Access Control Policies: Modeling and Validation

Mahdi Mankai, Luigi Logrippo
Université du Québec en Outaouais
Gatineau (QC)

***Abstract*— Access control requires authorization rules and constraints. To express access control policies, several languages, such as XACML, EPAL or PONDER, are used. These languages specify which subjects can (or cannot) access sets of resources or services to perform specific actions. A user can define several access control policies and rules, but these access control languages do not offer any mechanism to avoid conflicts and inconsistencies among them. In fact, it can happen that more than a rule or a policy, with opposite decisions, is applicable in a given context.**

We propose a method based on first order logic modeling to detect and visualize possible conflicts within sets of access control policies expressed in XACML. We translate the model into a relational first order logic language called Alloy. Alloy allows to specify sets of predicates and assertions defining the desired properties of a system. We can then analyze interactions and conflicts among access control policies by using the Alloy Analyzer tool.

I. Introduction

Selective access control is an important mechanism in distributed system security. It can be used in order to allow a user to access only certain information, for certain purposes, at certain times, or when he or she plays a certain role. Access control is enforced by mechanisms that need to be programmed by means of policies. An organization may have many such policies, which may have been established at different times, by different people, perhaps without a clear view of all the consequences. Inconsistencies can then exist in such sets of policies. While policy mechanisms may be able to solve inconsistencies at a higher level, users and administrators still need to be aware of them, because they may lead to unintended system behavior. For example, a policy may be added to prevent a certain access, however in fact the access is still allowed because of another policy of higher priority, or the new policy may prevent access of someone who should remain authorized. We will show in this paper that such inconsistencies can be detected by using logic model checking tools.

We demonstrate our method with application to the standard policy language XACML, (eXtensible Access Control Markup Language) [13] which is an XML-based language for specifying access control in distributed systems. We have chosen XACML because it has already achieved a considerable degree of industrial acceptance. XACML has also been proven to be adaptable to specify several common access control methods, such as Role-Based Access Control (RBAC) [3]. The logic model checking tool that we use is Alloy [7] [8] [9], again a system that is gaining considerable acceptance and which we have found to be adequate to the task.

Appeared in : K. Adi, D. Amyot, L. Logrippo-Proceedings of the 5th NOTERE Conference, Gatineau, Canada, August 2005. p 85-91

II. XACML overview

A. The XACML framework

XACML is an OASIS standard that defines an architecture, policies and messages within an access control system. Figure 1 shows the XACML model that contains two main entities: the *Policy Enforcement Point* (*PEP*) and the *Policy Decision Point* (*PDP*). The PEP is the entity that protects the resources. It receives the access request and forwards it to the PDP. The PDP makes a decision according to the information contained in the request. In an XACML context, each request defines a subject, a resource and an action which are characterized by a set of valued attributes that express their properties.

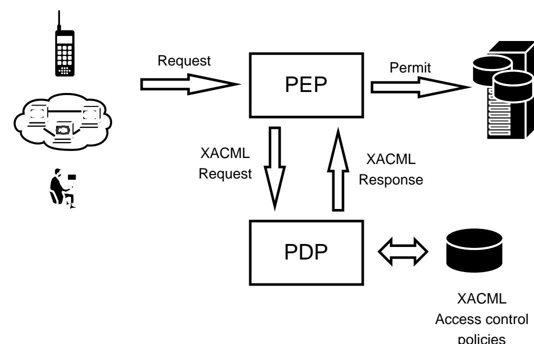

Fig. 1. The XACML framework

B. XACML policies structure

The access control policies are stored as XML [15] files, they are structured into rules, policies and policy sets. Several rules are grouped into policies and policies are grouped into policy sets. Rules, policies and policy sets define a *target* which indicates their domain of applicability. A target specifies a set of attributes and their values that should match those given by a request. Let's consider two policies. The first is applied when the subject is a student, the resource is the marks file and the action is printing. The second policy is applied when the subject is a professor, the resource is the marks file and the action is modification. If a request from a student to print the marks file comes to the PEP, the PDP will only select the first policy to make a decision.

Policies and policy sets are structural elements, they don't inform about possible decisions. On the other hand, a rule specifies an effect which could be *permit* or *deny*. Then,

if a rule is applied (its target is satisfied), the PDP will return its effect to the PEP, otherwise the returned response is *NotApplicable*. Rules could also specify additional constraints: *conditions*. A condition is a constraint that uses more complex functions over attributes. Conditions can add considerable complexity and have been ignored in this study.

