfy and show the composition algorithm is correct with respect to the composition property. Finally, we describe an alternative translation from the module language to the original relational transducer language that reports any no-ops that occur.

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, reminiscent of atomicity properties. We suggest two properties our composition operator must satisfy:

1. No extra decisions (including no-ops): A decision on a relation in the composition occurs only if that same decision is due to one or several modules being composed.
2. Module atomicity: To say a decision in the composition is due to one (or several) of the modules being composed, these modules being composed cannot make decisions that conflict with decisions made by other modules. Therefore, to say one applicable update in a module occurs, all of the applicable updates in the module must occur.

We combine these two properties into one stronger formal property stating that a decision occurs in the composition modules if and only if it also occurs as a decision in one of the modules such that all the decisions in that module do not conflict with those of any other modules being composed. A key result of this property is that any no-op that occurs is intentional and therefore not introduced by the composition operator.

In section 7.2 we formally describe the properties a composition operator must specify, in section 7.4 we write an algorithm to perform the composition of modules, and finally in section 8 we prove our algorithm is correct with respect to our desired composition properties.

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 relational transducer language that creates an additional outputs detailing whatever no-ops occur in a transition. The composition property shows that these no-ops 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 no-op derives, but we do not pursue this issue. Our intuition is that a technique similar to that used in reporting the original no-ops can be used.

7.1 Definition of a Module

We define a module in terms of ASM update rules. We assume an ASM program can be processed into a normal form consisting of a sequence 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 defined within the test condition of an ASM IF-THEN update rule. 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), the name of the relation r_k to update, the applicability guard ug_k for the update rule, the parameters of the update ${}_k\vec{p}$ with inline 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:

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

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

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

If we turn this into a module, we have:

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

$$m = (true, u_1) \quad (2)$$

While it may be possible to combine update and module guards, keeping them distinct makes sense: all of the variables of a relation being update are bound within the update guard, and when composing modules, the update rules do not change.

The ASM program $\pi_\mathbf{i} \equiv [\![\mathbf{m_i}]\!]$ representingmodule $\mathbf{m_i}$:
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:

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

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. Additionaly, for brevity, we will write $f(\vec{p}, UB)$ as $\vec{q}$ in future sections.

7.2 Desired Properties of Module Composition

We describe the main property a composition operator $\otimes : 2^{\{m_i\}} \rightarrow m_j$ must satisfy:

Given input module set $\{m_i\}$, the composed module $m_j = \otimes(\{m_i\})$ must translate into an ASM the makes 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.

Formally:

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

Where:

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

$$conflict(d, d\prime) = \begin{cases} false & d = d' \vee d = NA \vee d' = NA \\ true & else \end{cases}$$

7.3 No-op Reporting

The occurrence of no-ops during the transitions of an ASM can confuse the understanding of a policy because some updates written may not occur in the final transitioned state. Below, we describe an alternative translation from a module into an ASM, creating a new no-op relation $r_{NOOP,i}$ for every regular output relation r_i and adding the additional output of which relations' updates no-op during a transition:

No-op Reporting:

$$[\![m]\!]_{nreport} \equiv \begin{cases} IF \ mg_1 \ \wedge \ ug_1 \ THEN \\ \quad \forall ub_1, \cdots, \forall ub_{|UB_1|} \ s_1 r_1(f({}_1\vec{p}, UB_1)) \\ \quad IF \ (mg_{s,1,1} \ \wedge \ ug_{s,1,1} \ \wedge \ \vec{p}_1 = \vec{p}_{s,1,1}) \vee \\ \qquad \qquad \cdots \\ \qquad \vee \ (mg_{s,1,|similar(r_1,m)|} \ \wedge \ ug_{s,1,|similar(r_1,m)|} \ \wedge \ \vec{p}_1 = \vec{p}_{s,1,|similar(r_1,m)|}) \\ \qquad THEN \\ \qquad \quad \forall ub_1, \cdots, \forall ub_{|UB_1|} \ s_1 r_{NOOP,1}(f({}_1\vec{p}, UB_1)) \\ \quad \vdots \\ IF \ mg_n \ \wedge \ ug_n \ THEN \\ \quad \forall ub_n, \cdots, \forall ub_{|UB_n|} \ s_n r_n(f({}_n\vec{p}, UB_n)) \\ \quad IF \ (mg_{s,n,1} \ \wedge \ ug_{s,n,1} \ \wedge \ \vec{p}_n = \vec{p}_{s,n,1}) \vee \\ \qquad \qquad \cdots \\ \qquad \vee \ (mg_{s,n,|similar(r_n,m)|} \ \wedge \ ug_{s,n,|similar(r_n,m)|} \ \wedge \ \vec{p}_n = \vec{p}_{s,n,|similar(r_n,m)|}) \\ \qquad THEN \\ \qquad \quad \forall ub_n, \cdots, \forall ub_{|UB_n|} \ s_n r_{NOOP,n}(f({}_n\vec{p}, UB_n)) \end{cases}$$

7.4 Formal Algorithm

INPUT: $\{m_i\}$
$result \leftarrow \emptyset$

For each $r \in Memory \cup Output$:
function getPrime $(r_k, \{m_i\}) =$

$$\left(\bigvee_{\left\{\substack{(mg_k,u_k)\in\{m_i\}:\\ r_k=r,\\ s_k=+}\right\}} mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k \atop \bigwedge \bigwedge_{\left\{\substack{(mg_k,u_k)\in\{m_i\}:\\ r=r_k,\\ s=-}\right\}} \neg(mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k) \right)$$

$$\left(\bigvee_{\left\{\substack{(mg_k,u_k)\in\{m_i\}:\\ r_k=r,\\ s_k=-}\right\}} mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k \atop \bigwedge \bigwedge_{\left\{\substack{(mg_k,u_k)\in\{m_i\}:\\ r=r_k,\\ s=+}\right\}} \neg(mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k) \right)$$

$$\bigvee \left(\bigwedge_{\left\{\substack{(mg_k,u_k)\in\{m_i\}:\\ r=r_k}\right\}} \neg(mg_k \wedge ug_k \wedge \vec{q}_1 = \vec{q}_k) \right)$$

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

```
(1) $copy = \{m_i\}$
(2) FOR EACH $m_l \in copy$
(3)         FOR EACH $(mg_k, u_k) \in m_l$
(4)                 FOR EACH $(mg_x, u_x) \in m_l$
(5)                         $mg_x \leftarrow getPrime(r_k, \{m_i\}) \wedge mg_x$
(6)         FOR EACH $(mg_k, u_k) \in m_l$
(7)                 $result \leftarrow result \cup \{(mg_k, u_k)\}$
```

(8) return $result$

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).

