An Integrated Approach for the Enforcement of Contextual Permissions and Pre-Obligations

Yehia El Rakaiby, Frédéric Cuppens, Nora Cuppens-Bouhlahia

An Integrated Approach for the Enforcement of Contextual Permissions and Pre-Obligations

Yehia Elrakaiby, TELECOM Bretagne, France

Frédéric Cuppens, TELECOM Bretagne, France

Nora Cuppens-Boulahia, TELECOM Bretagne, France

Pre-obligations denote actions that may be required before access is granted. The successful fulfillment of pre-obligations leads to the authorization of the requested access. Pre-obligations enable a more flexible enforcement of authorization policies. This paper formalizes interactions between the obligation and authorization policy states when pre-obligations are supported and investigates their use in a practical scenario. The main advantage of the presented approach is that it gives pre-obligations both declarative semantics using predicate logic and operational semantics using Event-Condition-Action (ECA) rules. Furthermore, the presented framework enables policy designers to easily choose to evaluate any pre-obligation either (1) statically (an access request is denied if the pre-obligation has not been fulfilled); or (2) dynamically (users are given the possibility to fulfill the pre-obligation after the access request and before access is authorized).

Keywords: *Contextual Policies, ECA Rules, Formal Pre-Obligation Enforcement, Pre-Obligation Policies, Usage Control*

INTRODUCTION

Traditional security policy systems provided a simple yes/no answer to access requests. However, it was recognized that access often depends on some user-actions being performed before access is granted. For instance, an access rule may specify that users are allowed to download music files provided that they pay 1$ first. In this case, if a user requests to download, for example, the latest single of Muse, s\he is asked to pay 1$. If the payment is made successfully, the user is allowed to download the requested file. Such requirements are called pre-obligations. Neither traditional access control models such as DAC (NCSC, 1987) and RBAC (Ferraiolo & Kuhn, 1992) nor more recent contextual security models such as ASL (Jajodia, Samarati, & Subrahmanian, 1997) and OrBAC (Abou El Kalam et al., 2003) support preobligations: In these models, an access request is only allowed if the conditions associated with a permission authorizing the access are true when the access request is made.

There are several advantages of supporting pre-obligations in the policy language. First, this provides additional expressiveness since it enables policy administrators to specify that subjects may fulfill some of the access requirements after the access request. Furthermore, it separates the expression of requirements from the functional specification (the code) of the application. Thus, the analysis of policy requirements is simplified and administrators are able to modify the behavior of the system by updating policy rules without recoding the application.

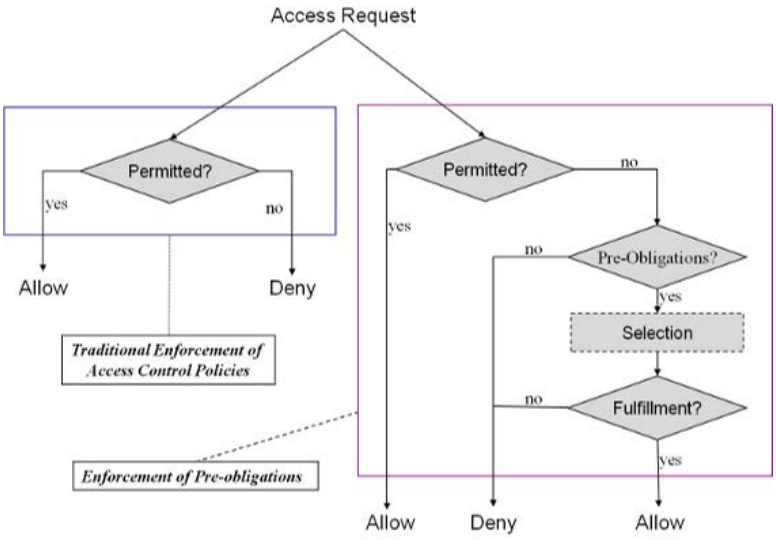
To support pre-obligations, a number of works (Bettini, Jajodia, Wang, & Wijesekera, 2002, 2003 ; Ni, Bertino, & Lobo, 2008) subordinate obligations to access control rules. This approach has some limitations. For instance, obligations are only activated after access requests and general obligations are not supported. In addition, this approach generally produces intricate access control policies since permissions and obligations are often specified within the same rule. This is the approach used in (Ni et al., 2008) to specify permissions and their associated pre-obligations. The main limitation of previous works on pre-obligations is however that none formalized the effects of supporting pre-obligations on the evolution of the authorization and obligation policy states. This is essential to provide a deeper understanding of pre-obligations and their enforcement in information systems. In addition, this formal approach allows the study and the analysis of change in the authorization and obligation policy states in the presence of pre-obligations. Therefore, it enables, for instance, to derive plans to reach some particular authorization states (Becker & Nanz, 2008 ; Craven et al., 2009) or to explain the deactivation of pre-obligations after permission activation.

In this paper, we study the specification and the enforcement of pre-obligations. In our approach, we formalize the enforcement of pre-obligations using an extension of the language Lactive (Baral & Lobo, 1996). Lactive enables the description of change in state using concepts from action specification languages (Gelfond & Lifschitz, 1993).Thus, it enables reasoning about state evolution and the study of interactions between pre-obligations and the authorization and obligation policy states. Lactive also supports the specification of reactive behavior using active rules. This feature enables us to provide formal operational semantics for the enforcement of pre-obligations.

