Composition with Consistent Updates for Abstract State Machines

Colin Gordon, Leo Meyerovich, Joel Weinberger, and Shriram Krishnamurthi

Brown University, Providence RI 02912, USA
Contact: sk@cs.brown.edu

Abstract. Abstract State Machines (ASMs) offer a formalism for describing state transitions over relational structures. This makes them promising for modeling system features such as access control, especially in an environment where the policy's outcome depends on the evolving state of the system. The current notions of modularity for ASMs, however, provide insufficiently strong guarantees of consistency in the face of parallel update requests. We present a real-world context that illustrates this problem, discuss desirable properties for composition in this context, describe an operator that exhibits these properties, formalize its meaning, and outline its implementation strategy.

1 Motivation

This work arises from modeling working software systems. The last author and his collaborators have developed and deployed a series of Web-based applications for managing conference papers, faculty candidate applications, homeworks, etc., all built using the PLT Scheme Web programming framework [1, 2]. These applications see significant daily use, and some have even been deployed commercially. Though this framework considerably reduced both development time and errors relative to traditional Web programming methods, we found one significant source of errors remained: the proper management of the access-control policy. Our subsequent work [3, 4] has therefore focused on this problem. In this paper, we motivate our approach with reference to the CONTINUE conference manager [2, 5].

Conference managers are rife with access-control restrictions; furthermore, their demands are representative of many content-management systems. Several key operations, such as sending mail to the authors of papers, are restricted for use by the chairs. Some operations, such as creating reviewers, are in the hands of an administrator. Reviewers should not be able to see the reviews of papers with which they have a conflict. Sub-reviewers should have reviewer-like privileges, but only for the paper they are asked to examine. Authors should have very limited privileges: submitting papers, reading reviews, and no more.

It is subtle enough to state these properties and get them precisely right. What greatly complicates this situation is that the policies are "dynamic": they appear to change over time.[1] Here are two concrete examples:

1. The conference proceeds in phases: Setup, Submission, Review, Response, etc. Privileges change across phases. For instance, authors can only submit during Submission, read reviews during Response, and have no privileges at other times.
2. Even within a phase, there are fine-grained changes of privilege. For instance, many conferences adopt the rule that, to increase independence of views, a reviewer can read the other reviews of a paper only after they have submitted their own.

Because these changes refer to program data that change dynamically (such as who has submitted reviews for what papers), the access-control policy cannot be specified entirely independently of the program's dynamic execution. For this reason we investigated the use of abstract state machines (ASMs), as introduced in [6, 7] and analyzed in [8–10], to model the system's access-control behavior.

To make matters worse, conferences and workshops routinely require small adjustments to suit their needs: for instance, it is common for some conferences to have a live program committee meeting (which activates the live meeting mode, and its corresponding access-control rules) while others don't; some conferences have a response phase during which authors can view their reviews and offer comments, while others do not. As a result, there is no single "conference manager policy"; rather, we have a *product line* [11] of policy features that individual conferences mix-and-match. The set of products is not fixed, either, as conferences often ask for new features (over the life of CONTINUE we have experienced about half-a-dozen such requests, some of which do have access-control implications).

This last requirement demands that our specifications be modular and composable. When modeling CONTINUE with ASMs, we found that while ASMs were concise, our specifications were often incorrect: our first hundred or so lines contained over twenty errors! We were able to trace these problems to *inconsistent updates* resulting from combining fragments of policy. These problems were therefore the inspiration for this work.

This paper therefore presents a new composition mechanism for ASMs. We identify three key properties of composition, which we call *containment*, *confined inconsistency*, and *atomicity*. We propose a new composition operator that exhibits these three properties, and sketch proofs of correctness. We present an algorithm that realizes this operator, and show that we can implement it by a source-to-source transformation of ASMs, thereby leaving their semantics intact. We have validated this new operator by using it to author the policy for CONTINUE [12].

