Model Based Analysis and Validation of Access Control Policies

J. W. Bryans, J. S. Fitzgerald and P. Periorellis.

TECHNICAL REPORT SERIES

No. CS-TR-976 July, 2006

TECHNICAL REPORT SERIES

No. CS-TR-976 **July, 2006**

Model Based Analysis and Validation of Access Control Policies

Jeremy W. Bryans and John S. Fitzgerald and Panos Periorellis

Abstract

We present a model based approach to describing, analysing and validating access control policies. Access control policies are described using VDM - a model oriented formal method. Policy descriptions are concise and may be easily manipulated. The structure of the VDM description is derived from the OASIS standard access control policy language XACML. It is therefore straightforward to translate between XACML policies and their corresponding VDM models. We show how the existing tool support for VDM enables a number of ways of validating these policies, each of which are valuable at different stages of the development and maintenance life cycle.

Bibliographical details

BRYANS, J. W., FITZGERALD, J. S., PERIORELLIS, P..

Model Based Analysis and Validation of Access Control Policies
[By] J. W. Bryans, J. S. Fitzgerald and P. Periorellis.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-976)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-976

Abstract

We present a model based approach to describing, analysing and validating access control policies. Access control policies are described using VDM - a model oriented formal method. Policy descriptions are concise and may be easily manipulated. The structure of the VDM description is derived from the OASIS standard access control policy language XACML. It is therefore straightforward to translate between XACML policies and their corresponding VDM models. We show how the existing tool support for VDM enables a number of ways of validating these policies, each of which are valuable at different stages of the development and maintenance life cycle.

About the author

Jeremy has been a post doctoral research fellow at the School of Computing Science in Newcastle since December 2002. His background is in Theoretical Computer Science, and he has held posts in Stirling and Canterbury. His work in Newcastle is involved with the security of computer-based systems, and he is employed on th DIRC and GOLD projects.

John Fitzgerald is a specialist in the engineering of dependable computing systems, particularly in rigorous analysis and design tools. He returned to the University in 2003, having established design and validation activities at Transitive, a successful new SME in the embedded processor market. John took his BSc (Computing & Information Systems) and his PhD (in modularity and formal proof) at the University of Manchester, before joining the newly-formed BAESYSTEMS Dependable Computing Systems Centre at Newcastle, where he developed specification and design techniques for real-time avionic systems. He went on to study the potential for industrial application of formal modelling (specifically, VDM and its support tools) as an EPSRC Fellow and later as a Lecturer at Newcastle. He has led several EU and industry-supported research projects in the communications, aircraft, software and power sectors. John has worked as a visiting scholar at institutes in Denmark, Japan and Poland and serves on several international program committees. He is Chairman of Formal Methods Europe, the main European body bringing together researchers and practitioners in rigorous methods of systems development. He is a member of the BCS, the ACM and the IEEE Computer Society. He is a member the Committee of the BCS Special Interest group on Formal Aspects of Computing (BCS-FACS).

Panos Periorellis joined the department in June 2000 as a research associate after successfully completing his Ph.D. in the area of Enterprise Modelling under the supervision of Prof. John Dobson. Panos was promoted to Senior research Associate in March 2004 and started work on the GOLD project looking into issues of trust, privacy and security.

Suggested keywords

ACCESS CONTROL,
MODEL BASED SPECIFICATION,
TESTING, VALIDATION

Model Based Analysis and Validation of Access Control Policies

Jeremy W. Bryans, John S. Fitzgerald and Panos Periorellis

School of Computing Science
Newcastle University
NE1 7RU, UK
{Jeremy.Bryans, John.Fitzgerald, Panayiotis.Periorellis}@newcastle.ac.uk

Abstract. We present a model based approach to describing, analysing and validating access control policies. Access control policies are described using VDM – a model oriented formal method. Policy descriptions are concise and may be easily manipulated. The structure of the VDM description is derived from the OASIS standard access control policy language XACML. It is therefore straightforward to translate between XACML policies and their corresponding VDM models. We show how the existing tool support for VDM enables a number of ways of validating these policies, each of which are valuable at different stages of the development and maintenance life cycle.

Keywords: Access Control, Model Based Specification, Testing, Validation.

1 Introduction

As the information technology infrastructure of an organisation increases in size and complexity, the access control policies it needs will grow correspondingly more complex. In a service-oriented environment, organisations can quickly come together to work in *virtual organisations* (VOs), with all the resource and information sharing that that entails. The structure of these VOs may be constantly changing, as companies join to provide necessary new skills, or members complete their part of a task and withdraw from the VO. The access control policy of the VO will be a composite of relevant parts of the policies from the individual organisations. It will therefore also be continuously evolving, as new members provide new resources and leaving members remove resources. We present here initial progress towards developing tool support for building and maintaining these access control policies.

