Validation of Policy Integration Using Alloy*

Manachai Toahchoodee and Indrakshi Ray

Department of Computer Science,
Colorado State University,
Fort Collins CO 80523-1873
{toahchoo, iray}@cs.colostate.edu

Abstract. Organizations typically have multiple security policies operating together in the same system. The integration of multiple policies might be needed to achieve the desired security requirements. Validating this integrated policy is a non-trivial process. This paper addresses the problem of composing, modeling and validating the security policies. We show how the various approaches for composing security policies can be modeled and verified using Alloy, a lightweight modeling system with automatic semantic analysis capability.

1 Introduction

Organizations typically enforce multiple policies to achieve security. For instance, each department in a hospital will have its own policy about disclosing the information of a patient. To get the information of a patient belonging to multiple departments, we need to combine the existing policies and check whether the requesters have enough permission to receive the information they need. Manually analyzing whether the integrated policy's behavior complies with the given requirement for a large-scale application is a tedious and error-prone process. Towards this end, we show how the process can be automated to some extent.

Our approach consists of developing a model of the system whose policies we are verifying. A model is an analyzable representation of a system. To be useful, it must be simpler than the system itself, but faithful to it. In our approach, we use the Alloy language [5, 6, 8, 10] to specify the model. The specification in Alloy can be automatically verified using the Alloy Analyzer. We show how the policy composition approaches proposed by Bonatti et al. [2] can be verified using Alloy.

The rest of this paper is organized as follows. Section 2 discusses some work on policy composition. Section 3 contains summary of the principle and the features of Alloy system. Section 4 describes algebra for composing access control policies and its representation in Alloy language. Examples are provided in section 5. Section 6 concludes the paper and gives some future directions.

2 Related Work

Several researchers have worked on the problem of policy composition. Hosmer [3] identified shortcomings in the unified security policy paradigm, such as inflexibility,

* This work was partially supported by AFOSR under Award No. FA9550-04-1-0102.

G. Chakraborty (Ed.): ICDCIT 2005, LNCS 3816, pp. 420–431, 2005.
© Springer-Verlag Berlin Heidelberg 2005

difficulty in exchanging data between systems having different policies, and poor performance. These shortcomings are eliminated in multipolicy paradigm proposed by the author. Bidan and Issarny [1] proposed a solution for reasoning about the coexistence of different security policies and how these policies can be combined. The authors also address issues pertaining to the completeness and the soundness of the combined security policy. Jajodia et al. [7] authors introduce the *Flexible Authorization Framework* (FAF) that allows users to specify policies in a flexible manner. The language allows the specification of both positive and negative authorizations and incorporates notions of authorization derivation, conflict resolution, and decision strategies. Such strategies can exploit the hierarchical structures in which system components are organized as well as any other relationship that the system security officer (SSO) may wish to exploit. Bonatti et al. [2] proposed an algebra for representing and composing access control policies. Complex policies are formulated as expressions of the algebra. Different component policies can be integrated while retaining their independence. This framework is flexible and keeps the composition process simple by organizing compound specifications into different levels of abstraction. Our work is based on this work. Siewe et al. [9] have developed a compositional framework for the specification of access control policies using specific language called ITL. Complex policies are created by composition using several operators. Multiple policies can be enforced through composition, and their properties reasoned about. The effect of the combined policy can be understood by using the simulator called Tempura. Zao et al. [10] and Schaad et al. [8] have investigated how Alloy can be used for verifying Role-Based Access Control (RBAC) policies. But none of these work address how to verify integrated policies.

3 Alloy Lightweight Modeling System

Alloy ([4], [5], [6], [10]), is a textual language developed at MIT by Daniel Jackson and his team. Unlike a programming language, an Alloy model is declarative. This allows very succinct and partial models to be constructed and analyzed. It is similar in spirit to the formal specification languages Z, VDM, Larch, B, OBJ, etc, but, unlike all of these, is amenable to fully automatic analysis in the style of a model checker.