C. Conflicts Resolution

In a distributed system, a PDP can use several policy databases. Then, several rules, policies or policy sets can be applied to a specific single request so, different decisions can be obtained. XACML provide a mechanism, called *Combining Algorithms*, to resolve such conflicts. A Combining Algorithm is a precedence rule that tells which unique and final decision the system should return. Each policy and policy set will have its own specified combining algorithm.

Four standard combining algorithms are defined by XACML:

- deny-overrides: if a rule or a policy evaluates to *deny*, the final decision will be deny
- permit-overrides: if a rule or a policy evaluates to *permit*, the final decision will be permit
- first-applicable: the final decision is the one provided by the first rule or policy in the policy file (an XML file) that applies to a request context
- only-one-applicable: if more than one rule or policy apply, no decision will be returned by the involved policy or policy set (*indeterminate* response).

III. A Logical Model of XACML

We propose in this section a logical model of XACML access control policies. We will consider the elements of the language as sets related by functions and relations. We first focus on XACML structures then we will consider access control constraints. Only a subset of XACML will be covered by this paper for purposes of simplicity.

A. Modeling structures

1) Attributes and Values: Let *Values* be a set of values and *Attribute* a set of attributes. Each attribute can have a set of possible values. Thus, we define a binary relation $values : Attribute \rightarrow Value$ which maps each attribute to its possible values.

2) Subjects, resources and actions: First, let *Element* be a set of elements which could be subjects, resources or actions. Each element is defined by a set of couples *(Attribute, Value)* that denote the element attribute and its actual value. So, we define a ternary relation $attributes : Element \rightarrow Attribute \rightarrow Value$ which maps each element to its attributes and their values.

As an example, consider an attribute called *role*, its possible values can be *professor* and *student*. If Bob is student, the *attributes* relation will include the tuple $(Bob, role, student)$.

Then, we can consider the sets *Subject*, *Resource* and *Action* that are partitions of the set *Element*. In this way, an element could be only a subject, a resource or an action.

3) Requests: An access control request contains information about the subject, the resource to be accessed and the action to be performed. We define *Request* as a set of requests in XACML. Each request defines one subject, one resource and one action. So, we define the following functions:

- $subject : Request \rightarrow Subject$
  defines the subject of a request
- $resource : Request \rightarrow Resource$
  defines the resource of a request
- $action : Request \rightarrow Action$
  defines the action of a request

4) Targets: Let *Target* be a set of targets. Each target is defined by a set of subjects, resources and actions specifying a domain of application. So, we define the following relations:

- $subjects : Target \rightarrow Subject$
  maps a target to its subjects
- $resources : Target \rightarrow Resource$
  maps a target to its resources
- $actions : Target \rightarrow Action$
  maps a target to its actions

5) Effects: *Effect* is a set of effects. The possible values of *Effect* are: *Permit*, *Deny*, *NotApplicable* and *Indeterminate*. *Indeterminate* effect is ignored for purposes of analysis.

6) Rules: *Rule* is a set of XACML access control rules. Each rule is defined by an effect, a target and a condition. So, we define the following functions:

- $ruleTarget : Rule \rightarrow Target$
  defines the target of a rule
- $effect : Rule \rightarrow Effect$
  defines the effect of a rule

As mentioned, we will not consider conditions in the present paper.

7) Combining Algorithms: *CombiningAlgo* is a set of XACML combining algorithms. This set contains the standard combining algorithms mentioned in subsection II-C.

8) Policies: *Policy* is a set of XACML access control policies. Each policy is defined by a target, a set of rules and a combining algorithm. So, we define the following functions and relations:

- $policyTarget : Policy \rightarrow Target$
  a function that defines the target of a policy
- $rules : Policy \rightarrow Rule$
  a relation that maps each policy to its child rules
- $ruleCombiningAlgo : Policy \rightarrow CombiningAlgo$
  a function that defines for a policy its rule combining algorithm

9) Policy sets: *PolicySet* is a set of policy sets. Each policy set is defined by a target, a set of policies and a combining algorithm. So, we define the following functions and relations:

- $policySetTarget : PolicySet \rightarrow Target$
  a function that defines the target of a policy set
- $policies : PolicySet \rightarrow Policy$
  a relation that maps each policy set to its child policies
- $policyCombiningAlgo : PolicySet \rightarrow CombiningAlgo$
  a function that maps each policy to its policy combining algorithm

In XACML, policy sets could be grouped into parent policy sets. We have not considered this aspect in this article in order to simplify the model.

