Formalization and Verification of Rewriting-Based Security Polices

By

Roman Veselinov

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in Computer Science

May 2008

APPROVED:

Professor Daniel Dougherty, Thesis Advisor

Professor Stanley Selkow, Reader

Professor Michael Gennert, Head of Department

Abstract

Term rewriting – an expressive language based on equational logic – can be used to author and analyze policies that are part of an access control system. Maude [25] is a simple, yet powerful, reflective programming language based on term rewriting that models systems along with the subjects, objects and actions within them. We specify the behavior of the system as a theory defined by conditional rewrite rules, and define the access control policy as an equational theory in a separate module. The tools that Maude provides, such as the Maude Model Checker and the Sufficient Completeness Checker, are used to reason about the behavior and verify properties of access control systems in an automated manner.

Acknowledgements

I would like to thank Professor Dan Dougherty, for his support, patience, and commitment throughout the completion of this thesis. I want to thank Professor Stanley Selkow for his encouragement and technical guidance. I want to express my gratitude to the members of the ALAS group, especially to Danny Yoo and Chris King, for their perceptive contributions.

Contents

List of Figures

1 Introduction

In modern systems authorization mechanisms have moved beyond the classical access control matrix [24] and access control decisions are represented by sets of rules comprising an access control policy. In this setting we want to be able to reason about the policy. Some of the relevant questions that will be addressed are:

- is the policy consistent, i. e., is it impossible to compute different decisions for the same access control request?
- is the policy complete, i. e., is there a decision specified for all requests?
- does the policy allow certain actions, i. e., is there an accessible state satisfying some Boolean expression over policy facts?

In order to achieve this goal we will use formal approaches to create a model of the system: the existence of a model gives us the ability to apply formal proofs and techniques to verify these security properties. We will explore the use of term rewriting [2] as a language for access control and use the environment of the term rewriting system Maude [25] to represent the system comprising the program and the policy, and to verify policy properties using Maude's model checking tool.

1.1 Dynamic Access Model

One of the most important problems in access control is the the problem of formalization of expressive access control models. The availability of such models facilitates the authoring of security policies and their verification according to a set of properties. Thus, in order to verify the correctness of a policy, it is useful to examine it as a separate module which, coupled with the environment, defines a dynamic access model [16]. The separation of the policy from the program is desirable since it promotes reusability, facilitates policy maintenance and provides a module that is verifiable with automated systems.

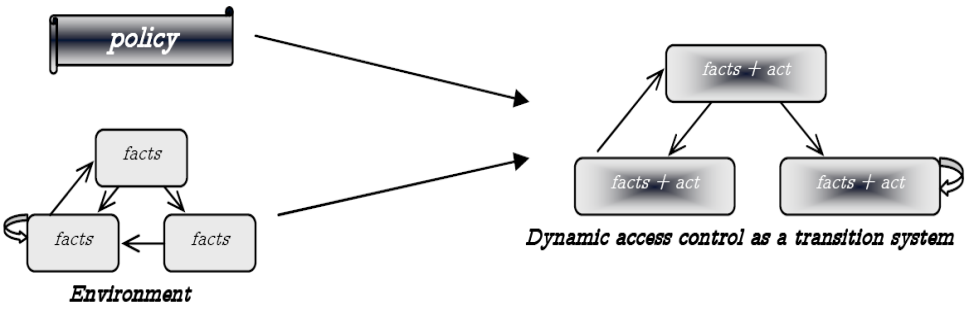

Figure 1: Dynamic Access Model

An important aspect of this interaction is the fact that the generated decisions are not based solely on the information contained in the incoming request or the policy but also on external factors in the environment. The set of facts relevant to the policy obtained from the program, the end-users, or the policy framework can be modeled as a transition system [8] (see figure 1). Transitions in the policy's environment are prompted by either user/application actions or other events. In general, the environment models will be interpretations partially derived from the application.

As an example, consider the policy from Figure 2 governing student access to assignments and grades: a student is required to submit an assignment and a grader has to assign a grade before the user is allowed to view the grade.

1. A student may read only his own grade for an assignment
2. A student cannot grade any assignments
3. A TA may read & grade any assignments
4. A student can submit an assignment if the assignment is open

Figure 2: Candidate policy V for controlling access to student records

The application maintains the information about submitted assignments by students, such as whether the assignment is still open, etc; the policy reference

monitor consults the information when a request comes along before issuing an access decision. This information forms the *environment* of the policy, which can be represented also as a list of atomic sentences. A request from a student to submit an assignment does not include information about whether the assignment is still open: that fact might have changed due to an administrator being granted a request to close the assignment. The environment data can be affected by external factors, while the policy remains static.

1.2 Term Rewriting

Term rewriting [2] offers a solution to representing the transition blueprint of the program along with the access control policy. It is a well-established paradigm whose applications include theoretical foundations for functional programming languages and theorem provers. A fairly simple computational standard built on the repeated application of rules, term rewriting is organized around the concepts of signature, terms, substitution and matching. It is a process that starts with a set of rewrite rules and an initial term (built from variables, constants and function symbols), which is reduced to a term that might be in its normal form, i. e. further irreducible term. Matching – linear on the size of the term – occurs when two given terms, are equal after applying a subset of the rules. The distinction between equational logic and term rewriting is the fact that equations are used as directed replacement rules: the arrow in the example group theory axioms below suggest one-directional term replacement:

$$(1)\ x \circ e \to x$$
$$(2)\ i(x) \circ e \to e$$
$$(3)\ (x \circ y) \circ z \to x \circ (y \circ z)$$

Figure 3: Group Theory Axioms as Rewrite Rules

Rewriting with conditional identities, also know as conditional rewriting, is an expanded form of the above concept, where a term on the left side will be rewritten

only if a specific condition applies. For example, a stack pop rule can be expressed as the conditional term rewriting rule

$$push(pop(\mathrm{s})) \rightarrow \mathrm{s}\ if\ \mathrm{s} \neq nil$$

where *nil* is a predefined constant corresponding to the empty stack.

1.3 Term Rewriting for Access Control

The expressive nature of term rewriting is an important reason for applying rewrite techniques to access control, policy specification and access request checking. A rule from our policy from Figure 2 could be written, based on [13], as:

$$auth(student(x), checkGrade, assignment(x, g, a)) \rightarrow permit$$

stating that a student (identified by his/hers unique ID) may check grades for any assignment that is authored by him or her. Here, the function *student* maps student ID to students as an element of the Subjects set and *assignment* maps student ID, grade and assignment ID to assignments as an element of the Objects set, while *checkGrade* is a constant symbol of sort Actions. The variable x is implicitly universally quantified, so that the rewriting above captures the generality of the access rule; the repetition of the variable as a parameter has the effect of enforcing the binding between the student and his/her assignment. In this manner the policy is transposed into rewrite rules and additional conditions can be imposed to capture subtle points. We distinguish ourselves from such approaches for they do not entirely match our dynamic access model based on the distinction between program, policy and system.

1.4 Maude

Maude [9] is a high-level language and a high-performance system supporting executable specification and both equational and rewriting logic computation [10].

Because of these features Maude is suitable for modeling nondeterministic concurrent computations. Its main modules are *functional* and *system*, corresponding respectively to the logics it supports. It must be noted that Maude also contains OBJ [20] as a sublanguage representing object-oriented computation enclosed in object-oriented modules. We are focusing on exploring Maude's environment in relation to formalization of security polices in a way that is easy to comprehend and replicate for a wide range of applications.

1.4.1 Functional Modules

Functional modules are equational-style functional programs with user-definable syntax in which a number of sorts, their elements, and functions on those sorts are defined [9]. They are theories in *membership equational logic* [7] extending *order-sorted equational logic* [19], support subsort relations, and overloading of operators. Functional modules are assumed to be Church-Rosser [2] and terminating (pg. 16) – if equations are applied repeatedly then eventually an irreducible term will be computed (pg. 15). Using these types of modules we define our policy signature and authorization mechanism.

1.4.2 System Modules

A level higher, system modules are theories in rewrite logic having three types of statements: equations, memberships and rules, all of which can be conditional [9]. The rules, contrary to equations in a functional modules, do not have to be Church-Rosser or terminating. A TRS will specify concurrent transitions in our system comprising of the policy and the program. More specifically, a set of conditional rules will define the transitions in our system - the system will proceed depending on decisions computed by the authorization mechanism.

1.4.3 Tools

Because we specify security policies using rewrite logic we can employ formal methods to reason about certain properties. We can use Maude's built-in search capabilities to reason about the performance of the system. A rewrite theory is most of the times nondeterministic: to analyze all possible behaviors from a given initial state, one can use Maude's high performance search capabilities and/or Maude's built-in Model Checker. The Model Checker provides fully automated decidable program verification of finite-state systems. In addition to the Model Checker, Maude comes with a Termination Tool used to prove termination of functional modules, which is an important property for access control policies. Another useful feature is the Sufficient Completeness Checker (SCC) that verifies the property whether every operation in a specification is defined on all valid inputs, which is important for policy composition.

1.4.4 Example

Following is a sample approach for defining the policy V from Figure 2 using Maude. Regardless of the programming technique for implementing the authorization mechanism, it is essential that we indicate the main sorts, such as Subject, Action and Object, and their subsorts. We also need to define the available actions. The order-sorted nature of the terms avoids common mistakes, because type-checking is performed on the specifications.

```
fmod SIGNATURES-SORTS is

protecting STRING .
protecting NAT.

---general sorts
sorts Subject Object Action SysAction SubjAction Fact Facts
Decision Node .
subsort SysAction SubjAction < Action .
subsort Fact < Facts .

---policy domain specific sorts
sorts Student Grader Assgmt Grade .
subsorts Student Grader < Subject .
subsorts Assgmt Grade < Object .

---constructs
op student :  Nat ⟶ Student [ctor] .
op as :  Nat Grade Student ⟶ Assgmt [ctor] .
---respectively, assgnmt number, grade, and student who submitted
the assgmt

op grade :  Nat ⟶ Grade [ctor] .

---actions
op checkGrade :  Ass ⟶ SubjAction .
op closeAssgmt :  Nat ⟶ SysAction .
⋮

endfm
```

Figure 4: Sorts Partitioning in Maude

In the signature for the rewriting-based access control policies we can represent requests ground terms containing the 3-tuple: *subject*, *action* and *object*; *permit* and *deny* serve as decision replies. A policy module in Maude can be viewed as a static module containing a set of conditional equations that evaluate the facts in the environment (a multiset of atomic sentences) upon a request:

```
fmod SIGNATURES-POLICY is

protecting SIGNATURE-SORTS .

---variables
vars N G S S1 :  Nat .

---decisions
op permit deny :  ⟶ Decision [ctor] .

---authorization construct
op auth :  Facts Action Subject ⟶ Decision .

---access rules defining the auth function
eq auth(F, checkGrade(as(N,G,student(S))), student(S1)) = if S ==
S1 then permit
else deny fi .
(corresponds to rule 1.  A student may read only his own grade for
an assignment)
⋮

endfm
```

Figure 5: Policy Signature in Maude

Modules such as the one in Figure 5 could be created for any real world policy regardless of the structure or hierarchy it possesses. We expand and apply these initial steps to analyze an Assignment Submission Application, to encode the specifications following the above ideas, and to check for properties that are not explicit policy rules using the Model Checker tool.