Z was a major influence on Alloy. Unlike Z, Alloy is first order, which makes it analyzable but is also less expressive. Alloys composition mechanisms are designed to have the flexibility of Z's schema calculus, but are based on different idioms: extension by addition of fields, similar to inheritance in an object-oriented language, and reuse of formulas by explicit parameterization, similar to functions in a functional programming language. Alloy is a pure ASCII notation and does not require special typesetting tools.

The Alloy Analyzer's analysis is fully automatic, and when an assertion is found to be false, the Alloy Analyzer generates a counterexample. It's a "refuter" rather than a "prover". When a theorem prover fails to prove a theorem, it can be hard to tell what's gone wrong: whether the theorem is invalid, or whether the proof strategy failed. If the Alloy Analyzer finds no counterexample, the assertion may still be invalid. But by picking a large enough scope, you can usually make this very unlikely. The tool can generate instances of invariants, simulate the execution of operations (even those defined implicitly), and check user-specified properties of a model.

4 An Algebra for Composing Access Control Policies and Its Representation in Alloy

Bonatti et al. [2] propose an algebra for composing access control policies. In this section, we show how these policy expressions can be represented in Alloy.

4.1 Definitions Used in Bonatti's Work

We begin by giving some definitions used in Bonatti's work.

Definition 1. [Authorization Term] *Authorization terms are triples of the form* (s,o,a)*, where s is a constant in S or a variable over S, o is a constant in O or a variable over O, and a is a constant in A or a variable over A where S, O, and A represent the set of subjects, objects, and actions, respectively.*

At a semantic level, a policy is defined as a set of ground (i.e., variable-free) triples.

Definition 2. [Policy] *A policy is a set of ground authorization terms. The triples in a policy P state the accesses permitted by P.*

The algebra (among other operations) allows policies to be restricted (by posing constraints on their authorizations) and closed with respect to inference rules.

- An *authorization constraint language* L_{acon} and a semantic relation *satisfy* $\subseteq (S \times O \times A) \times L_{acon}$; the latter specifies for each ground authorization term (s,o,a) and constraint $c \in L_{acon}$ whether (s,o,a) satisfies c.
- A *rule language* L_{rule} and a semantic function *closure*: $\wp(L_{rule}) \times \wp(S \times O \times A) \rightarrow \wp(S \times O \times A)$; the latter specifies for each set of rules R and ground authorizations P which authorizations are derived from P by R.

These languages have been chosen with the goal of keeping the presentation as simple as possible, focusing attention on policy composition, rather than authorization properties and inference rules.

Compound policies can be obtained by combining policy identifiers through the algebra operators. Let the metavariables P_1 and P_2 range over policy expressions.

Addition/Union $(+)$**.** It merges two policies by returning their union. Formally, $P_1 + P_2 = P_1 \cup P_2$. Intuitively it means that if access is permitted by either of the policies, then the access will be allowed by the resulting composed policy.

Conjunction/Intersection $(\&)$**.** It merges two policies by returning their intersection. Formally, $P_1 \& P_2 = P_1 \cap P_2$. This means the access will be permitted only if both the component policies allow access.

Subtraction $(-)$**.** It restricts a policy by eliminating all the accesses in a second policy. The formal definition is $P_1 - P_2 = P_1 \setminus P_2$. The resulting policy permits access only if the access is allowed by P_1 and not by P_2.

Closure $(*)$**.** It closes a policy under a set of inference (derivation) rules. Formally, $P * R = closure(R,P)$. It basically signifies the set of policies that can be generated from the policy P given the derivation rule R.

Scoping restriction ($\hat{}$)**.** It restricts the application of a policy to a given set of subjects, objects, and actions. Formally, $P\hat{}c = \{(s,o,a)\theta \mid (s,o,a)\theta \in P, (s,o,a)\theta \text{ satisfy } c\theta\}$ where $c \in L_{acon}$ and θ is a ground substitution for variables s,o,a. Scoping is particularly useful to "limit" the statements that can be established by a policy and to enforce authority confinement. Intuitively, all authorizations in the policy that do not satisfy the scoping restriction are ignored, and therefore ineffective.