XACML [16] has recently emerged as a *de facto* standard access control policy language in service-oriented environments. The benefits of XACML are considerable: it is very flexible, and policies may refer to other policies, which may be in remote locations. XACML therefore facilitates the development of larger policies from distributed component policies. Although the single standard language

XACML offers makes the combining of these policies more straightforward, the developer is still faced with a difficult task. This is particularly true when working in the context of a VO. Parts of the overall access control policy may be developed and maintained by each partner organisation, and it may be that no one person is solely responsible for the resultant policy. Updates to policies need to be handled carefully, as policies may refer to each other. A developer needs to have confidence in a policy before it is deployed. Validating policies to build this confidence can be done in various ways. They can be tested against individual access requests or scenarios; they can be checked for internal consistency or coherence; or they can be checked against some conditions imposed on the access control policy, such as for example legal obligations.

What is required is a formal model to support the full range of analysis techniques. To be most effective strong tool support must also be provided. At present research into tools supporting such pre deployment analysis is at an early stage. In this paper we present some of our preliminary results on automated analysis and validation of access control policies. Some promising complementary approaches are considered in Section 5.

We translate policies into the formal modelling language VDM-SL – the language of VDM. We show how we can test policies against individual requests, compare policies with each other and check the internal consistency of policies. The structure of the XACML is preserved in the translation, and therefore a faulty policy can be fixed within the VDM framework and translated back to XACML.

In Section 2 we give an overview of XACML. Section 3 gives a VDM-SL description of the data types and algorithms common to all access control policies. Section 4 shows how these data types may be instantiated to describe a specific policy and how the VDMTools framework may be used to test the instantiated policies, and compare them with each other. We also consider formalising the requirements on the example policy, describe our progress towards automatically extracting VDM-SL specifications from XACML documents. Section 5 describes related work and makes some comparisons. Section 6 draws some conclusions and describes several promising avenues for further work. A preliminary version of this work appeared as [3].

2 XACML and access control

XACML [16] is the OASIS standard for access control policies. It is written in XML and provides two languages: a *policy language* for describing access control policies, and a *request language* for interrogating these policies, to ask the policy if a given action should be allowed.

A simplified description of the way XACML policies work is as follows. An XACML policy has an associated *Policy Enforcement Point* (PEP) and a *Policy Decision Point* (PDP) (See Figure 1.) The PDP can be thought of as holding the policy. The PEP intercepts access requests from users. It translates them into the XACML request language and sends the XACML request to the PDP.

The PDP evaluates the request with respect to the access control policy it holds and sends a response to the PEP. The request is then *enforced* (permitted or denied) by the PEP. In this way the combined effect of the PEP and the PEP is to act as a sort of *execution monitor* for the system, permitting only allowed actions.

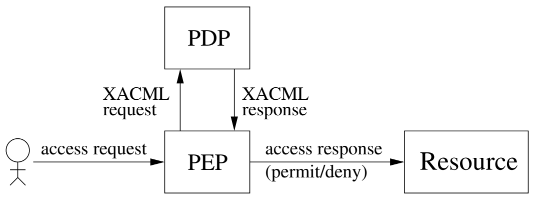

Fig. 1. XACML overview.

We will assume that a PDP contains a set of *Policies*[1], each of which contain a set of *Rules*[2]. Rules in XACML contain a *target* (which further contains sets of resources, subjects and actions) and an *effect* (permit or deny). If the target of a rule matches a request, the rule will return its *effect*. If the target does not match the rule *not applicable* is returned.

As well as a set of rules, a policy contains a *rule combining algorithm* and a target. All rules within a policy are evaluated by the PDP. The results are then combined using the rule combining algorithm,and a single effect is returned.

The PDP evaluates each policy and combines the results using a *policy combining algorithm*. The final effect is then returned to the PEP to be enforced. We choose to model a *deny biased* PEP: if permit is received from the PDP then the PEP will permit access; any other response will cause access to be denied. A *permit biased* PEP will only deny requests for which the PDP decision is deny.

More precisely, when a request is made, the PDP will return exactly one of:

– Permit: if the subject is permitted to perform the action on the resource,
– Deny: if the subject is not permitted to perform the action on the resource, or
– NotApplicable: if the request cannot be answered by the service.

The full language also contains the response *Indeterminate*. This is triggered by an error in evaluating the conditional part of the rule. In this work we make the simplification that requests and rules are environment independent. This still allows us to consider a rich class of policies. In particular, it allows us

[1] This means that a single access control policy can be made up of a number of XACML policies.

[2] Strictly, it contains a set of policy sets, each of which contain a set of policies. Each policy contains a set of rules. Extending our model to include this additional complexity would be straightforward.

to omit the environment and condition components. We therefore do not use INDETERMINATE.