To simplify the expression of pre-obligations in access control rules, we specify pre-obligations in the form of contexts. A security rule context (Cuppens & Cuppens-Boulahia, 2008) denotes a set of conditions which have to be true for the security rule to be effective. For instance, a context *during_working_hours* may hold (be true) every working day from 8 in the morning until 6 in the afternoon. In our approach, context rules may be used to specify requirements which state that some user-action should be taken. These contexts are called pre-obligation contexts. We support two evaluations of pre-obligation contexts: The static (traditional) evaluation requires that pre-obligation actions be taken before access requests are made. The dynamic evaluation, on the other hand, enables the fulfillment of pre-obligation requirements after access requests.

This is an extended version of the paper (El-rakaiby, Cuppens, & Cuppens-Boulahia, 2010) which appeared in ARES 2010. In particular, we extend our pre-obligation selection algorithm to clarify the formal model and we detail the different aspects of our approach. Furthermore, we consider *state* contexts in the policy language to simplify policy specification. The remainder of the paper is organized as follows. Section two presents some motivating examples. In Section three, we present our formalization language and introduce the basic entities used to describe the application domain. In Section four, we introduce our policy language. Section five formalizes policy management and enforcement using active rules. In Section six, we present the derivation of pre-obligations from the domain description and then present the enforcement of the policy. In Section seven, we

Figure 1. Enforcement of access control policies



present an application example. Finally, Section eight discusses related works and Section nine concludes the paper.

Motivating Example

We consider the following access control requirement:

r1. Mobile users may use the Video on Demand (VoD) service provided that they have paid 2$. In a traditional access control system, this requirement is enforced as follows: when a user requests to use the VoD service, the request is authorized only if the subject has paid 2$. Otherwise, the request is denied. This means that the verification of the fulfillment of pre-obligations consists of checking a history of previous action occurrences. This approach is inflexible for the enforcement of *r1* since it would be more convenient to allow the subject to pay for the service after s\he requests to use it. Then, when the subject successfully makes the payment, s/he is allowed access.

Thus, when pre-obligations are evaluated dynamically, the system would appear more flexible to the user. To provide such flexibility in the enforcement of access control requirements, we consider that requirements denoting user-actions may be defined as pre-obligations. In this case, when an access request is made, the subject is requested to satisfy the missing pre-obligation requirements (pay 2$). When these pre-obligations are fulfilled, the requested access is granted. Figure 1 compares the traditional enforcement of access control policies with their enforcement when pre-obligations are supported.

We now consider this second access control rule:

r2. Mobile users having WiFi coverage may use the VoD service provided that they have paid 1$.

Assume that the policy includes both the rules r1 and r2 and that it is possible to ask users to move to an area where there is WiFi coverage. In this case, when a user who has not paid for the VoD Service and has not WiFi

Table 1. Sorts of Lactive

Type	Description
Fluents	Facts describing the system state.
Actions	Possible actions in the system. Action occurrences update the fluent state by adding or removing fluents to or from the state.
Events	Define moments at which the policy needs to be updated.
Rule Names	ECA rule identifiers. An ECA rule states that when some *event* occurs and if some *conditions* are true, then some *actions* are executed. ECA rules (also called active rules) update the applied policy when particular events are detected.

coverage requests to use the VoD service, two alternative sets of pre-obligations are possible: (1) pay 2$ as specified in *r1*, (2) or pay 1$ and move to a WiFi covered area as specified in *r2*. One possible way to deal with this situation is to randomly select one of these two pre-obligation sets and ask the user to fulfill it. This however clearly represents an unacceptable behavior. Therefore, we choose to allow the association of pre-obligations with weights. For instance, if the pre-obligation to pay 2$ is given a lower weight than the sum of the weights of the two preobligations to pay 1$ and to move to a WiFi covered area, the pre-obligation set with the lowest weight (pay 2$) is selected. On the other hand, if the user is located in an area which has WiFi, s/he is asked to pay 1$ since the pre-obligation to pay 1$ would have lower weight than the one associated with the pre-obligation to pay 2$. This situation illustrates the importance of the dynamic selection of pre-obligations which takes into account which pre-obligations are and which are not fulfilled at the moment of the access request.

The Formalization Language

To formalize the effects of the support of pre-obligations on the authorization and policy states and to enable the study of their properties, we consider the language *Lactive* (Baral & Lobo, 1996). *Lactive* enables the description of change in state using concepts from action languages. It also supports the specification of reactive behavior in the form of Event Condition Action (ECA) rules. This gives operational semantics for the enforcement of pre-obligations. Sorts and propositions of Lactive are given in Tables 1 and 2.

