Reasoning about XACML Policies using CSP

Jeremy Bryans
School of Computing Science
University of Newcastle upon Tyne, UK
Jeremy.Bryans@newcastle.ac.uk

ABSTRACT

In this work we explore the use of process algebra in formalising and analysing access control policies. We do this by considering a standard access control language (XACML) and show how the core concepts in the language can be represented in CSP. We then show how properties of these policies may also be described in CSP, and how model checking may be used to verify that a policy meets the property.

We further consider how we may introduce a notion of workflow into this framework, and show that a simple appreciation of the workflow context may limit the things we need to verify about a policy.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification — *formal methods*; D.3.2 [**Programming Languages**]: Language Classifications — specialized application languages; D.4.6 [**Operating Systems**]: Security and Protection — access controls

General Terms

Security, Theory, Verification

Keywords

Access Control, XACML, CSP, Semantic Models

1. INTRODUCTION

Through the use of web services, web-enabled businesses are able to respond quickly to emerging market opportunities by combining to form Virtual Organisations (VOs) designed precisely to meet these opportunities. These VOs operate in many repsects as an ordinary business organisation: in bidding for market share and in designing, developing and marketing a product.

Although not as tightly coupled as a single organisation with several departments, the VO partners must move from mutual mistrust to having an increased trust and reliance in each other's business processes. Individuals in one partner may be allowed access to sensitive material in another partner's database, or a joint VO database may be set up for shared data, and the access policy of that database must be consistent with the policies of the VO partners. When this happens, partners need to be assured that the VO policies are consistent with their own internal policies, and that the combination of policies does not introduce any unwanted behaviour.

We present a method for verifying the behaviour of access control policies, and for comparing access control policies with each other. Taking XACML as our example, we show that many of the core concepts of access control can be represented concisely in the process algebra Communicating Sequential Processes (CSP). By assigning an already well understood semantic model to XACML, we are able to take advantage of the associated theory, as well as its model checker FDR.

We then show how we can check policies against desired properties, and thereby demonstrate that the policies uphold the properties. We can also compare policies with each other to demonstrate, for example, that all behaviours forbidden by one policy are also forbidden by the other.

A VO can form around an agreed *workflow*, where each partner has a clear understanding of the tasks it is to perform within the consortium. It may be that these tasks do not require any access control decisions over which the policies conflict. We show how, in a suitably restrictive workflow context, certain conflicts between policies may be ignored.

Communicating Sequential Processes (CSP) [5, 9, 13] is a well established formal method that has already proved highly effective in security applications, in particular the analysis of security protocols [12] and the formulation of security policies [10, 11]. The work presented here builds on the work reported in [11]. In Section 2 we briefly review the parts of CSP which we shall need.

XACML [8] is the OASIS standard for access control policies. It provides a XML-based language for describing role-based access control policies, and a language for checking the legality of actions with respect to a policy. In Section 3 we present the elements of XACML which we consider.

In Section 4 we show how to capture XACML policies using CSP, and how we model the request/response behaviour of a policy. In Section 5 we demonstrate the approach using an extended example, taken from [3]. In Section 6 we show how two access control policies may be compared, and in Section 7 we show how the concept of workflow may be incorporated into this work, and how even a simple abstraction of a workflow may be make some of the conflicts between access control policies irrelevant.

2. CSP

Here we briefly introduce the elements of CSP that we shall use: more detail can be found in [9, 13]. A CSP description of a system contains *processes* which can perform *events*. We adhere to the convention that processes are written in upper case, events in lower case, and names of sets of events begin with a capital letter.

$STOP$ is the trivial process which can perform no events. $SKIP$ is the process which successfully terminates, and then behaves as $STOP$. This is used when processes are to be executed in sequence. The process $P; Q$ will behave as process P until P successfully terminates, and then process Q will begin. The process $a \rightarrow STOP$ will perform the event a, then behave as $STOP$. We will use the *channel* notation: events may take the form $c.a$ where c is a channel and a is a value drawn from the channel type. Channel types may be compounded: if c has the type $A \times B$, then $c.a.b$ represents an event $a.b$ on channel c, where $a \in A$ and $b \in B$.

If P and Q are CSP processes, then the process $P \sqcup Q$ may behave as P or Q and the choice is made by the external environment. The process $P \sqcap Q$ may also behave as P or Q, but here the choice is made internally, and the environment has no control. We will also used the parameterised versions of internal and external choice: $\Box\, a : A \bullet a \rightarrow P(a)$ offers the environment a choice of the events from set A, and $\bigsqcap a : A \bullet a \rightarrow P(a)$ resolves the same choice internally.

$P \mid\mid\mid Q$ is the interleaving of P and Q, and each component process proceeds independently. $\mid\mid\mid\, a : A \bullet P(a)$ allows $card(A)$ processes to proceed independently, each parameterised by a different value from A.

If A is a set of events, the process $P \,|[\, A \,]|\, Q$ need only synchronise on events from A. $P \setminus A$ behaves as P, except that events from A are hidden from the environment.