8 Proof of Algorithm Correctness

We prove the module composition algorithm of section 7.4 is correct with respect to the property stated in section 7.2.

- Lemma 1: Every update rule in a composition comes from an update rule in some module being composed, with the module guard having an additional formula mg' appended to the original module guard by conjunction.
  Formally, both:
  1. $\forall m_j = \otimes(\{m_i\}), \forall(mg_k'', u_k) \in m_j, \exists mg_k' \wedge mg_k = mg_k'' \exists m_l \in \{m_i\} \ . \ (mg_k, u_k) \in m_l$
     Proof:
     (a) There is no instruction that adds or removes tuples to the set *copy* after it is initialized from the unaltered $\{m_i\}$. The only modifications to set *copy* modify module guards, which occurs on line 5. Thus, any u_k in any tuple in the set *copy* can be found in the original input set.
     (b) At any step of execution of the algorithm, any module guard of any update rule in any reachable module in *copy* can be written as the conjunction of some formula with the original module guard of some update rule from the original input set, $\{m_i\}$:
        i. Inductive Assumption: Prior to any instruction, for any tuple $(mg_k'', u_k'') \in m_l \in copy, \exists mg_k' \ . \ mg_k'' = mg_k' \wedge mg_k$ for some tuple $(mg_k' \wedge mg_k, u_k) \in m_l$ for some m_l in the original input.
        ii. Base Case: Prior to any instruction, every tuple $(mg_k, u_k) \in m_l \in copy$ can be found in the original input set $\{m_i\}$ because the set *copy* is a duplicate of the input set, and can be rewritten as $(true \wedge mg_k, u_k)$. Therefore $mg_k' = true$.
        iii. Inductive Step: The set *copy* is only mutated at one line after duplication. The left hand side of the mutation is mg_k''. The left side of the conjuction is mg_α.

By the inductive assumption, the right side of the conjuction can be written as $mg_\beta \wedge mg_k$. Let $mg_k\prime = mg_\alpha \wedge mg_\beta$. Let $mg_k\prime = mg_\alpha \wedge mg_\beta$. u_k is not modified therefore the inductive assumption is true.

(c) Our result set starts as an empty set, thus satisfying the property. The only modifications to the result set occur on line 7, which adds elements of a module of a modified set, *copy*. Any modification of the result set yields a set satisfying the property because by (b) all the elements from a modified set, *copy*, satisfy the property.

2. $\forall m_j = \otimes(\{m_i\}), \forall m_l \in \{m_i\}, (mg_k, u_k) \in m_l,\ \exists mg_k\prime\ .\ (mg_k \wedge mg_k\prime, u_k) \in m_j$
   (a) Fix some update rule $(mg_k, u_k) \in m_l \in \{m_i\}$. By 1.(b), this update rule will always be in the input set, $\{m_i\}$, in some modified form satisfying the property.
   (b) As any modified set, *copy* contains a form of the update rule that satisfies the property, and lines 2, 3, and 6 iterate over the entire modified set, *copy*, the update rule must be in the outputted result set.

- Lemma 2: All formulas extending module guards (the $mg\prime$) during composition are the same for a given module:

$$\forall m_j = \otimes(\{m_i\}), m_l \in \{m_i\}, \forall(mg_{a,l}, u_{a,l}), (mg_{b,l}, u_{b,l}) \in m_l\ .\ mg_{a,l}\prime = mg_{b,l}\prime$$

1. Inductive Assumption: Before every iteration of the loop on line 4 ($(mg_k, u_k) \in m_l$), the appended guard of every module guard in the corresponding module is the same.
2. Base Case: Same base case as Lemma 1, part 1.b.ii (implicitly append *true*)
3. Inductive Step: Each module guard is modified only by appending $getPrime(r_k, \{m_i\})$, line 5. This is done for every rule in a given module. Because *getPrime* is a function of r_k and $\{m_i\}$, neither of which is ever modified in the algorithm, the appended guards will all be the same.

- Lemma 3: A module conflict makes the mg' appended to the original mg by the algorithm false:

$$\forall m_j = \otimes(\{m_i\})\ .\ \forall m_l \in \{m_i\}, \forall r, \vec{q}$$
$$\exists m_k \in \{m_i\}\ .\ conflict(decision(r, \vec{q}, m_l), decision(r, \vec{q}, m_k))$$
$$\Rightarrow$$
$$\forall(mg_a, u_a) \in m_l \cup m_k\ .\ \neg mg_a\prime$$

Without loss of generality, consider two cases such that $conflict(d, d\prime)$:

1. Case $d = +, d\prime = -$:
   $\exists m_k \in \{m_i\}, r, \vec{q}\ .\ decision(m_l, r, \vec{q}) = + \wedge decision(m_k, r, \vec{q}) = -$
   Therefore, the four clauses of $getPrime()$ are false for r.

For each update rule u in m_l, $mg_{u\prime} = (... \wedge getPrime(r, \{m_i\}) \wedge ... \wedge mg_u)$ and thus mg_u is $false$ for all $u \in m_l$ by Lemma 1.

2. Case $d = +, d\prime = NOOP$:
   $\exists m_k \in \{m_i\}, r, \vec{q}$. $decision(m_l, r, \vec{q}) = + \wedge decision(m_k, r, \vec{q}) = NOOP$
   Therefore, the four clauses of $getPrime()$ are false for r.
   For each update rule u in m_l, $mg_{u\prime} = (... \wedge getPrime(r, \{m_i\}) \wedge ... \wedge mg_u)$ and thus mg_u is $false$ for all $u \in m_l$ by Lemma 1.

The remaining cases can be proved by communtivity of boolean logic and α renaming.

- Lemma 4: $\forall m_l, m_k \in \{m_i\}, \forall r, \vec{q}, \neg\exists conflict(decision(r, \vec{q}, m_l), decision(r, \vec{q}, m_k)) \Rightarrow \forall n \, mg_{n,l\prime}$
  Without loss of generality, consider two cases such that $\neg conflict(d, d\prime)$:

  1. Case $d = +, d\prime = +$:
     $decision(m_l, r, \vec{q}) = + \wedge decision(m_k, r, \vec{q}) = +$
     Therefore, the first clause of $getPrime()$ is $true$.
  2. Case $d = NA, d\prime = NA$:
     $decision(m_l, r, \vec{q}) = NA \wedge decision(m_k, r, \vec{q}) = NA$
     Therefore, the third clause of $getPrime()$ is $true$.
  3. Case $d = +, d\prime = NA$:
     $decision(m_l, r, \vec{q}) = + \wedge decision(m_k, r, \vec{q}) = NA$
     Therefore, the first clause of $getPrime()$ is $true$.
  4. Case $d = NOOP, d\prime = NA$:
     $decision(m_l, r, \vec{q}) = NOOP \wedge decision(m_k, r, \vec{q}) = NA$
     Therefore, the fourth clause of $getPrime()$ is $true$.

