Formal Engineering of XACML Access Control Policies in VDM++

Jeremy W. Bryans and John S. Fitzgerald

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

Abstract. We present a formal, tool-supported approach to the design and maintenance of access control policies expressed in the eXtensible Access Control Markup Language (XACML). Our aim is to help developers evaluate the consequences of policy decisions in complex situations where security requirements change and access decisions may depend on the external dynamic environment. The approach applies the model-oriented specification language from the Vienna Development Method (VDM++). An executable formal model of XACML access control is presented in VDM++. The use of the model to analyse and revise both policies and requirements on the environment is illustrated through an example. An approach to the practical problem of analysing access control in virtual organisations with dynamic membership and goals is proposed.

1 Introduction

For a multi-user computer system to be secure, the developer must ensure that the people or systems using it are only allowed to perform legitimate actions. The functionality that achieves this is often separated out into a distinct *access control policy* which defines the response to access requests. In many situations, the response given to a request depends on the environment in which the request is evaluated. For example, a request may be disallowed outside of regular working hours, and so each time the request is made a clock will need to be consulted. Such policies are termed *context-sensitive* (also context-aware or dynamic). Context-sensitivity adds complexity to the development and validation of access control policies. They often need to satisfy requirements from different domains, for example legal, technical and commercial. Conformance with each of these sets of requirements must be checked any time the access control policy is modified. Modern virtual organisations are composed of separate agents, each with their own access control policies. They often work to volatile functional requirements and so policies need to be updated accurately and quickly.

The goal of our current work is to assist the access control policy developer by providing rapid feedback on design decisions. Formal techniques are well suited to

this task because of the breadth and rigour of analysis that they afford. However, our goal is pragmatic, so it is vital that formal techniques for policy design are relevant to industry practice. Our specific aims are therefore to support policy analysis in an existing and widely-used access control framework (rather than to propose a new formalism), to focus on methods that can be supported by tools, and to exploit the benefits of formal approaches but provide a low technical barrier to their use by integrating with current and emerging industry practice.

The contribution of this paper is to provide a semantics for XACML in a language (VDM++) which is executable and has strong tool support (VDMTools.) Consistent with the lessons of previous industrial applications of VDM++ we emphasise modelling and analysis code verification and pay special attention to using the strong tool support for model validation by testing. The intention behind this approach to provide entry-level access to formal methods technology without requiring users to learn advanced modelling and analysis techniques at the outset.

We approach this by focussing on a substantial subset of the OASIS standard eXtensible Access Control Markup Language (XACML), providing a formal semantics for it in the formal specification language of VDM++ [10]. The semantics is similar in structure to XACML and includes environments, enabling analysis of context-sensitive policies. The semantic model can be executed directly using the VDMTools interpreter, providing a basis for testing and analysing proposed policies and environments. Test suites may be run against these interpreted models, providing rapid feedback to the developer.

Section 2 contains a brief overview of access control, and Section 3 gives an overview of semantics for XACML policies in VDM++. The validation and evolution of policies using the formal model are illustrated in Section 4. The application of the model in the volatile environment of dynamic virtual organisations is outlined in Section 5. We conclude by comparing related work and identifying potential improvements to the coverage of the formal model and the range of analyses supported.

2 Context-Sensitive Access Control in XACML

This section gives a simplified overview of XACML [16], pointing out how it is used to describe context-sensitive access control policies. XACML provides a *policy language* for describing access control policies and a *request language* for interrogating these policies.

Access control systems that implement XACML have the abstract structure shown in Figure 1. A request to perform an operation on SYS (the system under access control) is forwarded to the *Policy Enforcement Point* (PEP). The PEP translates the request into the XACML request language and passes it on to the *handler*. In XACML a request is a triple containing multiple subjects, actions and resources. However, we will follow [7] and [6] in restricting request triples to contain a single subject, action and resource. We assume that the PEP is defined

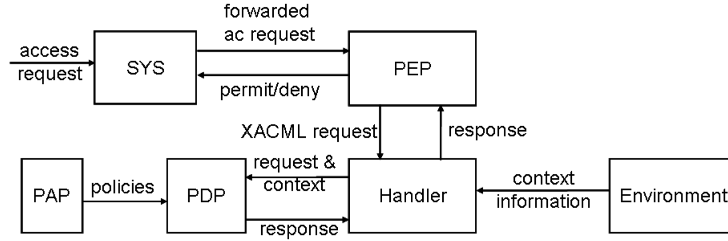

Fig. 1. XACML overview.

to break any compound request into a set of requests and submit them singly, then combine the results in whatever way the developer chooses.

The handler forwards the request, together with any relevant information from the context, to the *Policy Decision Point* (PDP). The PDP retrieves the relevant policies from the *Policy Access Point* (PAP), which contains all the access control policies for SYS. The PDP then evaluates the relevant access control policies in the context and sends the response to the handler. The result is returned to the PEP where it is enforced as either permission or denial. The decision of how to enforce the result returned by the PDP is in general application-dependent and is not in the scope of our current formalisation.

The PAP may contain a number of arbitrarily nested policies. When faced with a request, each policy must produce a single result, so a parent policy contains an algorithm for combining the responses of all the child policies into a single result. Policies may also contain a *target* – a set of subjects, resources, and actions denoting the limits of the applicability of the policy. A policy is applied to a request if the request *matches* the policy target, i.e. if the request subject, resource and action are in the set of policy target subjects, resources and actions respectively. If a request does not match the policy target – i.e. it relates to a subject, resource or action outside the policy target – the policy will generate the result *"not applicable"*. If the target is not present in the policy every request will match.