A full XACML request includes a set of *subjects* (e.g. a user, the machine the user is on, and the applet the user is running could all be subjects with varying access rights) to perform a set of *actions* (e.g. read, write, copy) on a set of *resources* (e.g. a file or a disk) within an environment (e.g. during work hours or from a secure machine). It may also contain a *condition* on the environment, to be evaluated when the request is made.

3 Modelling XACML

The Vienna Development Method (VDM) [14, 8] is a model-oriented formal method incorporating a modelling or specification language (VDM-SL) with formal semantics [1], a proof theory [2] and refinement rules [14]. VDM's origins lie in work on the formal definition of programming languages and compilers at IBM's Vienna Laboratory in the 1970s [13]. The characteristics of the specification language make it well suited for our work, in which we effectively provide a VDM semantics for a subset of XACML. In this paper, we will use the mathematical syntax of ISO Standard VDM-SL [1].

3.1 The VDM-SL language

A VDM-SL model is based on a set of data type definitions, which may include invariants (arbitrary predicates characterising properties shared by all members of the type). Functionality is described in terms of functions over the types, or operations which may have side effects on distinguished state variables. Functions and operations may be restricted by preconditions, and may be defined in an explicit algorithmic style or implicitly in terms of postconditions. The models presented in this paper use only explicitly-defined functions. We remain within a fully executable subset of the modelling language, allowing our models of XACML policies to be analysed using an interpreter.

VDM has strong tool support. The CSK VDMTools (`www.vdmbook.com`) include syntax and type checking, an interpreter for executable models, test scripting and coverage analysis facilities, program code generation and pretty-printing. These have the potential to form a platform for tools specifically tailored to the analysis of access control policies in an an XACML framework.

3.2 XACML in VDM-SL

An access control policy is essentially a set of complex data types, and the XACML standard is a description of these data types and of the evaluation functions over them. Thus VDM-SL, with its separation of data types and functionality, is a suitable language to describe access control policies.

We now describe the data types and functionality of an XACML policy. The description is presented in VDM-SL. We impose the simplifications mentioned in the previous section. In particular, we limit targets to sets of subjects, actions and resources, and exclude consideration of the environment. This means we only consider the effects *permit*, *deny* and *not applicable*.

Data types: The first definition introduces the type *PDP*. Elements of this type are composite values (pairs), and each field is labelled. The first field has label *policies* and contains a set of the elements of the type *Policy*. The second field contains a single enumerated value[3] of the type *CombAlg*, defined immediately below.

$$\begin{aligned} \mathit{PDP} :: \quad \mathit{policies} &: \mathit{Policy}\text{-}\mathbf{set} \\ \mathit{policyCombAlg} &: \mathit{CombAlg} \end{aligned}$$

$$\mathit{CombAlg} = \textsc{DenyOverrides} \mid \textsc{PermitOverrides}$$

DenyOverrides and PermitOverrides will act as pointers to the appropriate algorithms, to be defined later. Other possible combining algorithms are given in [16] but for simplicity we will model only these two here.

A policy contains a *target* (the sets of *subjects*, *resources* and *actions* to which it applies), a set of *rules*, and the name of a *rule combining algorithm*.

$$\begin{aligned} \mathit{Policy} :: \quad \mathit{target} &: \mathit{Target} \\ \mathit{rules} &: \mathit{Rule}\text{-}\mathbf{set} \\ \mathit{ruleCombAlg} &: \mathit{CombAlg} \end{aligned}$$

$$\begin{aligned} \mathit{Target} :: \quad \mathit{subjects} &: \mathit{Subject}\text{-}\mathbf{set} \\ \mathit{resources} &: \mathit{Resource}\text{-}\mathbf{set} \\ \mathit{actions} &: \mathit{Action}\text{-}\mathbf{set} \end{aligned}$$

Each rule has a target and an *effect*. If a request target overlaps with the rule target, in a way made precise by the function *targetmatch*, then the rule effect is returned. Otherwise *not applicable* is returned. The brackets [..] denote that the target component may be Null. In this case, [16] requires that the value is the target of the parent policy.

$$\begin{aligned} \mathit{Rule} :: \quad \mathit{target} &: [\mathit{Target}] \\ \mathit{effect} &: \mathit{Effect} \end{aligned}$$

The effect of the rule can be *permit*, *deny*, or *not applicable*. These are modelled as enumerated values.

$$\mathit{Effect} = \textsc{Permit} \mid \textsc{Deny} \mid \textsc{NotApplicable}$$

Functionality: Instances of the above types are evaluated with respect to a *request*, which is simply a *target*:

$$\mathit{Request} :: \mathit{target} : \mathit{Target}$$

[3] In VDM-SL, The only operator defined over enumerated values is equality.