RUN_A is the process which is always ready to perform any event from the set A. It is defined as $RUN_A = \Box\, a : A \rightarrow RUN_A$. If A is understood, then $STOP_{\{a\}}$ is the process which blocks the event a, but is always ready to perform any other event from A. It is defined as $STOP_{\{a\}} = RUN_A \,|[\, \{a\} \,]|\, STOP$.

CSP has various theories of process equivalence associated with it. The most simple of these, *trace equivalence*, is sufficient for us here. In it a process P is identified with the set of traces that it can perform (given by $tr(P)$), and P and Q are equivalent if and only if $tr(P) = tr(Q)$.

We also use the notion of *refinement*. P is a (trace) refinement of Q (written $P \sqsubseteq_{tr} Q$) when the set of traces it can perform is a subset of those of Q. Thus

$$P \sqsubseteq_{tr} Q \Leftrightarrow tr(Q) \subseteq tr(P)$$

Trace refinement is sufficient for capturing *safety* properties. If Q is all the safe behaviours, then any refinement P of Q will only have safe behaviours. No guarantees are made about liveness, however. In particular, the process $STOP$ is a refinement of all other processes. For our present task safety properties will suffice, and we will use $\sqsubseteq$ for $\sqsubseteq_{tr}$.

3. XACML AND ACCESS CONTROL

XACML [8] is an OASIS standard that describes (in XML) both a policy language and an access control decision language. The decision language allows access control *requests* to be made of a policy, and provides a *response* based on the policy.

Every XACML policy will have an associated *Policy Decision Point* (PDP) and a *Policy Enforcement Point* (PEP). Requests are formed in the decision language by the PEP and sent to the PDP. The PDP implements the policy under consideration, and requests are either *permitted* or *denied* (unless some error occurs), and the decision is returned to the PEP. The PDP may query the context to learn the value of environmental attributes, and make its decision based on these values. In this work we will focus on the interaction between the PDP and the PEP.

A request is from a *subject* (e.g. a user or a process) to perform an *action* (e.g. read, write, copy) on a *resource* (e.g. a file or a disk) within an environment (e.g. during work hours or from a secure machine).

If a subject requests permission for an action on a resource in an environment, a policy will return exactly one of

- `permit` if the subject is permitted to perform the action on the resource in the environment,
- `deny` if the subject is not permitted to perform the action on the resource in the environment,
- `Indeterminate` if some error occurs, or
- `NotApplicable` if the request can't be answered by the service.

Rules in XACML contain a *target* (which further contains resources, subjects, actions, and the environment of the rule), an *effect* (permit or deny) and a *condition* on the application of the rule. In this paper we restrict our attention to targets and effects. We consider the environment in Section 4.4.

Policies comprise sets of rules, and a rule combination algorithm. Like rules, they also contain a target (again resources, subjects, actions and environment). They also contain a notion of *obligation*. These obligations are a separate responsibility of the PEP. Since we are focusing on the interaction between the PDP and the PEP, we do not consider them further.

4. XACML AND CSP

We form the CSP model of an access control policy by beginning with an unconstrained model of the Policy Decision Point. We then compose the constraints that represent the rules, in a way which remains true to the particular rule combining algorithm used by the policy.

The Policy Enforcement Point is modelled as a process PEP which can attempt to synchronise with PDP on any event from the shared interface. These attempted synchronisations represent requests. If the synchronisation occurs, the request is considered to be granted. If it fails, the request is considered to be refused. Responsibility for avoiding deadlock lies entirely with the PEP, which is also the real-world situation.

The CSP model will thus look like

$$PEP \mid[\, Interface \,]\mid PDP$$

Strictly speaking, a request from the PEP can be permitted under two different circumstances in this model: by not including it in the set *Interface*, (and by default allowing it to proceed unhindered) or explicitly allowing it within *PDP*.

`NotApplicable` is the result returned by a policy, if the policy has nothing to say (either in favour or against) about a particular request. We assume that all requests about which the PDP has rules are in *Interface*, and so we do not model `NotApplicable` as a separate return from the PDP.

`Indeterminate` is returned if, for example, an error occurs within the environment when evaluating a rule. In our model we do not go outside the PDP and PEP, and therefore consider that each synchronisation request is either accepted or refused.

Our focus here is on the interaction between the PDP and the PEP, so we will not develop an explicit environment process here. However this model could easily be extended to include one. We show in Section 4.4 how simple environmental behaviour may be encapsulated within the PDP process.

4.1 Modelling requests

Following the approach taken in [11], actions, subjects and resources are straightforwardly modelled using compound channels. We take the name of the action to be the name of the channel, and the type of the channel to be *Subject* × *Resource*. Thus, the channel defined

$$channel\ \ read : Subject.Resource$$

denotes a channel called *read* with the type *Subject.Resource*, and events on this channel take the form $read.s.r$, where s and r are members of the sets *Subject* and *Resource* respectively.

4.2 Modelling roles

A *role* is a set of rights and obligations that a user may choose to assume. Users may be entitled to a number of roles, and may swap between them. Typically, although a user may hold a number of roles simultaneously, any request must be with respect to a single role. Requests to perform actions will be evaluated by the policy in the context of the current role of the user. In [8] roles are represented as attributes.