In the language, a state is a set of fluents. A fluent literal is either a fluent symbol or a fluent symbol preceded by ¬ (¬¬f is equivalent to f). The semantics of *Lactive* defines a transition function which given a state and a (possibly empty) sequence of actions produces

Table 2. Propositions of Lactive

Type	Syntax	Description
Effect Law	*a* **causes** *f* **if** $p_1,...,p_n$	An effect law proposition states that the execution of *a* in a state where the fluents $p_1,...,p_n$ are true causes *f* to be true in the next state.
Event Definition	*e* **after** *a* **if** $p_1,...,p_n$	An event definition proposition states that if the conditions $p_1,...,p_n$ are true in the state following the execution of the action *a*, then event *e* is produced.
Active Rule	*R: e* **initiates** α **if** $p_1,...,p_n$	An active rule proposition states that every new detection of the event *e* triggers the execution of the sequence of actions α if the rule conditions are true.

a new state as follows. Actions in the input sequence are processed successively. For every action, effect laws are evaluated and the fluent state is updated. If after the execution of the action, conditions in some event definition are true, the event is generated. The newly generated events trigger active rules. Identifiers of these triggered rules are added to the triggered rules set. When the last action in the input sequence is evaluated, if the triggered rules set is not empty, an action selection function selects the sequence of actions appearing in one of the rules in the triggered rules set to process. Active rules are assigned priorities.

Therefore, the action selection function returns the sequence of actions appearing in one of the rules which have the highest priority in the triggered rule set. The state stops evolving after the processing of all the actions in an input sequence if the triggered rule set is empty.

Basic Entities of the Application Domain

We consider that the application domain includes finite sorts of the entities: subjects *S*, objects *O*, actions *A* and contexts *C*. Entities may have attributes. For instance, the application dependent *Name(s,n)* means that the name of *s* is *n*. We also consider three relations to enable the specification of security rules for groups of subjects, actions and objects respectively: Subjects are empowered into roles using the relation *Empower(Subject;Role)*, actions, *i.e.* programs, are considered implementation of some activity using the relation *Consider(Action; Activity)* and objects are used in views using the relation *Use(Object; View)*. Security rules may be specified using the abstract entities of roles, activities and views or using the concrete entities of subjects, actions and objects.

Description of Change in the Application Domain

To study the evolution of the policy state when change in state occurs, we assume that the system state is dynamic. More precisely, the system state may change after the execution of actions. We consider that actions of the form *Do(S,A,O)* indicate that subject *S* has taken the action *A* on the object *O*.The effects of the execution of actions on the state are described using effect law propositions.

For instance, we may specify the effect of the action *pay_2$* on the state as follows.

```
Do(S,pay_2$,payment server)
causes Paid_2$(S)
```

This effect law specifies that the fluent *Paid_2$(S)* starts to hold (be true) in the state after the action *pay_2$* is executed by *S* on a payment server. In our example, we will assume that a payment of 2$ is consumed when users use the VoD service. Therefore, we specify that the fluent *Paid_2$(S)* ceases to hold when *S* uses the VoD service as follows.

```
Do(S,use,video_on_demand)
causes ¬Paid_2$(S)
```

A set of effect laws is consistent if it does not contain two effect laws for the same action which have contradictory effects and whose conditions are not disjoint. These conditions are verified by considering the ground instances of effect laws in the application domain: If there is two effect laws "*a* **causes** *f* if $p_1, \ldots p_i, \ldots, p_n$" and "*a* **causes** *g* **if** $q_1, \ldots, q_j, \ldots, q_m$", then they should have either non-contradictory effects ($f \neq \neg g$) or disjoint conditions ($\exists$ i; j: $p_i = \neg q_j$).

The Policy Language

In this section, we first introduce our context language and show how we manage context activation and deactivation. We then present our security rules and show how they are used to specify system requirements.

CONTEXT LANGUAGE AND CONTEXT MANAGEMENT

We separate the definition of security rule conditions from the definition of security rules using contexts (Cuppens & Cuppens-Boulahia,

2008). A context defines a set of security rule conditions. The association of security rules with contexts allows the abstraction of complex conditions in security rules and thus, simplifies the interpretation of the policy. Contexts also allow context reuse in different security rules.

Context Rules: Security rule conditions define when some subject S is allowed, prohibited or obliged to take some action A on some object O. Therefore, contexts enable the definition of constraints on the security rule triple (S,A,O). Our context rules are expressions of the following form:

$Hold_e$(S,A,O, start/end(Ctx))
after *Do(S,A,O)* **if** $p_1,\ldots,p_n$

Context rules define the moments at which the conditions identified by the context *Ctx* start and seize to be true for the subject *S*, action *A* and object *O*. More precisely, the context rules for *start(Ctx)* define the conditions at which *Ctx* begins to hold. On the other hand, context rules for *end(Ctx)* specify when *Ctx* ceases to hold.

For instance, consider the following context rules.

$Hold_e$(S,A,O, start(in_WiFi_Area))
after *Do(S,enter,L)* **if** *WiFi_Area(L)*
$Hold_e$(S,A,O, end(in_WiFi_Area))
after *Do(S,exit,L)* **if** *WiFi_Area(L)*

These two rules specify that the context *in_WiFi_Area* remains true for some subject *S* from the moment this subject enters a location which is covered by WiFi until the moment the subject exists such location. These two moments are defined in terms of the event contexts *start(in_ WiFi_Area)* and *end(in_WiFi_Area)*.

We also consider a second type of context rules which we call state context rules. State context rules define conditions on the system state and are particularly suitable for the specification of conditions of permission and prohibition rules. State contexts are specified using expressions of the following form.

Hold(S,A,O,Ctx) ← $L_1,\ldots,L_n$

Where $L_1,\ldots,L_n$ are conditions on the state. To support this form of context rules called state context rules, we transform state context rules into event context rules (given a domain description) (Elrakaiby, Cuppens, & Cuppens-Boulahia, 2009b). In other words, we transform every state context rule into event context rules of the form *start(Ctx)* and *end(Ctx)*. For instance, consider the following rule.