  The remaining cases can be inferred. Any non-conflicting decision will satisfy at least one clause of $getPrime()$. Since these clauses are joined by disjunction, every $mg_{l\prime}$ will be $true$.

Proof of correctness of the algorithm in section 7.4 with respect to the formal property stated in section 7.2:

Fix any r, $\vec{q}$ and show both directions of the bidirection hold:
Proof of $\Rightarrow$:

1. Case $decision(r, \vec{q}, m_j) = +$:
   a. $\exists m_l \in \{m_i\}$. $decision(r, \vec{q}, m_l) = +$
      Let A $= \left\{(mg_{k,l} \wedge mg_{k,l\prime}, u_{k,l}) : {}^{(mg_{k,l}, u_{k,l}) \in mg_l \in \{m_i\},}_{r = r_k, \vec{q} = \vec{q}_k, s_k = +}\right\}$
      By Lemma 1 and $decision(r, \vec{q}, m_j) = +$:
         $\exists m_l \in \{m_i\}, (mg_{k,l} \wedge mg_{k,l\prime}, u_{k,l}) \in A. \; mg_{k,l} \wedge mg_{k,l\prime} \wedge ug_{k,l}$
      Assume for contradiction $\exists (mg_{a,l}, u_{a,l}) \in m_l$. $r = r_a \wedge s_a = - \wedge \vec{q} = \vec{q}_a \wedge mg_{a,l} \wedge ug_{a,l}$
         By Lemma 2, $mg_{k,l\prime} \rightarrow mg_{a,l\prime}$
         By Lemma 1, $(mg_{a,l} \wedge mg_{a,l\prime}, u_{a,l}) \in m_j$
         Thus, $decision(r, \vec{q}, m_j) = NOOP$

The assumption led to a contradiction in $decision(r, \vec{q}, m_j)$, so $decision(r, \vec{q}, m_l) = +$

b. $(mg_{k,l} \wedge mg_{k,l}\prime, ug_{k,l}) \in A \;\wedge\; mg_{k,l} \wedge mg_{k,l}\prime \wedge ug_{k,l} \Rightarrow$
$\neg\exists r\prime, \vec{q}\prime, m_n \in \{m_i\}.conflict(decision(r\prime, \vec{q}\prime, m_l), decision(r\prime, \vec{q}\prime, m_n))$:
Assume for contradiction $\exists r\prime, \vec{q}\prime, m_n \in \{m_i\}$.
$conflict(decision(r\prime, \vec{q}\prime, m_l), decision(r\prime, \vec{q}\prime, m_n))$
By Lemma 3, $\neg mg_l\prime$
Contradiction so assumption false:
$\neg\exists r\prime, \vec{q}\prime, m_n \in \{m_i\}.conflict(decision(r\prime, \vec{q}\prime, m_l), decision(r\prime, \vec{q}\prime, m_n))$

c. Thus, we showed at least one module in $\{m_i\}$ makes the appropriate positive update decision, and this same module does not conflict with any others.

2. Case $decision(r, \vec{q}, m_j) = -$:
The proof follows by switching the '+' and '−' symbols in the preceding case.

3. Case $decision(r, \vec{q}, m_j) = NOOP$:
The structures is similar to case 1, with the following modifications:
a. Modify set A so that instead of containing applicable positive update rules for the given relation and input, it contains pairs of applicable rules for a given relation and input where one member of the pair is positive, and the other negative.
b. The contradiction is achieved by assuming at least one of the elements in the pair for a given relation and input is not present in the module m_l. Then the decision will not be NOOP, creating a contradiction.

4. Case $decision(r, \vec{q}, m_j) = NA$:
a. Consider some module m_l in the input set. By the other 3 cases:
1. Case $decision(r, \vec{q}, m_l) = +$:
By II.1, if m_l has no conflicts, then $decision(r, \vec{q}, m_j) = +$.
This is a contradiction, thus
$\neg decision(r, \vec{q}, m_l) = + \rightarrow \exists m_k \in \{m_i\}.conflict(decision(r, \vec{q}, m_l), decision(r, \vec{q}, m_k))$
2. Case $decision(r, \vec{q}, m_l) = -$:
This case follows from II.2.
3. Case $decision(r, \vec{q}, m_l) = NOOP$:
This case follows from II.3.
b. As decisions are well defined, with the range of $\{+, -, NOOP, NA\}$, all modules with decisions that are not NA have been shown to have conflicts.

Proof of $\Leftarrow$:
1. Case $decision(r, \vec{q}, m_l) = +$ and no conflicts for m_l:
a. $\exists(mg_{k,l} \wedge mg_{k,l}\prime, u_{k,l}) \in m_j.r_{k,l} = r \;\wedge\; s_{k,l} = + \;\wedge\; \vec{q}_{k,l} = \vec{q}$:
$decision(r, \vec{q}, m_l) = + \Rightarrow \exists(mg_{k,l}, u_{k,l}) \in m_l \in \{m_i\}$.
$mg_{k,l} \wedge ug_{k,l} \wedge\; r_{k,l} = r \;\wedge\; s_{k,l} = + \;\wedge\; \vec{q}_{k,l} = \vec{q}$
By Lemma 4, no conflicts for $m_l \Rightarrow \forall k.mg_{k,l}\prime$
Thus, $\exists(mg_{k,l} \wedge mg_{k,l}\prime, u_{k,l}) \in m_i$.

