Simulating Access Control Policies and their Environments in Maude

Justin Pombrio

January 2, 2010

1 Introduction

1.1 Environments

I will call the system a policy regulates an *environment*. *Facts* are predicates that take zero or more *entities* as arguments. Typically, entities are divided into *subjects* and *resources*. I will speak loosely and use the term *fact* to refer to both a fact itself and to a fact applied to some arguments.

For example, borrowed("Jane", "Macbeth") could be a fact representing the idea that Jane borrowed the book Macbeth. Both "Jane" and "Macbeth" would be entities. An environment E may *satisfy* a fact F, written $E \models F$.

There can be *transitions* between one environment and another. They represent the system moving from one state to another. Associated with each transition is an *action*, with zero or more entities as arguments. For instance, checkout("Jane", "Macbeth"), could be an action.

1.2 Policies

Policies are written as datalog rules. For now, recursion is allowed while negation is not. The head of a rule is usually permit(a) or deny(a), where a is some action with variables in place of its arguments. Rule heads of this form are called *decisions*. The body of a rule is a set of *Propositions*, which are either facts or decisions.

When manipulating policies, for instance when minimizing a policy, it is often useful to treat the body of a rule like an environment. Then it can be checked whether a body "satisfies" a proposition.

1.3 Type Hierarchy

In summary, here are the types of everything presented so far (in the notation of Maude),

op permit : Action $\rightarrow$ Decision .
op deny : Action $\rightarrow$ Decision .

subsorts Environment RuleBody < State .
subsorts Fact Decision < Prop .

op $\_\models\_$: State Set{Prop} $\rightarrow$ Bool .

1.4 Policy Semantics

A *policy* takes an action and produces one of three answers: "permit", "deny", or "na" (for "not applicable"). I will denote policies by P and actions by a.

The empty policy is written P_ϕ. For all actions a,

$$P_\phi(a(x_1, x_2, ..., x_n)) = \text{na}$$

The intersection, union, difference, and composition of policies is defined as follows. Notice that taking the union of two policies only makes sense when their intersection is empty.

$$\begin{array}{rcl} (P_1 \cap P_2)(a) & = & \text{permit, if } P_1(a) = \text{permit} \wedge P_2(a) = \text{permit} \\ & | & \text{deny, if } P_1(a) = \text{deny} \wedge P_2(a) = \text{deny} \\ & | & \text{na, otherwise} \end{array}$$

$$\begin{array}{rcl} (P_1 \cup P_2)(a) & = & \text{permit, if } P_1(a) = \text{permit} \vee P_2(a) = \text{permit} \\ & | & \text{deny, if } P_1(a) = \text{deny} \vee P_2(a) = \text{deny} \\ & | & \text{na, otherwise} \end{array}$$

$$\begin{array}{rcl} (P_1 - P_2)(a) & = & \text{permit, if } P_1(a) = \text{permit} \wedge P_2(a) \neq \text{permit} \\ & | & \text{deny, if } P_1(a) = \text{deny} \wedge P_2(a) \neq \text{deny} \\ & | & \text{na, otherwise} \end{array}$$

$$\begin{array}{rcl} (P_1 P_2)(a) & = & \text{permit, if } P_1(a) = \text{permit} \vee (P_1(a) = \text{na} \wedge P_2(a) = \text{permit}) \\ & | & \text{deny, if } P_1(a) = \text{deny} \vee (P_2(a) = \text{deny} \wedge P_1(a) = \text{na} \\ & | & \text{na, otherwise} \end{array}$$

2 Policy Differencing

I have found the following view of the difference of two policies to be helpful.

Suppose we are given two policies and an environment and wish to know the difference between them. First, we care about the difference between the possible *transitions*, not the *states*. Second, the notable transitions are the ones *reachable under one policy but not under the other*. This partitions the transitions allowed by policy A but not by policy B, into three categories:

- The initial state of the transition is reachable under both A and B.
- The initial state is reachable under A but not B.
- The initial state is reachable under B but not A.