Following [11], we introduce roles into our model of requests by enriching the space of events. If *Role* is the set of all roles known to the policy, then the channel definition

$$channel\ \ read : Subject.Role.Resource$$

includes the role that a subject holds when making a read request on some resource. So, for example, the event $read.anne.superuser.file1$ represents the request by *anne* as *superuser* to read *file*1.

4.3 Modelling single rules

We consider adding to a policy a rule that relates to a request by subject s to perform action *read* on resource r. $RULE1$, which permits this request, is then defined:

$$RULE1 = read.s.r \rightarrow RULE1$$

A rule which denies a request can be captured in the CSP model simply by including the request event in the interface to the policy, and ensuring that the policy says nothing further about this request. However, it is possible (and illustrated in Section 4.5) that a policy contain two rules, both of which refer to the same request. In this case, it becomes important to retain an explicit CSP representation of each rule, in order to accurately reproduce the semantics of the rule combining algorithm. We therefore choose to capture the rule which denies $read.s.r$ as:

$$RULE2 = RUN \mid[\, \{read.s.r\} \,]\mid STOP$$

which is the process that permits any requests except $read.s.r$.

To allow a subject $s1$ to *read* any member of the set *Resource*, we write

$$RULEANY = \Box\, r : Resource \bullet read.s1.r \rightarrow RULEANY$$

and we can similarly allow a choice of subjects.

4.4 Capturing the environment

In the XACML specification [8], the *environment* is defined as the set of attributes that are relevant to an authorisation decision and independent of a particular subject, resource or action. Thus far we have presented rules which either always permit or always deny a request, and are independent of their environment. Simple environment-sensitive rules can be formed using processes similar to the above, as well as the CSP sequencing and choice operators. For example, consider a rule which permits access during work hours, and forbids it otherwise. To encode this, we first introduce two new events into the alphabet of the PDP, *start-of-day* and *end-of-day*. The rule can then be written as two mutually-recursive processes:

$$\begin{array}{rcl} OUTOFHOURS & = & start\_of\_day \rightarrow WORKHOURS \\ WORKHOURS & = & read.s.r \rightarrow WORKHOURS \\ & & \Box \\ & & end\_of\_day \rightarrow OUTOFHOURS \end{array}$$

This can then be incorporated into the PDP as an ordinary rule, using *OUTOFHOURS* if the *PDP* begins operation outside of normal working hours, and *WORKHOURS* otherwise.

More complex environmental conditions can be formed as combinations of this type of rule, although a more comprehensive modelling of environment behaviour will probably require a separate environment process.

4.5 Combining rules into policies

There are a number of rule combining algorithms in the XACML standard. We consider as exemplars the `deny-overrides` and the `permit-overrides` algorithms, [8]. The `deny-overrides` algorithm denies a request if any one rule in the PDP denies it. The `permit-overrides` algorithm permits a request if any one rule in the PDP permits it. Different algorithms will result in different overall processes, even if they begin with the same set of rules.

There is a very close fit between the CSP parallel combinators and the two `*-overrides` algorithms. `deny-overrides` is naturally encoded as a communicating parallel composition of rules, communicating on the request event. `permit-overrides` is naturally encoded as a parallel interleaving combination of rules, with no communication between the rules.

If both $RULE1$ and $RULE2$ refer to the request $read.s.r$, we can combine the rules into a single policy using the `deny-overrides` rule combining algorithm as:

$$RULE1 \,|[\, \{read.s.r\} \,]|\, RULE2$$

Any request for $read.s.r$ will be denied if either $RULE1$ or $RULE2$ deny it. For the request $read.s.r$ to be permitted, both $RULE1$ and $RULE2$ would have to permit the action $read.s.r$. If $RULE1$ and $RULE2$ are defined as in section 4.3, then all requests from PEP to perform the action $read.s.r$ will be denied, because $RULE2$ denies it. Care must be taken when populating the interface set between rules, as unless both rules explicitly allow actions in the interface set they will be denied.

The behaviour of the `permit-overrides` algorithm is also easily described, this time using CSP interleaving as:

$$RULE1 \;|||\; RULE2$$

This time, any request to perform $read.s.r$ will be permitted.

Other algorithms are discussed in [8]. An interesting one is `first-applicable`, where the first applicable rule is used to answer any query. This would prove much more of a challenge to describe in CSP, because the CSP parallel operators are commutative.

Other rules in [8] include `ordered-deny-overrides`, and `ordered-permit-overrides`. These differ from their unordered variants only in the order of the evaluation of the rules. In particular, they will return identical values to their unordered variants, at the level of abstraction we use here.

In XACML, policies may be collected into sets, and these sets may be combined according to different algorithms. Within CSP we would combine policies in the same way that we combine rules, by using appropriate choices of the parallel composition operators.

5. INTRODUCTORY EXAMPLE

Our example is derived from the example given in [3], which describes the access control requirements of a university database which contains student grades. We show how specify the appropriate policy at each stage. We then show how properties of these policies can be formally defined and checked using the CSP model checker FDR [7], and how we can use this to discover mistakes and oversights in the policies.