The lowest-level policies contain sets of rules, together with a combining algorithm. Like policies, rules in XACML may contain a target. They will also contain an effect (permit or deny). Optionally, they may also contain a *condition.*

A *condition* is the part of a policy which is *context-sensitive*. It is a Boolean function, and may range over the subject, resource and action of the request as well as arbitrary environment variables. If a request matches a rule target the condition of the rule is evaluated. Evaluating the condition will involve consulting the environment of the system. If the condition evaluates to true the effect of the rule will be returned (which may be either permit or deny). If the condition evaluates to false the rule will return not applicable. If the condition cannot be evaluated (perhaps because some system-level environmental information is

missing) the rule will return *indeterminate*. Finally, if the request did not match the rule target *not applicable* is returned.

If several rules return a result for a request, the results are combined using a *rule-combining-algorithm*. The ones considered in this paper are *denyOverrides*, which will deny a request if a single rule denies it, and *permitOverrides*, which will permit a request if a single rule permits it.

As an example, consider the access control policy for a document management system designed to control access to documents within a chemical engineering plant. Documents are held on a central database and access must be carefully controlled. Each document must be reviewed after it is written. Suppose the initial stages of a project involve two documents – the hazard analysis and the production plan. We begin by developing the access control policy needed for these early stages. Suppose the developer is initially given two rules to implement:

1. The hazard analysis must be signed off before anyone may write to the production plan.

2. An author of a document cannot be the reviewer of that document.

An outline of a possible XACML realisation of the first rule is:

```
<Rule RuleId="hazanBeforePPRule" Effect="Deny">
  <Description> Deny PP write before haz_an signed off </Description>
  <Target>
     <Subjects><AnySubject/></Subjects>
     <Resources>...ProductionPlan...</Resources>
     <Actions>...write...</Actions>
  </Target>
  <Condition FunctionId="...:function:not">
    <Apply FunctionId="...signed-off">...haz_an...</Apply>
  </Condition>
</Rule>
```

The effect (to deny access) is in the top line, the target of the rule tells us that the rule is about write requests on the production plan and finally the rule has a condition (which will vary according to the precise context.) If a request is made to write to the production plan the rule must be able to check if the hazard analysis has been signed off. The second rule, requires that it be possible to check authorship of any document.

The structure available in the environment to answer the context-related queries may vary in each case. The documents may be in XML format, with "signed-off" and "authorship" fields, or the information may be recorded separately in a database. The policy developer will not necessarily know in advance which of these alternative implementations to assume. It is therefore important that the developer can model the environment abstractly, without requiring a particular implementation, but ensuring that the abstract model of the environment will have all the relevant behaviours that the real environment may exhibit.

3 A Semantics for XACML

This section introduces the VDM++ formalism, then presents a semantics for XACML in terms of an executable formal model in VDM++.

3.1 The VDM++ Formalism and Tools

The model is expressed in VDM++ [10], the object-oriented extension of the Vienna Development Method (VDM) specification language [13, 1]. For a detailed introduction to VDM and its support tools the reader is referred to [8] and the VDM portal [17]. VDM++ models are composed of class definitions, each of which may contain local state specified by typed instance variables. Type definitions are given in terms of abstract basic types including token types for representing structureless values. Complex types are constructed using record and union constructors, set, sequence and mapping types. Type membership is restricted by invariants. Functionality in each class is expressed in terms of operations that may modify the values of the instance variables and auxiliary functions that do not affect the local state. Operations and functions may be specified implicitly by means of postconditions, or explicitly. In either case, restrictions on their domains are expressed by logical preconditions. Constants may also be declared (as `values`). In our work on XACML, we remain within an executable subset of the modelling language, allowing our models of access control policies to be readily analysed using the VDMTools interpreter.

VDM++ benefits from strong tool support and a record of industrial use [9]. The existing tool set (VDMTools) includes a syntax and type checker, 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 XACML framework. Advanced static analysis for VDM models includes automatic proof obligation generation and automated proof support is under development.

An access control policy in XACML can be thought of as a data structure based on a set of complex data types. The XACML standard is a description of these data types and of the evaluation functions over them. VDM++, with its separation of data types and functionality, is an appropriate language to describe access control policies. VDM++ also allows a clear encapsulation of functionality within classes, which makes it particularly suitable for describing the architectural model of an access control environment.

3.2 A Model of XACML

This section gives an overview of a formal model of access control in VDM++. Throughout this section and Section 4 we use the interchange syntax of VDM++. The extracts are drawn from the full VDM++ model[1].

[1] Available at *http://homepages.cs.ncl.ac.uk/jeremy.bryans/VDMPPacmodel/*

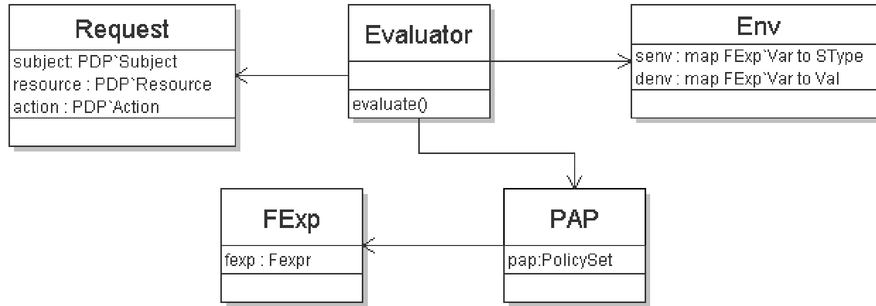

Fig. 2. Overview of the VDM++ model.