[1] We use quotes around 'dynamic' because, to be precise, the policies themselves are not dynamic; they are written down before system execution and stay fixed throughout. Rather, it is the *outcomes* that are dynamic [3].

2 ASM Composition: An Illustrative Example

Consider the following three rules, and then their composition. These rules are taken from our model for CONTINUE, though of course they are simplified for presentation. (The accompanying technical report [12] provides full details.)

First, rule R1, on becoming an administrator:

```
IF ChangeJobToAdmin(u) AND isReviewer(u) THEN
   isAdmin(u)
   NOT isReviewer(u)
```

This declares that a user with suitable privilege (usually a chair) who wants to change their currently active role to Administrator can do so while shedding their current role (to avoid inadvertent information leakage through role combination). For instance, a request

```
ChangeJobToAdmin(fred) and isReviewer(fred)
```

would yield the effect[2]

```
{+isAdmin(fred), -isReviewer(fred)}
```

Now consider rule R2, which governs when a reviewer may be assigned to review a particular paper:

```
IF AddPaperReviewer(u1, p) AND isReviewer(u1)
AND isAuthor(u2, p) AND NOT Conflict(u1, u2) THEN
   isPaperReviewer(u1, p)
```

This ensures that `u1` is indeed a reviewer, and that this reviewer has no conflicts-of-interest with work by the paper's authors. Thus, a request

```
AddPaperReviewer(bob, iliad) and ...
```

would, if the other conditions are satisfied, yield the outcome

```
{+isPaperReviewer(bob, iliad)}
```

Finally, rule R3 is a means for removing administrative privilege:

```
IF RemoveAdmin(u) THEN
   NOT isAdmin(u)
```

A request of the form

```
RemoveAdmin(alice)
```

yields

[2] We use the form {+rel1(args), -rel2(args)} to represent the addition and removal (respectively) of a tuple in a relation at a given state in a run of an ASM. That is, this represents the change to the state caused by a transition.

```
{-isAdmin(alice)}
```

Having seen these three rules and examples of requests, let us now consider what happens if the requests are made concurrently. Typical ASM semantics would leave us with the result set

```
{+isAdmin(fred), -isReviewer(fred),
 +isPaperReviewer(bob, iliad),
 -isAdmin(alice)}
```

which appears perfectly reasonable. If, however, the last request is to remove administrative privilege for `fred` instead of for `alice`, the result set is instead

```
{+isAdmin(fred), -isReviewer(fred),
 +isPaperReviewer(bob, iliad),
 -isAdmin(fred)}
```

which of course is inconsistent because of conflicting updates to the `isAdmin` relation. What should happen now?

The traditional definition dictates total failure. The justification for this outcome is that a pair of updates that conflict are the result of an error in the specification; consequently, the system should halt. While this is a very safe outcome, it is sometimes too harsh: in the example above, it is clear that there are several outcomes less drastic than termination that we can consider reasonable.

A simple way to make progress is to simply disregard these two conflicting updates, treating their net effect as a no-op. That is, the resulting effect would therefore be:

```
{-isReviewer(fred),
 +isPaperReviewer(bob, iliad)}
```

In principle, the system can recognize such a partial update and retry the request (as we discuss in more detail [12]). However, we reject such an outcome as unreasonable (in the very literal sense). Though the inconsistent updates are gone, we still have `-isReviewer(fred)` in the resulting set of updates. This removes `fred` from the reviewer role, while failing to reassign him to any other role. Therefore, any local reasoning performed by the author of R1 has now been destroyed, leaving the system in an inconsistent state. That is, while we have eliminated the inconsistent update, we have ended up with a semantically inconsistent model.

Irrespective of which composition semantics we choose, to ensure complete updates and progress an author is forced to encode the dependency between the rules R1 and R3 in the model. For instance, we can rewrite R3 to R3′:

```
IF RemoveAdmin(u) AND NOT ChangeJobToAdmin(u) THEN
    NOT isAdmin(u)
```

Now, whenever R1 and R3 would have been in conflict, R3′ will fail to fire, resolving our immediate problem.