$mg_{k,l} \wedge mg_{k,l'} \wedge ug_{k,l} \wedge r_{k,l} = r \;\wedge\; s_{k,l} = + \;\wedge\; \vec{q}_{k,l} = \vec{q}$

b. $\neg\exists(mg_{k,l} \wedge mg_{k,l'}, u_{k,l}) \in m_j \;.\; r_{k,l} = r \;\wedge\; s_{k,l} = - \;\wedge\; \vec{q}_{k,l} = \vec{q}$:

Assume to achieve a contradiction that:

$\exists(mg_{e,f} \wedge mg_{e,f'}, u_{e,f}) \in m_j.$

$r_{e,f} = r \;\wedge\; s_{e,f} = - \;\wedge\; \vec{q}_{e,f} = \vec{q}$

By Lemma 1,

$\exists(mg_{e,f}, u_{e,f}) \in m_f \in \{m_i\}.$

$mg_{e,f} \wedge ug_{e,f} \wedge r_{e,f} = r \;\wedge\; s_{e,f} = - \;\wedge\; \vec{q}_{e,f} = \vec{q}$

$decision(r, \vec{q}, m_f) \in \{-, NOOP\}$

Contradiction as $conflict(decision(r, \vec{q}, m_l), decision(r, \vec{q}, m_f))$, contrary to given.

c. By (a.) and (c.), all update rules in m_j for $r, \vec{q}$ yield are either additive (+) or not applicable, and there exists at least one applicable additive update in m_j for $r, \vec{q}$, so $decision(r, \vec{q}, m_j) = +$

2. Case $decision(r, \vec{q}, m_l) = -$ and no conflicts for m_l:

The proof follows by switching the '+' and '−' symbols in the preceding case.

3. Case $decision(r, \vec{q}, m_l) = NOOP$ and no conflicts for m_l:

Similar to case 1 part a, as the decision of a nonconflicting module being composed is a NOOP, there are both applicable positive and negative update rules for the given relation and input. Irrespective of any other update rules in the composition, the decision for the given relation will be a NOOP.

4. Case $decision(r, \vec{q}, m_l) = NA$ or there exists a conflcit for m_l:

Case $decision(r, \vec{q}, m_j) = +$:

By I.1, $decision(r, \vec{q}, m_j) = + \rightarrow \exists m_k \in \{m_i\}.decision(r, \vec{q}, m_k) = +.$

$\forall\neg m_b \in \{m_i\}.conflict(decision(r, \vec{q}, m_k), decision(r, \vec{q}, m_b))$

By assumption, for any such m_l there is a conflict.

Therefore there is a contradiction and thus $decision(r, \vec{q}, m_j) \neq +$.

Case $decision(r, \vec{q}, m_j) = -$:

By I.2, $decision(r, \vec{q}, m_j) = - \rightarrow \exists m_k \in \{m_i\}.decision(r, \vec{q}, m_k) = -.$

$\forall\neg m_b \in \{m_i\}.conflict(decision(r, \vec{q}, m_k), decision(r, \vec{q}, m_b))$

By assumption, for any such m_l there is a conflict.

Therefore there is a contradiction and thus $decision(r, \vec{q}, m_j) \neq -$.

Case $decision(r, \vec{q}, m_j) = NOOP$:

By I.3, $decision(r, \vec{q}, m_j) = NOOP \rightarrow \exists m_k \in \{m_i\}.decision(r, \vec{q}, m_k) = NOOP.$

$\forall\neg m_b \in \{m_i\}.conflict(decision(r, \vec{q}, m_k), decision(r, \vec{q}, m_b))$

By assumption, for any such m_l there is a conflict.

Therefore there is a contradiction and thus $decision(r, \vec{q}, m_j) \neq NOOP$.

As decisions are well defined, with the range of $\{+, -, NOOP, NA\}$, all decisions of the output module m_j other than NA have been shown to have contradictions, therefore $decision(r, \vec{q}, m_j) = NA$.

9 Evaluation of Algorithm Output

We applied the algorithm to a set of modules representing the Continue policy.

9.1 Features Used in Practice in Continue

While rewriting our Continue policy to make use of modules, we noticed several interesting properties of our policy with modules before it was transformed into the ASMRT language. First, while intentional no-ops were allowed, taking advantage of them was neither necessary nor natural when writing the policy with modules. Furthermore, all of the conflicts between modules were conflicts between $+$ and $-$ decisions, rather than between either of those and $NOOP$ decisions. This is a result of the previous point, but still worth noting because much work was done to address the latter case. Both the correctness proof and algorithm itself could potentially be simplified if this concern were removed. Another feature we never utilized was the ability to nest modules within each other. This never felt necessary in writing the policy with modules, though it is probable that larger policies would benefit from such nesting. In contrast, previous work by Tschantz et al in modelling Continue [5] with XACML heavily utilized nesting.

9.2 Policy Bugs Encountered

The majority of the bugs encountered in the original policy were due to unintentional no-ops. It should be noted that we recognized this failing while writing the original policy which motivated the development of the composition operator. The module system fixes these bugs in the original policy. Adding one update rule without the module system could potentially introduce dozens or more unintentional no-ops, the prevention of which would require many additional guards. With the composition operator, the generation of these guards is automated. The module system limits the scope of conflicts by preventing no-ops between modules.

9.3 Policy Bugs Fixed by the Module System

There are well over 20 unintentional-no-op-related bugs, roughly uniformly distributed, in the original policy's first hundred lines of code, and the rest of the policy demonstrates similar flaws. These were fixed by converting to modules and composing.

9.4 Observations About the Module System

In the ASMRT output by our algorithm for the Continue policy, there is a significant increase in the size of the module guard for any module which has a rule updating a relation updated by another module. In our case, the guard of the module handling the **RemoveUser** input relation increases nearly ten times in the number of conjunctions because it updates almost every memory relation in the system, and therefore must be guarded against no-ops with almost every other module. However, this is the worst case scenario and the exception for the Continue policy. Most modules in the modularized Continue policy update a very small number of relations, usually not greater than 4. In most systems, there are very few events which will require updates to the majority of the system. This could also be a result of the way we constructed the Continue policy; every action and relation in our policy is parameterized by user. It is possible that a different parameterization of the system would not have any individual input which could cause system-wide updates. It is worth noting that correctly writing the policy without modules such that unintentional no-ops do not occur would require the author to generate these same guards by hand. Thus the resulting policy is no larger or more complex to verify than the correct policy generated without using modules, despite being easier to write.

The module system allows a large degree of flexibility in that an arbitrarily large number of update rules can appear in a module. These sets of update rules succeed and fail atomically which introduces a new type of programming error. By including too many updates in a module, it is possible to have some updates unnececssarily fail to execute. Similarly, including too few updates to address certain input relations has the potential to allow some updates to execute even when correctly modeling the system at hand requires that they not. This suggests that conversion of a policy into modules is not absolutely straightforward. In practice, however, we have found that the choice of where to separate blocks of policy into modules is actually clear. We chose to construct a module for every potential individual input relation, and have the body of each module deal with any and all updates relevant to that one input relation.