There are n users of the database, each of which may take the role *faculty* or *student*. The database contains *internal* and *external* grades, and the relevant actions are *receive*, *assign* and *view*. We begin with the following datatype and channel declarations:

$$\begin{array}{ll} datatype & User = \{1 \ldots n\} \\ datatype & Role = faculty \mid student \\ datatype & Grade = internal \mid external \\ channel & assign, receive, view : User.Role.Grade \end{array}$$

All the possible events on this database are given by $Allevents$:

$$\begin{aligned} Allevents \;=\; & \{assign.u.r.g, view.u.r.g, receive.u.r.g \mid \\ & \quad u \leftarrow User, r \leftarrow Role, g \leftarrow Grade\} \end{aligned}$$

We begin by defining the unconstrained database policy as the CSP process D_BASE which allows any of these actions. We then add the constraints incrementally.

$$\begin{aligned} D\_BASE \;=\; & \Box\, u : User \bullet (\Box\, r : Role \bullet (\Box\, g : Grade \bullet \\ & \quad (view.u.r.g \rightarrow D\_BASE \\ & \quad \Box \\ & \quad assign.u.r.g \rightarrow D\_BASE \\ & \quad \Box \\ & \quad receive.u.r.g \rightarrow D\_BASE))) \end{aligned}$$

The policy imposed on this database is

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

We describe this policy by defining processes which capture the effect it has on users. The same policy applies to each user, so we first describe the policy as it applies to a single user, parameterised by i.

$$\begin{aligned} USER(i) \;=\; & receive.i.student.ext \rightarrow USER(i) \\ & \Box \\ & \Box\, g : Grade \bullet assign.i.faculty.g \rightarrow USER(i) \\ & \Box \\ & \Box\, g : Grade \bullet view.i.faculty.g \rightarrow USER(i) \end{aligned}$$

Any user in the role of student may receive external grades, and any user in the role of faculty may assign all grades, and view all grades. The interleaved combination of all these policies gives us our model of the policy.

$$ALL\_USERS \;=\; \big|\big|\big|\, u : User \bullet (USER(u))$$

The user policy runs in parallel with the database policy, communicating on $Allevents$. PDP is thus described as:

$$PDP = ALL\_USERS \,|[\, Allevents \,]|\, D\_BASE$$

To validate a model, we attempt to prove properties (or *validation conjectures*) which we believe to be true. The following property is found in [3].

Property 1. *There do not exist members of Student who can assign external grades.*

Following [11], we begin by defining a set of events $Breach1$. The occurrence of any of these events indicates a violation of Property 1.

$$Breach1 \;=\; \{assign.u.student.ext \mid u \rightarrow User\}$$

The property is then proved by asserting that no event from the set $Breach1$ can occur.

$$PDP \,|[\, Breach1 \,]|\, STOP \;\sqsubseteq\; PDP$$

FDR quickly tells us that this assertion holds. This means that when PDP is unable to perform any events from $Breach1$, it still has at least as many traces as the unrestricted PDP. In fact, we are only removing possible events by putting PDP in parallel with $STOP$, so the two processes above have the same traces. Explicitly forbidding all events from the set $Breach1$ makes no difference to the behaviour of PDP.

5.1 Enforcing roles within the PDP

Going back to property 1, we must be careful to distinguish between two different interpretations of this property. We could be referring to *user in the role of a student* or *user who is able to hold the role of a student*. The distinction becomes important if it is possible for a user to take different roles at different times. We therefore make the property more precise, as:

PROPERTY 2. *No user who has previously assumed the role of Student can assign external grades.*

To validate this property, we need to redefine the PDP to track previous user behaviour. We allow a user to make the choice between acting as a student or a faculty member. The user behaviour is then constrained by the PDP, according to the choice they have made. We introduce two new events:

$$channel\ \ choose\_stu, choose\_fac : User$$

which are used by the PDP to track the choices the users make. ENF_USERS is defined as the parameterised process which forces users to choose between *faculty* and *student*, then allows the users to perform a single action from the chosen role. The users then choose roles again.

$$\begin{aligned}
ENF\_USERS(i) &= choose\_stu.i \rightarrow STUDENT(i);\\
&\qquad ENF\_USERS(i)\\
&\quad \Box\\
&\quad choose\_fac.i \rightarrow FACULTY(i);\\
&\qquad ENF\_USERS(i)\\
STUDENT(i) &= receive.i.student.ext \rightarrow\\
&\qquad SKIP\\
FACULTY(i) &= \Box\, g : Grade \bullet (assign.i.faculty.g \rightarrow\\
&\qquad SKIP)\\
&\quad \Box\\
&\quad \Box\, g : Grade \bullet (view.i.faculty.g \rightarrow\\
&\qquad SKIP)
\end{aligned}$$

The users are combined and a new PDP ($PDP2$) is then built up as

$$\begin{aligned}
ENF\_ALL\_USERS &= \big|\big|\big|\, u : User \bullet (ENF\_USERS(u))\\
PDP2 &= ENF\_ALL\_USERS\\
&\quad |[\, Allevents \,]|\\
&\quad D\_BASE
\end{aligned}$$

A breach of Property 2 corresponds to: for some i an $assign.i.stu.ext$ event is preceded by a $choose\_stu.i$ event. We therefore want to assert that, for any user i, after a $choose\_stu.i$ event the $assign.i.stu.ext$ event cannot happen.