Fig. 2 is an informal class diagram showing the structure of the VDM++ model. The structure of the model closely reflects that of XACML. The `Request` class describes requests that a user may make. Policies are represented in objects of the `PAP` class; individual rules may include conditions that are expressed in expressions of the `FExp` class. The `Evaluator` collects together the functionality of the XACML handler and Policy Decision Point. It evaluates a `request` with respect to a `PAP` policy in an environment (an instance of the `Env` class). We briefly consider each of the main classes in turn.

Policy Model. The policy model is described in the Policy Access Point (PAP) class. In the VDM++ model, objects from the PAP class contain a single instance variable `pap` of type `PolicySet`. A `PolicySet` contains an optional `Target`, a set of elements from `Policy` and `PolicySet`, and the name of the policy combining algorithm (of type`CombAlg`).

```
PolicySet :: target : [Target]
         components : set of (Policy|PolicySet)
      policyCombAlg : CombAlg;

Target :: subjects : set of Subject
         resources : set of Resource
           actions : set of Action;

Policy :: target : [Target]
           rules : set of Rule
     ruleCombAlg : CombAlg;

CombAlg = <denyOverrides> | <permitOverrides>;

Rule :: target : [Target]
        effect : Effect
          cond : [FExp]
```

A policy contains an Target (optional, indicated by the [..]), a set of Rules, and a combining algorithm name. The enumerated values <denyOverrides> and <permitOverrides> act as pointers to the appropriate algorithms, defined in the Evaluator class. Other possible combining algorithms are given in the XACML Standard [16], but for simplicity we will model only these two here. Both these algorithms may be applied to both policies and rules; a slight modification would allow us to include algorithms which only apply to either policies or rules. The effect of the rule is the value returned by the rule when a request is evaluated against it. It can be one of the enumerated values <Permit>, <Deny>, <Indet> or <NotApplicable>. The condition cond may contain an expression of the class FExp.

Dynamic Context. The context in which the rules are evaluated is given by the environment, which contains two mappings. senv represents a static environment mapping expression variables to types, and denv represents a dynamic environment mapping expression variables to values.

```
senv : map FExp`Var to SType;
denv : map FExp`Var to Val
```

The senv component contains a record of the types of the variables in the dynamic component. This allows us to type check expressions evaluated within this environment. The dynamic component of the environment contains the values of each the variables. The abstract syntax of values is given below.

```
Val = AtomicVal | StructuredVal;
AtomicVal = bool | int | <Indet>;
StructuredVal = BoolArray | IntArray | VarArray;
BoolArray = map FExp`Var to bool;
IntArray = map FExp`Var to int;
VarArray = map FExp`Var to (map FExp`Var to bool)
```

An XACML implementation must provide support for arbitrary run time environments, whereas in our approach we build a simpler abstract model of the environment that manifests the behaviour necessary to test the rules and policies that we design. For this reason our syntax of expressions is currently less expressive than that offered by XACML. However, it is readily extensible.

Atomic values may be Boolean or integer, as well as the special value Indet used to model situations where the environment fails to return a result. A BoolArray is a map from variables to Booleans and an IntArray is a map from variables to integers. A VarArray is a map to BoolArrays.

Condition Expressions. The syntax of expressions and the structure of the environment are closely related. Objects from the FExp class are used to build the conditions that form the context-sensitive part of rules. Each object is an expression. In general, an environment and a request may need to be provided in order to fully evaluate an expression, as it may contain references to elements in both of them. An ordinary expression is one which only uses variable names

declared in the environment, and a full expression also uses the reserved terms `requester` or `resource`. Before a full expression is evaluated, these are instantiated with the name of the request subject and resource. Ordinary (non-full) expressions take one of the following forms:

```
Expr = Var | Unary | Infix | Literal | ArrayLookup | VarArrayLookup;

Unary :: op : <NOT>
       body : Expr;

Infix :: left : Expr
           op : <AND>|<OR>|<LT>
        right : Expr;

ArrayLookup :: aname : Var
               index : Var;

VarArrayLookup :: aname  : Var
                  index1 : Var
                  index2 : Var
```

A `Var` expression is a record with a single token field, and a `Literal` may be a Boolean literal or an integer literal. Expressions may be negated using the `Unary` type. `Infix` expressions may be conjunctions or disjunctions of Boolean expressions, and integer expressions may be combined using less-than. Expressions may also look up values in environment `Arrays`. An `ArrayLookup` contains the name of the array and the index to be looked up. A `VarArrayLookup` contains the name of a `VarArray` and two indices. The first index is the name of the Boolean array to be looked up, and the second is the name of the index to be addressed within the Boolean array.

The syntax of full expressions (not shown here) extends the above by including `requester` and `resource` as uninstantiated variables.

```
UnVar :: <requester>|<resource>
```

These may be used in the same way as instantiated variables, so full expressions contain `Var|UnVar` where `Var` appears above.

Evaluating Requests The `Evaluator` class describes the evaluation of requests, combining the functionality of the XACML handler and Policy Decision Point.