The module system is fairly intuitive to use, powerful, and helps to prevent many occurrences of potentially serious errors. On top of this, it makes policy updates much easier. Updating a properly-guarded hand-written policy would require revisiting the entire policy to ensure that all parts were appropriately guarded. With the module system, it is possible to update one small module appropriately, and know that the impact of the change is limited, guaranteed by the composition algorithm which satisfies our saftey property.

9.5 Related Work

Spielmann's work [12, 13, 14] provides us with a basis for the verification of ASM Relational Transducers. His work introduces the idea of modeling Relational Transducers as ASMs in order to increase their power and to connect them to a language shown to be useful in practice. Spielmann does not develop an atomic update system because his focus is simply on developing the use of ASMs as Relational Transducers. However, atomic sets of updates turn out to be an extremely useful capability for Relational Transducers and other database-oriented systems.

Because Spielmann's work does not provide atomic updates, actually modeling a parallel system with his ASM Relational Transducers becomes very cumbersome. If one desires an atomic set of updates, it becomes necessary to effectively implement the module system described earlier, hand encoding each dependency. This is especially important when considering the modeling of applications such as web programs that will almost ceratinly have multiple points of parallel access, each one potentially adding multiple conflicts.

Existing abstract state machine systems, including ASM-Workbench [4], XASM [2], and ASML [8], provide means of composing programs. The general approach in this area is similar to our own by allowing the external definition of a subprogram and then including it in the parent. Unless there is a subsequent program transformation to enforce interesting properties, we will not dwell on this case in the following descriptions. Despite the variety of composition techniques, each of these systems are still susceptible to the consistent update problem and do not prevent the introduction of inconsistencies when composing consistent programs.

ASM-Workbench [4], with the underlying ASML-SL language, utilizes rule-macros. Large rule-macros are called programs and support naming and thus embedding. Additionally, appeals to the outside world or fucntions that are not know at the time of modeling can be described with exteranl functions that are included in basic type checks. Rule-macros support composition similar to that found in [3].

XASM [2] provides relatively rich support of modules. External functions may be called within an ASM program, running in their own time to return a value as well as an update set to be merged in to the normal updates in a transition. The return of this update set signifies that the consistency problem still exists. There is a similar ability to call environment functions as in ASM-Workbench.

ASML provides means of handling inconsistencies: it supports transactional semantics through exception handling. Rule may throw an exception that may be caught elsewhere, restoring state. However, both of these operations are manually specified with try/catch/throw constructs instead of being automatically caught at the module boundry. Additionally, it is not clear how to detect inconsistencies with this approach.

Nicolosi-Asmundo and Riccobene [3] present two forms of composition for building ASMs, the first being the naive approach and the second adding explicit structure to components in order to form a basis for guarantees. Feature composition is the intuitive composition technique: simply run multiple smaller ASMs in parallel. In feature composition, compatibility is assumed by appealing to external techniques, such as tests for arity and name checking. The second approach, called component composition, assumes certain relations to be used as communication channels between component ASMs in a style akin to message passing. It is necessary to check that these particular relations are used appropriately by each component, and the technique renames other relations in each component to effectively create separate namespaces for each component, modulo the communication channels.

Both composition techniques are useful. The former is well-suited for situations where basic composition is all that is necessary, but only if ther are no potential inconsistencies. The latter is well suited for complex systems with internal state within each component of a system. Both maintazin a safety property that any run of an individual component can be found in the runs of the composed system. However we do not strive to provide this guarantee as we are more interested in easily maintaining consistency and ensuring progress can be made.

Module composition provides an easier-to-specify approach to composition than component composition, while being safer than feature composition in situations where modified behavior is acceptable.

The Marianna et al. approach provides a useful safety property, and describes an intuitive technique and a structured technique for specifying composition, a task we automate.

10 Conclusion

We have shown that the Continue Conference Manager can be modeled by Abstract State Machine Relational Transducers (ASMRT). The business model supported by Relational Transducers provides a good abstraction for the web application realm of Continue. Furthermore, using first order logic to describe every state in ASMs provides the expressiveness needed to develop a full model for Continue.

Through this example, we have shown that ASMRTs can be used to model common ideas of fine-grained security policies. Revocation, separation of duties, and separation of privilege can all be simply modeled in ASMRTs.

We chose ASMRTs so we could automatically verify properties of the Continue Conference Manager thus guaranteeing correctness. This is done with First Order Temporal Logic (FTL), allowing us to verify fine-grained temporal properties of our model. We have shown that some of the weaknesses in the property language are balanced by additional expressiveness in the modeling language in the form of mutable state.

ASMRTs provide a simple way of resolving conflicting relation updates through the use of no-ops: if there is a conflict, do nothing with the involved relations. This is helpful since the modeler does not need to deal extensively with the handling of conflicts.

However, the automatic generation of no-ops is problematic. There is no language support for dealing with no-ops, such as exception handling. Thus, in order to explicitly manage no-ops, extra encoding is needed to identify and handle conflicts. It is also of concern that when a no-op occurs, the user of the system is not notified: the system silently fails. Thus, the user is unable to react appropriately.

In order to solve these problems, we have introduced modules and no-op reporting. By compartmentalizing an ASMRT model into modules, we have proved that one can apply our composition operator to generate an ASMRT model that does not allow inter-module conflicts. Furthermore, we showed a method of automatically generating user output notification every time a no-op occurs. Thus, users can react accordingly to intentional no-ops and properties can be verified about the occurance of no-ops.

In our example use of the composition operator on our Continue model, we demonstrate the usefulness of modules. In our original policy without modules, there were numerous bugs caused by unintentional no-ops. By using modules, we eliminated all of these.

The techniques that we developed, no-op reporting and the module system, are applicable not just to ASMs for security policies, but to ASMs in general. Thus, these two systems will allow for ease of development across various problem domains that utilize ASMs.

11 Future Work

As pointed out in section 9.4, it is possible to unnecessarily constrain some updates from occurring by including too many update rules in one module. However it may not always make sense from the policy-writer's perspective to have such a module separated into two modules with very similar module guards. It is possible that it may not matter if one or more of the updates in a given module conflict with other modules, but only matters if a conflict occurs within a specific subset. Therefore it may be worth investigating a way to mark a module such that only a specific set of relations within it are permitted to be involved in conflicts, and others relations are permitted to conflict with other modules without preventing the entire module from firing. Furthermore, if a module system is used, it may be of benefit to see which modules caused a no-op, not just that a no-op occurred. Finally, we believe separating policy from business logic and providing a transformation to join the two is crucial in future research in policy language models. Recent work by Pucella is moving in this direction [11].