The general approach is as follows. For each user, we define $PERMIT\_A\_UNTIL\_B$ as the process which permits any number of *assign* events until the point where a *choose_stu* event occurs. After the *choose_stu* event, no further *assign* event is permitted. These processes allows only behaviours which meet Property 2. If, when placed in parallel with this process, the behaviour of $PDP2$ is altered, then the unconstrained $PDP2$ must allow an illegal event, and the FDR model checker will return a trace which violates the property.

We define this as follows:

$$\begin{aligned}
&PERMIT\_A\_UNTIL\_B(i) =\\
&\quad \Box\, r : Role \bullet (assign.i.r.ext \rightarrow\\
&\qquad PERMIT\_A\_UNTIL\_B(i))\\
&\quad \Box\\
&\quad choose\_stu.i \rightarrow JUST\_B(i)
\end{aligned}$$

where

$$JUST\_B(i) \quad = \quad choose\_stu.i \rightarrow JUST\_B(i)$$

For all roles, any external grades may be assigned, but as soon as a user performs a $choose\_stu.i$ event, they are no longer allowed to make any assignments to external grades. They are only allowed (by $JUST\_B$) to continue to perform the action $choose\_stu.i$. Now, for any i, the $choose\_stu.i$ event is continually allowed until the first event from the set $\{assign.i.fac.ext, assign.i.stu.ext\}$ occurs.

Finally, the $RESTRICT$ process is defined as

$$RESTRICT \quad = \quad |||\ u : User \bullet PERMIT\_A\_UNTIL\_B(u)$$

and when we place it in parallel with $PDP2$, communicating only on Res_events, where

$$\begin{aligned}
&Res\_events =\\
&\{choose\_stu.u, assign.u.r.ext \mid u \leftarrow User, r \leftarrow Role\}
\end{aligned}$$

and check

$$PDP2\, |[\, Res\_events \,]|\, RESTRICT \sqsubseteq PDP2$$

FDR quickly tells us that this fails to check. The trace returned by the debugger is:

$$\langle choose\_stu.i, choose\_fac.i, assign.i.fac.ext\rangle$$

which shows that a user may choose the student role, then later choose the faculty role and assign external grades.

This issue is the well-known one of *separation of duty*. The problem is that users are allowed to swap in and out of roles, and a student can re-login as a faculty member and acquire all the privileges associated with that. We address this in the next section.

5.2 Separation of duty

Separation of duty is an important aspect of access control, and a possible way to implement that is *history-based authorisations* [2]. We show that is possible in our model. To specify a once-for-all separation of duty constraint, we define (following [11]):

$$\begin{array}{lcl} \mathit{SEP\_USERS}(i) & = & (\mathit{choose\_stu}.i \rightarrow \mathit{STUDENT}(i) \\ & & \square \\ & & \mathit{choose\_fac}.i \rightarrow \mathit{FACULTY}(i)) \end{array}$$

where

$$\begin{array}{lcl} \mathit{STUDENT}(i) & = & \mathit{receive}.i.\mathit{student}.\mathit{ext} \rightarrow \\ & & \quad \mathit{STUDENT}(i) \\ \mathit{FACULTY}(i) & = & \square\, g : \mathit{Grade} \bullet \\ & & \quad (\mathit{assign}.i.\mathit{faculty}.g \rightarrow \\ & & \qquad \mathit{FACULTY}(i)) \\ & & \square \\ & & \square\, g : \mathit{Grade} \bullet \\ & & \quad (\mathit{view}.i.\mathit{faculty}.g \rightarrow \\ & & \qquad \mathit{FACULTY}(i)) \end{array}$$

Users now must choose their role before they can do any other action, and are compelled to remain within their role. The totality of the user behaviour is now given by

$$\mathit{ALL\_SEP\_USERS} \;=\; \big|\big|\big|\; i : \mathit{User} \bullet (\mathit{SEP\_USERS}(i))$$

and the new system is defined as

$$\mathit{PDP}3 \;=\; \mathit{ALL\_SEP\_USERS} \,|[\, \mathit{Allevents} \,]|\, \mathit{D\_BASE}$$

This time the assertion

$$\mathit{PDP}3 \,|[\, \mathit{Interface} \,]|\, \mathit{RESTRICT} \sqsubseteq \mathit{PDP}3$$

holds, demonstrating that separation of duty using a history-based authorisation is a possible method for ensuring property 2.

6. COMPARING POLICIES

To compare two policies $P1$ and $P2$, we can check for trace refinement in each direction. If $P1 \sqsubseteq P2$, the behaviours $P1$ allows are a subset of those that $P2$ allows, and $P1$ is more restrictive that $P2$. If $P1 \sqsubseteq P2$ and $P2 \sqsubseteq P1$, the two processes are equivalent. By hiding events we are not concerned about, we can consider only the relevant subset of events, and compare behaviours with respect to that subset.