4.2 Representation of the Integrated Access Control Policy in Alloy

We apply the following rules in order to represent policies in Alloy:

Base Elements. Each base element in the access control policy (that is, the set of subjects, objects and actions) is represented by using signature (see example 1 for details).

Authorization Term. The authorization term is expressed using a special signature that comprises subject, object and action (see Example 1 for details).

Policy. Policy is described using a special signature which is composed of the set of authorization terms (see example 1 for details).

Closure. We define closure as a fact in Alloy. This feature ensures that all elements in our model must satisfy the predefined rules in the fact section. Fact content can be either attached to or separated from the content of the signature. If we attach fact to the signature, this means the fact is applied to the signature only (see example 2 and 7 for details).

Policy expressions. Other policy expressions, such as, scoping restriction, addition, conjunction and subtraction, are represented using predicates. We can think of a predicate as a function that will change the value of parameters (if any) according to the expressions stated in the predicate's context, and return true or false. Using predicates allows us to verify our integrated access control policy in the following ways:

- Model consistency check: We can check from the predicate whether there exists an input for our model or not by using run command in Alloy. If there is any input that satisfies the model, Alloy will show it. Otherwise it will report an error.
- Model correctness check: The correctness of the policy composition can also be validated by using test cases. The test cases are specified using the assert feature. In the assert part, we define the result we expect from the policy composition and use check command in Alloy to evaluate it. If the assertion does not hold, a counterexample is produced. To define the expected result in the assert part, we define the preconditions and the expected post conditions then concatenate them by the imply operator ($\Rightarrow$).

5 Example Scenario

We illustrate our approach by using an example application. Consider a hospital composed of three departments, namely, *Radiology*, *Surgery*, and *Medicine*. Each department is responsible for granting access to data under their authority domains, where domains are specified using scoping restrictions. In addition there are administrators who may or may not be a member of a department. These relationships are shown in Figure 1.

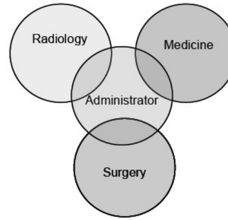

Fig. 1. Hospital policy relationship diagram

Example 1. We represent the set Subject, Object, and Action as signatures in Alloy. The authorization term is represented as a signature whose constituent elements are members of the set of subject, object and action, respectively. This specification given below shows how the authorization term and policies are represented in Alloy.

```
sig Subject {}
sig Radiology, Surgery, Medicine extends Subject {}
sig Administrator in Subject {}
sig Object
{
    owner: Subject
}
sig File, Form extends Object {}
sig Action {}
sig ReadOnly, Write, Execute extends Action {}
sig AuthorizationTerm
{
    subject: Subject,
    object: Object,
    action: Action
}
sig Policy
{
    auth: set AuthorizationTerm
}
```

Example 2. Every authorization term in the hospital policy must satisfy a set of ground rules. These ground rules will be used when computing the closure of some policy. The ground rules are as follows.

- The elements of set *Subject* must come from the union of the elements from the set *Radiology*, *Surgery*, *Medicine* and *Administrator* only.
- The elements of set *Object* must come from the union of the elements from the set *File* and *Form* only.
- The elements of set *Action* must come from the union of the elements from the set *ReadOnly*, *Write* and *Execute* only.

– Every department has their own *administrators*.
– There must be at least one member for each department.

The specification given below shows how these can be represented as facts in Alloy.

```
fact PolicyGroundRules
{
    // Specify the elements in the universe (Subject, Object, Action)
    Subject = Radiology + Surgery + Medicine + Administrator
    Object = File + Form
    Action = ReadOnly + Write + Execute
    // Every departments have administrators
    Radiology & Administrator != none
    Surgery & Administrator != none
    Medicine & Administrator != none
    // Each subset of subject must not empty
    Radiology != none
    Surgery != none
    Medicine != none
    Administrator != none
}
```

Example 3. To represent the scope restriction, we use Alloy's predicate feature to represent the scope operation. We can check whether the combination of each condition in the predicate can generate the output or not by using command run in the Alloy Analyzer. If there are any conflicts of conditions, the analyzer will send an error message. If the analyzer can find valid input for the predicate, it will show the graph of the input as in Figure 2.