This seems like a reasonable solution, although it is not difficult to see that such encoded dependencies could easily become quite complex because system modelers will need to consider every possible combination of dependencies.

A more subtle problem lurks, and it's even worse. Tracking dependencies only makes sense in a closed model, where the designer can account for all dependencies. In a product-line, such as a conference manager, each collection of rules (corresponding to a feature) must minimize assumptions about the structure of the rest of the model. In particular, subsequent changes to the policy may remove the dependence, as a result of which the augmented guard above may prevent R3′ from firing—even though that was not the designer's intent. Put differently, the augmentation to the guard of R3 expresses not a natural dependency between R1 and R3 that arises from the domain, but an artificial one due to the ASM update semantics. We should avoid forcing designers to introduce and manage such dependencies.

One natural solution is, of course, to eschew concurrency entirely. We can force the designer to make the system behave entirely sequentially and ensure this by checking, for instance, that no guards overlap. However, we find this an onerous burden and one inconsistent with the modeling principles engendered by ASMs. We therefore seek ways to compose collections of rules in a way that enables a reasonable measure of concurrency while guaranteeing agreeable semantics for the composed behavior.

3 Desired Properties for Composition

We begin by supposing the rules have been grouped into modules, with each module presumably representing one feature of the product-line. Our implicit assumption is that each module has been validated locally, and we focus on the effect of module interactions.

Recall the outcome from the original three rules:

R1: `{+isAdmin(fred), -isReviewer(fred)}`
R2: `{+isPaperReviewer(bob, iliad)}`
R3: `{-isAdmin(fred)}`

Let us treat each rule as its own module. We argue that one acceptable result from composing these three rules is the following outcome:

```
{+isPaperReviewer(bob, iliad)}
```

This output is the consequence of the three properties described below, which we name and justify.

Containment: First, if the final update set contains an update, that update came from an actual update set (in this case, that of R2). We would like all final update sets to satisfy this property: that every update in the final composition must be an effect from one of the individual update sets. That is, we reject algorithms that try to combine update sets and in the process create artificial

updates that did not result from any one particular rule. (It is possible to loosen this criterion if, for instance, the updates satisfy some ordering relationship that enjoys limit and closure properties—such as a lattice—but the examples we have worked with do not naturally subscribe to such an order, so we do not consider this possibility further in this paper.)

Confined Inconsistency: Since our goal is to eliminate conflicts introduced by composition, we eliminate all conflicting updates—with one caveat. Suppose one module (not shown in our running example) itself results in a conflict. In such a case, we permit the conflict to remain in the composed output on the grounds that, because the module was presumably validated by the designer, this conflict was somehow intended. That is, we want to prevent conflicts from the *composition* of modules, but not take a position on conflicts that originate entirely within a single module. Thus, each inconsistency in the composed update set must be traceable to individual modules.

Atomicity: Finally, eliminating inter-module conflicting updates can result in a module having only partial effect. As we have argued above, this can violate invariants intended to be established by that module. As a result, we demand that if part of a module's effect is pruned due to conflict, that entire effect should be nullified. We argue that from a system design perspective, it is much easier to establish whether a module has fired at all (and, if it has not, to re-try it) than to identify and—more importantly—correct for *partial* commitment. Furthermore, this is entirely consistent with traditional transactional practice in databases. As a corollary, if one update from a rule's update set occurs, we can be sure that all the updates occurred.

To be more precise about these properties, let us define some basic terminology. First, for purposes of defining these formulas, let us consider each module as a set of update rules, where each rule takes the form:

$$\texttt{IF } mg \wedge ug \texttt{ THEN } s\ r(\boldsymbol{p})$$

which can be condensed into a tuple

$$(mg, ug, s, r, \boldsymbol{p})$$

where mg is the condition under which the entire module fires (if there is such a general condition), ug is the set of any additional conditions under which that particular update would fire, s is the sign of the update (none or + for additive updates, − for removal), r is the relation, and $\boldsymbol{p}$ is the set of parameters to which this update applies. We will set aside the issue of universally bound quantifiers to simplify notation, but these can be abstracted out of every update as well, and in our technical report [12] we do so. However, for the purposes of matching parameter lists, universally bound quantifiers can be considered to match any term to which they are compared.