References

[1] Serge Abiteboul, Victor Vianu, Brad Fordham, and Yelena Yesha. "Relational Transducers for Electronic Commerce". 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] 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*, page 324. Springer Berlin / Heidelberg, 2003.

[4] 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.

[5] 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.

[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] Daniel Jackson. "Alloy: a lightweight object modelling notation". *Software Engineering and Methodology*, 11(2):256–290, 2002.

[10] 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.

[11] Riccardo Pucella and Vicky Weissman. "Reasoning about Dynamic Policies". In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 453–467. Springer, 2004.

[12] 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.

[13] 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.

[14] 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.

Appendix: Output from Composing Continue Policy Modules

```
inputs:
    AdvancePhase (curuser)
    AdvancePaperPhase (curuser, paper)
    ModifySubmission (curuser, new-paper)
    AddConflict (curuser, user, paper)
    DecidePaper(curuser, paper, decision)
    ReviewPaper (curuser, paper, review)
    AddConflict (curuser, user, paper)
    ModifyUserPassword (curuser, old-password, new-password)
    ModifyUserInfo (curuser, user, new-name)
    EditConferenceInfo (curuser, new-info)
    UnassignPaper (curuser, user, paper)
    AssignPaper (curuser, user, paper)
    UnbidPaper (curuser, paper)
    BidPaper (curuser, paper)
    RemovePaper (curuser, paper)
    RemoveUser (curuser, user)

    ReadPaper (curuser, paper)
    ReadReview (curuser, review)
    CreateAuthor (username, name)
    CreateReviewer (curuser, newuser, name)

    //Job Changes
    changeUserJobToReviewer(curuser, user)
    changeUserJobToNotReviewer(curuser, user)
    changeUserJobToAdmin(curuser, user)
    changeUserJobToNotAdmin(curuser, user)

memory:
    Admin (user)
    Author (user)
    Reviewer (user)
    ConferenceInfo ()
    User (user, name)
    Password (user, password)
    Conflicts (user, paper)
    DecidedPapers (paper, decision)
    AssignedReviews (user, paper)
    PaperReviews (user, paper, review)
    PaperBids (user, paper)
    AcceptedPapers (paper)
    Papers (user, paper)
    CurrentPhase ()
    PaperPhase (paper, phase)

db:
    //DECISIONS
    Accepted
    Rejected

    //CONFERENCE PHASES
    Initialization
    PreSubmission
    Submission
    Bidding
    Assignment
    Reviewing
    Discussion
    Notification
    Publishing

    //PAPER PHASES
    PAssignment
    PReviewing
    PDiscussion
    PDecided

    InitialInfo
    InitialAdmin
    DefaultPassword

output:
    PhaseAdvanced(phase)
    PaperStateAdvanced(phase)
    NewPaperSubmission(user)
    ConflictAdded(user, paper)
    PaperDecision(paper, decision)
    UserModified(user)
    InfoChanged
    AssignmentModified(user,paper)
    BigModified(user,paper)
    PaperDeleted(paper)
    UserDeleted(user)
    PaperIsRead(user,paper)
    ReviewIsRead(user,paper)
    AuthorCreated(user,paper)
    ReviewerCreated(user,paper)
    ActionFailed(user)
    ReviewSubmitted(user, paper, review)

log: input U ouput


//===================================================================//


memory rules:
/*Initialization*/
    //Set up conference if it hasn't started.
    if NOT exists state . CurrentPhase = state then
        ConferenceInfo = InitialInfo
        Admin(InitialAdmin)
        CurrentPhase = Initialization

/* Other memory rules */
    //Phase Advancement
    if AdvancePhase(curuser) AND Admin(curuser)
      AND NOT (AdvancePaperPhase(curu, paper) AND Admin(curu)) then
       if CurrentPhase = Initialization then
            CurrentPhase = PreSubmission
            PhaseAdvanced(PreSubmission)
       else if CurrentPhase = PreSubmission then
            CurrentPhase = Submission
            PhaseAdvanced(Submission)
       else if CurrentPhase = Submission then
            CurrentPhase = Bidding
            PhaseAdvanced(Bidding)
       else if CurrentPhase = Bidding then
            CurrentPhase = Assignment
            PhaseAdvanced(Assignment)
            forall u, p | Papers(u, p):
                    PaperPhase(p, PAssignment)
       else if CurrentPhase = Assignment then
```

```
        CurrentPhase = Reviewing
        PhaseAdvanced(Reviewing)
   else if CurrentPhase = Discussion then
        CurrentPhase = Notification
        PhaseAdvanced(Notification)
   else if CurrentPhase = Notification then
        CurrentPhase = Publishing
        PhaseAdvanced(Publishing)
   else
        ActionFailed(curuser)
else
    ActionFailed(curuser)


//Submitting a paper
if ModifySubmission(curuser, new-paper) AND Author(curuser)
    AND CurrentPhase = Submission
  AND NOT (RemoveUser(curu, user) AND Admin(curu))
  then
    forall p: NOT Papers(curuser, p)
    Papers(user, new-paper)
    NewPaperSubmission(curuser)
 else
    ActionFailed(curuser)


//Creating conflicts
if AddConflict(curuser, user, paper) AND (Admin(curuser) OR ((curuser = user)
    AND Reviewer (user))
  AND NOT (BidPaper(user, paper) AND (currentPhase = Bidding) AND Reviewer(user)
    AND NOT Conflicts(user, paper))
  AND NOT (ReviewPaper (user, paper, review) AND AssignedReviews (user, paper))
  AND NOT (AssignPaper(curu, user, paper) AND currentPhase = Assignment
    AND Admin(curu) AND Reviewer(user) AND NOT Conflicts(user, paper))
then
    Conflicts(user, paper)
    NOT AssignedReviews(user, paper)
    forall r : NOT PaperReviews(user, paper, r)
    NOT PaperBids(user, paper)
    ConflictAdded(user, paper)
else
    ActionFailed(curuser)
```

```
//Paper Phases
if AdvancePaperPhase(curuser, paper) AND Admin(curuser)
  AND NOT (AdvancePhase(user) AND Admin(user)) then
    if PaperPhase(paper, phase) and (phase = PAssignment) then
        if AssignedReviews(reviewer, paper) then
            NOT PaperPhase(paper, phase)
            PaperPhase(paper, PReviewing)
            PaperStateAdvanced(paper, PReviewing)
    else if PaperPhase(paper, phase) and (phase = PReviewing) then
        if PaperReviews(reviewer, paper, review) then
            NOT PaperPhase(paper, phase)
            PaperPhase(paper, PDiscussion)
            PaperStateAdvanced(paper, PDiscussion)
    else if PaperPhase(paper, phase) and (phase = PDiscussion) then
        if DecidePaper(curuser, paper, decision) then
            NOT PaperPhase(paper, phase)
            PaperPhase(paper, PDecided)
            PaperStateAdvanced(paper, PDecided)
    else
        ActionFailed(curuser)
else
    ActionFailed(curuser)


//Change Jobs
if changeUserJobToReviewer(curuser, user) AND Admin(curuser)
  AND NOT (changeUserJobToNotReviewer(curu, user) AND Admin(curu))
  AND NOT (CreateAuthor (user, name)) then
   Reviewer(user)
   NOT Author(user)
   UserModified(user)

if changeUserJobToNotReviewer(curuser, user) AND Admin(curuser)
  AND NOT (changeUserJobToReviewer(curu, user) AND Admin(curu))
  AND NOT (CreateReviewer(u, newuser, name) AND Admin(u)) then
   NOT Reviewer(user)
   UserModified(user)

if changeUserJobToAdmin(curuser, user) AND Admin(curuser)
  AND NOT (changeUserJobToNotAdmin(curu, user) AND Admin(curu))
  AND NOT (RemoveUser(u, user) AND Admin(u))
  AND NOT CreateAuthor(user, name) then
   Admin(user)
   NOT Author(user)
   UserModified(user)

if changeUserJobToNotAdmin(curuser, user) AND Admin(curuser)
  AND NOT (changeUserJobToAdmin(curuser, user) AND Admin(curuser)) then
    NOT Admin(user)
    UserModified(user)

if ReadPaper(curuser, paper) AND NOT Conflicts(curuser, paper) then
        PaperIsRead(curuser,paper)