As indicated in Section 2, we restrict requests to a single `subject`, `resource`, and `action`. In the VDM++ these are instance variables of the `Request` class. The `Subject`, `Resource` and `Action` types are drawn from the PAP class but could be pulled out into a separate unit.

```
subject  : PAP`Subject;
resource : PAP`Resource;
action   : PAP`Action
```

The Evaluator class contains four instance variables: a `PAP`, an environment `Env`, a `request` and the instantiation mapping `inst`. The `inst` mapping will map the reserved terms `requester` and `resource` to the actual requester and resource in the access control request triple. This allows `requester` and `resource` in any of the conditions in the `PAP` to be bound to the current request `subject` and `resource`. The constructor takes a request, a PAP and an environment, and instantiates `Inst` from the request. The operation `evaluate` then returns the effect of evaluating the request `req` against the PAP `pap` in the environment `env`.

```
  pap : PAP;                          -- an object of class PAP
  env : Env;                          -- an object of class Env
  req : Request;                      -- an object of class Request
  inst: Inst := mk_Inst({|->});       -- the instantiation mapping
```

The `evaluate` operation (below) invokes the appropriate evaluator on the basis of the combining algorithm.

```
public evaluate: () ==> PAP`Effect
evaluate() ==
  if (pap.GetpolicyCombAlg() = <denyOverrides>) then
    return(evaluatePAPDenyOverrides())
  elseif (pap.GetpolicyCombAlg() = <permitOverrides>) then
    return(evaluatePAPPermitOverrides())
  else
    return(<NotApplicable>)
```

The tree of policies is traversed and each policy combines the results of the subcomponents of that policy using the stated combining algorithm. At the level of single rules, the effect of the rule is returned if there is no condition or the condition evaluates true in the current environment. Otherwise the rule returns `<NotApplicable>` if the condition evaluates false and `<Indet>` if the condition can not be evaluated (e.g. if a variable reference in the condition does not point to a variable in the environment).

```
evaluateRule : PAP`Rule ==> PAP`Effect
evaluateRule(rule) ==
  if targetmatch(rule.target) then
     if rule.cond = nil
     then return(rule.effect)
     else
       cases (rule.cond).Evaluate(req,env):
                  true   -> return(rule.effect),
                  false  -> return(<NotApplicable>),
                 <Indet> -> return(<Indet>)
       end
  else
    return(<NotApplicable>)
```

4 Validating Access Control Policies using the Model

Our overall aim is to support the design and evolution of context-sensitive access control policies. In particular, we wish to provide rapid feedback on the characteristics of policies, or on the effects of changing policies, before they are implemented. The formal model of XACML presented in Section 3 can provide a basis for this form of evaluation.

The XACML model is written in the executable subset of VDM++. Initialised with an environment and a proposed policy, a set of test requests can be evaluated by direct execution of the semantic model on the VDMTools interpreter. Based on the outcome of such an evaluation, the designer may choose to modify either or both of the policy and the environment. For example, a policy might be modified to add specific constraints, or the environment might have to be extended in order to include external information required to facilitate a new rule. The designer has an eye to the data that can be obtained from the environment as well as the content of the policy itself. This process of test and modification can go through several iterations, eventually yielding a model policy and environment. The policy can be implemented in a real XACML Policy Access Point. The environment model specifies the external data that must be obtainable from the real environment in order for the policy to behave correctly.

This section illustrates the process of iterative development using the executable semantics of XACML. Using the simple example introduced in Section 2, we first show an initial encoding of two elementary rules in an access control policy and then illustrate the testing and subsequent modification of the policy. Throughout we show how policies and tests are represented in VDM++, but in practice we expect the user of a tool based on the semantic model to access it through a more friendly interface.

4.1 Designing Rules

Consider first the implementation of the two rules given in Section 1, namely

1. The hazard analysis must be signed off before anyone may write to the production plan.

2. An author of a document can not be the reviewer of that document.

The developer begins by defining, in VDM++, a sample environment in which the rules will be exercised. From the example rules, necessary elements in the environment will include names for the two documents in question (`haz_an` for the hazard analysis and `pp` for the production plan) and the names of the two relevant actions (`write` and `review`). To fully specify an example environment, in which different subjects will have different privileges, the names of some subjects (here `Anne` and `Bob`) are included. All these are defined in VDM++ as variables from the class `FExp`.

```
haz_an : FExp`Var = mk_FExp`Var(mk_token("hazard_analysis")),
pp     : FExp`Var = mk_FExp`Var(mk_token("production_plan")),

write  : FExp`Var = mk_FExp`Var(mk_token("write")),
review : FExp`Var = mk_FExp`Var(mk_token("review")),

Anne   : FExp`Var = mk_FExp`Var(mk_token("Anne")),
Bob    : FExp`Var = mk_FExp`Var(mk_token("Bob"))
```

To construct the environment, we consider the two rules. Rule 1 contains a reference to hazard analysis being signed off. Since this will change over the course of the document lifetime, it must be possible when the policy is being evaluated to determine it from the environment. The developer does not need to know at this stage the way in which the environment stores this information, merely that it must be available. One possibility open to the developer is to include a Boolean variable to indicate whether or not the document is signed off. This is a straightforward solution, but if the policy expands, an environment with a large number of such Boolean variables could become complex to maintain and update. This becomes more important when we come to test the policy in environments with different values. We therefore add a variable called `signed_off`, of type `BoolArray`, to the environment. It is instantiated in the dynamic environment `denv` as a mapping from documents to Booleans. In the example environment given we populate it with the map {`haz_an |-> true, pp |-> false`}.

```
signed_off : FExp`Var = mk_FExp`Var(mk_token("signed_off")),

env : Env = new Env({...
                     signed_off |-> <BoolArray>},
                    {...
                     signed_off |->
                       {haz_an |-> true, pp |-> false}})
```