We begin with the component function *targetmatch*. In [16] a request target (*target1*) and a rule/policy target (*target2*) are said to match if they each have at least one element in common.

```
$targetmatch : Target \times Target \to Bool$
$targetmatch(target1, target2) \quad \triangleq$
    $(target1.subjects \cap target2.subjects) \neq \{\,\} \wedge$
    $(target1.resources \cap target2.resources) \neq \{\,\} \wedge$
    $(target1.actions \cap target2.actions) \neq \{\,\}$
```

A request is evaluated against a rule using *evaluateRule*. If the rule target is Null the targets are assumed to match, since the parent policy target must match. If the targets match the effect of the rule is returned, otherwise NotApplicable is returned.

```
$evaluateRule : Request \times Rule \to Effect$
$evaluateRule(req, rule) \quad \triangleq$
    if $rule.target = \textsc{Null}$
    then $rule.effect$
    else if $targetmatch(req.target, rule.target)$
        then $rule.effect$
        else $\textsc{NotApplicable}$
```

A policy is invoked if its target matches the request. It then evaluates all its rules with respect to that request, and combines the returned effects using its *rule combining algorithm*.

```
$evalPol : Request \times Policy \to Effect$
$evalPol(req, pol) \quad \triangleq$
    if $targetmatch(request.target, policy.target)$
    then cases $pol.ruleCombAlg$ of
            $\textsc{DenyOverrides} \to evalRulesDenyOverrides(req, pol.rules)$
          $\textsc{PermitOverrides} \to evalRulesPermitOverrides(req, pol.rules)$
          others $\textsc{NotApplicable}$
          end
    else $\textsc{NotApplicable}$
```

The implementation of the deny override algorithm is

```
$evalRulesDenyOverrides : Request \times Rule\text{-}\mathbf{set} \to Effect$
$evalRulesDenyOverrides(req, rs) \quad \triangleq$
    if $\exists r \in rs \cdot evaluateRule(req, r) = \textsc{Deny}$
    then $\textsc{Deny}$
    else if $\exists r \in rs \cdot evaluateRule(req, r) = \textsc{Permit}$
        then $\textsc{Permit}$
        else $\textsc{NotApplicable}$
```

If any rule in the policy evaluates to DENY, the policy will return DENY. Otherwise, if any rule in the policy evaluates to PERMIT, the policy will return PERMIT. If no rules evaluate to either PERMIT or DENY, the policy will return NOTAPPLICABLE.

The permit override algorithm (omitted) is identical in structure, but a single PERMIT overrides any number of DENYS.

The evaluation of the PDP and its rule combining algorithms has an equivalent structure to the policy evaluation functions already presented.

```
$evaluatePDP : Request \times PDP \rightarrow Effect$

$evaluatePDP(req, pdp) \quad \triangleq$
    **cases** $pdp.policyCombAlg$ **of**
      DENYOVERRIDES $\rightarrow evalPDPDenyOverrides(req, pdp)$
    PERMITOVERRIDES $\rightarrow evalPDPPermitOverrides(req, pdp)$
    **others** NOTAPPLICABLE
    **end**


$evaluatePDPDenyOverrides : Request \times PDP \rightarrow Effect$

$evaluatePDPDenyOverrides(req, pdp) \quad \triangleq$
    **if** $\exists p \in pdp.policies \cdot evalPol(req, p) =$ DENY
    **then** DENY
    **else if** $\exists p \in pdp.policies \cdot evalPol(req, p) =$ PERMIT
        **then** PERMIT
        **else** NOTAPPLICABLE
```

The above functions and data types are generic. Any XACML policy in VDM-SL will use these functions. In the next section we show how to instantiate the data types with a particular policy.

4 Populating and testing the policy

In this section we present the initial requirements on an example policy. In Section 4.1 we instantiate the model presented in Section 3.2 with a policy aimed at implementing these requirements. In Section 4.2 and Section 4.3 we show how we can use the testing capabilities of VDMTools [5] to find errors in these policies. In Section 4.4 we consider how we might formalise the properties of Section 4.1 in order to check them automatically and in Section 4.5 we discuss the our work on automatic translation from XACML to VDM-SL.

4.1 A simple example

This example is taken from [6], and describes the access control requirements of a university database which contains student grades. There are two types of resources (*internal* and *external* grades), three types of *actions* (*assign*, *view* and *receive*), and a number of subjects, who may hold the roles *Faculty* or *Student.* We therefore make the following definitions to produce a derivative specification from our general model.

$$Action = \textsc{Assign} \mid \textsc{View} \mid \textsc{Receive}$$

$$Resource = \textsc{Int} \mid \textsc{Ext}$$

Subjects are enumerated values:

$$Subject = \textsc{Anne} \mid \textsc{Bob} \mid \textsc{Charlie} \mid \textsc{Dave}$$

We populate the student and faculty sets as:

$$\begin{aligned} Student &: Subject\text{-}\mathbf{set} = \{\textsc{Anne}, \textsc{Bob}\} \\ Faculty &: Subject\text{-}\mathbf{set} = \{\textsc{Bob}, \textsc{Charlie}\} \end{aligned}$$

making Bob a member of both sets. In practice, an access control request is evaluated on the basis of certain attributes of the subject. What we are therefore saying here is that the system, if asked, can produce evidence of Anne and Bob being students, and of Bob and Charlie being faculty members.