if ReadReview (curuser, review) AND
      exists paper, user : PaperReviews(user, paper, review)
      AND NOT Conflicts(curuser, paper) then
        ReviewIsRead(curuser,review)

if CreateAuthor(username, name)
  AND NOT (changeUserJobToReviewer(curuser, username) AND Admin(curuser))
  AND NOT (changeUserJobToAdmin(curu, username) AND Admin(curu))
  AND NOT (RemoveUser(u, username) AND Admin(u)) then
    User(username, name)
    Author(username)
    Password(username, DefaultPassword)
    UserModified(username)

if CreateReviewer(curuser, newuser, name) AND Admin(curuser)
  AND NOT (RemoveUser(u, username) AND Admin(u))
  AND NOT (changeUserJobToNotReviewer(curu, user) AND Admin(curu)) then
    User(newuser, name)
    Password(newuser, DefaultPassword)
    Reviewer(newuser)
    UserModified(username)

if RemoveUser(curuser, user) AND Admin(curuser)
  AND NOT (ModifySubmission(user, new-paper) AND Author(user))
  AND NOT (AddConflict(curu, user, paper) AND
           (Admin(curu) OR ((curu = user) AND Reviewer(user)))
  AND NOT (changeUserJobToReviewer(u, user) AND Admin(u))
  AND NOT (CreateAuthor(user, name))
  AND NOT (CreateReviewer(cu, user, name) AND Admin(cu))
  AND NOT (ModifyUserInfo (cur, user, name) AND (Admin(curuser)
            OR (curuser = user)))
  AND NOT (AddConflict(curu, user, paper) AND (Admin(curu)
            OR ((curu = user) AND Reviewer(user)))
  AND NOT (ReviewPaper(curusr, paper, review)
            AND AssignedReviews(curusr, paper) AND Papers(user, paper))
  AND NOT (BidPaper(curus, paper) AND (currentPhase = Bidding)
            AND Papers(user, paper))
  AND NOT (BidPaper(user, paper) AND (currentPhase = Bidding)
            AND Reviewer(user))
  AND NOT (AssignPaper(curuser, user, paper) AND currentPhase = Assignment
            AND Admin(curuser) AND Reviewer(user)
            AND NOT Conflicts(user, paper))
  AND NOT (DecidePaper (cur1, paper, decision) AND Admin (cur1)
            AND Papers(user, paper))
  AND NOT (ModifySubmission(user, new-paper) AND Author(user))
  AND NOT (ReviewPaper (user, paper, review)
            AND AssignedReviews(user, paper))
  AND NOT (BidPaper(user, paper) AND (currentPhase = Bidding)
            AND Reviewer(user) AND NOT Conflicts(user, paper))
  AND NOT (changeUserJobToAdmin(curuser, user) AND Admin(curuser))
  AND NOT (ModifyUserPassword (curu, user, password, new-password)
            AND (((curu = user) AND Password(user, password))
                    OR (Admin(curu))))
  then
    if (exists n : User(user, n)) and Admin(curuser) then
        forall n NOT User(user, n)
    if Conflicts(user, p) then
        NOT Conflicts(user, p)
    if Reviewer(user) then
        if PaperReviews(user, p, r) then
            NOT PaperReviews(user, p, r)
        if PaperBids(user, paper) then
            NOT PaperBids(user, paper)
        if AssignedReviews(user, paper) then
            NOT AssignedReviews(user, paper)
        NOT Reviewer(user)
    else if Author(user) then
            if Papers(user, paper) then
                if AcceptedPapers (paper) then
                    NOT AcceptedPapers (paper)
```

```
                NOT Papers(user, paper)
            if PaperReviews(reviewer, paper, review) then
                NOT PaperReviews(reviewer, paper, review)
            if PaperBids(reviewer, paper) then
                NOT PaperBids(reviewer, paper)
            NOT Author(user)
    if Admin(user) then
        NOT Admin(user)
    NOT User(user, name)
    forall password : NOT Password (user, password)
    UserDeleted(user)

if RemovePaper(curuser, paper) AND (Papers(curuser, paper) OR Admin(curuser))
  AND NOT (AssignPaper(cur, curuser, paper) AND currentPhase = Assignment
            AND Admin(cur) AND Reviewer(curuser)
            AND NOT Conflicts(curuser, paper))
  AND NOT (BidPaper(curuser, paper) AND (currentPhase = Bidding)
              AND Reviewer(curuser) AND NOT Conflicts(curuser, paper))
  AND NOT (DecidePaper(user, paper, decision) AND Admin (user))
  AND NOT (ReviewPaper(user, paper, review) AND AssignedReviews(user, paper))
  then
        if AssignedReviews(user, paper) then
            NOT AssignedReviews(user, paper)
        if PaperBids(user, paper) then
            NOT PaperBids(user, paper)
        if AcceptedPapers(paper) then
            NOT AcceptedPapers(paper)
        if DecidedPapers(paper) then
            forall dec : NOT DecidedPapers(paper, dec)
        if PaperReviews(user, paper, r) then
            NOT PaperReviews(user, paper, r)
        PaperDeleted(paper)

if BidPaper(curuser, paper) AND (currentPhase = Bidding)
    AND Reviewer(curuser) AND NOT Conflicts(curuser, paper)
  AND NOT (AddConflict(curu, curuser, paper) AND (Admin(curu)
            OR ((curu = curuser) AND Reviewer (curuser)))
  AND NOT (RemoveUser(user, curuser) AND Admin(user))
  then
        PaperBids(curuser, paper)
        BidModified(curuser,paper)
else
```

```
    ActionFailed(user)

if UnbidPaper(curuser, paper) AND currentPhase = Bidding
  AND NOT (RemoveUser(user, curuser) AND Admin(user))
  then
    if Reviewer(curuser) then
        NOT PaperBids(curuser, paper)
        BidModified(curuser, paper)
    else
        ActionFailed(curuser)

if AssignPaper(curuser, user, paper) AND currentPhase = Assignment
    AND Admin(curuser) AND Reviewer(user) AND NOT Conflicts(user, paper)
  AND NOT (AddConflict(curu, user, paper) AND (Admin(curu)
            OR ((curu = user) AND Reviewer (user)))
  then
    AssignedReviews(user, paper)
    AssignmentModified(user,paper)
else
    ActionFailed(curuser)

if UnassignPaper(curuser, user, paper) AND currentPhase = Assignment
    AND Admin(curuser)
  AND NOT (AssignPaper(cur2, user, paper) AND currentPhase = Assignment
            AND Admin(cur2) AND Reviewer(user)
            AND NOT Conflicts(user, paper))
  then
    NOT AssignedReviews(user, paper)
    AssignmentModified(user,paper)
    else
        ActionFailed(curuser)

if EditConferenceInfo (curuser, conference-info) AND Admin (curuser) then
    ConferenceInfo = conference-info
    InfoChanged()
    else
        ActionFailed(curuser)

if ModifyUserInfo (curuser, user, name) AND (Admin (curuser)
    OR (curuser = user))
  AND NOT (CreateAuthor(username, name))
  AND NOT (CreateReviewer(curuser, newuser, name) AND Admin(curuser))
  then
    (forall names : NOT User(user, names))
    User (user, name)
    UserModified(user)
    else
        ActionFailed(curuser)

if ModifyUserPassword (curuser, user, password, new-password)
    AND (((curuser = user) AND Password(user, password)) OR (Admin(curuser)))
  AND NOT (RemoveUser(curu, user) AND Admin(curu))
  then
    NOT Password (user, password)
    Password (user, new-password)
    UserModified(user)
  else
    ActionFailed(curuser)

if DecidePaper (curuser, paper, decision) AND Admin (curuser)
  AND NOT (DecidePaper(curu, paper, d2) AND Admin(curu) AND d2 != decision)
  AND NOT (RemovePaper(curu, paper)
    AND (Papers(curu, paper) OR Admin(curu)))
  then
    if exists d . DecidedPapers (paper, d) then
        NOT DecidedPapers (paper, d)
    DecidedPapers (paper, decision)
    if d = Accepted
        AcceptedPapers(paper)
    PaperDecision(paper, decision)
    else
        ActionFailed(curuser)

if ReviewPaper(curuser, paper, review) AND AssignedReviews(curuser, paper)
  AND NOT (AddConflict(curuser, user, paper)
        AND (Admin(curuser) OR ((curuser = user) AND Reviewer (user)))
  AND NOT (RemoveUser(curu, user) AND Admin(curu) AND Papers(user, paper))
  AND NOT (RemovePaper(user, paper) AND (Papers(user, paper) OR Admin(user)))
  then
    PaperReviews (curuser, paper, review)
    ReviewSubmitted(curuser, paper, review)
else
    ActionFailed(curuser)
```