1.5 Rewriting-Based Approaches

In the short example above we observed that a policy written in natural language can be transformed into a sentence in a rule-based language such as term rewriting. There are other existing approaches employing similar techniques using Datalog [16], first order logic [21], or temporal logic [12]. In the following two sections we refer to a rewrite formalization of a medical corporation policy based on an XACML [26] specification and to a formalism of Role-Based Access Control (RBAC) using the same paradigm. We distinguish ourselves from the approach for RBAC using term rewriting by Barker and Fernandez [3] but we are influenced by the work of de Oliveira [13] on his Maude implementation concept.

1.5.1 Fernandez

In [3] the authors explore how to represent access control models and policies using term rewriting systems; in particular representing role-based models and models based on access control lists. The rewrite system R_{ACL} for the former constitutes rewrite rules of the type $acl(1, r, U) \rightarrow grant$, where a request is expressed as $access(u, req)$ – with u denoting a user and req denoting an r (read), w (write) or x (execute) action – is evaluated using the list of rules. In contrast to this model, we use term rewriting to specify a complete system with the program separate from the policy specification. While Fernandez et al. mention the idea of using Maude as the specification environment, we go further by providing an approach to translating Datalog specifications into Maude modules along with a set of a Maude implementation of *Turnin* (pg. 44).

1.5.2 Oliveira

In [13], Oliveira proposes a formalization of rewriting-based access control policies with the policy being represented as a set of rewrite rules as opposed to equational theory axioms (pg. 26). He presents an implementation of a medical corporation policy using Maude modules. Requests are represented as ground terms containing the 3-tuple: subject, action and object:

$$req : Subject \times Action \times Object \quad \rightarrow \quad Request$$

and the system is represented by a set of states and state transitions triggered by the requests. We have adopted the use of a reference monitor as a part of the system, which evaluates the system states but we make the clean distinction between a system and a program. We also attempt to model a behavior of a system comprising of a policy and a program rather than analyzing a policy rule and a program state as part of the same algebraic term.

1.6 Contributions

This thesis builds on existing work in the area of formalization of access control policies using term rewriting. We adopt the ideas and base our implementation partially on the work of Dougherty et al. [16]. Because of the availability of proof techniques for verifying properties and the availability of rewrite engines, term rewriting was chosen as the language for this project. The main contribution lies in the fact that we provide an implementation of the theories using Maude and take advantage of its built in tools to verify certain properties. We test our approach on Turnin [17], an Assignment Submission Application, using the available specifications [17] and offer an approach to translate Datalog policy specification to Maude implementation (pg. 30).

It is imperative to stress the realization of the interaction between facts in the environment and the place of the latter in the request authorization process. We have exemplified the above as Maude modules (pg. 44) using equational theory axioms (pg. 25) and conditional rules (pg. 27) – both part of the term rewriting paradigm. The former express the static nature of the policy and define the authorization function (pg. 25), while the latter simulates the system that embodies both the policy and the program (pg. 25). In the chapters to follow, we provide definitions that comprise the basis for our approach to modeling the policy, the program and the system.

2 Foundation of Rewriting-Based Security Policies

Access control in information systems refers to the mechanism of granting access to data and available resources to authorized entities. These entities comprise agents capable of performing certain computations such as the invocation of a command or a program, the initiation of a thread of execution, etc. According to Lampson [24], access by a subject (pg. 11) of an object (pg. 11) in the system is decided by an access matrix with entries determined by certain rules. The rules

comprise the *access control policy*, which maps each entity – user, resource and action (pg. 11) – to a decision. Upon an action initiated by a user, the reference monitor consults the policy and issues a decision based on the information from the request and relevant facts in the environment.

2.1 Terms and Concepts

We have defined below the relevant access control terms that are used throughout this document.

(Object) The set of all protected entities relevant to the system protection requirements is called the set of *objects* [6]. Lampson treats entities such as processes, domains, records, fields, files, segments and terminals as objects and the level of abstraction depends on the protection requirements of the system.

(Subject) A *subject* is an active entity, such as a process or a user. The subject may initiate a transition from the current state of the system depending on the information in the request (pg. 11) and the authorization decision (pg. 11).

(Action) An *action* is the initiation of an operation by a subject. Depending on the level of abstraction we can differentiate between *system actions*, initiated by the system, and *subject actions*, initiated as requests by subjects.

(Request) A *request* is a triple consisting of an object being accessed, a user initiating an action and the action itself.

(Decision) A *decision* is the result issued by the authorization mechanism, either *deny* or *permit*, upon evaluation of the information provided by the request and the current state of the environment.

(Environment) The *environment* is a collection of relevant policy domain facts.

2.2 Historical Perspective on Access Control

Examples of access control depend on the specific requirements of the domain and the security policy to be in use. For example, applications that manage personal records, being financial, health or other, require a sophisticated mechanism to maintain confidentiality or integrity properties by means of access rules. These sets of global constraints on the system comprise the *access control policy*.

Access control policy is just one of the mechanisms to enforce a given security policy. When an application or user requests to act upon a resource, the decision tool provides, or restricts, access to the resource based on information in the security policy. While access control policies can pertain to specific applications, they can also refer to user actions within the context of a certain domain: they may govern resource usage based on users' roles, obligations, permissions, etc. depending on the properties the security policy aims to satisfy.

Lampson [24] organized the objects that need protection, the subjects that perform actions on these objects and the actions themselves into a matrix of privileges regulating the access of subjects to objects. In this framework, a singular entry in the matrix represented the right a given subject has over an object. Unfortunately, he does not specify precisely how the access control matrices should be used by the defined domains.

The need for a mathematical model of security policy describing levels of access led to the formalization of military access control rules by Bell and Lapadula [5]. The BLP model described the military classification system where user with a certain authorization level is only allowed to access information labeled at or below their designated classification level. Having the policy designate the appropriate labels according to the security levels does not make the information flow transparent upon implementation. The subtleties of the interaction between entities in the system required a rigorous model that could be analyzed in detail.

Harrison, Ullman and Ruzzo [22] address the issue of protection in a more technical manner. They gave a mathematical definition of "safety" and address the

question of reasoning about the safety of a system; in particular, the decidability of safety. "Autonomous" entities relinquish some of their rights for higher protection levels – in the modeled system this fact undoubtedly remains. This leads to the conclusion that in general it is undecidable if "a particular generic right is entered at some place in the matrix where it did not exist before." Nevertheless, if the problem is narrowed down to some restricted cases, then safety is decidable concerning the "leaking" of certain rights. The remark of Harisson et al. showed that if the system has little control over the flow of rights between subjects, then there exists a chain of delegation of rights that will eventually grant access to an object to an unauthorized user.

All of the above works are considered central to the field of computer security for they introduced the idea of formalization and verification of security policies. The need for a mathematical model was recognized and in this setting security properties could be checked. Because of the limitations of the early models to describe the current system, a new, rule-based approach for specifying policies emerged as a flexible solution to a wider scope of systems. The availability of a rule-based specification language does not make the search for a better representational model obsolete.

2.3 Languages & Logics for Access Control

Because of the wide range of applications that demand the enforcement of security policies, specific languages must be used to express the policy statements. A number of languages for access control have been proposed. A role-based language and system for expressing policies, Cassandra [4] supports credential-based access control written in Datalog with constraints. In this setting, policies are viewed as declarative statements over data from requests and over relations that capture information gathered by the application. Domain-specific languages such as EPAL [27] are also widely used. EPAL specifies data handling enterprise privacy policies; a policy in EPAL defines lists of data-categories along with sets of actions and is

used to model the services in the system's domain.

With the increased number of applications utilizing XML for data modeling, additional languages have been proposed such as WS-Policy and XACML [26]. The latter is a standard that describes both a policy language and an access-control decision language written exclusively in XML. The authors of [18] present a tool (Margrave) for analyzing XACML policies (for ex. RBAC) through a verifier written in XACML. They discuss two forms of policy analysis, namely, verification relative to known properties and change analysis, and present techniques for performing the analysis in Margrave.

Recently, language designs have been structured around concepts from logic programming such as Languages such as Li et al.'s D1LP and RT, Jim's SD3, and DeTreville's Binder [15]. Binder [14] utilize the formal declarations, flexibility and expressiveness providing for an access control language requiring modest programming proficiency. Because of their formal nature, logic based languages provide an attractive framework for specifying policies and we explore the flexibility and expressiveness of one such language in this work, namely, term rewriting.

Several proposals introduce term rewriting as a language for describing access control policies. In [3], the authors model role-base access control (RBAC) and access control lists (ACLs). They explore the use of term rewriting for evaluating access requests and for proving properties of an access control policy. Rather than focusing on the dynamic aspect of the access control environment, they focus mainly on characterization of consistency of policies. Use of an automated tool to reason about policy properties is briefly mentioned and an executable specification of ACL policy is given but implementation is not suggested.

Work in the area of defining security policies using term rewriting can be seen in Oliveira's proposal [13] on formalization for access control. He represents the state of the system to which policies are enforced as an algebraic term and discusses the relation between properties of TRS, such as confluence and termination.