Informally, we can argue that populating the student and faculty sets so sparsely is adequate for testing purposes. All rules we go on to define apply to roles, rather than to individuals, so we only need one representative subject holding each possible role combination.

The properties to be upheld by the policy are

1. No students can assign external grades,
2. All faculty members can assign both internal and external grades, and
3. No combinations of roles exist such that a user with those roles can both receive and assign external grades.

Our initial policy (following the example in [6]) is

> *Requests for students to receive external grades, and for faculty to assign and view internal and external grades, will succeed.*

Implementing this policy naïvely leads to the following two rules, which together will form our initial (flawed) policy. Students may receive external grades,[4]

$$StudentRule : Rule = ((Student, \textsc{Ext}, \textsc{Receive}), \textsc{Permit})$$

and faculty members may assign and view both internal and external grades.

$$FacultyRule : Rule = ((Faculty, \{\textsc{Int}, \textsc{Ext}\}, \{\textsc{Assign}, \textsc{View}\}), \textsc{Permit})$$

The policy combines these two rules using the permit overrides algorithm. The target of the policy is all requests from students and faculty members.

[4] The correct VDM-SL description is
$StudentRule : Rule = mk\_Rule(mk\_Target(Student, \{\textsc{Ext}\}, \{\textsc{Receive}\}), \textsc{Permit})$
For ease of reading we omit the *mk_* constructs and brackets around singleton sets.

$\mathit{PolicyStuFac} : \mathit{Policy} =$
$((\mathit{Student} \cup \mathit{Faculty}, \{\textsc{Int}, \textsc{Ext}\}, \{\textsc{Assign}, \textsc{View}, \textsc{Receive}\}),$
$\{\mathit{StudentRule}, \mathit{FacultyRule}\}, \textsc{PermitOverrides})$

In fact, this policy would have the same effect for all requests if the two rules were combined using the deny overrides algorithm. This is because there is no request which one rule permits and another denies.

The PDP is a collection of policies; in this case only one.

$\mathit{PDPone} : \mathit{PDP} = (\mathit{PolicyStuFac}, \textsc{DenyOverrides})$

We use the deny overrides algorithm here but because there is only one policy, there will only be one effect, and no opportunity for overriding.

4.2 Testing the derivative specification

VDMTools [5] provides considerable support for testing VDM specifications. Individual tests can be run at the command line in the interpreter. The test arguments can be also read from pre prepared files, and scripts are available to allow large batches of tests to be performed.

A systematic approach to testing requires that we have some form of oracle against which to judge the (in)correctness of test outcomes. This may be a manually-generated list of outcomes (where we wish to assess correctness against expectations) or an executable specification (if we wish to assess correctness against the specification). In this section we use a list of expected results as our oracle. Section 4.3 uses one version of a VDM-SL specification as an oracle against another version.

Tests on *PDPone* are made by forming requests and evaluating the PDP with respect to these requests, using the function *evaluatePDP* from Section 3.2. Below we show four example requests and the results anticipated by the three properties in Section 4.1. The results from *PDPone* are in the third column.

Request	Prediction	*PDPone*
(Anne, Ext, Assign)	Deny	NotApplicable
(Bob, Ext, Assign)	Deny	Permit
(Charlie, Ext, Assign)	Permit	Permit
(Dave, Ext, Assign)	Deny	NotApplicable

In the first test, *PDPone* returns *not applicable* when user Anne (a student) asks to assign an external grade. This is because there is no rule which specifically covers this situation. A deny biased PEP would resolve this by denying the request. We choose to keep the *not applicable* result to give the tester more comprehensive information.

The second test points out an error, because Bob (who is both a faculty member and a student) is allowed to assign external grades, in violation of property one, which states that no student may assign external grades. This policy has been written with an implicit assumption that the sets student and faculty are disjoint. Constraining these sets to be disjoint when we populate them allows us to reflect this assumption. In practice this constraint would have to be enforced at the point where roles are assigned to individuals rather than within the PDP.

The third test is permitted, as expected, since Charlie is a member of faculty, and the fourth test returns *not applicable*, because Dave is not a student or a faculty member.

Multiple requests The policy as defined can be broken if multiple access control requests are combined into one XACML request. For example the request below (identified in [6])

$(\textsc{Anne}, \{\textsc{Ext}\}, \{\textsc{Assign}, \textsc{Receive}\})$

is permitted. As pointed out in [6], this breaks the first property, because Anne (a student) is piggybacking an illegal request (assigning an external grade) on a legal one (receiving an external grade). In future, therefore, we make the assumption that the PEP only submits requests that contain singleton sets. Given this assumption, we can limit the test cases we need to consider to those containing only single subjects, actions and resources.