To illustrate all this, we extend our policy to include another role and then compare the original to the extended version. Following [3], we first introduce a teaching assistant role. Students may opt to be teaching assistants, and teaching assistants may assign and view internal grades but not external ones. We then show how to compare this policy with the previous one using FDR.

The *Role* datatype is extended to include teaching assistants:

$$\mathit{datatype} \;\; \mathit{Role} = \mathit{faculty} \mid \mathit{student} \mid \mathit{ta}$$

$\mathit{ENF\_USERS}$ enforces the initial choice between student and faculty as before, and $\mathit{EXT\_USERS}$, which we go on to define, allows students to drop into the teaching assistant role. We include an extra channel to mark this choice.

$$\mathit{channel} \;\; \mathit{choose\_stu}, \mathit{choose\_fac}, \mathit{choose\_ta} : \mathit{User}$$

$$\begin{array}{lcl} \mathit{EXT\_USERS}(i) & = & \mathit{choose\_stu}.i \rightarrow \mathit{EXT\_STU}(i) \\ & & \square \\ & & \mathit{choose\_fac}.i \rightarrow \mathit{FACULTY}(i) \end{array}$$

$\mathit{EXT\_STU}$ process can behave as $\mathit{OLD\_STUDENT}$, or make the choice to upgrade to teaching assistant privileges.

$$\begin{array}{l} \mathit{EXT\_STU}(i) = \\ \quad \mathit{receive}.i.\mathit{student}.\mathit{ext} \rightarrow \mathit{EXT\_STU}(i) \\ \quad \square \\ \quad \mathit{choose\_ta}.i \rightarrow (\mathit{assign}.i.\mathit{ta}.\mathit{int} \rightarrow \mathit{EXT\_STU}(i) \\ \qquad\qquad\qquad \square \\ \qquad\qquad\qquad \mathit{view}.i.\mathit{ta}.\mathit{int} \rightarrow \mathit{EXT\_STU}(i)) \end{array}$$

Faculty is as before:

$$\begin{array}{l} \mathit{FACULTY}(i) = \\ \quad \square\, g : \mathit{Grade} \bullet (\mathit{assign}.i.\mathit{faculty}.g \rightarrow \mathit{FACULTY}(i)) \\ \quad \square \\ \quad \square\, g : \mathit{Grade} \bullet (\mathit{view}.i.\mathit{faculty}.g \rightarrow \mathit{FACULTY}(i)) \end{array}$$

The users are combined to form the systems

$$\mathit{ALL\_EXT\_USERS} = \big|\big|\big|\; i : \mathit{User} \bullet (\mathit{EXT\_USERS}(i))$$

$$\mathit{EXT\_PDP} = \mathit{ALL\_EXT\_USERS} \,|[\, \mathit{Allevents} \,]|\, \mathit{PDP}3$$

We compare this policy $\mathit{EXT\_PDP}$ with $\mathit{PDP}3$ by comparing the sets of traces that they both produce. An obvious immediate difference is that $\mathit{EXT\_PDP}$ can perform $\mathit{choose\_ta}.i$ events, but what concerns us here is whether we have any new behaviour with respect to the original set of events, so we hide all $\mathit{choose\_ta}.i$ events. We can now meaningfully compare trace refinement in both directions.

$$\mathit{EXT\_PDP} \setminus \{\mathit{choose\_ta}.u \mid u \leftarrow \mathit{User}\} \sqsubseteq \mathit{PDP}3$$

This check passes, indicating that, excluding the $\mathit{choose\_ta}$ events, no traces forbidden by $\mathit{EXT\_PDP}$ are permitted by $\mathit{PDP}3$.

However, when we try to check:

$$\mathit{PDP}3 \sqsubseteq \mathit{EXT\_PDP} \setminus \{\mathit{choose\_ta}.u \mid u \leftarrow \mathit{User}\}$$

it fails, indicating the presence of a trace allowed by $\mathit{EXT\_PDP}$ which is not allowed by $\mathit{PDP}3$. The trace returned by FDR shows

$$\langle \mathit{choose\_stu}.1, \mathit{choose\_ta}.1, \mathit{assign}.1.\mathit{ta}.\mathit{ext} \rangle$$

pointing out that a student may choose the teaching assistant role, then use this role to assign external grades.

This sort of problem could be dealt with in a number of ways. We have already demonstrated how enforcing separation of duty is one solution, and this could again be used here. In the next section we show how taking the workflow context into account may be a possible solution.

7. INCLUDING WORKFLOW

A possible context in which we may want to combine access control policies is a situation such as a Virtual Organisation (VO), where a number of independent actors want to work together on a project, and each has a security policy they wish to enforce on joint assets of the VO. In a situation like this, there may be many points of conflict between the policies. It is possible, however, the the particular workflow being enacted means that there will be no access requests which cause the conflicting rules to be consulted. If the task the VO was proposing was relatively benign, only the consistent parts of the policies may need to be consulted.

To show how a VO might take advantage of this, in this section we show how to combine a representation of workflow with a policy. We then show how this representation of workflow may limit the points at which we have to consider access control issues.