3 Term Rewriting

The following rewriting definitions are adopted from "Term Rewriting and All That" [2]. In the term rewriting paradigm, *terms* (pg. 15) are built from function symbols and variables thus, in order to know which are the domain symbols, we introduce *signatures*.

Definition 1 (Signature)**.** A *signature* Σ is a set of function symbols, where each $f \in \Sigma$ is associated with a non-negative integer n denoting the arity of f. This definition will be extended for our purposes to include the set of sorts S (pg. 23), thus a *many-sorted* signature is defined as a set F of function symbols and a set of sorts S where each $f \in \Sigma$ has a profile $f : S_1 \times ... \times S_n \longrightarrow S$. If $n = 0$ then f is defined as a constant symbol.

Definition 2 (Term)**.** Let X be a set of variables such that $\Sigma \bigcap X = \emptyset$, then the set $T(F, X)$ of all well-sorted $\Sigma-$terms is defined inductively as

- $X \subseteq T(F, X)$, i. e., each $x \in X$ is a term with $x : S$ denoting the sort of x
- $\forall\, n \geq 0$, all $f \in \Sigma^{(n)}$

If t_1 is a variable-free term then t_1 is called a *ground term* and the set of such terms is denoted $T(F)$. The term t_1 is said to be in its *normal form* or *irreducible* if there is no term t_2 such that $t_1 \longrightarrow t_2$. Many-sorted terms are built on many sorted signatures [23].

Definition 3 (Substitution)**.** A *substitution* σ is a function, $\sigma : X \longrightarrow T(F, X)$, such that $\sigma(x) \neq x$ for only finitely many xs. The finite domain of σ is written as $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$, with $\forall i \in \{1, \ldots, k\}$, x_i and t_i have the same sort. Applying σ to t simultaneously replaces all occurrences of variables by the respective σ-images. Recursively,

- if t is x_i then $\sigma(x) = t_i$,

- if t is $x \neq x_i$ then $\sigma(x) = t$,
- if t is a term $f(u_1, \ldots, u_k)$ with $u_1, \ldots, u_k \in T(F, X)$ then, $\sigma(t) = f(\sigma(u_1, \ldots . \sigma(u_k))$.

Definition 4 (Multiset)**.** A *multiset* M over a set of elements A is a function $M : A \longrightarrow \mathbb{N}$. Using finite multisets we can build terminating orders (because $\longrightarrow$ is finitely branching). For example, $\{t_1, t_2, t_2\}$ and $\{t_2, t_1, t_2\}$ are identical, while $\{t_1, t_2, t_1\}$ is distinct from them.

Definition 5 (Rule)**.** A *rewrite rule* is an identity $t_1 \approx t_2$ denoted $t_1 \longrightarrow t_2$ with t_1 and t_2 called respectively left-hand side and right-hand side of the rule. The following restrictions apply for rules:

- set of variables of $t_1 \supseteq$ set of variables of t_2
- $t_1 \notin X$ and $t_1, t_2 \in T(F, X)$
- t_1 and t_2 are of the same sort

A set of rewrite rules is called a term rewriting system (TRS).

Definition 6 (Rewrite Relation)**.** A relation on $T(F, X)$ is a *rewrite relation* if and only if it is compatible with Σ-operations and closed under substitutions. Given a TRS R, the rewrite relation associated to R over $T(F, X)$ is denoted $\longrightarrow_R$ and is well-defined.

Definition 7 (Termination)**.** A reduction $\longrightarrow$ is *terminating* if and only if, there does not exist an infinite descending chain $t_1 \longrightarrow t_2 \longrightarrow \ldots$.

Definition 8 (Consistency)**.** *Consistency* is a security policy property that ensures that a unique decision is computed from the current program state (pg. 19) and a given request.

Definition 9 (Confluence)**.** A set of rules A is *confluent* when any two rewritings of a term can always be unified by further rewriting: if $t \longrightarrow {}^{*}_{A} t_1$ and $t \longrightarrow {}^{*}_{A} t_2$, then there exists a term $t^{'}$ such that $t_1 \longrightarrow {}^{*}_{A} t^{'}$ and $t_2 \longrightarrow {}^{*}_{A} t^{'}$ [9].

Definition 10 (Completeness)**.** *Completeness* is a security policy property that ensures that for any given request there is a corresponding authorization decision.

We model the program and the policy using equational theory – a sublogic of term rewring logic – while the system is modeled using conditional rules.

3.1 Equational Theory

Any given equality formulas is built from the set of terms defined above and the equality predicate " = ".

Definition 11 (Equality or Axiom)**.** An *equality (axiom)* is a pair of two terms $< t_1, t_2 >$, which is denoted $t_1 = t_2$. Both t_1 and t_2 are of the same sort, with the variables universally quantified.

Definition 12 (Matching)**.** Let t_1 and t_2 are terms of the same sort then, if there exists a substitution σ such that $\sigma(t_1) = t_2$ then we say that there is a *matching* between t_1 and t_2.

3.2 Conditional Rules

Definition 13 (Conditional Rule)**.** A *conditional rule* is an oriented condition equality denoted $t \longrightarrow s$ if Γ , where Γ is a conjunction of equalities [23].

Definition 14 (Conditional Rewriting)**.** A *conditional TRS* R is a set of conditional rules R over a set of terms $T(F, X)$.

4 Conceptual Model

Our conceptual model describes policy enforcement of security rules guarding transitions in the program states. The idea is that we would like to preserve policy/program separation since the program can exist independently of a mechanism applying a given policy. The latter should be defined separately and implemented on top of the running program as a reference monitor (pg. 18).

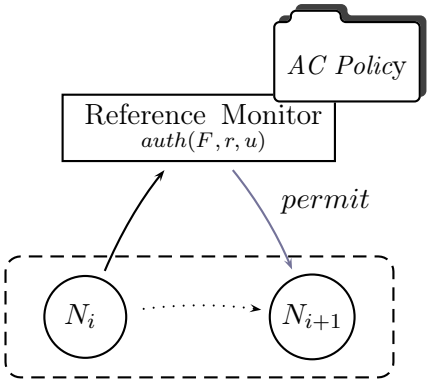

Figure 6: Reference Monitor Model

In the above representational schema the current set of facts and the request contained in the the node N_i are delivered to the reference monitor, which evaluates every request according to the set of equational rules A (pg. 26). If the policy permits the request the application proceeds with no error indication. Otherwise, the system terminates.

4.1 The Program

It is clear that a program can exist without a guarding mechanism. Hence, we model the program as an independent labeled transition system.

Definition 15 (Program LTS)**.** A *program* P is modeled as a labeled transition system (LTS). The LTS is a quadruple (V, L, $\rightharpoonup$, I) where

- V is a set of vertices with $v \in V$ corresponding to a program state
- L is a set of labels with $l \in L$ corresponding to an action
- $\rightharpoonup \subseteq V \times L \times V$ is a ternary relation of labeled transitions
- $I \subseteq V$ is a set of initial states

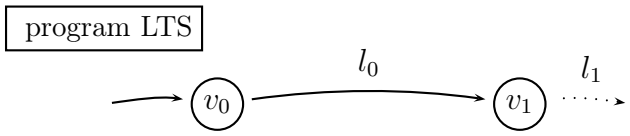

Figure 7: Program LTS

4.2 The Policy

A security policy is a statement of what actions are permitted and what are denied in a given environment. For that purpose we need to define positive (permit) and negative (deny) authorizations. These are elements of the set of decisions.

Definition 16 (Decision)**.** We define the set D of *decisions* depending on the policy's requirements as either

- $D = \{permit, deny\}$, or
- $D = \{permit, deny, undefined\}$.

Definition 17 (Policy)**.** A *policy* A is a function $\Theta : V \times L \longrightarrow D$. It generates a decision $d \in D$ for each $< V, L >$ pair.

Policy consistency is inherent in this definition. The interesting question arises when we encode the policy as a set of equational theory axioms – does the set define a policy according to the above definition?

4.3 The System

In our model the rewrite system is the program extended with the policy. A *system state* is a node $n \in V$ with $l \in L$ labeling the transition.

Definition 18 (System)**.** We denote the *system* T as ordered pair, $T \triangleq < P, A >$.

4.4 The Logic

We use *linear temporal logic* (LTL) for property specification and *model checking* to decide whether – given our model – a property holds. Using LTL we can specify the safety, liveness or other properties (pg. 1).

Definition 19 (LTL)**.** Let AP be a set of atomic propositions, then the propositional $LTL(AP)$ formulas are defined as [9]:

- **True**: $T \in LTL(AP)$.
- **Atomic Propositions**: if $p \in AP$, then $p \in LTL(AP)$.
- **Next operator**: if $\alpha \in LTL(AP)$, then $\bigcirc\alpha \in LTL(AP)$.
- **Until operator**: if $\alpha, \beta \in LTL(AP)$, then $\beta \mathcal{U} \alpha \in LTL(AP)$.
- **Minimal set of Boolean connectives**: if $\alpha, \beta \in LTL(AP)$, then $\neg\alpha$ and $\alpha \vee \beta \in LTL(AP)$

Definition 20 (Kripke Structure)**.** A Kripke structure [8] is a *4-tuple* $M = (S,\ I,\ R,\ L)$ consisting of

- a $finite$ set of states S
- a set of initial states $I \subseteq S$
- a transition relation $R \subseteq S \times S$ where $\forall s \in S$, $\exists s' \in S$ such that $(s, s') \in R$
- a labeling (or "interpretation") function $L : S \to 2^{AP}$

The labeling function associates to each $s \in S$ the set $L(s) \subseteq AP$ of propositions that hold at s.

Definition 21 (Satisfaction Relation)**.** A *satisfaction relation* between M, s and α is denoted as

$M, s \vDash \alpha$

and holds if and only if $\forall \pi \in Path(M)_s$ the *path satisfaction relation*

$M, \pi \vDash \alpha$

where the set $Path(M)_s$ of *computation paths* from s is the set of functions of the form $\pi : \mathbb{N} \longrightarrow M$ s.t. $\pi(0) = s$ and $\pi(n) \longrightarrow_M \pi(n+1)$ for each $n \in \mathbb{N}$ [9].