After we have created the predicate, we can verify the correctness of our operation (in this case, the scoping restriction) by using the assert feature in Alloy. Assert will try to find the counterexample for our predicate and show us the counterexample graph if any counterexample exists as in figure 3.

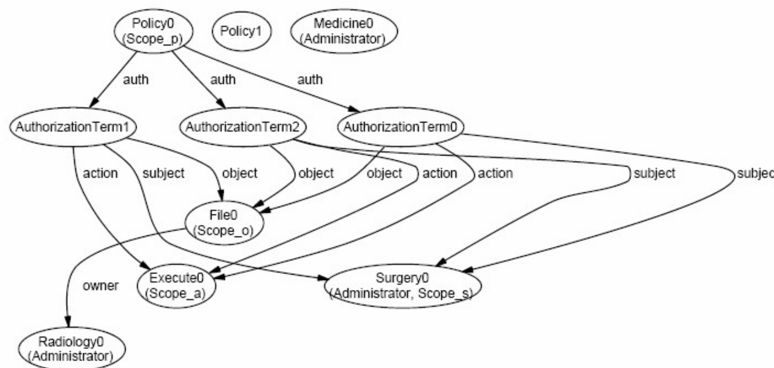

Fig. 2. Example of input for Scope

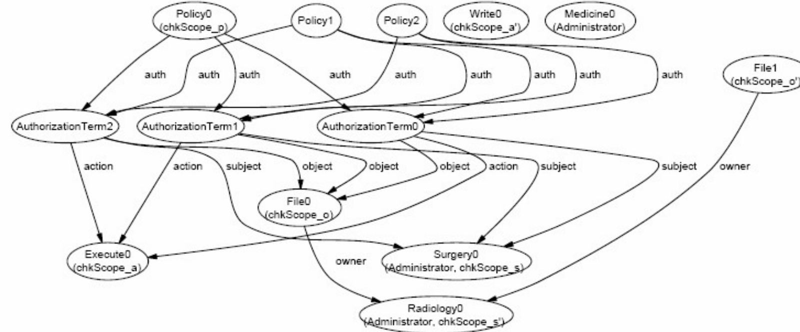

Fig. 3. Counterexample graph

In this example, we make the assumption that if we have two arbitrary sets of authorization terms and we do the scoping restriction based on the first authorization term, the result of the scoping must not equal to the authorization terms of the second set. Then we test our predicate based on this assumption by command check. Obviously, the analyzer could not find the counterexample. We can specify the number of testing objects that we want Alloy to generate for us by adding the parameter for after command run, check. The Alloy code for the predicate Scope and the corresponding assert command is shown below.

```
pred Scope (p: Policy, s: Subject, o: Object, a: Action)
{
    p.auth.subject = s
    p.auth.object = o
    p.auth.action = a
}
run Scope
// After scoping, the remaining authorization terms must satisfy
// the scope condition
assert chkScope
{
    all s, s’: Subject, o, o’: Object, a, a’: Action, p: Policy|
    ((s != s’) && (o != o’) && (a != a’) &&
    Scope(p, s, o, a)) =>
    ((p.auth.subject -> p.auth.object -> p.auth.action) !=
    (s’ -> o’ -> a’))
}
check chkScope
```

Example 4. To do the addition of two policies, we create the predicate called PolicyUnion. This predicate will accept three policies as input parameters, then it will union the first two policies together and store the result in the third parameter.