4.3 Comparing specifications of PDPs

Here we show how we can determine the differences between two versions of a policy. In effect, we use the results from the first version as an oracle, compare them with the results of the second version and alert the user to any inconsistencies.

To demonstrate this, we suppose (following [6]) that teaching assistants (TAs) are to be employed to help with the internal assignment of grades. They are not, however, allowed to help with external assignment of grades. A careless implementation, that merely included the names of the TAs as faculty members, would overlook the fact that students are often employed as TAs.

A more robust implementation, that makes TAs a separate role and develops rules specific for them, is given below. Note that in *TArule*2, TAs are explicitly forbidden to assign or view external grades; their role is restricted to dealing with the internal grades.

$TArule1 : Rule = ((TA, \{\textsc{Int}\}, \{\textsc{Assign}, \textsc{View}\}), \textsc{Permit})$
$TArule2 : Rule = ((TA, \{\textsc{Ext}\}, \{\textsc{Assign}, \textsc{View}\}), \textsc{Deny})$

The rules are combined into a (TA-specific) policy

$PolicyTA : Policy =$
$((TA, \{\textsc{Int}, \textsc{Ext}\}, \{\textsc{Assign}, \textsc{View}, \textsc{Receive}\}),$
$\{TArule1, TArule2\}, \textsc{PermitOverrides})$

which is combined with *PolicyStuFac* from Section 4.1 to give a new Policy Decision Point:

$PDPtwo : PDP = (\{PolicyTA, PolicyStuFac\}, \textsc{DenyOverrides})$

This new PDP can of course be tested independently, but it can also be compared with the previous one. We do this with respect to a test suite. The policies are small and so this test suite can be comprehensive. In order to have a fully instantiated policy to test, we populate the roles as

$$\begin{aligned}
&\mathit{Student} : \mathit{Subject}\text{-}\mathbf{set} = \{\text{Anne, Bob}\}\\
&\mathit{Faculty} : \mathit{Subject}\text{-}\mathbf{set} = \{\text{Charlie}\}\\
&\mathit{TA} : \mathit{Subject}\text{-}\mathbf{set} \quad = \{\text{Bob, Dave}\}
\end{aligned}$$

taking care that there is a person holding each possible combination of roles that we allow. Every request that each person can make is considered against each PDP. This is easily automated using a simple shell script, which creates every possible request and feeds them all to the PDP. The observed changes are summarised below.

Request	*PDPone*	*PDPtwo*
(Bob, Int, Assign)	NotApplicable	Permit
(Bob, Int, View)	NotApplicable	Permit
(Bob, Ext, Assign)	NotApplicable	Deny
(Bob, Ext, View)	NotApplicable	Deny
(Dave, Int, Assign)	NotApplicable	Permit
(Dave, Int, View)	NotApplicable	Permit
(Dave, Ext, Assign)	NotApplicable	Deny
(Dave, Ext, View)	NotApplicable	Deny

As a TA, Bob's privileges now include assigning and viewing internal grades, as well as all the privileges he has as a student. Everything else is now explicitly denied.

All requests from Dave, who is a now TA but not a student, are judged *not applicable* (and consequently denied) by the first policy, but the second policy allows him to view and assign internal grades. It explicitly forbids him to assign or view external grades.

Internal consistency of a PDP: We consider a set of rules to be consistent if there is no request permitted by one of the rules which is denied by another in the set. A set of policies is consistent if there is no request permitted by one of the policies which is denied by another in the set.

Rule consistency within a policy and policy consistency within a PDP can each be checked using the method outlined above, using the functions *evaluateRule* and *evaluatePol* from Section 3.2.

4.4 Satisfaction of Policy Requirements

In Section 4.1 we gave a formalisation of a specific policy for the student assignment example. We identified three properties to be upheld by the policy, expressed informally. We validated the VDM model of a policy by testing and manually inspecting the results of the tests to gain confidence in the conformance to informal requirements (Section 4.2). It is interesting to consider the formalisation of these requirements and the potential for checking them automatically when policies are developed or updated.

Using a formal model encourages us to consider the precise meaning of policy requirements. For example, the first property in Section 4.1 is “ No students can assign external grades”. This might be considered to be primarily a requirement on a policy (or a PDP embodying a policy), so it might be expressed as follows (for *PDPone*):

$$\neg \exists s \in \mathit{Student} \cdot \mathit{evaluatePDP}((s, \textsc{Ext}, \textsc{Assign}), \mathit{PDPone}) = \textsc{Permit}$$

The informal statement of the requirement, however, relates to the overall effect of the policy. In XACML this is not necessarily the same thing as the value returned by the PDP; decisions are implemented at the enforcement point (PEP). Since the formal constraint above allows a student's request to assign external grades to yield *not applicable*, the PEP may yet allow the access. The formalisation above is therefore valid if the PEP is deny biased, but not if it is permit biased.