Associating a Kripke structure – a model of LTL – to a rewrite theory (our system) is accomplished by defining system states as *Kripke structure states* (ks-states) and defining the *ks-state predicates*.

5 Policy Formalization in Maude

The conceptual model from figure 6 is translated into Maude modules with each module representing corresponding specification. The Program and the Policy modules each inherit the Signature module, which contains domain specific sort definitions and function symbols (see *signature* on pg. 15). The System module includes the Program and the Policy, and contains conditional rewrite rules defining the system. The Kripke module includes the predefined Model Checker module and specifies the system transformation to a Kripke structure. Batch is the main executable, which loads the all of the modules and contains sample reductions.

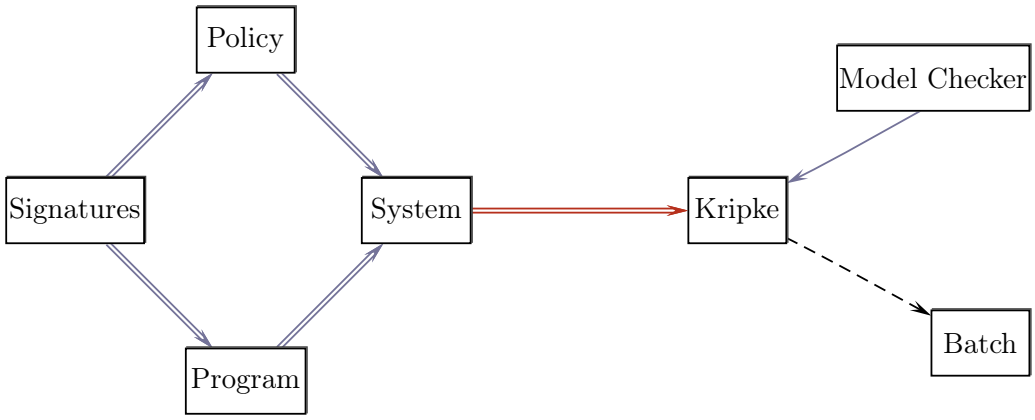

Figure 8: Maude Modules Importation Graph

In the above module relationship a single arrow represents an *including* [25] importation and a double arrow represents a *protecting* [25] importation.

5.1 Signatures

For a given access control specification we model the static policy using equational theory. We will use the following running example to illustrate the implementation in Maude:

Let us suppose that a school policy for registering students lacks granularity and its specifics are contained in a few simple rules:

1. A student is permitted to register only if the student has paid
2. A student is denied registration if the student has not paid or the student paid but the payment has been compromised
3. A student is permitted to pay if the student has not paid already

Figure 9: Register Policy

Formalizing the problem and the system requires the use of order-sorted specifications. In general, we need to indicate, which sorts are subsorts of *Subject, Action, Object* - this list can be modified depending on the requirements. The im-

plementations of the following definitions will be part of the Maude SIGNATURE module from Figure 7.

Definition 22 (Sorts)**.** Let S be the set of sorts and subsorts

$$S \quad = \quad \Big\{ Subject,\ Action,\ Object,\ SysAction,\ SubjAction,\ Fact,\ Facts,\ Decision,\ Node,\ S' \Big\}$$

- *SysAction* and *SubjActions* subsorts of *Action*
- S' denotes additional sorts specific to the environments, i. e., it is possible for $S' = \emptyset$
- *Node* denotes the system configuration

```
---general sorts
sorts Subject Object Action SysAction SubjAction Fact Facts
Decision Node .
subsort SysAction SubjAction < Action .
subsort Fact < Facts .

---specific sorts to the environment
sort Individual .
subsort Individual < Subject .
⋮
```

Figure 10: Constructs and Sorts

We define the policy environment as a set of facts. Important notion here is the *fact satisfiability*: the environment satisfies a given fact if the fact is an element of the set of facts in the current state:

Definition 23 (Fact Satisfiability)**.** Let P be the set of environment typical n-ary predicates with n dependent exclusively on the specifications, and *has* is the binary function

$$has \ : \ Facts \times P \quad \rightarrow \quad Boolean$$

then for $F \in Facts$ and $p \in P$ the function is defined by the following axioms:

- $has(F, p) = True$ if $p \in F$
- $has(F, p) = False$ otherwise.

```
---facts specific to the policy domain
ops paid regd bncd :  Individual -> Fact .

---fact satisfiability axioms
var F : Facts .
var P : Fact .

op _has_ :  Facts Fact -> Bool .
eq F P has P = true .
eq F has P = false [owise] .
⋮
```

Figure 11: Facts and Fact Satisfiability

Definition 24 (Multiset of Facts)**.** The set of environment relevant *Facts* is a multiset of predicates $P \subseteq Facts$ composed together by a defined associative and commutative constructor.

```
---multiset of facts

op nil :  -> Facts [ctor] .
op __ :  Facts Facts -> Facts [assoc comm id:  nil] .
⋮
```

Figure 12: Facts

Definition 25 (System Actions)**.** The set *SysAction* contains actions that are system specific and do not require authorization (see *auth* function on pg. 26).

Definition 26 (Subject Actions)**.** The set *SubjAction* contains actions specific to the environment and are defined by n-parameter functions with n dependent on the specifications. Such functions map $S' \subseteq S$ to the sort *SubjAction*.

```
---requests specific to the domain

op wtr :  Individual -> SubjAction .  ---subject wants to register

op wtp :  Individual -> SubjAction .  ---subject wants to pay
op bnc :  Individual -> SubjAction .  ---subject check bounce
⋮
```

Figure 13: Subject Actions

5.2 The Program

We represent the program as a set of equations defining the transition system of the environment facts as depicted in Figure 1.

Definition 27 (State-update Function)**.** The function $nextF$ is defined as

$$nextF : Facts \times Action \times Subject \longrightarrow Facts$$

This function evaluates the current set of environment facts along with the information in the request and computes the new set of facts.

```
*** ---facts state-update function---
*** next Facts equations over the requests

op nextF : Facts Action Subject -> Facts .

vars I I1 :  Individual .

** ---PROGRAM---
eq nextF(F, wtr(I), I1 ) = F paid(I) regd(I) .
eq nextF(F, wtp(I), I1 ) = F paid(I) .
eq nextF(F, bnc(I), I1 ) = F bncd(I) .
⋮
```

Figure 14: Program Module

5.3 The Policy

The Policy module contains constructs for the policy's decisions and set of rules defining what actions are allowed according to the facts in the environment.

```
*** generic decision constructs
ops permit deny :  -> Decision .
```

Figure 15: Decisions

The goal of the policy is to ensure that all requests are accounted for, i. e., there is a decision for each request. Since the resulting decision does not depend exclusively on the request but on the current facts we define a function that returns decision derived from the normalization process issued from a request term and from the information contained in the environments.

Definition 28 (Authorization Function)**.** Let the policy P be a set of authorization rules A where each request is defined using equational theory axioms. We define the binary function *auth* as

$$auth : Facts \times Action \times Subject \rightarrow Decision$$

An authorization rule is a conditional equation where the left side is a term of the form $auth(F, Act, Subj)$ with $F \subseteq Facts$, $Act \in Action$ and $Subj \in Subject$. The equation is reduced to a $d \in Decision$ if a condition is true about the environment, i. e., if the set of facts satisfies a policy relevant predicate.

```
op auth :  Facts Action Subject -> Decision .

***pr 1 :  I is permitted to register if I paid

eq auth(F, wtr(I), I) = if F has paid(I) and not F has bncd(I) and
not F has regd(I)
then permit
else deny fi .

***pr 2 :  I is permitted to pay if I did not pay already
⋮
```

Figure 16: Policy: set of authorization rules A

The goal is to to have the policy static and on top of the system – this way

we can freely make changes to A without altering the program, which may exist without the policy.

5.4 The System

In the program LTS, transition from an environment state F to environment state $F^{'}$ depends on the information contained in either a system action (pg. 11) or a subject action (pg. 11). We differentiate between two implementation of the system – a *weak* and a *strong* one. We consider a weak implementation one where the program checks the policy as a guard on actions whereas a strong implementation suggests that the program is not aware of a guarding mechanism (see our model on pg. 18) but it is the system that enforces the use of the reference monitor.

Definition 29 (Node Function)**.** Let the system T be a set of conditional rewrite rules TRS where node transition of T corresponds to a permitted request from a subject to execute an action over a resource considering the current set of environment facts. We define the function *node* as

$$node : Facts \times Action \times Subject \rightarrow Node$$

An important note must be made here about the fact that we have to encode the system's transition labels, i. e., the action inside of the system nodes .

A transition is a defined as a conditional rule where the left-hand side is a term of the form $node(F, Act, Subj)$ with $F \subseteq Facts$, $Act \in Action$ and $Subj \in Subject$, and the right-hand side is a term of the form

$$node(nextF(F, Act, Subj), Act^{'}, Subj^{'})$$

where $Act^{'} \in Action$ and $Subj^{'} \in Subject$ are a new action and a subject allocated respectively by the system.

As defined above, the *nextF* function will compute the new set of facts depending on the information from the request. The conditional part of the rule refers to the authorization mechanism – the transition will be allowed if the the *auth* function returns a *permit* decision for the next node in the system.

Naturally, we would like to restrict the system to be composed of only *good* nodes. That is, if N is the set of all possible states, the set $G^A \subseteq N$ is the set of good nodes as specified by the policy A. A node $g \in N$ is considered good if the authorization mechanism allows the transition from an initial state **to** g based on the request and the facts.

Definition 30 (Good Node)**.** We define a node of the form $node(F, Act, Subj)$ *good* if

$$auth(node(F, Act, Subj)) \quad = \quad permit$$

.

In order to restrict the system so that it is composed only of good nodes we have to consider the next state, i. e., starting form an initial node $n_0 \in G^A$, the only allowed transitions $n_i \rightharpoonup n_{i+1}$ are those that produce good nodes.

Initially, our model of the system included a *denied* node, which served as an error node for transitions for which the *auth* function returned a *deny* decision. We realized that this implementation caused the generation of unwanted, or rather trivial counterexamples when we used the Model Checker to verify properties.