To simplify later presentation, we first define a function that captures the decision of a module.

Definition 1. *Let the* decision *of a module m_j for a particular relation r and parameter list $\boldsymbol{q}$ be in the set $\{+, -, NOOP, NA\}$, whose elements respectively denote addition to that relation, removal, a module-local conflicting update, and lack of an update to r with the parameters $\boldsymbol{q}$. A function for a decision can formally be defined as $decision(r, \boldsymbol{q}, m_j) =$*

$$\begin{cases} NA & \forall (mg_i, ug_i, s_i, r_i, \boldsymbol{q}_i) \in m_j.(r \neq r_i \vee \boldsymbol{q}_i \neq \boldsymbol{q}) \\ + & \exists (mg_i, ug_i, s_i, r_i, \boldsymbol{q}_i).(r = r_i \ \wedge \ mg_i \ \wedge \ ug_i \ \wedge \ \boldsymbol{q} = \boldsymbol{q}_i \ \wedge \ s_i = +) \\ & \quad \wedge \ \forall (mg_f, ug_f, s_f, r_f, \boldsymbol{q}_f).(r = r_f \ \wedge \ mg_f \ \wedge \ ug_f \ \wedge \ \boldsymbol{q} = \boldsymbol{q}_f) \ s_f = + \\ - & \exists (mg_i, ug_i, s_i, r_i, \boldsymbol{q}_i) \in m_j.(r = r_i \ \wedge \ mg_i \ \wedge \ ug_i \ \wedge \ \boldsymbol{q} = \boldsymbol{q}_i \ \wedge \ s_i = -) \\ & \quad \wedge \ \forall (mg_f, ug_f, s_f, r_f, \boldsymbol{q}_f).(r = r_f \ \wedge \ mg_f \ \wedge \ ug_f \ \wedge \ \boldsymbol{q} = \boldsymbol{q}_f) \ s_f = - \\ NOOP & \exists (mg_i, ug_i, s_i, r_i, \boldsymbol{q}_i), (mg_f, ug_f, s_f, r_f, \boldsymbol{q}_f) \in m_j. \\ & \quad r = r_i = r_f \ \wedge \ mg_i \ \wedge \ ug_i \ \wedge \\ & \quad mg_j \ \wedge \ ug_f \ \wedge \ \boldsymbol{q} = \boldsymbol{q}_i = \boldsymbol{q}_f \ \wedge \ s_i \neq s_f \end{cases}$$

Definition 1 is what one would expect. If no update in the module addresses the relation in question, or no update to that relation ever matches the parameters specified, the decision is not applicable. If there is a matching update that adds $\boldsymbol{p}$ to the relation, and no update in that module which removes it, then the decision is positive. The case is analogous for negative decisions. The decision is a no-op (inconsistent update) if both positive and negative updates occur in the same module.

With this, we can formally define the properties that the composition must satisfy.

Definition 2. *A composition m_j of a set of modules $\{m_i\}$ satisfies the* containment *property if*

$$\forall r \forall \boldsymbol{q} \forall d \in \{+, -\} \quad decision(r, \boldsymbol{q}, m_j) = d \Longrightarrow \exists m_l \in \{m_i\}.decision(r, \boldsymbol{q}, m_l) = d$$

According to Def. 2, if the composition reaches a decision on a particular relation and parameter list, then there was a module in the input that reached that decision.