Formal versions of the other two requirements from our example are stated below, again assuming a deny biased PEP:

$$\begin{aligned}&\forall f \in \mathit{Faculty} \cdot\\ &\quad \mathit{evaluatePDP}((f, \textsc{Int}, \textsc{Assign}), \mathit{PDPone}) = \textsc{Permit} \wedge\\ &\quad \mathit{evaluatePDP}((f, \textsc{Ext}, \textsc{Assign}), \mathit{PDPone}) = \textsc{Permit}\end{aligned}$$

$$\begin{aligned}&\neg \exists u \in \mathit{Student} \cap \mathit{Faculty} \cdot\\ &\quad \mathit{evaluatePDP}((u, \textsc{Ext}, \textsc{Assign}), \mathit{PDPone}) = \textsc{Permit} \wedge\\ &\quad \mathit{evaluatePDP}((u, \textsc{Ext}, \textsc{Receive}), \mathit{PDPone}) = \textsc{Permit}\end{aligned}$$

Being able to formalise the requirements at this level allows us to check conformance by encoding the requirements as boolean functions in the model. In general, for a larger-scale system, evaluating those functions will amount to an exhaustive test of the request space. In spite of this, it may sometimes be feasible to make the appropriate checks when policies are updated.

4.5 Automatic generation of the VDM-SL from XACML

The GOLD project [18, 4] has been researching into enabling technology to enable the formation, operation and termination of VOs within the high-value chemicals industry. Access control is a key issue within GOLD. To support this work we have produced an experimental Java API for automatic generation of the VDM-SL representation of an access control policy from a number of XACML policies. It produces a VDM-SL description representing the combined set of XACML policies. It operates on policies which conform to the XACML role based access control profile [15] (XACML RBAC). The RBAC profile insists that permissions are assigned to roles, and that roles are assigned to individuals, rather than that permissions are assigned directly to individuals. It requires three XML files for each role. The *role declaration file* simply names the role and points to the relevant *permissions file* for that role. The permissions file contains the policy for that role. The information extracted from the permissions file is used to create instances of the VDM types *Rule* and *Policy*. The *role assignment* *file* contains a list of the user identities of permitted holders of that role. It is used to retrieve the target users for that role.

The API exposes a WSDL interface which, in combination with VDMTools, can be used to streamline the validation of XACML policies in each of the ways outlined in Sections 4.2 and 4.3. It affords the developer of a policy a quick way to test (prior to deployment) that certain access requests are permitted or denied. In addition it can be used to compare policies before and after an alteration, to confirm the extent of any change.

5 Related Work

There is a large body of work in the development and analysis of access control policies. The work most closely related to our own takes the approach of model checking. We identify three strands whose aims are closely aligned with our own.

In [10] the authors propose a simple propositional access control language called *RW*. An access control system written in *RW* includes a set of agents, a set of propositional variables and two possible actions R and W, which allow agents to read and overwrite the values of propositional variables. In [20] it is shown how access control systems written in *RW* may be model checked with respect to a goal, where a goal involves a combination of reading and manipulation propositional variables. In [19] it is shown how access control systems written in *RW* may translated into XACML.

In providing a translation from XACML to VDM-SL, we are able to consider policies already written in XACML. This is in keeping with a long term aim of our work, which is to provide tools which make the benefits of model based specification and testing available and useful to developers.

In [6] the authors present Margrave – a tool for analysing policies written in XACML. Margrave transforms XACML policies into Multi-Terminal Binary Decision Diagrams (MTBDDs) and users can verify policies against properties using these representations.

In [11], Alloy [12] is used to verify access control policies. The authors develop a language for describing access control policies. Partial orderings are defined over these policies. Properties of policies are encoded as propositions over these partial orderings. These may be checked by translating both policies into the Alloy language and using the Alloy Analyzer to check for refinement between the policies.

The testing approach advocated in this paper can be complementary to a model checking approach. Offering the developer the facility to check a XACML policy against a range of carefully selected test cases, including pathological ones, will be a relatively easy way to provide a developer with a first level of confidence in their policy. Our work leaves open the possibility of model checking (and of machine assisted proof) because, although we are using an executable abstract model, it does have a formal semantics. Model checking VDM-SL specifications is an area we hope to consider, and in this we hope to benefit from each of the strands of work outlined above.

6 Conclusions and Further Work

We have presented a formal approach to modelling and analysing access control policies. We have used VDM, a well-established method, as our modelling notation. This has allowed us to use VDMTools to analyse the resultant formal models. We have shown that rigorous testing of access control policies is possible within VDMTools, also that policies may be compared with each other and checked for internal consistency.