We implement the system as a Maude system module, which inherits the signatures of the policy and the program.

```
op node :  Facts Action Subject -> Node .
op denied :  -> Node .

*** rule for "wants to pay" request
crl node(F, wtp(I), I) => node(nextF(F,wtp(I), I), R1, I) if
auth(nextF(F,wtp(I), I), R1, I) = permit /\ (R1 restRequest) :=
(wtr(I) bnc(I)) .
:
```

Figure 17: System Module

5.5 The Logic

Exploration of all the possible rewrites from a given state is accomplished trough the Model Checker module provided with Core Maude. To be able to check the policy for certain properties we need to associate the Kripke structure above to rewrite theory. This is accomplished with the Kripke module, in which we make explicit the Boolean state predicates and define the ks-states as a superset of our configuration.

Definition 31 (Model Checking)**.** We define a new module, which extends the predefined policy, program and system modules along with the `MODEL-CHECKER` module, and contains

- the sort *Node* defined as a subsort of *State* (predifined sort inherited from the `SATISFACTION` module),
- relevant state predicates as operators of sort *Prop*, and
- set of equations specifying for what states a predicate evaluates to *True*.

These propositions are specific to the and are need for stating the desired safety (something bad never happens, $G\neg\varphi$), liveness (something good keeps happening, $G(\varphi \rightarrow F\varphi)$) or any other properties.

```
subsorts Node < State .

ops ispaid isregd isbncd :  Individual -> Prop .
op P : -> Prop .
var F : Facts .
var Act :  Action .
var Subj :  Subject .
var I : Individual .

*** equations defining when each of the state prediactes holds in
a given state
ceq node(F,Act, Subj) |= ispaid(I) = true if F has paid(I) .
ceq node(F,Act, Subj)|= isregd(I) = true if F has regd(I) .
ceq node(F,Act, Subj) |= isbncd(I) = true if F has bncd(I) .
ceq node(F,Act, Subj) |= P = false [owise] .
.
.
.
```

Figure 18: Model Checking Module

5.6 Analysis using Maude Tools

The Model Checker module's operator *modelCheck* takes a ks-state and a LTL formula α and return either a Boolean *True* if α is satisfied or counterexample when it is not. The counterexample is a pair of a state and a rule label applied at each state. Examples of specifying reductions corresponding to policy properties are listed below.

```
op a :  -> Individual [ctor] .
*** property 1:  if a has registered implies that a has paid
red modelCheck(node(nil,wtp(a), a), [] (isregd(a) -> ispaid(a)) )
.

*** property 2:  if a has paid but the check has bounced a cannot
register
red modelCheck(node(paid(a) bncd(a),wtr(a), a), [] ~ isregd(a) )
.
.
.
```

Figure 19: Reductions of *modelCheck*

6 Datalog Policies to Maude Specifications

In this section we give an approach for transcribing policies captured as Datalog programs into Maude modules.

6.1 Definitions

We have adopted the following Datalog terms and definitions [11].

(atom) An *atom* is a string of letters, digits, underscores; an atom string starts with a lowercase letter.

(variable) A *variable* is a string of letters, digits, underscores; a variable string starts with a capital letter.

(identifier) An *identifier* is a string omitting special characters; an identifier strings does not start with a capital letter.

(term) A *term* is a variable or constant or underscore.

(literal) A *literal* is a *predicate symbol* followed by optional parenthesized list of comma-separated terms.

(predicate symbol) A *predicate symbol*or a relation is a string or identifier

(facts) A *fact* is a true assertion about a part of the environment

(rule) A *rule* is a sentence allowing for a fact or facts deduction from given facts

(goal) A *goal* is a literal preceded by the symbol "?–" used for formulating queries

To avoid symbol confusion when we talk about facts and rules, we use $\mathcal{R}$ and $\mathcal{F}$ to denote Datalog rules and facts respectively as apposed to R and F in the rewriting paradigm.

Both facts and rules are represented as *clauses* of the form

$$\mathcal{R}_0(\vec{u}_0) \ :- \ \mathcal{R}_1(\vec{u}_1), \ldots, \ \mathcal{R}_n(\vec{u}_n)$$

where $\mathcal{R}_i$ are predicates, $\vec{u}_i$ are tuples of variables and constants. The left-hand side of the rule is called a *head*, the right-hand side is considered the rule's *body*. If a clause has empty body then it is considered a fact, and a rule if it contains at least

one predicate. For a set of rules $\mathcal{R}$, $edb(\mathcal{R})$ denotes *extensional* predicates, i. e., ones occurring only in the body, and $idb(\mathcal{R})$ denotes *intentional* predicates, i. e., ones occurring in the head. A Datalog policy's signature $\Sigma_{\mathcal{R}}$ is $edb(\mathcal{R}) \cup idb(\mathcal{R})$. The set $\mathcal{F}$ of APs over $\Sigma_{\mathcal{R}}$ is defined as a set of *facts*. For the policies considered in this paper, rules are non-recursive, meaning that no *idb* predicate (permit or deny) occurs in the right-hand side of any rule.

A non-recursive Datalog rule is essentially a *conjunctive query* [1] over the set of *edb* facts. A set of such rules, then, constitutes a definition of the *idb* predicates in terms of the *edb* facts as follows. If Q is an *idb* predicate and $a_1, ...a_n$ are constants, then $Q(a_1, ...a_n)$ is defined to hold if and only if there is a rule

$$Q(\vec{u}) \ :- \mathcal{R}_1(u_1), \ldots, \mathcal{R}_n(u_n)$$

and a substitution σ mapping $\vec{u}$ to $\vec{a}$, such that for each $i, R_i(\sigma(u_1))$ is an *edb* fact.

6.2 Policy Example

A given a Datalog policy $\mathcal{R}$ includes the predicates *permit* and *deny* – called *decisions* – of the form $decision(Subject, Action, Object)$, where Subject, Action and Object are sorts. In this setting, a rule over $\Sigma_{\mathcal{R}}$ has a head either permit or deny. The policy $\mathcal{R}$ will be then a set of rules over $\Sigma_{\mathcal{R}}$ as in the example below of a students policy for reading/grading homeworks:

1. permit(X, readHW, Y) :– student(X), hw(Y), author(X, Y)
2. deny(X, gradeHW, Y) :– student(X), hw(Y)
3. permit(X, gradeHW, Y) :– faculty(X), hw(Y)

Figure 20: Datalog Policy $\mathcal{R}$

For $\mathcal{R}_1$ and a given set of Datalog facts $\mathcal{F}_1$ we can determine the truth value of the $edb(\mathcal{R}_1)$. For example if $\mathcal{F}_1$= {student($John$), student(Ana), hw(1), hw(2), author($John, 1$), author($Ana, 2$)} following (1) we add to $\mathcal{F}_1$ the computed facts,

$idb(\mathcal{R}_1)$, from $\mathcal{R}_1$, which are permit(John, readHW, 1), permit(Ana, readHW, 2), etc.

6.3 Datalog Policy to Term Rewriting Policy

Given a Datalog policy $\mathcal{R}$ there exists a policy R where the body of a $\mathcal{R}$ rule is the left-hand side of the corresponding R rule, and the head of $\mathcal{R}$ is the right-hand side of R. The relational symbol ":–" is replaced with "$\longrightarrow$" and the conjunction operator "," is replaced with the associative and commutative operator "$\wedge$" as in the example below:

$$\mathcal{R} = \text{permit}(X, \text{readHW}, Y) \text{ :– } \text{student}(X), \text{hw}(Y), \text{author}(X, Y)$$

$$R = \text{student}(X) \wedge \text{hw}(Y) \wedge \text{author}(X, Y) \longrightarrow \text{permit}(X, \text{readHW}, Y)$$

where the left-hand side of the rewriting policy rule is the set of environment facts.

In this direct transformation, the terms F_1 on the left-hand side will not be added to the list of F. One approach is to add them to the right-hand site with the conjunction operator:

$$\text{student}(X) \wedge \text{hw}(Y) \wedge \text{author}(X, Y) \longrightarrow \text{permit}(X, \text{readHW}, Y) \wedge F_1.$$

This direct transformation of Datalog rules to rewrite rules might not be suitable for implementing a policy as Maude module. Even more so, if we would like to separate the program (the transition system of environment facts) from the policy using the above direct transformation will be inadequate.

6.4 Maude Policy Module

To model $\mathcal{R}$ as a Maude specification module we need to define the required sorts and constructs beforehand (see section 6). The implementation of the small policy example above depends on the program specifications (definitions of relevant sorts and ops) but since we are concerned here only with the set of policy rules

we will partially ignore that fact. Recall, that we have defined F as a multiset of facts (pg. 16) relevant to the policy domain and the *auth* function (pg. 26) as the decision computing mechanism. Then, in general, the transformation from $\mathcal{R}$ to R as a Maude specification becomes as trivial as translating the set of Datalog policy rules to a set of conditional equations.

Definition 32 (Datalog to Maude)**.** Given a policy rule $r \in \mathcal{R}$, a decision d of the form $d(Subject,\ Action\ , Object)$, and a set of facts $\mathcal{F}_1 \subseteq \mathcal{F}$ with

$$r\ = d\ \text{:-}\ \mathcal{F}_1$$

then the corresponding rewriting rule in Maude is of the form

$$auth(\mathcal{F},\ Action,\ Subject)\ =\ d$$

For instance, the following Datalog rule from the policy above

$$\mathcal{R}\text{= permit}(X,\ \text{readHW},\ Y)\ \text{:-}\ \text{student}(X),\ \text{hw}(Y),\ \text{author}(X, Y)$$

will become either the following policy rule as an equation in Maude

$$\text{eq}\ auth(\text{student}(X)\ \text{hw}(Y)\ \text{author}(X, Y)\ ,\ \text{readHW},\ \text{hw(Y)})\ =\ \text{permit}$$

or – depending on the complexity of the predicates – the following conditional equation:

$$\text{ceq}\ auth(\text{F},\ \text{readHW},\ \text{hw(Y)})\ =\ \text{permit if F has student}(X)\ \text{hw}(Y)\ \text{author}(X, Y)\ .$$

7 Conclusions

We presented a process of formalizing security policies that are integral part of access control systems using term rewriting techniques. The implementation of the policy, the program and the system as distinct Maude modules was consistent with our conceptual model of the separation between policy and program. In view of

the fact that security policies could written as Datalog programs we provide steps to translate Datalog policy rules as sets of equational theory axioms contained in a functional Maude module. We further explored the flexibility of our approach to formalize and reason about Turnin.