B. Access Control Constraints

In order to express access authorization constraints, XACML offers two mechanisms: targets and conditions. The PDP performs some tests and comparisons to check if attributes of requests obey the constraints imposed by targets and conditions. These constraints are included in our model as logical functions and predicates. We describe in the next paragraphs the evaluation of targets against requests and the response of rules and policies.

1) Target Verification: Targets define a set of subjects, resources and actions. In order to apply a rule, a policy or a policy set, the subject, resource and action of a request should match respectively at least one subject, one resource, and one action of a target.

Saying that a request element matches the target element means that all target attributes should match those specified by the request.

As an example let P be an access control policy. Its target is defined by:

- Subjects:
  - Subject:
    - role = Professor
  - Subject:
    - name = Bob
    - role = Student
- Resources:
  - Resource:
    - resource name = Course marks file
- Actions:
  - Action:
    - action name = Modify
    - action name = Read

The policy P is applied to the following requests:

- Request 1:
  - Subject:
    - name = John
    - role = Professor
  - Resource:
    - resource name = Course marks file
  - Action:
    - action name = Modify
- Request 2:
  - Subject:
    - name = Bob
    - role = Student
  - Resource:
    - resource name = Course marks file
  - Action:
    - action name = Modify

But P is not applied to the following requests:

- Request 3:
  - Subject:
    - name = John
    - role = Student
  - Resource:
    - resource name = Course marks file
  - Action:
    - action name = Modify
- Request 4:
  - Subject:
    - name = Bob
    - role = Student
  - Resource:
    - resource name = Course marks file
  - Action:
    - action name = Print

P is not applied to Request 3 since the subject attributes don't match those of the target. In fact, to apply policy P, a subject should be professor or a student named Bob. Request 3 specifies a student named John.

P is not applied to Request 4 because the requested action is *Print* which is not specified in the policy's target.

In the following paragraphs we assume that *targetMatch* is a logical function that checks if a target matches a request.

2) Rule Response: The response on an XACML access control rule depends on its target and its specified effect. If a request matches a rule's target, the response of the rule will be the rule's effect, otherwise the rule's response will be *NotApplicable*.

3) Policy Response: The response of an XACML access control policy depends on its target, its child rules and its rule combining algorithm. As for a rule, the policy's target is evaluated against a request to determine whether the policy is applied or not. If it is applied, we will consider the responses of its child rules that apply. A unique decision will be returned according to the combining algorithm of the policy as mentioned in subsection II-C.

4) Policy set Response: The response of an XACML access control policy set depends on its target, its child policies and its policy combining algorithm in a similar way as for policies.

IV. Modeling with Alloy

Alloy is a modeling language based on relational first order logic. The *Alloy Analyzer* [10] is a tool that permits to verify and analyze Alloy models. Alloy is a structural and declarative language. It is suitable to describe complex structures and logical constraints. In this section, we provide a brief overview of Alloy. Then, we provide a translation of the XACML logical model into Alloy notation.

A. Alloy Overview

To model structures, Alloy uses the concepts of signature and relation. A signature is a type in Alloy. It can be considered equivalent to a class in the object oriented paradigm since a signature can be instantiated.

A relation is a structure that relates signatures and their instances. Functions are special binary relations: they map each instance from the left signature to only one instance from the right signature e.g. the function *effect* maps each rule to only one effect.

Constraints are represented in Alloy by facts. A fact is a logical formula that always holds. Alloy uses first order logic in an ASCII format. We can also specify predicates that could be evaluated to return true or false and functions that could return signature instances. Alloy is able to automatically instantiate and evaluate predicates.

We can also define assertions in Alloy. An assertion is a logical formula representing a system property. Assertions can be analyzed to check if they hold or not. The Alloy Analyzer will show a counterexample if an assertion is inconsistent. A counterexample is a set of instances that respects the constraints defined by the model and doesn't respect the analyzed assertion.

B. XACML Alloy Model

In section III we have proposed a logical model for XACML. To obtain the Alloy model we need to translate the logical notation into Alloy. Every set is defined by a signature. These signatures are related by relations and functions. The constraints in XACML are translated into Alloy's facts and functions. We present some examples in the following paragraphs.

1) XACML Policies:

```
abstract sig Policy {
  policyTarget : one Target,
  rules : set Rule,
  ruleCombiningAlgo : one CombiningAlgo}
```

We declare a signature *Policy* for the set of policies. This signature is declared as abstract to force Alloy not to generate random policies automatically, but let us define each specific policy. The function *policyTarget* is represented as a relation that relates each policy to only one target. The relation *rules* maps each policy to a set of rules. The function *ruleCombiningAlgo* relates each policy to one combining algorithm.