Rule 1 applies to any subject trying to write to the production plan, so we introduce a symbol `all_subjects` to represent the set of all subjects in the rule target. This is initialised to {`Anne,Bob`} and allows us to change the set of test subjects without having to change the rule target. The XACML convention is that if the target field is empty it applies to all possible subjects; we allow the developer to use that convention but retain `all_subjects` for clarity.

```
all_subjects : set of PAP`Subject = {Anne,Bob},

hazanBeforePPRule: PAP`Rule =
 mk_PAP`Rule(
   mk_PAP`Target(all_subjects,{pp},{write}), <Deny>,
                 new FExp(mk_FExp`FUnary(<NOT>,
                     mk_FExp`FArrayLookup(signed_off,haz_an))))
```

In order to evaluate Rule 2, it must be possible at any time to determine the authorship of any document. To support this, a mapping `author` is added to the

environment as a `VarArray` which has type `map Var to (map Var to bool)`. It is instantiated in the dynamic environment as a map from documents to people to Booleans:

```
author : FExp`Var = mk_FExp`Var(mk_token("author")),

environment1 : Env = new Env({author     |-> <VarArray>,
                              signed_off |-> <BoolArray>},
    {author |-> {haz_an |-> {Anne |-> false, Bob |-> true},
                 pp     |-> {Anne |-> true, Bob |-> false}},
      signed_off |-> {haz_an |-> true, pp |-> false}})
```

Rule 2 itself is a separation of duty constraint applying to all requests to `review` a document, and denying any author the ability to review their own document. This demonstrates the need to vary the result of a rule evaluation depending on the relationship between the requester and the resource. It therefore makes use of both the `resource` and `requester` reserved words. In order to evaluate the effect of this rule, the evaluator must instantiate both of these variables via the `inst` mapping.

```
requester : FExp`UnVar = mk_FExp`UnVar(<requester>),
resource  : FExp`UnVar = mk_FExp`UnVar(<resource>),

reviewRule : PAP`Rule =
   mk_PAP`Rule(mk_PAP`Target(all_subjects,{haz_an,pp},{review}),
           <Deny>,
     new FExp(mk_FExp`FVarArrayLookup(author,resource,requester)))
```

Following the XACML structure, these two rules are combined in a policy within a PAP. Both the policy and the PAP have a `denyOverrides` rule combining algorithm, so, unless a request is specifically permitted by the rules, it will be denied. Formally:

```
CompanyPolicy : PAP`Policy =
 mk_PAP`Policy(mk_PAP`Target(all_subjects,{haz_an,pp},{review,write}),
       {hazanBeforePPRule,reviewRule},
        <denyOverrides>),

pap : PAP =  new PAP({CompanyPolicy}, <denyOverrides>)
```

4.2 Testing and Modifying Rules

To build confidence in the design of the policy, a number of tests are created. Each test is a single request which may be made of the policy. The developer evaluates the policy with respect to these tests, and decides if the results of the tests correspond to his or her understanding of the requirements. The testing process can be repeated in several environments.

For example, we might define a second test environment (`environment2`) in which authorship of each of the documents is as in `environment1`, but `haz_an` has not been signed off:

```
environment2 : Env = new Env({author     |-> <VarArray>,
                              signed_off |-> <BoolArray>},
            {author |-> {haz_an |-> {Anne |-> false, Bob |-> true},
                         pp     |-> {Anne |-> true, Bob |-> false}},
             signed_off |-> {haz_an |-> false, pp |-> false}})
```

Table 1 shows the results of evaluating various test requests in each of these two environments. At this stage the two given rules have been designed, but it is clear that we are some way from a comprehensive access control policy.

Request	Environment 1	Environment 2
`Request(Anne,haz_an,{write})`	`<NotApplicable>`	`<NotApplicable>`
`Request(Anne,haz_an,{review})`	`<NotApplicable>`	`<NotApplicable>`
`Request(Bob,haz_an,{write})`	`<NotApplicable>`	`<NotApplicable>`
`Request(Bob,haz_an,{review})`	`<Deny>`	`<Deny>`
`Request(Anne,pp,{write})`	`<NotApplicable>`	`<Deny>`
`Request(Anne,pp,{review})`	`<Deny>`	`<Deny>`
`Request(Bob,pp,{write})`	`<NotApplicable>`	`<Deny>`
`Request(Bob,pp,{review})`	`<NotApplicable>`	`<NotApplicable>`

Table 1. Results of request evaluation in two environments.

The kind of error that is usually highlighted in this way is *policy incompleteness* – there are legitimate requests for which the policy has no rule. These are the requests for which the policy returns `<NotApplicable>`.

We could use this approach to highlight inconsistencies between rules within a policy by applying the same set of tests to two different rules. Note, however, that an inconsistency – one rule returns `Permit>` where another returns `<Deny>` – is not necessarily a cause for concern. It is relatively common in access control policies to have general rule with specific exceptions, and the combining algorithm takes care of the resolution.