Hold(S,A,O,paid_2$) ← Paid_2$(S)

The rule above specifies that the context paid 2$ holds for the subject *S* and any action/object while the fluent *Paid_2$(S)* is true. Given this state context rule and the effect laws presented in the previous section, we use an algorithm (Elrakaiby et al., 2009b) which transforms this state context into two event contexts *start(Ctx)* and *end(Ctx)*. For instance, the following event context rules are derived for the context *paid_2$*.

$Hold_e$(S,A,O,start(paid_2$))
after *Do(S,pay_2$,payment_server)*
$Hold_e$(S,A,O,end(paid_2$))
after *Do(S,use,video_on_demand)*

Our context language allows the expression of other important context types. For example, our context language supports the specification of temporal contexts. Temporal contexts are specified using the action *Clock*. This action updates fluents which represent calendars available in the system, such as Minutes, Hours, Day, etc. Temporal contexts enable the specification of absolute and periodic temporal conditions.

For instance, we may specify a temporal context working hours which holds everyday from 8 until 18 as follows.

```
Hold_e(S,A,O, start(working_hours))
afterClockifHours(08)
Hold_e(S,A,O,end(working_hours))
afterClockifHours(18)
```

We also consider the specification of relative temporal deadlines for obligations using the state context *delay(Nb.TimeUnit)*. This special context holds for some security rule after the elapse of Nb time units after its activation. We also allow context composition (Cuppens & Cuppens-Boulahia, 2008) using the logic operators of conjunction (&), disjunction ($\oplus$) and negation ($-$). The semantics of these operators is defined by the following rules.

```
Hold(S,A,O, C1&C2) ←
Hold(S,A,O,C1) ∧ Hold(S,A,O,C2)
Hold(S,A,O, C1⊕C2) ←
Hold(S,A,O,C1) ∨ Hold(S,A,O,C2)
Hold(S,A,O, -C1) ← ¬Hold(S,A,O,C)
```

Context Management: In this paper, we consider persistent contexts. A persistent context *Ctx* holds from the moment the event *start(Ctx)*until the occurrence of *end(Ctx)*. To enable the reasoning about which contexts hold in every state, we associate every persistent context *Ctx* with a fluent *Hold(S,A,O,Ctx)*. This fluent holds from the detection of the event context *start(Ctx)*until the occurrence of the event context *end(Ctx)*. This is enforced using the following two active rules.

```
activate_Context:
Hold_e(S,A,O,start(Ctx))
initiatesInsert(Hold(S,A,O,Ctx))
deactivate_Context:
Hold_e(S,A,O,end(Ctx))
initiatesRemove(Hold(S,A,O,Ctx))
```

The rules above specify that the fluent *Hold(S,A,O,Ctx)* should be inserted into (removed from) the state when *start(Ctx)* (*end(Ctx)*) is detected. We consider the context state to be the subset of fluents which are of the form *Hold(S,A,O,Ctx)*. In our framework, the context state is always updated before the evaluation of the policy. Therefore, the previous active rules are given higher priority than the rules which enforce the security policy. We present policy enforcement in the following section.

Security Policy Language

We consider security rules which are close ground facts of the following form.

```
Permission(N, SR, AA, OV, Ctx)
Obligation(N, SR, AA, OV, Ctx, Ctxv)
```

Where *N* is a rule identifier, *SR* is a subject or a role, *AA* is an action or an activity and *OV* is an object or a view. These expressions are called abstract security rules. A permission rule has one state context *Ctx*. This context is called the permission context. A permission is effective only while this context is true, *i.e.* after the event context *start(Ctx)* occurs and before the event context *end(Ctx)* occurs.

For example, consider the permission "mobile users may use the VoD service if they have paid 2$". This permission is specified as follows:

```
Permission(p, mobile_users, use,
video_on_demand,
paid_2$)
```

This permission specifies that subjects assigned to the role of mobile users may use the VoD service when the context *paid_2$* is true.

On the other hand, obligations are associated with two contexts: an obligation context (*Ctx*) and a violation context (*Ctxv*). The obligation is effective while the context *Ctx* holds. It is violated if the context *Ctxv* is detected while the obligation is effective. An obligation ceases to be effective when it is fulfilled, *i.e.* when the subject executes the obliged action on the corresponding object.

For instance, consider the obligation "When users are in a WiFi covered area, they

should turn on their WiFi connectivity within 3 minutes". This rule may be specified as follows:

```
Obligation(o1,mobile_users,turn_
on,wifi_connectivity,
in_WiFi_ area,delay(3.minutes))
```

Specification of Pre-obligations: A permission rule is contextual. For instance, permission *p* specifies that users are allowed to use the VoD service if they have paid 2$. This contextual permission is enforced as follows in traditional systems: When a request to use the VoD service is made, if a payment of 2$ had been made, access is authorized. Otherwise, the request is denied. This enforcement model may be sometimes too inflexible since it may be required to ask users to pay after the access request (if the payment is not already made). To simplify the specification that some requirements should be evaluated dynamically, we associate every user defined context in the policy *Ctx* with another context denoted *d_Ctx*, called the dynamic version of *Ctx*. When some context *d_Ctx* is used in some security rule, this means that this requirement may be fulfilled dynamically after the access request.