7.1 Analysis of Turnin

One of the questions we can ask about a security policy is whether the policy is consistent, i. e., is it impossible to compute different decisions for the same access control request. Because we have implemented the policy as a Maude functional module consisting of equational theory axioms, Maude enforces the corresponding set of rules to be confluent hence consistent (pg. 16).

The modular implementation of the concept allows us to reason about the policy independent of the program – we can make changes to the policy without modifying the program and vice versa. The structure provides for adequate policy or program modification if a certain desired security property is proved false.

Using the Model Checker we can verify properties written as LTL formulas. Below we list several general statements in English one can ask about Turnin and their corresponding *modelCheck* reduction statement. We reason about the system starting after a predefined good initial state. The properties below hold true for all paths.

Property A: If a student s can read a submitted by student t file, then s and t are partners

Since the LTL operators *next* $\bigcirc$ and *henceforth* $\Box$ commute we can translate the above sentence into

1. $\bigcirc\Box$ (wbPer(view(f(1)), u(1)) $\longrightarrow$ isSubmitted(c(1), a(1), f(1),u(2), Sh,Bl) $\wedge$ isPartners(c(1), a(1), u(1), u(2)))).

Property B: If a student s can read a submitted file f, then either s is the owner of f or s is viewing f submitted by their partner

1. $\bigcirc\Box$ (wbPer(view(f(1)), u(1)) $\longrightarrow$ isSubmitted(c(1),a(1),f(1),u(2),Sh,Bl) $\wedge$ isPartners(c(1),a(1),u(1),u(2)) $\vee$ isSubmitted(c(1), a(1), f(1),u(1),Sh,Bl))).

Property C: Every user has at most one role per course

1. $\bigcirc\Box$ (isUserRole(c(Cn), u(Un), Rl) /\ isUserRole(c(Cn), u(Un), Rl1) $\longrightarrow$ isRoleEq(Rl, Rl1))) .

The analysis of the above properties can be taken further to include significant changes in the policy axioms and observation of how that affects the outcome of the model checking reduction. For example, questions such as does a modification to make a property hold lead to making a previously desired property not hold, can be further explored. Another important feature that our model provides is that we can also change the program and reason about the effects of those changes

7.2 Problems & Concerns

The majority of the problems of modeling Turnin appeared to be concentrated only in one area of our modular implementation - the system. Formalizing the policy as a axioms defining the *auth* function was not problematic, and neither was modeling the program's facts transition system but the simulation of random actions within the system posed as a challenge. Furthermore, due to the complexity of the domain relevant predicates and the restriction of variable rule position, we were forced to apply clumsy techniques to simulate nondeterministic, pseudo-random actions in the system.

The latter could be a likely reason for the several segmentation fault errors that caused termination of model checking reductions. It will be interesting to explore further the reasons for that and whether it was due to checking for specific types of LTL formulas or due to inappropriate definition of proposition satisfiability equations.

7.3 Future Work

Even though this work presents a model for formalization and verification of rewriting-based security polices and further provides an implementation in Maude's environment, there are still many avenues for future work. We believe that conceptually the foundation is solid but we have not explored various encoding schemes. Although we present a general way to implement policies using Maude specifications it might not be applicable to large complex systems due to the fact that we are simulating interactions between subjects and objects rather than simply reasoning about the policy as a stand-alone entity.

- The current system can be improved to better simulate nondeterministic actions. A possible way of addressing the issue is exploring the use of Object Oriented modules in Maude or Real Time Maude. It might be possible that the new environment will facilitate the simulation of system with random transitions in an agile way.

- The attempt to reason about the "correctness" of Turnin, that is, verifying properties using the Model Checker could be more successful if we introduce more specific propositions and define carefully the satisfiability axioms for them. Furthermore, specific properties can be readily analyzed after changes to the policy or the program.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. *Reading, MA*, 1995.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, August 1999.

[3] Steve Barker and Maribel Fernández. Term rewriting for access control. In *DBSec*, pages 179–193, 2006.

[4] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records.

[5] E. D. Bell and L. J. LaPadula. In *Secure computer systems: Mathematical foundations. Technical Report Mitre Report ESD-TR-73-278 (Vol. I-III)*. Mitre Corporation, 1974.

[6] M. Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2004.

[7] A. Bouhoula and J. Meseguer. Specification and Proof in Membership Equational Logic. *Tapsoft'97: Theory and Practice of Software Development: 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997: Proceedings*, 1997.

[8] Edmund M. Clarke. Model checking. *Lecture Notes in Computer Science*, 1346:54–??, 1997.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.3). January 2007.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[11] The MITRE Corporation. Datalog user manual, 2004.

[12] F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad: a security model with non atomic actions and deadlines. *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 186–196, 2005.

[13] Anderson Santana de Oliveira. Rewriting-based access control policies. *Electr. Notes Theor. Comput. Sci.*, 171(4):59–72, 2007.

[14] John DeTreville. Binder, a logic-based security language. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105, Washington, DC, USA, 2002. IEEE Computer Society.

[15] S. di Vimercati, P. Samarati, and S. Jajodia. Policies, models, and languages for access control, 2005.

[16] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR*, pages 632–646, 2006.

[17] Kathi Fisler. Turnin - assignment submission application: Specification document, November 2005.

[18] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA, 2005. ACM Press.

[19] J. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, overloading, exeptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[20] J.A. Goguen and G. Malcolm. *Software Engineering with Obj: Algebraic Specification in Action*. Kluwer Academic Publishers, 2000.

[21] JY Halpern and V. Weissman. Using first-order logic to reason about policies. *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 187–201.

[22] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[23] C. Kirchner and H. Kirchner. Rewriting, solving, proving. *A preliminary version of a book available at www. loria. fr/ckirchne/rsp. ps. gz*, 1999.

[24] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, 1974.

[25] Theodore McCombs. Maude 2.0 primer version 1.0, 2003.

[26] T. Moses. extensible access control markup language (xacml) version 1.0., February 2003.

[27] Gunter Karjoth Calvin Powers Matthias Schunter Paul Ashley, Satoshi Hada. Enterprise privacy authorization language (epal), March 2003.

A Turnin Specification [17]

Version 3, November 1 2005

Core Domains

- User(Pwd:Password) : u, u1, u2 ... - File: f, f1, ...

- Role={student, ta, faculty, admin}: r, r1, ...

- Assignment(Status: AsgmtStatus, Partners:Set[Set[User]], Submitted:File x User x SharingMode x Late Comments:User/Partnering x CommentFile) : a

- Courses(Members:set[User], Asgmts:set[Assignment], UserRoles:Members x Role): c, c1, ...

Minor Domains

- Password

- SharingMode={self, group}

- Late={true, false}

- AsgmtStatus={new, open, closed, late, deleted}: s, s1, ...

System Actions

- Change (add/remove) users: add-user(u,c,r), rem-user(u,c)

- Create assignment: create-asgmt(c)

- Change assignment status: change-status(a,c,new-status)

- Submit file: submit(u,c,a,f,mode)

- Download/View file: view(f)

- Change password: change-pwd(u,new-pwd)

- Change (add/remove) partners: make-partners(u1,u2,a), unpartner(u,a)

- Upload grading comment: upload-comments(u,c,a,cf)

Functions

- RIC(user, role, course) == (user,role) in course.UserRoles

- Partners(u,u1,a) == Exists grp: set[User] | g in a.Partners & u in g & u1 in g - cell(u,a) = {u1 | Partners(u,u1,a)}

- new-assign() returns new assignment not already in some course

---

Access policies

- Change (add/remove) user

<u1, add-user(u,c,r)> if RIC(u1,faculty,c) + RIC(u1,admin,c) <u1, rem-user(u,c)> if RIC(u1,faculty,c) & u in c.student+ta + RIC(u1,admin,c)

- Create assignment

<u1, create-asgmt(c)> if RIC(u1,faculty,c)

- Change assignment status

<u1, change-status(a,c,new-status)> if a in c.Asgmts & (RIC(u1,faculty,c) + RIC(u1,ta,c))

- Submit file [u and u1 allows policy where staff can submit for a student]

<u1, submit(u,c,a,_,_)> if u=u1 & RIC(u1,student,c) & a.Status = open

- Download/View files [Implementation may enforce instantiation of c & a]

<u1, view(f)> if Exists c in Courses, a in c.Asgmts | Exists (f,u1,_,_) in a.Submitted + Exists (f,u,group,_)
in a.Submitted & Partners(u,u1,a) + Exists (f,_,_,_)
in a.Submitted & (RIC(u1,faculty,c) + RIC(u1,ta,c)) + Exists (u1,f)
in a.Comments + Exists (_,f)
in a.Comments & (RIC(u1,faculty,c) + RIC(u1,ta,c))

- Change password

<u1, change-pwd(u,new-pwd)> if u1=u + Exists c in Courses | u

in c.Members & RIC(u1,admin,c) + RIC(u1,faculty,c) & (RIC(u,student,c) + RIC(u,ta,c)) + RIC(u1,ta,c) & RIC(u,student,c)

- Change partners

<u1, make-partners(u,u2,a)> if Exists c in Courses | a in c.Asgmts & RIC(u1,faculty,c) + RIC(u1,ta,c)

<u1, unpartner(u,a)> if Exists c in Courses | a in c.Asgmts & RIC(u1,faculty,c) + RIC(u1,ta,c)

- Upload grading comment

<u1, upload-comments(u,c,a,cf)> if RIC(u1,ta,c) + RIC(u1,faculty,c)