In Table 1 no permission is given for any requests: they are all either denied or out of scope of the policy, returning the effect `<NotApplicable>`. This highlights a choice for the developer: what is to be the default behaviour of the Policy Enforcement Point if `<NotApplicable>` is returned from the policy evaluation? The two choices are *permit-biased* and *deny-biased*. If the PEP is permit-biased it will permit requests for which the policy returns `<NotApplicable>`, and rules must be designed to deny all requests that should be forbidden. An alternative is a *deny-biased* PEP, which treats `<NotApplicable>` as `<Deny>`. In this case a policy must be designed to permit all requests which are to be allowed.

If the developer wishes to make the behaviour of the policy independent of the style of the PEP, they must ensure that the policy does not return `<NotApplicable>` for any requests which match the policy target. In this example, we pursue this latter option. This is done by including a general permission rule (`permitRule`) in the policy. This permits any requests to write

and review the two documents. When combined with the other rules using `<denyOverrides>`, this means that requests not specifically denied by the other rules in the policy will be permitted. The `permitRule` has no expression in the condition clause, because it is not conditional on any aspect of the environment:

```
permitRule: PAP`Rule = mk_PAP`Rule(mk_PAP`Target(
              all_subjects,{pp,haz_an},{write,review}),
              <Permit>,nil)
```

The `CompanyPolicy` is altered to include the new rule:

```
CompanyPolicy : PAP`Policy =
   mk_PAP`Policy(mk_PAP`Target(all_subjects,{haz_an,pp},{review,write}),
         {hazanBeforePPRule,reviewRule,permitRule},
          <denyOverrides>)
```

Testing these three rules in each of our environments has the anticipated results: each `<NotApplicable>` is set to `<Permit>` (Table 2).

Request	Environment 1	Environment 2
Request(Anne,haz_an,{write})	<Permit>	<Permit>
Request(Anne,haz_an,{review})	<Permit>	<Permit>
Request(Bob,haz_an,{write})	<Permit>	<Permit>
Request(Bob,haz_an,{review})	<Deny>	<Deny>
Request(Anne,pp,{write})	<Permit>	<Deny>
Request(Anne,pp,{review})	<Deny>	<Deny>
Request(Bob,pp,{write})	<Permit>	<Deny>
Request(Bob,pp,{review})	<Permit>	<Permit>

Table 2. Test results for policy independent of PEP bias.

4.3 Environment Modification

Another problem highlighted by the test results is that requests by both Anne and Bob to write to `haz_an`, and by Bob to review `haz_an`, are permitted, even though it has been signed off. This is an inadequacy of the rules as given, which do not cover that explicitly. To deal with this, we could propose a rule which refuses any action on a document after it has been signed off, for example:

```
NoActionAfterSignoffRule : PAP`Rule =
   mk_PAP`Rule(mk_PAP`Target(all_subjects,{haz_an,pp},{review,write}),
          <Deny>,
     new FExp(mk_FExp`FArrayLookup(signed_off,resource)))
```

Let us assume that, through analysing the test results, the developer realises a further requirement

3. For each document, writing and reviewing phases must not overlap.

Implementing this rule will require a new environment variable, a Boolean array (`completed`). For each document, this is to be set to `true` when the writing phase is over and it is ready for review. The following two rules will keep the writing and reviewing phases separate:

```
NoWriteIfCompletedRule : PAP‘Rule =
   mk_PAP‘Rule(mk_PAP‘Target(all_subjects,{haz_an,pp},{write}),
           <Deny>,
   new FExp(mk_FExp‘FArrayLookup(completed,resource))),

NoReviewUntilCompletedRule : PAP‘Rule =
   mk_PAP‘Rule(mk_PAP‘Target(all_subjects,{haz_an,pp},{review}),
           <Deny>,
   new FExp(mk_FExp‘FUnary(<NOT>,
         mk_FExp‘FArrayLookup(completed,resource))))
```

These three rules can be combined into a separate policy:

```
CompletionPolicy : PAP‘Policy =
  mk_PAP‘Policy(mk_PAP‘Target(all_subjects,{haz_an,pp},{write,review}),
        {NoWriteIfCompletedRule,NoReviewUntilCompletedRule,
         NoActionAfterSignoffRule},
         <denyOverrides>)
```

This new policy can then be added to the PAP:

```
pap : PAP =
  new PAP({CompanyPolicy,CompletionPolicy}, <denyOverrides>)
```

A value for the mapping can be added to the dynamic part of each test environment. In this case we add {`completed |-> {haz_an |-> false, pp |-> false`}}. Testing the new PAP in the two environments gives the results shown in Table 3. In Environment 1, the only requests now permitted are requests by Anne and Bob to write to `pp`. In Environment 2 there is an inconsistency. Requests to write to both `haz_an` and `pp` are allowed. This has happened because `haz_an` has been signed off without being completed. This inconsistency would be unacceptable in the implementation, and an appropriate safeguard would need to be put in place. However, modelling such an inconsistent environment is valuable, because doing so can help to identify environmental assumptions made by the policy.