For instance, we would like to create the new access control policy from our existing policy. This new policy allows the access from both Radiology department and Surgery department. After the composition, we will verify our model to ensure that the result of composition must be the combination of the set of authorization terms from Radiology departments policy and the set of authorization terms from Surgery departments policy. To satisfy these requirements, our model and the verification command in Alloy will be as below.

```
    pred PolicyUnion (p1, p2, p3: Policy)
      {
p3.auth = p1.auth + p2.auth
    }
    run PolicyUnion
    assert chkPolicyUnion
    {
        all p1, p2: Policy| some p3: Policy|
        (Scope(p1, Radiology, Object, Action) &&
        Scope(p2, Surgery, Object, Action)) &&
        PolicyUnion(p1, p2, p3) =>
        ((Radiology in p3.auth.subject) &&
        (Surgery in p3.auth.subject) && (p3.auth != none))
    }
    check chkPolicyUnion
```

Example 5. To do the conjunction of policies, we create the predicate called Policy-Intersection. This predicate will accept three policies as input parameters, then it will intersect the first two policies together and store the result in the third parameter.

For example, we would like to create a new access control policy which allows only the administrators of the Radiology department to access the resource. The integrated policy is created from the intersection between policy of the Radiology department and the policy of the administrator. To verify the correctness of the model, we create the test that the result policy's authorization term is restricted to the Radiology department and the staff must be the administrator. The Alloy code to support the requirement is as follows.

```
    pred PolicyConjunction (p1, p2, p3: Policy)
    {
    p3.auth = p1.auth & p2.auth
    }
    run PolicyConjunction
    assert chkPolicyConj
    {
        all p1, p2: Policy| some p3: Policy|
        (Scope(p1, Administrator, Object, Action) &&
        Scope(p2, Radiology, Object, Action)) &&
        PolicyConjunction(p1, p2, p3) =>
```

```
    ((p3.auth.subject in Radiology) &&
    (p3.auth.subject in Administrator) &&
    (p3.auth != none))
}
check chkPolicyConj
```

Example 6. To do the subtraction of policies, we create the predicate called Policy-Subtraction. This predicate will accept three policies as input parameters, then it will subtract policy p2 from p1 and store the result in the third parameter (policy p3).

To demonstrate the idea, suppose we want to create a new policy which allows only the staff from Radiology department who is not the administrator to access the resource. To achieve this goal, we subtract the members who are the administrator of the Radiology department policy from the policy of the Radiology department itself. To verify the correctness of the integrated policy, we check if the result from subtraction is the set which members are from Radiology set but not from Administrator set. The Alloy code for this example will be as below.

```
pred PolicySubtraction (p1, p2, p3: Policy)
{
    p3.auth = p1.auth - p2.auth
}
run PolicySubtraction
assert chkPolicySubtraction
{
    all p1, p2: Policy| some p3: Policy|
    (Scope(p1, Radiology, Object, Action) &&
    Scope(p2, Administrator, Object, Action)) &&
    PolicySubtraction(p1, p2, p3) =>
    ((p3.auth.subject in Radiology) &&
    (p3.auth.subject not in (Administrator)) &&
    (p3.auth != none))
}
check chkPolicySubtraction
```

Example 7. In this example, we will show how to combine the different kinds of policy expressions together. Each department is responsible for granting access to data under their authority domains, where domains are specified by scoping restrictions. The statements made by the departments are then unioned, meaning the hospital considers an access as authorized if any of the department policies so states.

For privacy regulations, the hospital will not allow any access (even if authorized by the departments) to *lab_tests* data unless there is patient consent for that, stated by policy $P_{consents}$. In terms of the algebra, the hospital policy can be represented as

$$[(P_{rad} + P_{surg} + P_{med}) - (P_{rad} + P_{surg} + P_{med})^\wedge(\text{object} = \\ lab_tests)] + [P_{consents} \& (P_{rad} + P_{surg} + P_{med})^\wedge(\text{object} = lab_tests)]$$