In order to combine it with a CSP model of a policy, a model of workflow must be a CSP process. Also, the alphabet of this workflow process must overlap with the alphabet of the policy, or it will have no impact.

Workflow is a rich concept. Here, however, we limit our representation of workflow to the impact it has on access control decisions. We are not interested, for example, in what a user must do with a document, simply in whether or not they have access to it.

We therefore *enforce* the workflow in one sense, by insisting that the only access control actions that are taken are those necessary for the workflow. We do not mean enforcement in the workflow sense of ensuring that obligations are met.

As an example, consider a workflow that stipulates that teaching assistants assign internal grades, then faculty members assign external grades, then the students receive their external grades. We let the workflow take account of the role choices, and only complete each subprocess when every role chosen within that subprocess has been rescinded. Consider the definition

$$\begin{aligned} \mathit{WORKFLOW} \;=\; & \mathit{ASSIGN\_INT\_GRADES};\\ & \quad \mathit{ASSIGN\_EXT\_GRADES};\\ & \qquad \mathit{REC\_GRADES};\ \mathit{STOP} \end{aligned}$$

where the components are described below.

$$\begin{aligned} &\mathit{ASSIGN\_INT\_GRADES} =\\ &\qquad \textstyle\bigsqcap i : \mathit{User} \bullet \mathit{choose\_ta}.i \rightarrow \mathit{ASSIGN\_INT}(i)\\ &\qquad \sqcap\ \mathit{SKIP} \end{aligned}$$

$$\begin{aligned} &\mathit{ASSIGN\_INT}(i) =\\ &\qquad \mathit{assign}.i.\mathit{ta}.\mathit{int} \rightarrow \mathit{ASSIGN\_INT}(i)\\ &\qquad \sqcap\\ &\qquad \mathit{drop\_ta}.i \rightarrow \mathit{ASSIGN\_INT\_GRADES} \end{aligned}$$

One (non-deterministically chosen) user at a time may choose the teaching assistant role and assign internal grades. They can continue making assignments or rescind the teaching assistant role, in which case either another user may pick up the role, or the process can successfully complete (by choosing *SKIP*.)

$$\begin{aligned} &\mathit{ASSIGN\_EXT\_GRADES} =\\ &\qquad \textstyle\bigsqcap i : \mathit{User} \bullet \mathit{choose\_fac}.i \rightarrow \mathit{ASSIGN\_EXT}(i)\\ &\qquad \sqcap\ \mathit{SKIP} \end{aligned}$$

$$\begin{aligned} &\mathit{ASSIGN\_EXT}(i) =\\ &\qquad \mathit{assign}.i.\mathit{faculty}.\mathit{ext} \rightarrow \mathit{ASSIGN\_EXT}(i)\\ &\qquad \sqcap\\ &\qquad \mathit{drop\_fac}.i \rightarrow \mathit{ASSIGN\_EXT\_GRADES} \end{aligned}$$

The same sort of process is then gone through for the assignment of external grades by staff.

$$\begin{aligned} &\mathit{REC\_GRADES} =\\ &\qquad \textstyle\bigsqcap i : \mathit{User} \bullet \mathit{choose\_stu}.i \rightarrow \mathit{REC\_EXT}(i)\\ &\qquad \sqcap\ \mathit{SKIP} \end{aligned}$$

$$\begin{aligned} &\mathit{REC\_EXT}(i) =\\ &\qquad \mathit{receive}.i.\mathit{student}.\mathit{ext} \rightarrow \mathit{REC\_EXT}(i)\\ &\qquad \sqcap\\ &\qquad \mathit{drop\_stu}.i \rightarrow \mathit{SORT\_REC}2 \end{aligned}$$

Finally, the students may receive their external grades.

Workflow_interface must include the *choose* and *drop* channels:

$$\begin{aligned} &\mathit{Workflow\_interface} =\\ &\qquad \{\mathit{choose\_stu}.u, \mathit{choose\_fac}.u, \mathit{choose\_ta}.u,\\ &\qquad \mathit{drop\_stu}.u, \mathit{drop\_fac}.u, \mathit{drop\_ta}.u,\\ &\qquad \mathit{assign}.u.\mathit{faculty}.g, \mathit{assign}.u.\mathit{ta}.g,\\ &\qquad \mathit{receive}.u.\mathit{student}.\mathit{ext} \mid\\ &\qquad u \leftarrow \mathit{User}, g \leftarrow \mathit{Grade}\} \end{aligned}$$

and the new policy is then defined as the parallel composition of the original policy (*EXT_PDP*) and the new workflow, communicating on those events that are common to both processes:

$$\mathit{PDP\_WF} \;=\; \mathit{PDP} \,|[\, \mathit{Workflow\_interface} \,]|\, \mathit{WORKFLOW}$$

To check if *PDP_WF* satisfies Property 2, we use the process *RESTRICT* from Section 5.1, which for all users allows external assign events until that user chooses the student role, and then denies all further external assign events. The assertion

$$\mathit{PDP\_WF} \,|[\, \mathit{Interface} \,]|\, \mathit{RESTRICT} \sqsubseteq \mathit{PDP\_WF}$$

checks as expected.