5 An Application: Access Control Design for Virtual Organisations

We have proposed a semantic model of XACML which serves as a tool for the iterative development of access control policies. One of our motivations for addressing this topic formally has been the need to develop and maintain access

Request	Environment 1	Environment 2
Request(Anne,haz_an,{write})	<Deny>	<Permit>
Request(Anne,haz_an,{review})	<Deny>	<Deny>
Request(Bob,haz_an,{write})	<Deny>	<Permit>
Request(Bob,haz_an,{review})	<Deny>	<Deny>
Request(Anne,pp,{write})	<Permit>	<Permit>
Request(Anne,pp,{review})	<Deny>	<Deny>
Request(Bob,pp,{write})	<Permit>	<Permit>
Request(Bob,pp,{review})	<Deny>	<Deny>

Table 3. Test results in environment augmented with separated phases.

control policies for virtual organisations. In this section, we show how the formal model can be used in this domain.

Virtual organisations (often called dynamic coalitions or virtual enterprises) are opportunistic collaborations that form around business needs and opportunities. Examples include coalitions of companies involved in designing, assessing and manufacturing a new chemical compound or agencies collaborating in an emergency relief scenario. Members of virtual organisations are agents, each having their own resources and policies for controlling access to them. However, within a coalition, some additional access is given to other coalition members in order to achieve the coalition's goal. The members agree a joint workflow for the common task, and adapt their individual access control policies if necessary to accommodate the workflow.

Virtual organisations evolve in real time as members join and leave, and as workflows are modified to adapt to changing business goals. At each step in the evolution, each member must make a trade-off between the risks and the benefits of modifying their access control policies to deal with the changed workflow or coalition membership. A source of complexity in this process is the need to determine the consistency of policy modifications with the member's own policies.

The executable formal model can be used to assist trade-off analysis by providing designer feedback on potential policy modifications. The process is illustrated in Fig. 3. An Access Control policy generator (which could be an off-the-shelf tool such as [5]) generates access control requirements from a workflow. These are split into sub-policies, one for each coalition member. These sub-policies represent the access policy that each member needs to deliver in order to satisfy the needs of the coalition. The translator (as demonstrated in [4] for context-free XACML) now translates the proposed policies into VDM++. These are then combined with the formal model of the current access control policy thus, for each member, giving a formal model of the new policy each member must enforce if they accept the workflow.

Each coalition member now has to decide independently if the proposed adaptions to their current policy are acceptable. They are faced with two questions: first, "What access privileges to my resources need to be granted to other

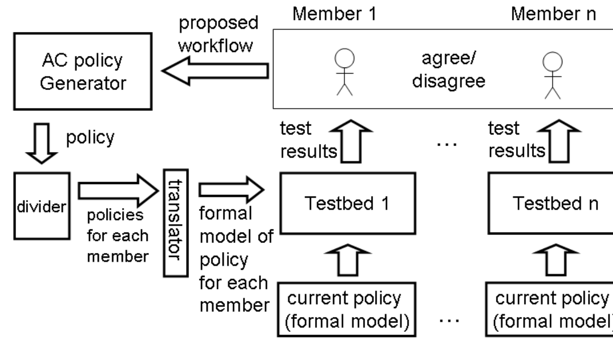

Fig. 3. Evolution of access control policies in virtual organisations.

members in the course of executing this workflow?" and second, "Will adding these new privileges to my existing access control policy violate my own security policy?"

These questions can be answered using the VDMTools interpreter. The tests are derived from the member's own information security policy (a high-level description of the company information security policy). Deriving these tests will take understanding and insight (although it need only be done when the information security policy changes) but performing them and understanding the results should be more straightforward, especially if the results are presented through a GUI interface.

Any failures are presented to the decision maker. If any member decides that the policy is unacceptable, they can do more than simply signal disapproval. For every test that fails they may use the formal model to investigate precisely why it failed, and which particular parts of the access control policy were invoked. They can use this information to propose alternative workflows to the collaboration. Once all members agree on a workflow, they add the new access control policy to their current policies and may then begin to execute the workflow.

As an example, consider the scenario developed earlier, but suppose that company A outsources the hazard analysis to company B, thus forming a style of coalition. Further, suppose that legislation requires that the author of the hazard analysis verify that the production plans implement any recommendations made. However, when a workflow capturing these requirements is proposed and tested by both companies, the first company discovers that the production plan is classified as commercially sensitive, and as such only company employees may see it. The onus is now the first company to propose a new workflow. Note that the tool highlights the issue but it does not enforce a particular solution.

6 Concluding Remarks

We have presented a VDM++ model for a substantial subset of XACML and shown how this can be used for developing context-sensitive access control policies. The utility of the approach has been demonstrated on a small but realistic example development of an access control policy fragment, showing how VDM-Tools can be used to analyse prototype policies and provide rapid feedback to the developer. In particular, the modelling and testing process helps to drive out requirements on both the policy and environment to be implemented. We have also outlined the value of the formal model in the development and evolution of access control policies in virtual organisations.

Current research addresses a variety of semantic models of context-sensitive access control policies. Constraint logic programming in Datalog has been used in several formalisations of access control in general(e.g. [6]) and of specific policy languages such as SecPAL [2]. Few, if any, of these formalisations deal directly with XACML. The policy modelling formalism *RW* [18] may be translated to XACML. Its semantics permit model checking with respect to a goal involving a combination of reading and manipulating propositional variables. Model checking of access control policies derived from XACML has also been demonstrated in Alloy [11], where the authors present several ordering relations between policies and use these to specify policy properties. Event systems have been proposed as a semantic model for dynamic access control policies in [15], where the authors show how to prove that an access control policy refines a given event system specification. A Haskell semantics of XACML 1.1 [12] covers more of the expression part of the language than we have done here for XACML 2.0, our focus being on the exploitation of the model in the policy design process.