2) Subjects, Resources and Actions:

```
abstract sig Element {
  attributes : Attribute -> Value}
  {attributes in values}
sig Subject, Resource, Action
    extends Element{}
```

Since subjects, resources and actions have the same structure, we define first an abstract general signature *Element*. Then, we define its subtypes that are *Subject*, *Resource* and *Action*.

After the *Element* signature definition we add a fact (between the brackets) that forces the couple *(Attribute,Value)* to be significant. For example, we consider two subject's attributes: *name* and *role*. The name attribute has two possible values *Bob* and *John*. The role attribute has two possible values *professor* and *student*. So, the attribute signature has two instances (*name* and *role*) and the value signature has four instances (*Bob*, *John*, *professor* and *student*). The relation *values* is a relation that defines for each attribute its possible values (as mentioned in paragraph III-A.1). In this case it includes the following couples:

- (*name*, *Bob*)
- (*name*, *John*)
- (*role*, *professor*)
- (*role*, *student*)

But the relation *attributes* maps each element to a set of couples of type *(Attribute,Value)*. So, it can map a subject to the couple (name, professor) that has no meaning since relation *attributes* is supposed to define the subject's attributes and their actual values. That's why we add the fact that says `attributes in values` meaning that the couples (Attribute,Value) defined by the relation *attributes* should be included in the couples above defined by the relation *values*.

3) Target Verification: The *targetMatch* function is defined by a predicate that returns *true* if some target's subject, resource and action in the policy, rule or policy set match a request's subject, resource and action.

```
pred targetMatch (t:Target,r:Request) {
  some s: t.subjects |
       elementMatch(s,r.subject)
  some s: t.resources |
       elementMatch(s,r.resource)
  some s: t.actions |
       elementMatch(s,r.action)}
```

4) Rule Response: The function *ruleResponse* will be translated into Alloy. It will return the effect returned by the function *ruleEffect*, if the rule's target matches a request, otherwise it will return the *NotApplicable* effect.

```
fun ruleResponse (r:Rule,req:Request)
    :Effect{
  if   targetMatch(r.ruleTarget, req)
  then r.ruleEffect
  else NotApplicable}
```

V. Access Control Verification and Validation

In this section we analyze actual XACML access control policies using the model defined above and the Alloy Analyzer.

A. An Example of access control policy

We consider a plain access control policy consisting of three rules. We assume that we control access to a file of course marks:

1) a professor can read or modify the file of course marks
2) a student can read the file of course marks
3) a student cannot modify the file of course marks

B. A Complete Model

The model described in section IV-A is an abstract model. It defines only abstract entities. We need to specify actual and specific policies, rules, targets, attributes and values. A subject is defined by an attribute that could be called *Role*.

The role attribute could have *Student* and *Professor* values. A resource is defined by the attribute *ResourceName* that could be *MarksFile*. Finally an action is defined by the attribute *ActionName* that could have value *Read* or *Modify*. So, the *values* relation will be defined by the following couples:

- (*Role*, *Student*)
- (*Role*, *Professor*)
- (*ResourceName*, *MarksFile*)
- (*ActionName*, *Read*)
- (*ActionName*, *Modify*)

We define four targets T, T1, T2 and T3, one policy P, and three rules R1, R2 and R3. The relation *policyTarget* is defined by the couple {(P,T)} that indicates that the target of P is T (recall that we need a target for the whole policy, although it is not used by any rule). The relation *ruleTarget* is defined by the tuples {(R1,T1), (R2,T2), (R3,T3)} that indicates that the targets of R1, R2 and R3 are respectively T1, T2 and T3.

We define three subjects SUB1, SUB2 and SUB3, one resource RES and three actions ACT1, ACT2 and ACT3. The relation attributes is defined by the tuples:

- (SUB1, *Role*, *Professor*) to indicate that SUB1 is a professor
- (SUB2, *Role*, *Student*) to indicate that SUB2 is a student
- (SUB3, *Role*, *Professor*) and (SUB3,*Role*,*Student*) to indicate that SUB3 could be a professor or student
- (RES1, *ResourceName*, *MarksFile*) to indicate that RES1 is a marks file
- (ACT1, *Action*, *Read*) to indicate that ACT1 is a read action
- (ACT2, *Action*, *Modify*) to indicate that ACT2 is a modify action
- (ACT3, *Action*, *Read*) and (ACT3, *Action*, *Modify*) to indicate that ACT3 could be a read or modify action