For instance, consider our example. To specify that users may be allowed to pay 2$ for the VoD service after they request to use it, we specify a permission rule using the context *d_paid_2$* as follows.

```
Permission(p1, mobile_users, use,
video_on_demand,
d_paid_2$)
```

In this case, when a user requests to use the VoD service and s\he has not paid for the service, the user is asked to pay 2$. This requirement is enforced using an obligation to pay 2$ for using the service. When this obligation is fulfilled, access is allowed.

It is necessary to associate every obligation with a deadline condition. For instance, it may be required to specify that the mobile user should pay within 5 minutes. For this reason, we allow the association of every dynamic context with a deadline in the form of an attribute *Violation*. For instance, we specify that the obligation associated with the context *d_paid_2$* has a deadline of 5 minutes by updating the value of its attribute *Violation* to *Violation(d_paid_2$,delay(5. minutes))*. For every dynamic context, a default deadline defined by the policy administrator is used unless this attribute is updated. We also give pre-obligations weights to enable the selection of the simplest set of pre-obligations for a given access request. Therefore, we consider a second attribute *Weight* for dynamic contexts. The default value of this attribute is 1. For instance, consider the following permission.

```
Permission(p2, mobile_users, use,
video_on_demand,
d_paid_1$ & d_in_WiFi_area)
```

Where the context *paid_1$* is defined similarly to the context *paid_2$*. To specify that a 1$ payment is simpler to a 2$ payment, we assign the contexts *d_paid_1$* and *d_paid_2$* the weights of 2 and 3 respectively. Now, assume that the context *d_in _WiFi_area* is given a weight of 4. In this setting, when a user requests to use the VoD service, there are several possibilities. For instance, if s\he has not paid and is in a WiFi covered area, s\he is asked to pay 1$. If s\he has not paid and is not in a WiFi covered area, s\he is asked to pay 2$.

Policy Management and Enforcement

We distinguish between abstract and concrete security rules as follows: Abstract policy rules describe the global system policy and is specified by policy administrators. Concrete rules, on the other hand, are the security rules which are derived from the abstract policy as follows.

```
Permission(N,S,A,O,Ctx) ←
Permission(N,SR,AA,OV,Ctx) Empower'(S,
SR),Consider'(A,AA),Use'(O,OV)
```

The predicate *Empower'(S,SR)* specifies that *S* should be either *SR* if *SR* is a subject or a subject empowered into the role of *SR* if *SR* is a role. It is specified as follows.

```
Empower'(S,SR) ← Subject(SR)
Empower'(S,SR) ← Role(SR,
Empower(S,SR)
```

Similarly, the predicate *Consider'(A,AA)* states that *A* should be either the action *AA* if *AA* is an action or an action considered in *AA* if *AA* is an activity. The predicate *Use'(O,OV)* dictates that *O* should be either the object *OV* or an object used in *OV* if *OV* is a view. Concrete obligation rules are also derived for individual subjects, actions and objects from abstract obligation rules. In the following, we give formal operational semantics for policies which consist of concrete security rules using active rules.

Permission Activation and Deactivation

Every concrete permission rule is associated with a context which defines when it is effective. We therefore associate every permission in the state with a fluent *Permitted(N,S,A,O,Ctx)*. This fluent starts to hold when the permission's context begins to hold. It ceases to hold when the permission's context is ended. This is specified using the following active rules:

```
activate_Permission:
Hold_e(S,A,O,start(Ctx))
initiates Insert(Permitted(N,S,A,O,C
tx)) if Permission(N,S,A,O,Ctx)
deactivate_Permission:
Hold_e(S,A,O,end(Ctx))
initiates Remove(Permitted(N,S,A,O,C
tx)) if Permitted(N,S,A,O,Ctx)
```

The rules above specify that the action *Insert(Permitted(N,S,A,O,Ctx))* should be taken when the context of some permission's is activated. This action makes the fluent *Permitted* hold as specified in the following effect law.

```
Insert(Permitted(N,S,A,O,Ctx))
causes Permitted(N,S,A,O,Ctx)
```

Reciprocally, we specify that *Permitted* ceases to hold after the execution of the action *Remove* on the fluent *Permitted*. In the policy, an access may be authorized by more than one permission. Therefore, we consider an additional fluent *Permitted(S,A,O)* which holds for some access (S,A,O) while this access is allowed. This fluent begins to hold for some access (S,A,O) whenever some permission for (S,A,O) is activated. It ceases to hold after the deactivation of a permission for (S,A,O) only if there is no other permission for (S,A,O) in the state.