Appendix: Example Continue Policy Properties for Verification

```
//conflictEnforcement - Nobody can read/review a conflicted paper
G(forall author, paper, reviewer | Paper(author,paper) AND Reviewer(reviewer)
    Conflicts(reviewer,paper) -> NOT PaperBid(reviewer,paper) AND NOT
    PaperAssign(reviewer,paper) AND (forall review NOT
    PaperReviews(reviewer,paper,review)) AND NOT ReadPaper(reviewer,paper))

//authorsDisabled - Authors can only submit in the Submission Phase
G(forall paper, user: NOT CurrentPhase = Submission ->
    (X Papers(user,paper) -> Papers(user,paper)))

//validConference
G(  (exists u : Admin(u))
    AND (forall u, n | Users(u,n),
    Reviewer(u) OR Admin(u) OR Author(u))
    AND (forall p, d | DecidedPapers(p,d),
    d in Accepted U Rejected)
    AND (forall u,p,r, PaperReviews(u,p,r) -> AssignedReviews(u,p)))

//checkPaperDeleted
G(  RemovePaper(curuser,paper) ->
    X( forall user, review, decision, paperphase:
NOT Papers(user, paper)
AND NOT DecidedPapers(paper, decision)
AND NOT AssignedReviews(user, paper)
AND NOT PaperReviews(user, paper, review)
AND NOT PaperBids(user, paper)
AND NOT PaperPhase(paper,paperphase)))

/* We don't have checkOtherPapersUnchanged because in our model,
   we actually can modify multiple papers simultaneously. */

//makeReviews
G(  forall curuser, paper, review:
    ReviewPaper(curuser, paper, review) AND AssignedReviews(curuser, paper) ->
X PaperReviews(curuser, paper, review))

//All papers in the Discussion phase have reviews
G( forall paper:
    CurrentPhase = Discussion ->
exists user, review | PaperReviews(user, paper, review))
```

```
//Admins can always read reviews
G(  forall u, reviewer, paper, review:
    Admin(u) AND PaperReviews(reviewer, paper, review) AND ReadReviews(u,review)
    ->
    X ReviewIsRead(u,review))

//noLongerRevise
G(  forall u, paper, newpaper:
    Papers(u,paper) AND ModifySubmission(u, newpaper) -> X Papers(u,paper))
```