In this case, we can view $P_{consents}$ as a policy which is closed by a set of rules (the permissions assigned by patient). In Alloy, we defined $P_{consents}$ as a special kind of

signature which inherit from the Policy signature and closed by the attached fact. To check the correctness of the model, we claim that there is no staff from the Medicine department that can access *lab_tests* data. The full Alloy model for the hospital policy can be shown as below

```
sig Subject {}
sig Radiology, Surgery, Medicine extends Subject {}
sig Administrator in Subject {}
sig Object
{
        owner: Subject
}
sig File, Form, LabTests extends Object {}
sig Action {}
sig ReadOnly, Write, Execute extends Action {}
sig AuthorizationTerm
{
    subject: Subject,
    object: Object,
    action: Action
}
sig Policy
{
    auth: set AuthorizationTerm
}
sig Consent extends Policy
{
}
{
// Allow only Radiology and Surgery Staff to access lab tests data
    auth.subject = Radiology + Surgery
    auth.object = LabTests
}
fact PolicyGroundRules
{
    Subject = Radiology + Surgery + Medicine + Administrator
    Object = File + Form
    Action = ReadOnly + Write + Execute
    Radiology & Administrator != none
    Surgery & Administrator != none
    Medicine & Administrator != none
    Radiology != none
    Surgery != none
    Medicine != none
    Administrator != none
}
```

```
pred Scope (p: Policy, s: Subject, o: Object, a: Action)
{
    p.auth.subject = s
    p.auth.object = o
    p.auth.action = a
}
// Policy Union
pred PolicyUnion (p1, p2, p3: Policy)
{
    p3.auth = p1.auth + p2.auth
}
pred HospitalPolicy (p1, p2, p3, p4: Policy)
{
    p4 = (p1 - p2) + (p3 & p2)
}
run HospitalPolicy
assert chkHospitalPolicy
{
    all Prad, Psurg, Pmed, Pradsurg, p1, p2: Policy,
    Pconsent: Consent|
    no p4: Policy|
    (Scope(Prad, Radiology, Object, Action) &&
    Scope(Psurg, Surgery, Object, Action) &&
    Scope(Pmed, Medicine, Object, Action)) &&
    PolicyUnion(Prad, Psurg, Pradsurg) &&
    PolicyUnion(Pradsurg, Pmed, p1) &&
    Scope(p2, p1.auth.subject, LabTests, p1.auth.action) &&
    HospitalPolicy(p1, p2, Pconsent, p4) =>
    ((Medicine->LabTests->Action in
    p4.auth.subject->p4.auth.object->p4.auth.action) &&
    (p4.auth != none))
}
check chkHospitalPolicy
```

6 Conclusion and Future Work

In this paper we have shown how the different policy composition operations can be represented in Alloy. Specifying the policies in Alloy allows for formal analysis most of which can be performed automatically using the Alloy Analyzer. A lot of work remains to be done. We assumed that policies can be defined using the set of authorization terms. Towards this end, we need to find the method to automatically transform the component policies to the set of authorization terms. Similar to the problem of software testing, we need a methodology to generate the good test cases that can help us to detect the flaws, if any, in our model.

References

1. Christophe Bidan and Valerie Issarny. Dealing with multi-policy security in large open distributed systems. In *ESORICS*, pages 51–66, 1998.
2. Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 5(1):1–35, February 2002.
3. Hilary H. Hosmer. The multipolicy paradigm for trusted systems. In *NSPW '92-93: Proceedings on the 1992-1993 workshop on New security paradigms*, pages 19–32, New York, NY, USA, 1993. ACM Press.
4. Daniel Jackson. Automating first-order relational logic. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139, New York, NY, USA, 2000. ACM Press.
5. Daniel Jackson. *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*. At http://sdg.lcs.mit.edu/alloy/referencemanual.pdf, 2002.
6. Daniel Jackson. *Alloy 3.0 reference manual*. At http://alloy.mit.edu/reference-manual.pdf, 2004.
7. Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.
8. Andreas Schaad and Jonathan D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 13–22, New York, NY, USA, 2002. ACM Press.
9. Franois Siewe, Antonio Cau, and Hussein Zedan. A compositional framework for access control policies enforcement. In *FMSE '03: Proceedings of the 2003 ACM workshop on Formal methods in security engineering*, pages 32–42, New York, NY, USA, 2003. ACM Press.
10. John Zao, Hoetech Wee, Jonathan Chu, and Daniel Jackson. *RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis*. At http://alloy.mit.edu/contributions/RBAC.pdf, 2002.