Obligation Activation and Deactivation

To manage obligations, we associate every concrete obligation with a fluent *Obliged(N,S,A,O,Ctx,Ctxv)*. This fluent represents that there is an effective obligation for *S* to take *A* on *O* before *Ctxv* is detected. An obligation is deactivated when its context *Ctx* is ended while it is effective. When an obligation is deactivated, the fluent *Obliged* ceases to hold. This is formalized using the following two active rules.

```
activate_Obligation:
Hold_e(S,A,O,start(Ctx))
initiates Insert(Obliged(N,S,A,O,Ctx,Ct
xv)) if Obligation(N,S,A,O,Ctx,Ctxv)
deactivate_Obligation:
Hold_e(S,A,O,end(Ctx))
initiates Remove(Obliged(N,S,A,O,Ctx,Ct
xv)) if Obliged(N,S,A,O,Ctx,Ctxv)
```

Obligation Fulfillment and Violation

As opposed to permissions, obligations may additionally be violated and fulfilled. An effective

obligation is fulfilled when its required action is taken. Actions required by obligations are monitored using the following context.

```
Hold_e(S,A,O,start(ctx_fulfillment))
after Do(S,A,O)
if Obliged(N,S,A,O,Ctx,Ctxv)
```

The context *ctx_fulfillment* holds for some (S,A,O) when the action *Do(S,A,O)* is taken and there is an effective obligation requiring (S,A,O). When *start(S,A,O,start(ctx_fulfillment))* is detected, effective obligations for (S,A,O) are fulfilled using the following active rule.

```
fulfill_Obligation:
Hold_e(S,A,O,start(ctx_fulfillment))
initiates Fulfill(N,S,A,O)
if Obliged(N,S,A,O,Ctx,Ctxv)
```

Reciprocally, the detection of the deadline context of an effective obligation violates this obligation. This is specified as follows.

```
violate_Obligation:
Hold_e(S,A,O,start(Ctxv))
initiates Violate(N,S,A,O)
if Obliged(N,S,A,O,Ctx,Ctxv)
```

The actions *Fulfill* and *Violate* indicate the fulfillment and violation of obligations respectively. In this paper, we assume for simplicity that obligations are deactivated whenever they are violated/fulfilled. Therefore, the fluent *Obliged* ceases to hold when the actions *Fulfill* and *Violate* are taken.

Derivation of Dynamic Contexts and Enforcement of Pre-Obligations

To simplify the specification of pre-obligations, we consider that every user-specified context *Ctx* has a corresponding dynamic context *d_Ctx*. This simplifies the specification of the policy by enabling policy administrators to easily choose whether a context should be statically or dynamically evaluated.

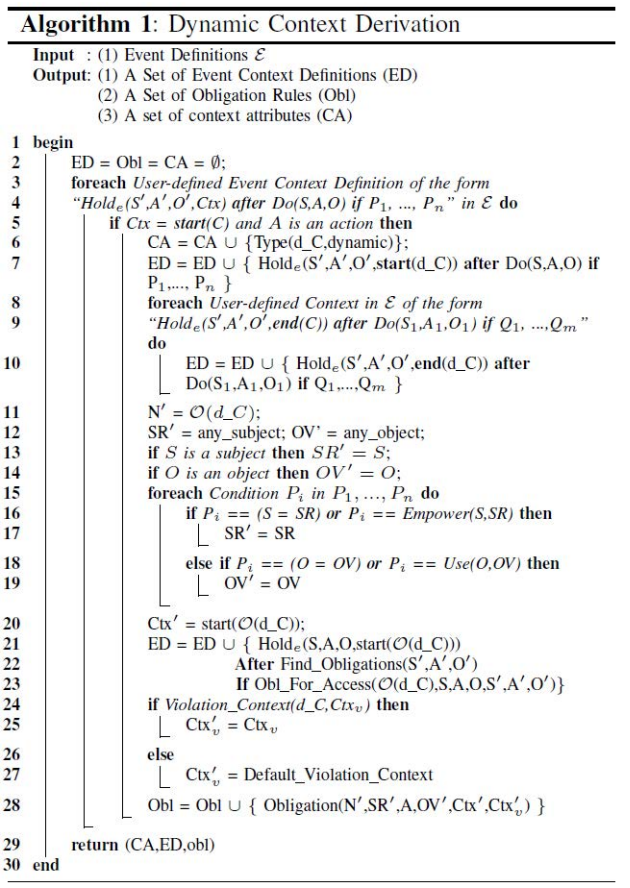
Dynamic contexts and their associated pre-obligations are automatically derived from the definition of user-specified contexts using Algorithm 1 (Figure 2). This algorithm takes the set of user-defined event context definitions *E* as input, and produces the definition of dynamic contexts. It also derives for every dynamic context *d_C* an obligation rule *O(d_C)*. This obligation *O(d_C)* defines the action which should be taken for the context *d_C* to be activated. The fulfillment of this obligation activates the context *d_C* (as well as the context *C*) for the access requester.

The algorithm verifies every user-defined event context rule as follows. First, if the context is of the form *start(C)* and is started by some action *A* (line 5), then a dynamic context *d_C* is defined similarly to *C*, *i.e. d_C* is associated with the same actions which start and end *C* (lines 7-10). An obligation is then constructed. The obligation's identifier is *O(d_C)* (line 12). Its role and view are initialized using the role *any_subject* and the view *any_object* (line 13). These entities represent all subjects and all objects in the system respectively.

Constraints over the parameters of the action which starts *C* in the *after* part (lines 13-14) and in the *if* part (lines 15-19) of the context definition of *start(C)* are then checked. If some constraint over the action's subject or object (S,A,O) is specified, it is used as the subject/role and object/view of the obligation respectively.