This example is a demonstration of how the workflow provides a context within which the system is secure, despite it being insecure in general. It does this because the sequential nature of the workflow coupled with the restrictions on logging on and off imposes a temporal requirement on when users may be active in a system. In the context of this workflow, it becomes impossible for students to log on before the external grades have been assigned.

8. RELATED WORK

Other efforts have been made to ascribe a formal semantics to XML-based languages. One of the most closely related approaches is [6], where the authors map WS-Security protocols to CSP (via an intermediate protocol notation). Properties of these protocols can then be formally checked using FDR. The underlying formalism (CSP) is the same as in this work, but the focus is on understanding protocols rather than policies.

WS-Security is also formalised in [1], where it is given a formal basis using the pi-calculus. Properties of these protocols can again be formulated and proved. The focus here is on the cryptographic protocols described by WS-Security.

Access control policies are considered in [3], from which the grades example in this paper is taken. Policies written in XACML are translated into Binary Decision Diagrams (BDDs), and thereby given a formal semantics. The *margrave* tool is presented, which implements a number of special purpose BDD procedures, allowing a user to query and compare policies. Queries are performed by limiting policies to attributes of interest, although the authors intend to develop a special-purpose query language.

The work in [4] and [14] considers the problem of generating already verified XACML policies. In [4] the authors define a simple programming language which allows specification and verification of access control policies. A propositional language for describing goals of agents is given, and decision procedure to determine if a goal is achievable with respect to a given policy. In [14] a tool is presented which generates an (already verified) XACML policy from this language. This work easily allows for policies which are conditional [2] on the state of the system.

9. CONCLUSIONS AND FURTHER WORK

We have successfully shown how access control models can be expressed, using the CSP framework and tools. We have therefore been able to use the CSP semantics to ascribe a semantics to XACML, and shown how properties formally verified. Further, we have shown how two access control policies may be compared, and the differences analysed using FDR. We have shown how a primitive notion of workflow can be considered alongside these models, and how the constraints imposed by the workflow may rule out possible security concerns.

Many avenues could be pursued further. We could consider further the notion of *obligation*, whereby a party is required to perform a particular action. Obligations would be expressed as conventional CSP events, but checking obligation properties formally would require consideration of *liveness*, which would require a richer semantic model that the traces used here.

We have limited our consideration of the environment to a simple consideration of time, also separation of duty constraints and workflow. Environment is clearly a much wider concept, and developing our model to include more detailed aspects of environment would provide a rich vein of work. Time could be explicitly modelled in Timed CSP, although tools are not available to mechanically check Timed CSP specifications.

Rules may contain a *condition*, which is evaluated at runtime to calculate the effect of the rule. It would be interesting to see to what extent these conditions can be understood within the CSP framework. This would perhaps stretch the abilities of a process algebra, and might be better described in a language which contains an explicit representation of state.

Acknowledgments

Thanks are due to colleagues who read previous drafts and gave valuable advice, in particular Jim Armstrong, John Fitzgerald, Michael Harrison and Peter Ryan.

This work is part-funded by the UK EPSRC under e-Science pilot project GOLD, also DIRC (Interdisciplinary Research Collaboration in Dependability) and DSTL.

10. REFERENCES

[1] K. Bhargavan, C. Fournet, and A. Gordon. A Semantics for Web Services Authentication. *Theoretical Computer Science*, 340(1):102–153, June 2005.

[2] Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. *Software - Practice and Experience*, 33:397–421, 2003.

[3] Kathi Fisler, Shriram Krishnamurthi, Leo Meyerovich, and Michael Carl Tschantz. Verification and Change-Impact Analysis of Access-Control Policies. In *ICSE*, 2005.

[4] D. Guelev, M. Ryan, and P. Schobbens. Model-checking Access Control Policies. In *ISC*, 2004.

[5] C. A. R. Hoare. *Commmunicating Sequential Processes*. Prentice-Hall, 1985.

[6] E Kleiner and A.W. Roscoe. Web Services Security: a preliminary study using Casper and FDR. In *ARPSA*, pages 160–174, 2004.

[7] Formal Systems (Europe) Ltd. Failures-Divergences Refinement: User Manual and Tutorial. Oxford University.

[8] T. Moses. eXtensible Access Control Markup Language (XACML) version 1.0. Technical report, OASIS, Feb 2003.

[9] A. W. Roscoe. *The Theory and Practice of Concurrency*. Pearson Education, 1998.

[10] Peter Ryan. Mathematical models of computer security. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, 2000.

[11] Peter Ryan and Ragni Ryvold Arnesen. A Process Algebraic Approach to Security Policies. In Ehud Gudes and Sujeet Shenoi, editors, *DBSec*, volume 256 of *IFIP Conference Proceedings*, pages 301–312. Kluwer, 2002.

[12] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols*. Pearson Education, 2001.

[13] Steve Schneider. *Concurrent and Real-time Systems: the CSP approach*. John Wiley & Sons, 2000.

[14] N. Zhang, M. Ryan, and D. Guelev. Synthesising Verified Access Control Systems in XACML. In *FMSE*, 2004.