Ongoing work is seeking to represent rules that are dependent on context. This will require extending the VDM-SL model with environmental variables and allowing rules to query these variables. In general, this will make the modelling of very fine grained policies possible. For example, we could model policies which permitted dynamic role activation and deactivation, by keeping track of the active roles of a user. Modelling attribute based access control, where decisions are made on the basis of attributes of the users and resources, rather than user and resource identifiers, is another related line of research.

With larger policies, testing all possible requests may become time consuming. Further work will look at techniques and tools for developing economical test suites for access control policies, perhaps by focusing on the significant resources or actions. It is also possible that resources may be categorised, similar to the way that roles categorise people. If all resources in a category were treated in the same way by a policy, our tests need only consider a representative resource from each combination of categories. We expect to benefit from work combining formal methods and testing, such as that reported in [9] and [17].

Following the approach of role based access control, the rules we have considered so far all contain a role as the *Subject*-**set** in the target. However in the XACML specification the *Subject*-**set** of a rule can be an arbitrary set of subjects. If this is the case, then in general *every* possible combination of subject, resource and action would need to be tested, rather than just a single representative from each role combination.

It may be possible to validate access control policies against workflow descriptions. For example a BPEL description of a workflow contains information about resource accesses necessary by some user or role as part of a particular task. We can use this information to generate tests for the VDM-SL description. This will allow us to check that the policies that govern access to those resources allow a particular workflow to proceed to completion. It may be possible to use these tests to generate a least-privilege access control policy.

Automating the translation from VDM-SL to XACML would allow a developer to make changes to VDM-SL specifications and then generate the corresponding XACML.

Acknowledgments: Thanks to Joey Coleman for guiding the first author through shell scripting. This work is part-funded by the UK EPSRC under e-Science pilot project GOLD, the Interdisciplinary Research Collaboration in Dependability (DIRC) and the Defence Science Technology Laboratory (DSTL).

References

1. D.J. Andrews, editor. *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language.* International Organization for Standardization, December 1996. International Standard ISO/IEC 13817-1.
2. J. C. Bicarregui, J.S. Fitzgerald, and P.A. Lindsay et. al. *Proof in VDM: A Practitioner's Guide.* Springer-Verlag, 1994.
3. Jeremy W. Bryans. Formal Analysis of Access Control Policies. In *UK e-Science All Hands Meeting*, Nottingham, 2006. To appear.
4. A Conlin, H. Hiden, and A. Wright. A chemical process development case study as a source of requirements for the gold project. Technical Report CS-TR:968, Newcastle University, June 2006.
5. CSK. VDMTools. available from `http://www.vdmbook.com`.
6. 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.
7. J. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors. *International Symposium of Formal Methods Europe. Newcastle, UK, July 2005*, volume 3582 of *LNCS*. Springer-Verlag, 2005.
8. John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Developement.* Cambridge University Press, 1998.
9. M.-C. Gaudel. Formal Methods and Testing: Hypotheses, and Correctness Assumptions. In Fitzgerald et al. [7], pages 2–8. Invited ralk.
10. D. Guelev, M. Ryan, and P. Schobbens. Model-checking Access Control Policies. In *ISC'04: Proceedings of the Seventh International Security Conference*, volume 3225 of *LNCS*, pages 219–230. Springer, 2004.
11. Graham Hughes and Tevfik Bultan. Automated Verification of Access Control Policies. Technical Report 2004-22, University of California, Santa Barbara, 2004.
12. D. Jackson. *Micromodels of software: Modelling and analysis with Alloy.* `http://sdg.lcs.mit.edu/ alloy/reference-manual.pdf`.
13. C.B. Jones. Scientific Decisions which Characterize VDM. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of the 1999 World Congress on Formal Methods in the Development of Computing Systems (FM '99)*, volume 1708 of *LNCS*, pages 28–47, Toulouse, France, 1999. Springer-Verlag.
14. Cliff B. Jones. *Systematic Software Developement using VDM.* International Series in Computer Science. Prentice-Hall, 1990.
15. OASIS. Core and heirarchical role based access control (RBAC) profile of XACML v2.0. Technical report, OASIS, Feb 2005.
16. OASIS. eXtensible Access Control Markup Language (XACML) version 2.0. Technical report, OASIS, Feb 2005.
17. A. Pretschner. Model-based testing in practice. In Fitzgerald et al. [7], pages 537–541.
18. The GOLD project. `http://gigamesh.ncl.ac.uk/`.
19. N. Zhang, M. Ryan, and D. Guelev. Synthesising Verified Access Control Systems in XACML. In *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, pages 56–65. ACM Press, 2004.
20. N. Zhang, M. Ryan, and D. Guelev. Evaluating access control policies through model checking. In J. Zhou, J. Lopez, R.H. Deng, and F. Bao, editors, *Eighth Information Security Conference (ISC'05)*, volume 3650 of *LNCS*, pages 446–460. Springer-Verlag, 2005.