An event context identifier of the form *start(O(d_C))* is then used to denote the activation conditions of the obligation (line 20). The context *start(O(d_C))* is then defined (lines 21-23). Its definition states that it should be detected after the execution of the action *Find_Obligations(S',A',O')* if the fluent *Obl_For_Access(O(d_C),S,A,O,S',A',O')* holds. The action *Find_Obligations* checks the policy for possible pre-obligations when the access (S′,A′,O′) is not authorized. The fluent *Obl_For_Access*, on the other hand, denotes that the obligation associated with *d_C* for the

Figure 2. Algorithm 1: derivation of dynamic contexts



Algorithm 1: Dynamic Context Derivation

Input : (1) Event Definitions $\mathcal{E}$
Output: (1) A Set of Event Context Definitions (ED)
(2) A Set of Obligation Rules (Obl)
(3) A set of context attributes (CA)

```
1  begin
2      ED = Obl = CA = ∅;
3      foreach User-defined Event Context Definition of the form
4      "Hold_e(S',A',O',Ctx) after Do(S,A,O) if P_1, ..., P_n" in E do
5          if Ctx = start(C) and A is an action then
6              CA = CA ∪ {Type(d_C,dynamic)};
7              ED = ED ∪ { Hold_e(S',A',O',start(d_C)) after Do(S,A,O) if
               P_1,..., P_n }
8              foreach User-defined Context in E of the form
9              "Hold_e(S',A',O',end(C)) after Do(S_1,A_1,O_1) if Q_1, ...,Q_m"
               do
10                 ED = ED ∪ { Hold_e(S',A',O',end(d_C)) after
                   Do(S_1,A_1,O_1) if Q_1,...,Q_m }
11             N' = O(d_C);
12             SR' = any_subject; OV' = any_object;
13             if S is a subject then SR' = S;
14             if O is an object then OV' = O;
15             foreach Condition P_i in P_1, ..., P_n do
16                 if P_i == (S = SR) or P_i == Empower(S,SR) then
17                     SR' = SR
18                 else if P_i == (O = OV) or P_i == Use(O,OV) then
19                     OV' = OV
20             Ctx' = start(O(d_C));
21             ED = ED ∪ { Hold_e(S,A,O,start(O(d_C)))
22                         After Find_Obligations(S',A',O')
23                         If Obl_For_Access(O(d_C),S,A,O,S',A',O')}
24             if Violation_Context(d_C,Ctx_v) then
25                 Ctx'_v = Ctx_v
26             else
27                 Ctx'_v = Default_Violation_Context
28             Obl = Obl ∪ { Obligation(N',SR',A,OV',Ctx',Ctx'_v) }
29     return (CA,ED,obl)
30 end
```

subject *S* to take the action *A* on the object *O* has been selected for the authorization of (S′,A′,O′).

Finally, if a user-specified Violation Context attribute for *d_C* exists, it is used as the obligation violation context. Otherwise, the default context is used (lines(24-27)). The algorithm returns the constructed event definitions, obligation rules and context attributes. These elements are added to the policy.

For instance, the application of the algorithm to the context *start(paid_2$)* produces: (1) the dynamic context definition for *d_paid_2$*. This context is defined similarly to the user-defined context *paid_2$,* (2) The context attribute *Type(d_paid_2$,dynamic),* and (3) The obligation and the event context rule specified below.

```
Obligation(O(d_paid_2$),any_
subject,pay_2$,any_object,
start(O(d_paid_2$)), delay(3.Minutes))
Hold_e(S, pay_2$, O, start(O(d_
paid_2$)))
after Find_Obligations(S',A',O')
if Obl_For_Access(O(d_paid_2$),S,
pay_2$,O, S',A',O')
```

This obligation rule defines an obligation *O(d_paid_2$)* which states that the action *pay_2$* should be taken by any subject on any object when the context *start(d_paid_2$)* is detected. This context is detected for the subject *S* and object *O* if *S* and *O* were selected to fulfill the obligation after the execution of the action *Find_Obligations* for the access

request (S′,A′,O′). The selected *S* and *O* for the obligation are the ones specified using the fluent *Obl_For_Access(O(d_paid_2$),S, pay_2$,O,S',A',O')*.

Authorization Policy Enforcement

When pre-obligations are supported, the authorization policy is enforced as follows: When an access request is made, access is granted if it is authorized by an effective permission. Otherwise, the authorization policy is checked for pre-obligations which would allow the access. If none is found, access is denied. If pre-obligations are activated, they are enforced as follows. Whenever an effective permission for the requested access is activated or if one of the pre-obligations is violated/deactivated, pre-obligations are deactivated. When all pre-obligations are successfully fulfilled, access is granted.

Authorization Policy Enforcement: To enforce the authorization policy, we consider the context *access_req_ctx*. This context is specified as follows.

```
Hold_e(S,A,O,start(access_req_ctx))
after Request(S,A,O)
```

The context *access_req_ctx* holds for an access (S,A,O) after the occurrence of the special action *Request(S,A,O)*. This action indicates that *S* has requested to take *A* on *O*. This context holds until this access request is honored, *i.e.* when the access is either allowed or denied. The end of *access_req_ctx* is therefore specified as follows.

```
Hold_e(S,A,O,end(access_req_ctx))
after Allow(S,A,O)
Hold_e(S,A,O,end(access_req_ctx))
after Deny(S,A,O)
```

When an access request is made, it is directly granted if it is authorized by an effective permission. This is specified using the following rule.