Definition 3. *A composition m_j of a set of modules $\{m_i\}$ satisfies the* confined inconsistency *property if*

$$\forall r \forall \boldsymbol{q} \quad decision(r, \boldsymbol{q}, m_j) = NOOP \Longrightarrow \exists m_l \in \{m_i\}.decision(r, \boldsymbol{q}, m_l) = NOOP$$

Definition 3 says that if the composition causes a conflicting update to occur, it is because there is a module in the input that causes a conflicting update when run in isolation.

Definition 4. *A composition m_j of a set of modules $\{m_i\}$ satisfies the* atomicity *property if*

$$\begin{array}{l}\forall d \in \{+, -, NOOP\} \forall r \forall \boldsymbol{q} \\ \quad decision(r, \boldsymbol{q}, m_j) = d \Longrightarrow \\ \quad\quad \exists m_k \in \{m_i\}. \forall r' \forall \boldsymbol{q'} \\ \quad\quad\quad decision(r', \boldsymbol{q'}, m_k) = decision(r', \boldsymbol{q'}, m_j) \vee decision(r', \boldsymbol{q'}, m_k) = NA\end{array}$$

Definition 4 says that for any decision other than NA, for any relation and parameters, if the composition makes that decision, then there is some module for which on all relations and parameters, the module and composition make the same decision or the module reached no decision on that relation and those parameters.

It is also worth noting that these properties provide no progress guarantees: a trivial composition algorithm which satisfies these properties is one that performs no updates. The algorithm described below provides stronger guarantees of progress. We prove in our technical report [12] that it actually satisfies the bidirectional implication versions of all three of these properties, which would imply more progress than these strictly require.

4 The Composition Algorithm

Suppose the user has chosen a set of modules that will be composed to create the current product. Our algorithm operates over this set of modules as a whole.

To keep discussion concise, we will define a notion of what it means for two modules to reach conflicting decisions on a relation. Two decisions on the same relation and parameters *conflict* if they are different and neither decision is NA. More precisely:

Definition 5.

$$conflict(d, r_d, \boldsymbol{p}_d, d', r_{d'}, \boldsymbol{p}_{d'}) = \begin{cases} false & \begin{array}{c} d = d' \vee d = NA \vee d' = NA \\ \vee\ r_d \neq r_{d'} \vee \boldsymbol{p}_d \neq \boldsymbol{p}_{d'} \end{array} \\ true & otherwise \end{cases}$$

The algorithm proceeds one module at a time. For each update in the current module, find all other modules with updates to the same relation but which might reach a conflicting decision on this relation. (We cannot be certain there will be a conflict until we have examined the parameters, which may not be known at the time of composition: for instance, one rule may add `alice` while the other deletes `fred`.) For each such rule, the algorithm augments the pre-conditions of

the current rule with the following conjuncts: the negation of the pre-conditions of the potentially conflicting rule, and argument checks to establish whether the identities coincide. Proceed in this fashion until all rules in all modules have been considered. Clearly, this algorithm can be implemented as a source-to-source transformation.

To demonstrate this algorithm, consider our running example (rules R1, R2 and R3), where each rule is treated as its own module. The first update in the R1, `isAdmin(u)`, conflicts with the update in the third module, which executes when `RemoveAdmin(u)` is true. Therefore, the pre-condition for R1 is augmented by conjoining checks that (a) there is no request to `RemoveAdmin(u2)` for some user `u2`, and (b) that `u2` is not the same as `u`. Thus, the rewritten rule is

```
IF ChangeJobToAdmin(u) AND isReviewer(u)
AND NOT (RemoveAdmin(u2) AND u = u2) THEN
    isAdmin(u)
    NOT isReviewer(u)
```

The second update in R1 has no conflicts, and neither does the sole update in R2. This leaves the sole update of R3. We must symmetrically modify R3's pre-conditions to ensure that the user surrendering administrative privilege is not simultaneously being given that privilege by R1:

```
IF RemoveAdmin(u)
AND NOT (ChangeJobToAdmin(u1) AND isReviewer(u1) AND u = u1) THEN
    NOT isAdmin(u)
```

Note that this is effectively identical to R3′, which we considered earlier. Recall that the objections to R3′ were not on the grounds of correctness, but rather on maintenance. By automating the process of modifying such rules, we avoid those difficulties. (Furthermore, we symmetrically modify both R1 and R3, whereas a manual adaptation is almost certainly liable to miss some such cases.)