State-update policies

- Change (add/remove) users

add-user(u,c,r) ==> Users' = Users U {u} [assumes u not already in system, U handles if not]

c.Members' = c.Members U {u} c.UserRoles' = c.UserRoles U {(u,r)}

rem-user (u,c) ==> [does not remove user from user domain, just from course]

c.Members' = c.Members - {u} c.UserRoles' = c.UserRoles - {(u, _)}

- Create assignment

create-asgmt(c) ==> c.Asgmts' = c.Asgmts U {new-assign()}

- Change assignment status

change-status (a,c,newstat) ==> a.Status' = newstat

- Submit file

submit (u,c,a,f,mode) ==> if a.Status=late c.a.Submitted' = c.a.Submitted U {(f,u,mode,true)}

else c.a.Submitted' = c.a.Submitted U {(f,u,mode,false)}

- Download/View files [NONE]

- Change password

change-pwd (u,new-pwd) ==> Users' = Users - {(u,_)} U {(u,new-pwd)}

- Change partners

make-partners(u,u2,a) ==> a.Partners' = a.Partners - cell(u,a) - cell(u2,a) U {(cell(u,a) U cell(u2,a))}

unpartner(u,a) ==> a.Partners' = a.Partners - {cell(u,a)} U {(u)} U {cell(u,a) - {u}}

- Upload grading comment — this doesn't handle replace

upload-comments(u,c,a,cf) ==> a.Comments' = a.Comments U {(u,cf)}

Mode Transition Information (and how that affects policies)

default policy contains rules for changing users, creating assignments, changing assignment status,

change password (things that don't take assignment as an argument)

[this suggests that policy must have different rules per actual assignment, rather than the abstract class of assignments]

Transition System over AsgmtStatus

init new

new : enables make-partners, unpartner -> open, deleted

open : enables submit, download/view -> late, deleted

late : [makes submissions late, no effect on access] -> open, closed, deleted

closed: disables submit,make-partners,unpartner enables upload-comments -> deleted, late, open

deleted : disables submit, download/view, make-partners, unpartner

[currently no support for restoration]

Invariants

- If status is open or late, all users/groups in c have writeable dir to submit to

- Every user has at most one role per course

- Partners sets in assignments partition student course members

- No faculty/staff/admin is in homework group for an assignment

- Open assignments have no submissions marked late

- Submission not allowed for asgnt/usr for which a grading comment exists

---

Non-invariants

- File need not be unique to asgmts (same file may be submitted to multiple asgmts)

B Turnin Maude Implementation

Signatures Module

```
fmod SIGNATURES is


*** sorts
inc SET{Nat} .

sorts Subject Object Action SysAction SubjAction Fact Facts
    Decision Node .
subsort SysAction SubjAction < Action .
subsort User < Subject .
subsort Fact < Facts .

sorts User File Role Assignment Asgmt AsgmtStatus Status
   Partners Submitted SharingMode Course Members UserRole
   Late .
subsort AsgmtStatus < Status .
subsort Late < Bool .

*** some of this is reduntant (can merge isStatus+isAsgmt)
op isStatus : Course Assignment AsgmtStatus -> Status .
```

```
op isPartners : Course Assignment User User -> Partners .
op isSubmitted : Course Assignment File User SharingMode
   Bool -> Submitted .
op isMember : Course User -> Members . --- maybe isUserRole
    covers isMember
op isAsgmt : Course Assignment -> Asgmt .
op isUserRole : Course User Role -> UserRole .


ops self group : -> SharingMode [ctor] . --- Sharing Mode
ops new open closed late deleted : -> AsgmtStatus [ctor] .
   --- Assignment Status


*** multiset of predicates / generic
op nil : -> Facts [ctor] .
op __ : Facts Facts -> Facts [assoc comm id: nil] .


*** predicates  / specific
subsort Status Partners Members Submitted Asgmt UserRole
   Course < Fact .


*** predicate satisfiability (if predicate is an element of
    a Facts then predicate is true)
op _has_ : Facts Fact -> Bool .

vars F : Facts .
var P : Fact .

eq F P has P = true .
eq F has P = false [owise] .


*** subjects in the environment / specific
op u : Nat -> User  .
op c : Nat -> Course  .

*** objects in the environment / specific
op f : Nat -> File  .
op a : Nat -> Assignment  .

*** roles in the environment / specific
```

```
ops student ta fac admin : -> Role [ctor] .


*** SysActions are actions associated to admin or faculty
    in some cases to ta but not to students
op addUser : Course User Role -> SysAction .
op rmvUser : Course User Role -> SysAction .

op createAsgmt : Course Assignment -> SysAction . ***
op changeStatus : Course Assignment AsgmtStatus ->
    SysAction .
op makePartners : Course Assignment User User -> SysAction
    .
op unPartners : Course Assignment User User  -> SysAction .

*** SubjActions are actions that are initiated by students
op submitFile : Course Assignment File SharingMode ->
    SubjAction .
op view : File -> SubjAction .


endfm
```

Program Module

```
fmod PROGRAM is

pr SIGNATURES .

var St St1 : Status .
var B : Bool .
var Sh : SharingMode .
var Rl : Role .
var F : Facts .
var P : Fact .

--- natural numbers for users courses files assignments
vars Un Un1 Un2 Cn Cn1 Fn  An An1 : Nat .


*** courses domain
op courses : -> Facts .
```

```
eq courses = c(1) c(2) .


*** ---STATE-UPDATE--- PROGRAM MODULES
***  next Facts equations over the requests

op nextF : Facts Action Subject -> Facts .
eq P P = P . --- no fact redundancy


---add user to a course
eq nextF(F, addUser(c(Cn), u(Un), Rl), u(Un1)) = F
   isUserRole(c(Cn), u(Un), Rl) .

---rmv user
eq nextF(F isUserRole(c(Cn), u(Un), Rl), rmvUser(c(Cn), u(
   Un), Rl), u(Un1)) = F .

---create asgmt
eq nextF(F, createAsgmt(c(Cn), a(An)), u(Un)) = F isAsgmt(c
   (Cn),a(An)) isStatus(c(Cn),a(An), new) . --- now if we
   combine these tables do we loose anything??? as in
   isAsgnmt(c,a,status)

---change status
eq nextF(F isStatus(c(Cn),a(An), St), changeStatus(c(Cn), a
   (An1), St1), u(Un)) = F isStatus(c(Cn),a(An), St1) .

---submit file
eq nextF(F, submitFile(c(Cn), a(An), f(Fn), Sh), u(Un)) =
   if F has isStatus(c(Cn),a(An), late)
then F isSubmitted(c(Cn), a(An), f(Fn),u(Un), Sh,true)
else F isSubmitted(c(Cn), a(An), f(Fn),u(Un), Sh,false) fi
   .

---view file no change
eq nextF(F, view(f(Fn)), u(Un)) = F .

---make partners
eq nextF(F , makePartners(c(Cn), a(An), u(Un), u(Un1)), u(
   Un2)) = F isPartners(c(Cn), a(An), u(Un), u(Un1)) .

eq isPartners(c(Cn), a(An), u(Un), u(Un1)) isPartners(c(Cn)
   , a(An), u(Un1), u(Un)) = isPartners(c(Cn), a(An), u(Un)
```

```
    , u(Un1)) .

endfm
```

Policy Module

```
fmod POLICY is

protecting SIGNATURES .

vars St : Status .
vars B : Bool .
vars Sh : SharingMode .
vars Rl Rl1 : Role .
vars F : Facts .
var Act : Action .

--- natural numbers for users courses files assignments
vars Un Un1 Un2 Cn Cn1 Cn2 Fn  An An1 : Nat .

--- admin is present in all courses and is u(0)
eq F has isUserRole(c(Cn),u(0),admin) = true .



*** ---POLICY---
ops permit deny : -> Decision .
op auth : Facts Action User -> Decision .


*** ACCESS RULES

---add/rmv users from courses AR1: only faculty in course c or
    the admin can add users to c ---EVERY USER HAS AT MOST ONE
    ROLE PER COURSE
ceq auth(F, addUser(c(Cn), u(Un), Rl), u(Un1)) = permit if  F
    has c(Cn) and
not F has isUserRole(c(Cn),u(Un),ta) and
not F has isUserRole(c(Cn),u(Un),student) and
not F has isUserRole(c(Cn),u(Un),fac) and
(F has isUserRole(c(Cn),u(Un1),fac) or F has isUserRole(c(Cn),u(
    Un1),admin))
and not Rl == admin .

eq auth(F, addUser(c(Cn), u(Un), Rl), u(Un1)) = deny [owise] .
```

```
***can the admin remove himself from the course with role admin?
     no
ceq auth(F , rmvUser(c(Cn), u(Un), Rl), u(Un1)) = permit if  F
    has c(Cn) and F has isUserRole(c(Cn),u(Un),Rl) and F has
    isUserRole(c(Cn),u(Un1),admin) and not Rl == admin .
ceq auth(F , rmvUser(c(Cn), u(Un), Rl), u(Un1)) = permit if  F
    has c(Cn) and F has isUserRole(c(Cn),u(Un),Rl) and F has
    isUserRole(c(Cn),u(Un1),fac) and not Rl == admin and not Rl
    == fac .
eq auth(F, rmvUser(c(Cn), u(Un), Rl), u(Un1)) = deny [owise] .


---create asgmt for a course AR2: only faculty in c can create
    assgmnts in c ( when you create an assignment a fact is added
     wtih status new
eq auth(F, createAsgmt(c(Cn), a(An)), u(Un)) = if  F has c(Cn)
    and not F has isAsgmt(c(Cn), a(An)) and F has isUserRole(c(Cn
    ),u(Un),fac)
then permit
else deny fi .


---change status for an assignment AR3: fac and students in c
    can change a if a is in c (if we look at the one below this
    becomes
***eq auth(F isAsgmt(c(Cn), a(An1), St), changeStatus(c(Cn), a(
    An1), St1), u(Un)) = if  (F has isUserRole(c(Cn),u(Un .....
eq auth(F, changeStatus(c(Cn), a(An1), St), u(Un)) = if  F has c
    (Cn) and F has isAsgmt(c(Cn), a(An1)) and (F has isUserRole(c
    (Cn),u(Un),fac) or F has isUserRole(c(Cn),u(Un),ta))
then permit
else deny fi .


---submit file AR4: //note: staff can submit for students? // we
     can merge isStatus and isAsgmnt as in isAsgnmt(c,a,s)
    instead of isA(c,a) (this will add a fact isSubmitted(Un..)
eq auth(F, submitFile(c(Cn), a(An), f(Fn), Sh), u(Un)) = if  F
    has c(Cn) and F has isAsgmt(c(Cn), a(An)) and F has
    isUserRole(c(Cn),u(Un), student) and F has isStatus(c(Cn), a(
    An), open)
then permit
else deny fi .


---view file AR5: fac and ta and owner or partners can view file
ceq auth(F c(Cn) isAsgmt(c(Cn), a(An)) isSubmitted(c(Cn), a(An),
     f(Fn),u(Un),Sh,B) , view(f(Fn)), u(Un1)) = permit if Un ==
```

```
    Un1 . ***owner can view file

ceq auth(F c(Cn) isAsgmt(c(Cn), a(An)) isSubmitted(c(Cn), a(An),
     f(Fn),u(Un),Sh,B)  , view(f(Fn)), u(Un1)) = permit if Sh ==
    group and F has isPartners(c(Cn), a(An), u(Un), u(Un1)) . ***
    partner can view file

ceq auth(F c(Cn) isAsgmt(c(Cn), a(An)) isSubmitted(c(Cn), a(An),
     f(Fn),u(Un),Sh,B) , view(f(Fn)), u(Un1)) = permit if F has
    isUserRole(c(Cn),u(Un1), fac) or F has isUserRole(c(Cn),u(Un1
    ), ta) .

eq auth(F, view(f(Fn)), u(Un1)) = deny [owise] .



---partners AR6: fac or ta can make/split partners (and other
    cases?); ta/fac cannot be partners with students
eq auth(F, makePartners(c(Cn), a(An), u(Un), u(Un1)), u(Un2)) =
    if F has c(Cn) and F has isAsgmt(c(Cn), a(An))
and (F has isUserRole(c(Cn),u(Un2),fac) or F has isUserRole(c(Cn
    ),u(Un2),ta))
and F has isUserRole(c(Cn),u(Un), student) and F has isUserRole(
    c(Cn),u(Un1), student)
and not F has isUserRole(c(Cn),u(Un),ta)
and not F has isUserRole(c(Cn),u(Un1),ta)
and not F has isUserRole(c(Cn),u(Un),fac)
and not F has isUserRole(c(Cn),u(Un1),fac)
then permit
else deny fi .

eq auth(F, unPartners(c(Cn), a(An), u(Un), u(Un1)), u(Un2)) = if
     F has c(Cn) and F has isAsgmt(c(Cn), a(An))
and (F has isUserRole(c(Cn),u(Un2),fac) or F has isUserRole(c(Cn
    ),u(Un2),ta))
and F has isUserRole(c(Cn),u(Un), student) and F has isUserRole(
    c(Cn),u(Un1), student)
and F has isPartners(c(Cn), a(An), u(Un), u(Un1))
then permit
else deny fi .


endfm
```