The first class is the most significant because any transition in the second or third class must be preceded by one in the first.

3 Algorithms

To check (uniform) policy containment and to minimize policies, I use algorithms outlined by Yehoshua Sagiv in *Optimizing Datalog Programs*.

Uniform policy containment checking is based on an intuitive fact: a rule is contained in a policy iff the body of the rule satisfies the head of the rule in the environment. In other words, evaluate the body of the rule in the environment. This will produce a set of propositions (typically *decisions*) as consequences. If the head of the rule is among them, the rule is contained in the policy; otherwise it is not. The extension from rule containment to policy containment is straightforward: one policy is contained in another iff each of its rules is.

A policy may be minimized by first eliminating redundant propositions in each rule and then eliminating redundant rules. First, for each proposition in each rule, eliminate the proposition if the rule would still be contained in the policy without it. Then eliminate each rule that is contained in the policy. Sagiv showed that running this algorithm a second time would have no further effect, so that the result is a *minimal* policy (not necessarily minimum). I suspect that in the more common non-recusive case the algorithm gives a minimum policy.

I believe that finding the difference between two policies, $P - Q$ may be as simple as minimizing P over their intersection $P \cap Q$. This fact should be checked, though.

The algorithms given check for *uniform* policy containment, which is weaker, and easier to test for, than policy containment. (In fact, containment of recusive programs is undecidable.) Testing for uniform containment is actually appropriate, though, because it guarentees that if policies A and B are (uniformly) equivalent, then so are $A + C$ and $B + C$.

4 Implementation

The user is expected to define four kinds of modules. A *setting module* defines facts and actions to be used in the environment and the policy. An *environment module* defines when an environment satisfies facts, and specifies transitions (expressed as rewriting rules) between environments. A *policy module* defines the policy (expressed as equations). A *system module* ties together both an environment module and a policy module. Since policies sometimes answer "na", the system module must also resolve this ambiguity.

Policy rules must have one of the following forms (where 'permit' may also be 'deny'):

```
var S : State .
var V1 : Entity.
var V2 : Entity.
var U : Entities.

ceq E |= permit(Event) = true
  if E |= FactList .

ceq E |= permit(Event) = true
  if (V1, V2, ..., U) := entities(E)
    /\ E |= FactList .
```

The variables $V1$, $V2$, etc. are existentially quantified. U is used only to capture the rest of the entities in the environment. Here is an example,

```
var S : State .
var B : Resource .
var U : Entities .

*** A book may be added if it is not already in the library's collection.
ceq S |= permit(add(B)) = true
  if (B, U) := entities(S)
    /\ S |= not checked-in(B), not is-borrowed(B) .
  if S |= not checked-in(B), not is-borrowed(B) .
```

The details of the other modules should be ascertainable from an example. See "library-cia.maude".

5 Further Work

There is one major outstanding problem. When writing policy rules in Maude, the subexpression

```
 if (V1, V2, U) := entities(E)
```

means (in Maude) that $V1$ and $V2$ must be distinct. But this is different from the semantics of datalog rules!

I have found Maude to be well-suited for writing access-control systems and their environment. Equations naturally capture policy rules, and term rewriting system rules naturally capture environment transitions. Maude's speed and built-in search capabilities are also helpful. Maude is, however, ill-suited for symbolic policy manipulation. The proccess of syntactically deconstructing policy rules and reconstructing modules using Maude's meta-level tool is clumsy and slow. See the code in "change-impact-analysis.maude".

There may be a way to manipulate policies without using the meta-level that I have missed. If none is found, I recommend writing a program in another language to translate policies and environment transitions into Maude. This would have the advantage of syntactically limiting policies to what is semantically sensible. As it is, there is a lot of leeway for users to write syntactially legal Maude code that doesn't make semantic sense for the access control system (such as writing a policy rule that doesn't match one of the two given forms). Symbolic policy manipulation could also be done in the new program. This way the meta-level would never have to be used, and the module resulting from differencing two policies would be a Maude module, not a Maude meta-module.