Note that when examining other rules for conflicts, we need to consider rules in their original form only, not with the augmented pre-conditions. Therefore, there is no danger of the pre-conditions growing without limit and the process failing to terminate. Indeed, because we consider each pair of rules, the algorithm takes time quadratic in the number of rules to compose a set of modules into a single ASM.

We provide the pseudocode for the composition operator in Fig. 1. (The technical report [12] proves that this algorithm satisfies our desired properties.) We assume the modules provided are in the normal form discussed earlier in Sec. 3. The outermost two loops iterate over all updates. The nested loops in the body of these two look through every possible update in the system. The loop that iterates over `m2` builds additional conditions for execution that negate the conditions under which conflicting updates would occur. The last loop mutates the module guard for each update in the current module, so that if an update conflicting with the update under consideration would fire, none of the updates in the current module will fire.

```
Compose({Mi}):
    copy = DeepCopy(Mi)
    foreach m in copy
        foreach (mg,ug,s,r,p) in m
            newguard = false
            foreach m2 in Mi
                 if conflict(decision(r,p,m),decision(r,p,m2)) then
                     foreach u2 = (mg',ug',s',r',p') in m2
                         newguard = newguard OR (mg' AND ug')
            foreach (mg'',ug'',s'',r'',p'') in m
                mg'' = mg'' AND NOT (newguard)
    return copy
```

Fig. 1. The Composition Algorithm

A more refined implementation would use only one copy of the module guard for each module, conjoining the guards in a single statement rather than in a loop. In contrast, this implementation requires repeated mutation because the choice of normal form duplicates the module guard. Note that the mutation affects copies, not the original input, so the appended guards do not cascade.

5 Related Work on ASM Composition

Nicolosi Asmundo and Riccobene propose two composition techniques [13]. The first, *feature composition*, joins ASM programs by straightforward concatenation, then checks for potential inconsistent updates; if any such updates are found, the entire composition is rejected. This works well for systems with relatively little overlapping state between rules, but is not useful for systems like CONTINUE that exhibit significant overlap in the state manipulated by different rules.

Nicolosi Asmundo and Riccobene also propose a second approach, called *component composition* [13]. This approach effectively converts each unit of composition into an isolated system with read-only and write-only communication channels for communicating with other parts of the larger system. Relations marked for communication are checked for appropriate use (i.e., no updates to read-only input relations, no reads from an output relation). Other relations are renamed on a per-component basis, effectively placing the non-communication relations of each component in separate namespaces. This works well for modelling systems which consist of subsystems with largely independent state. For such systems, it guarantees that composing these components will not introduce additional inconsistent updates. As with feature composition, however, component composition is not an appropriate choice for modelling CONTINUE, because the rules dealing with access control share too much state. To use component composition for CONTINUE would require either (a) writing additional components to merge output from various policy components, or (b) modifying each

component to deal with the global state changes itself. The former moves the problems with inconsistent updates to a new form, and the latter recreates the initial problem many times over, in each component. Using component composition for a system with large amounts of global, frequently-modified state presents many of the same problems as attempting to hardcode dependencies.

Gurevich and Rosenzweig point out that reasoning about partially-ordered runs of distributed ASMs can be used to reach useful conclusions about such systems [14]. Some implementations of multi-agent ASMs apply concurrent updates from separate agents in a nondeterministic, but serialized, order, which alleviates the problem of inconsistent updates from separate agents. This will, however, likely yield less parallelism than our composition operator, and may also not simplify the process of specifying the system.