System Module

```
mod SYSTEM is
```

```
pr PROGRAM .
pr POLICY .


vars St St1 : Status .
var B : Bool .
var Sh : SharingMode .
vars Rl Rl1 Rl2 Rl3 : Role .
var Act : Action .
var F : Facts .

--- natural numbers for users courses files assignments actions
vars Un Un1 Un2 Un3 Cn Cn1 Fn  An An1 Ac Ac1 Ac2 : Nat .
var Rest Rest1 Rest2 RestUn RestUn1 RestAc : Set{Nat} .

op node : Facts Action Subject -> Node .

op __ : Role Role -> Role [assoc comm] .


*** ---INITIAL STUFF---

--- a "minimal" initial set of facts and initail operation
op minimalFacts : -> Facts .
eq minimalFacts = courses .

op noop : -> SysAction .
eq nextF(F, noop, u(77)) = F .
eq auth(F, noop, u(77)) = permit .

--- so to model-check starting with minimal state use node(
    minimalFacts, noop, u(77))


--- a "generic" initial set of facts
op initialFacts : -> Facts .
eq initialFacts = minimalFacts isUserRole(c(1),u(1),student)
    isUserRole(c(1),u(2),student) isUserRole(c(1),u(3),ta)
isUserRole(c(1),u(4),fac) isUserRole(c(2),u(1),ta) isUserRole(c
    (2),u(5),student) isAsgmt(c(1), a(1)) isStatus(c(1), a(1),
    open)
isPartners(c(1), a(1), u(1), u(2)) isSubmitted(c(1), a(1), f(1),
    u(1), group,false) .



*** SYSTEM RULES ***
--- add user action
```

```
crl [--addUser--] : node(F, Act, u(Un)) => node(nextF(F, Act, u(
    Un)), addUser(c(Cn), u(Un1), Rl), u(Un2)) if
(Un2, RestUn) := (0,1,2,3,4,5) /\
(Cn, Rest1) := (1,2) /\ (Un1, Rest) := (1,2,3,4,5) /\ (Rl Rl1
    Rl2) := (student ta fac) /\
auth(nextF(F, Act, u(Un)), addUser(c(Cn), u(Un), Rl), u(Un2)) =
    permit .


--- remove user action
crl [--rmvUser--] : node(F, Act, u(Un)) => node(nextF(F, Act, u(
    Un)), rmvUser(c(Cn), u(Un1), Rl), u(Un2)) if
(Un2, RestUn) := (0,1,2,3,4,5) /\
(Cn, Rest1) := (1,2) /\ (Un1, Rest) := (1,2,3,4,5) /\ (Rl Rl1
    Rl3) := (student ta fac) /\
auth(nextF(F, Act, u(Un)), rmvUser(c(Cn), u(Un1), Rl), u(Un2)) =
     permit .


--- create assignment action
crl [--createAsgmt--] : node(F, Act, u(Un)) => node(nextF(F, Act
    , u(Un)), createAsgmt(c(Cn), a(An)), u(Un2)) if
(Un2, RestUn) := (0,1,2,3,4,5) /\
(Cn, Rest1) := (1,2) /\ (An, Rest) := (1,2) /\
auth(nextF(F, Act, u(Un)), createAsgmt(c(Cn), a(An)), u(Un2)) =
    permit .


--- change status action
crl [--changeStatus--] : node(F, Act, u(Un)) => node(nextF(F,
    Act, u(Un)), changeStatus(c(Cn), a(An), St1), u(Un2)) if
(Un2, RestUn) := (0,1,2,3,4,5) /\
(Cn, Rest1) := (1,2) /\ (An, Rest) := (1,2) /\ (St St1) := (new
    open closed late deleted) /\
auth(nextF(F, Act, u(Un)), changeStatus(c(Cn), a(An), St1), u(
    Un2)) = permit .


--- make partners action
crl [--makePartners--] : node(F, Act, u(Un)) => node(nextF(F,
    Act, u(Un)), makePartners(c(Cn), a(An), u(Un3), u(Un1)), u(
    Un2)) if
(Un2, RestUn) := (0,1,2,3,4,5) /\
(Cn, Rest1) := (1,2) /\ (An, Rest) := (1,2) /\ (Un3, Un1, Rest2)
     := (1,2,3,4,5) /\
auth(nextF(F, Act, u(Un)), makePartners(c(Cn), a(An), u(Un3), u(
    Un1)), u(Un2)) = permit .

--- separate partners action
```

```
crl [--unPartners--] : node(F, Act, u(Un)) => node(nextF(F, Act,
    u(Un)), unPartners(c(Cn), a(An), u(Un), u(Un1)), u(Un2)) if
(Un2, RestUn) := (0,1,2,3,4,5) /\
(Cn, Rest1) := (1,2) /\ (An, Rest) := (1,2) /\ (Un3, Un1, Rest2)
    := (1,2,3,4,5) /\
auth(nextF(F, Act, u(Un)), makePartners(c(Cn), a(An), u(Un3), u(
   Un1)), u(Un2)) = permit .

--- view submitted file action
crl [--viewFile--] : node(F, Act, u(Un)) => node(nextF(F, Act, u
   (Un)), view(f(Fn)), u(Un2)) if
(Un2, RestUn) := (0,1,2,3,4,5) /\
(Fn, Rest1) := (1,2) /\
auth(nextF(F, Act, u(Un)), view(f(Fn)), u(Un2)) = permit .


endm
```

Kripke Module

```
mod KRIPKE is

pr SYSTEM .


*** --- MODEL CHECKER ---

inc MODEL-CHECKER .


***associating system with KRS
subsort Node < State .
subsort Fact < Prop .

var Act Act1 : Action .
var F : Facts .
var Pr : Fact .
vars Un Un1 Cn : Nat .
var Sh : SharingMode .
var Bl : Bool .
vars Rl Rl1 : Role .
var Fn : File .


ops wbPer : Action Subject -> Prop .
ops wbDen : Action Subject -> Prop .
op isAct : Action Subject -> Prop .
```

```
eq node(F Pr, Act, u(Un)) |= Pr = true .
eq node(F , Act, u(Un)) |= Pr = false [owise] .


op isRoleEq : Role Role -> Prop .
eq node(F , Act, u(Un)) |= isRoleEq(student, student) = true .

---eq node(F , Act, u(Un)) |= isRoleEq(R1, R2) = true if R1 ==
    R2 .



ceq node(F , Act, u(Un)) |= isAct(Act1, u(Un1)) = true if Act ==
     Act1 and Un == Un1 .
eq node(F , Act, u(Un)) |= isAct(Act1, u(Un1)) = false [owise] .

ceq node(F, Act, u(Un)) |= wbPer(Act1,u(Un1)) = true if auth(
    nextF(F, Act, u(Un)),Act1,u(Un1)) = permit .
eq node(F, Act, u(Un)) |= wbPer(Act1,u(Un1)) = false [owise] .

--- THE MODEL CHECKER asks whether our property is True FOR ALL
    PATHS. If we want to find A path where it is true we have to
    negate the formula:
--- Example: red modelCheck(node(c(1), addUser(c(1), u(1), ta),
    u(0)), <> isUserRole(c(1),u(1),student)): the way we set up
    the system there WILL be a state with
--- isUserRole(c(1),u(1),student) eventually so the
    counterexample to the negation will be our "exists" path

endm
```

Batch Module

```
*** loading of modules

in SIGNATURES .
in POLICY .
in PROGRAM .
in SYSTEM .

load model-checker .
in KRIPKE .
```

Index