The goal of our current work is to assist the access control policy developer by providing rapid feedback on design decisions. How far have we gone towards achieving this? Our approach has a formal basis but the goal is pragmatic and this has led to the use of a tool-supported model-oriented formalism and a test-based approach to model exploration. The use of a model-oriented formalism such as VDM++ makes it relatively easy to build a formal model that has a very similar structure to the XACML framework, easing translation of the validated policy model into its implementation. The model-oriented formalism also makes validation by testing straightforward. We have made this a priority as we wish to support evolution of models that may already have legacy test sets and we also want to have a low technical barrier to the use of the formal model.

The formal model that we have developed is expressed in a notation that will not be familiar to the majority of security engineers. It would be preferable if users could invoke the model through a tailor-made interface that allows the definition of XACML rules, policies and environments and allows the execution of the model on test requests. It is straightforward to construct such an interface using the VDMTools API, allowing the model to be exercised directly by the user without them having to face the formalism directly.

Not all of the XACML Standard [16] is so far modelled, but many of the remaining extensions are technically straightforward, including the extension to

cover the remaining four policy combination algorithms and a capability for the policy developer to define their own combination algorithms. Requests can be extended to encompass sets of subjects, resources and actions. When returning an effect to the PEP, an XACML policy may also return *obligations* which must be carried out by the PEP in addition to enforcing the effect. Such obligations can modify the environment. For example, a policy might permit a user to log on if the number of unsuccessful attempts is suitably small, and oblige the PEP to increment this number with every denied request. Allowing the policy to modify its own environment directly will allow us to model the ability of a user to modify other users' access rights.

Currently our policy validation approach is based on testing. We would like to extend this to include the possibility of proving that policies meet key information security objectives. Automating this would utilise on proof technology for VDM which is currently being developed.

As remarked in Section 5, the semantics and tool developed so far allow the decision maker to see the implications of changes to access control policies. We are deliberately neutral about the surrounding processes which are the province of domain experts. These include the determination of environment models and the negotiation of resolutions to defects in policies for particular applications. The synthesis of specifications based on explicit assumptions about the surrounding computational and real-world environments [14] is an area of active research. The development of a tools framework for more domain-specific policy design is a natural next step for evaluating and tuning the formal engineering approach.

We would like to extend the work done so far to allow the user to explore the consequences of access control decisions on information flow in virtual organisations. In [3] we present a way of formally modelling a range of virtual organisations in order to investigate information flow properties between the members. Combined with the work presented here, this would allow the decision maker to ask "what-if" questions based on possible future scenarios. Here again, an intuitive interface, via the tool API, will be important.

Acknowledgments: We are grateful for support from the GOLD project and particularly Panos Periorellis. We are also grateful to our colleagues from Dstl, in particular Tom McCutcheon, Helen Phillips and Olwen Worthington. We acknowledge the many helpful comments of the anonymous reviewers.

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. Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. SecPAL: Design and semantics of a decentralized authorisation language. Technical Report MSR-TR-2006-120, Microsoft Research, September 2006.
3. Jeremy W. Bryans, John S. Fitzgerald, Cliff B. Jones, and Igor Mozolevsky. Formal modelling of dynamic coalitions, with an application in chemical

engineering. In *2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. IEEE, 2006. To appear. Also available as Newcastle University Technical Report CS-TR-981.
4. Jeremy W. Bryans, John S. Fitzgerald, and Panos Periorellis. Model based analysis and validation of access control policies. Technical Report CS-TR-976, Newcastle University, School of Computing Science, July 2006.
5. Dulce Domingos, António Rito-Silva, and Pedro Veiga. Authorization and access control in adaptive workflows. In Dieter Gollmann and Einar Snekkenes, editors, *Computer Security - ESORICS 2003*, volume 2808 of *Lecture Notes in Computer Science*, pages 23–38. Springer-Verlag, 2003.
6. Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *Proceedings of the International Joint Conference on Automated Reasoning*, August 2006.
7. 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.
8. John S. Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Developement*. Cambridge University Press, 1998.
9. John S. Fitzgerald and Peter Gorm Larsen. Triumphs and challenges for the industrial application of model-oriented formal methods. In *2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. IEEE, 2006. To appear. Also available as Newcastle University Technical Report CS-TR-999.
10. John S. Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. 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. Polar Humenn. The formal semantics of XACML. available at http://lists.oasis-open.org/archives/xacml/200310/pdf00000.pdf.
13. Cliff B. Jones. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice-Hall, 1990.
14. Cliff B. Jones, Ian J. Hayes, and Michael A. Jackson. Deriving specifications for systems that are connected to the physical world. In Jim Woodcock, editor, *Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occassion of their 70th Birthdays*, volume 4700 of *Lecture Notes in Computer Science*, pages 364–390. Springer Verlag, 2007.
15. Dominique Méry and Stephan Merz. Event systems and access control. In Dieter Gollmann and Jan Jürjens, editors, *6th International Workshop on Issues in the Theory of Security*, pages 40–54, Vienna, Austria, 2006. IFIP WG 1.7, Vienna University of Technology.
16. OASIS. eXtensible Access Control Markup Language (XACML) version 2.0. Technical report, OASIS, Feb 2005.
17. Overture Group: The VDM Portal. *http://www.vdmportal.org*. 2007.
18. Nan Zhang, Mark D. Ryan, and Dimitar Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 2007. In Print.