The *subjects* relation is defined by the couples:

- (T, SUB3) to indicate that the subject of T is SUB3
- (T1, SUB1) to indicate that the subject of T1 is SUB1
- (T2, SUB2) to indicate that the subject of T2 is SUB2
- (T3, SUB2) to indicate that the subject of T3 is SUB2

Similarly, the *resources* relation is defined by the tuples (T, RES1), (T1, RES1), (T2, RES1) and (T3, RES1) and the relation *actions* by tuples (T, ACT3), (T1, ACT3), (T2, ACT1) and (T3, ACT2).

R1 and R2 will result in permit and R3 will result in deny. Then, the relation *effect* will include the couples :

- (R1, *Permit)*
- (R2, *Permit)*
- (R3, *Deny)*

C. Model Analysis

The Alloy model of the access control policies can be analyzed, by using a set of predicates and assertions that represent the properties to be verified.

First we can ask Alloy to show an example in which a rule generates a Permit response. For this purpose, we define the following predicate in Alloy:

```
pred PermitRule(r : Request, r1 : Rule){
  ruleResponse(r1,r) = Permit}
```

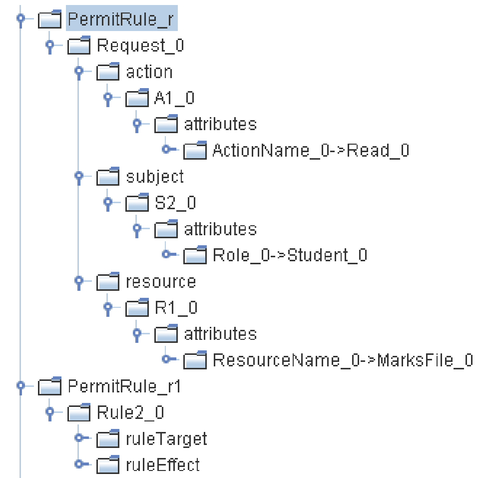

Fig. 2. A Permit Response

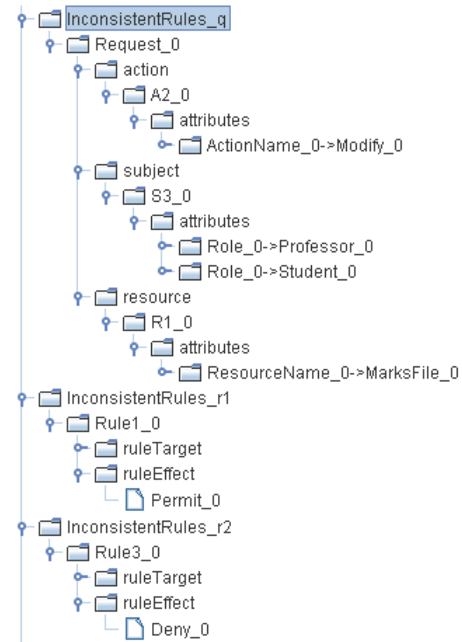

Fig. 3. Inconsistent Rules

Alloy will find the example shown in figure 2. In this example Alloy found a request and rule instances (the two parameters of the predicate) for which *PermitPolicy* is evaluated to true. Then, when a student asks to read the file of course marks, the access is granted by rule 2.

We can also check if there exist two rules that return two different decisions (permit and deny) in a context of a specific request. So, we define the following predicate in Alloy:

```
pred InconsistentRules
     (q : Request, r1, r2 : Rule){
  ruleResponse(r1,q) = Permit
  and ruleResponse(r2,q) = Deny}
```

Alloy returns the result shown in figure 3. In this figure, we can see that a request to modify the file of course marks processed

by a subject that has both student and professor role leads to two possible decisions generated by *Rule1* and *Rule3*. The user then knows that the access model must be corrected in some way.

We can finally verify that a professor's request to modify the marks file will be always accepted by the following assertion:

```
assert PermitForProfessor {
  all q : Request {
    {~(q.subject.attributes).Attribute =
    Professor}
    => policyResponse(P,q) = Permit}}
```

For this example Alloy didn't find a counterexample for this assertion. This means that this property holds in the specific context of this example.