6 Future Work

Our technique disables updates by selectively strengthening guards with conditionals. Naturally, this suggests that we can achieve different properties by using different forms of guard augmentation. Our prefixing algorithm is very conservative, by upholding the three properties described in this paper—especially atomicity—at the cost of progress. Currently, in case of conflict we disable *all* participating modules. We could achieve greater progress by being less pessimistic, enabling enough modules so that there is still no conflicting update but now at least some otherwise conflicted modules can complete their transition. (As a result, if the system is entirely deterministic and retries all failed modules, it is less likely to livelock.) Determining which module to permit is a matter of policy; a common policy used to resolve feature interactions is a simple priority scheme. Encoding such schemes using relations and inspecting them in the guards is usually straightforward.

Even prioritized composition is slightly conservative, as the only guarantee which must be made by the composition to prevent new inconsistent updates is that in any pair of modules with conflicting updates, only one of them executes. So in a set of modules with conflicts on a given relation, all modules yielding a positive (or negative) update to that relation might be permitted (subject to checking conflicts on other relations). Thus, with other modifications to the prefixing conditions, it is possible that even more progress might be permitted.

Another possible extension is to report no-ops. Doing this can help guide retries. Recording this information requires adding new output relations for notification of conflicts, and adding new updates to write to these relations when a conflicting update is detected.

Acknowledgments

We thank Dan Dougherty, Kathi Fisler, and students of the Spring 2006 cs296-1 course at Brown University for their comments. We are grateful to Egon Börger, Marianna Nicolosi Asmundo, Wolfram Schulte, Wolfgang Shoenfeld, and Yuri

Gurevich for discussions and for pointing us to resources, and to Don Batory for his extensive comments on an earlier draft. This work is partially supported by NSF grants CPA-0429492 and CNS-0627310.

References

1. Graunke, P.T., Krishnamurthi, S., van der Hoeven, S., Felleisen, M.: Programming the Web with high-level programming languages. In: European Symposium on Programming. (April 2001) 122–136
2. Krishnamurthi, S., Hopkins, P.W., McCarthy, J., Graunke, P.T., Pettyjohn, G., Felleisen, M.: Implementation and use of the PLT Scheme Web server. Higher-Order and Symbolic Computation (2007) To appear.
3. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Specifying and reasoning about dynamic access-control policies. In: International Joint Conference on Automated Reasoning. (August 2006) 632–646
4. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: International Conference on Software Engineering. (May 2005) 196–205
5. Krishnamurthi, S.: The CONTINUE server. In: Symposium on the Practical Aspects of Declarative Languages. Number 2562 in Springer Lecture Notes in Computer Science (January 2003) 2–16
6. Gottlob, G., Kappel, G., Schrefl, M.: Semantics of object-oriented data models—the evolving algebra approach. In: Next Generation Information System Technology, First International East/West Database Workshop. (October 1990) 144–160
7. Gurevich, Y.: Evolving algebras: An attempt to discover semantics. In: Current Trends in Theoretical Computer Science. (1993) 266–292
8. Spielmann, M.: Automatic verification of abstract state machines. In: International Conference on Computer-Aided Verification. (1999) 431–442
9. Spielmann, M.: Model checking Abstract State Machines and beyond. In: International Workshop on Abstract State Machines. (2000) 323–340
10. Spielmann, M.: Verification of relational transducers for electronic commerce. In: Symposium on Principles of Database Systems. (2000) 92–103
11. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
12. Meyerovich, L.A., Weinberger, J.H.W., Gordon, C.S., Krishnamurthi, S.: ASM relational transducer security policies. Technical Report CS-06-12, Computer Science Department, Brown University, Providence, RI, USA (2006)
13. Nicolosi Asmundo, M., Riccobene, E.: Consistent integration for sequential Abstract State Machines. In: International Workshop on Abstract State Machines. (2003) 324–340
14. Gurevich, Y., Rosenzweig, D.: Partially ordered runs: A case study. In: International Workshop on Abstract State Machines. (2000) 131–150