VI. Related Work

In [4], Multi-Terminal Binary Decision Diagrams (MTBDDs) were used to represent access control policies and analyze conflicts and change impact among them. [11] [12] have used other languages such as PONDER [2] to specify policies and have developed their own conflict detection tools [1] to analyze and verify conflicts. [16] has demonstrated a different approach: instead of checking a posteriori a given set of policies, it is shown how access control policies can be specified in a logic-based language [6], checked and then translated into XACML. Surely, a similar thing could be done with Alloy: first, policies could be specified and checked in Alloy and then translated into XACML. Or perhaps more pragmatically one can expect that policies will be specified by using GUIs and then checked at that stage before being translated into XACML or other languages. It remains to be seen how each method can fit into business models.

VII. Conclusion

We have shown how XACML policies can be formalized in logic and how inconsistencies in them can be detected by using a standard logic model checker. Of course, this detection process has a high computational complexity and it cannot be guaranteed that it can always complete in reasonable time. However all the examples of XACML policies that we have encountered so far (some taken from real-life examples) were not complex and the result of the analysis was available in an acceptable time (less than one minute). In principle, elements of our method can be used to detect inconsistencies in other policy languages such as EPAL [14] or PONDER, however the adaptation that must be done in each case in not negligible.

Part of this project are also a translator from XACML into Alloy, a translator from XACML into natural language, and an analyzer that scans Alloy outputs such as the one of figure 3 to yield natural language explanation.

A question that comes up is, how far can we go in this process before we encounter undecidability problems, or at least problems that will be intractable with common model checkers and theorem provers. Indeed, XACML allows the use of functions that can be like ordinary programs, and we have not taken into consideration such functions in our work. In this case, methods such as ours can still be useful to detect possible inconsistencies, which will be reported to users who will be able to do some manual inspections, or run tests. In this latter case, an automatic tool can still be useful in order to suggest which tests should be run, as has been shown, in a different context [5].

VIII. Acknowledgment

This research was supported in part by a grant from the National Sciences and Engineering Research Council of Canada. We are grateful to Ann Anderson of Sun Microsystems and the NOTERE reviewers for useful comments.

References

[1] N. Damianou, N. Dulay, E. Lupu, M. Sloman, and T. Tonouchi, "Tools for Domain-based Policy Management of Distributed Systems," in *IEEE/IFIP Network Operations and Management Symposium (NOMS2002)*, Apr. 2002, pp. 213–218.

[2] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," *Lecture Notes in Computer Science*, vol. 1995, p. 18, 2001. [Online]. Available: citeseer.ist.psu.edu/damianou01ponder.html

[3] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM Trans. Inf. Syst. Secur.*, vol. 4, no. 3, pp. 224–274, 2001.

[4] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and Change-Impact Analysis of Access-Control Policies," in *International Conference on Software Engineering*. ACM, 2005.

[5] N. Gorse, L. Logrippo, and J. Sincennes, "Formal Detection of Feature Interactions with Logic Programming and LOTOS," *Journal on Software and System Modelling*, 2005, (to appear).

[6] D. P. Guelev, M. Ryan, and P. Y. Schobbens, "Model-Checking Access Control Policies," in *Seventh Information Security Conference (ISC 2004). Lecture Notes in Computer Science*. Springer-Verlag, Sept. 2004, pp. 219–230.

[7] D. Jackson, "ALLOY Home Page." [Online]. Available: http://alloy.mit.edu/

[8] ——, *Micromodels of Software: Lightweight Modelling and Analysis with ALLOY*, Feb. 2002.

[9] ——, *ALLOY 3.0 Reference Manual*, May 2004.

[10] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: the alloy constraint analyzer," in *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. ACM Press, 2000, pp. 730–733.

[11] E. Lupu and M. Sloman, "Conflict Analysis for Management Policies," in *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network management IM'97, San Diego, CA*, 1997. [Online]. Available: citeseer.ist.psu.edu/lupu97conflict.html

[12] E. C. Lupu and M. Sloman, "Conflicts in Policy-Based Distributed Systems Management," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 852–869, Nov. 1999. [Online]. Available: citeseer.ist.psu.edu/lupu99conflicts.html

[13] OASIS, "eXtensible Access Control Markup Language (XACML)." [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

[14] M. Schunter and C. Powers, "The Enterprise Privacy Authorization Language (EPAL 1.1)," 2003. [Online]. Available: http://www.zurich.ibm.com/security/enterprise-privacy/epal/

[15] W3C, "eXtensible Markup Language (XML)." [Online]. Available: http://www.w3.org/XML/

[16] N. Zhang, M. Ryan, and D. P. Guelev, "Synthesising verified access control systems in XACML," in *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*. ACM Press, 2004, pp. 56–65.