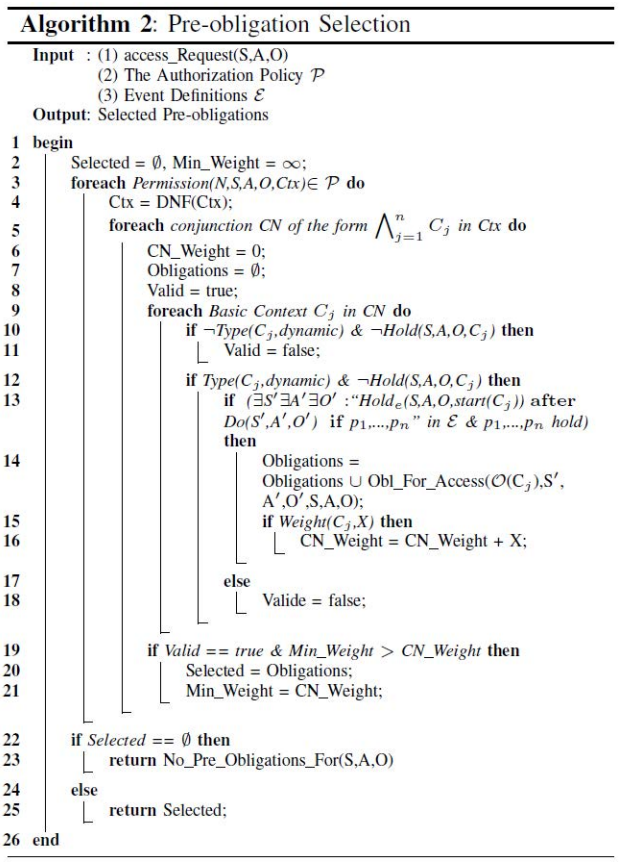
```
allow_Access:
Hold_e(S,A,O,start(access_req_ctx))
initiates Allow(S,A,O)
if Permitted(N,S,A,O,Ctx,Ctxv)
```

If there is no effective permission for the requested access, we check the authorization policy for pre-obligations.

```
find_Obligations:
Hold_e(S,A,O,start(access_req_ctx))
initiates Find_Obligations(S,A,O) if
¬Permitted(N,S,A,O,Ctx,Ctxv)
```

The action *Find_Obligations* selects (if possible) the simplest set of pre-obligations required to allow (S,A,O) by executing Algorithm 2 (Figure 3). This algorithm works as follows: it checks every permission which permits (S,A,O). First, the permission context *Ctx* into the disjunctive normal form (DNF) to identify the sets of basic contexts which have to hold simultaneously to allow the requested access. A set of basic contexts (*CN*) is considered valid if: (1) all its non-dynamic contexts are true, (2) each of its dynamic contexts which does not hold can be activated. A dynamic context *C* can be activated if there exists (S′,A′,O′) and a dynamic event definition $Hold_e(S,A,O,start(C))$ such that the conditions of this event definition are true. This ensures that when *Do(S',A',O')* is preformed, *C* is activated. For every inactive dynamic context which can be activated, a fluent of the form *Obl_For_Access(O(C),S',A',O',S,A,O)* is added to the set *Obligations*. This fluent specifies that *Do(S',A',O')* should be taken to activate the dynamic context *O(C)* and, subsequently allow the requested access (S,A,O). Then, the weight assigned with *O(C)* is added to the *CN_Weight*. After the evaluation of every CN_i, if the sum of the weights of its pre-obligations *CN_Weight* is less than the minimum weight *Min_Weight*, the pre-obligations of this CN_i are selected. After the evaluation of the authorization policy, the

Figure 3. Algorithm 2: selection of pre-obligations



Algorithm 2: Pre-obligation Selection

Input : (1) access_Request(S,A,O)
(2) The Authorization Policy $\mathcal{P}$
(3) Event Definitions $\mathcal{E}$
Output: Selected Pre-obligations

```
1  begin
2      Selected = ∅, Min_Weight = ∞;
3      foreach Permission(N,S,A,O,Ctx)∈ P do
4          Ctx = DNF(Ctx);
5          foreach conjunction CN of the form ⋀_{j=1}^{n} C_j in Ctx do
6              CN_Weight = 0;
7              Obligations = ∅;
8              Valid = true;
9              foreach Basic Context C_j in CN do
10                 if ¬Type(C_j,dynamic) & ¬Hold(S,A,O,C_j) then
11                     Valid = false;
12             if Type(C_j,dynamic) & ¬Hold(S,A,O,C_j) then
13                 if (∃S′∃A′∃O′ :"Hold_e(S,A,O,start(C_j)) after
                   Do(S′,A′,O′) if p_1,...,p_n" in E & p_1,...,p_n hold)
                   then
14                     Obligations =
                       Obligations ∪ Obl_For_Access(O(C_j),S′,
                       A′,O′,S,A,O);
15                     if Weight(C_j,X) then
16                         CN_Weight = CN_Weight + X;
17                 else
18                     Valide = false;
19             if Valid == true & Min_Weight > CN_Weight then
20                 Selected = Obligations;
21                 Min_Weight = CN_Weight;
22     if Selected == ∅ then
23         return No_Pre_Obligations_For(S,A,O)
24     else
25         return Selected;
26 end
```

algorithm returns *No_Obl_For_Access(S,A,O)* if no preobligations are possible for the access. Otherwise, the set of pre-obligations selected for the access is returned.

If no pre-obligations are returned after the execution of *Find_Obligations*, the context *no_pre_obligations* holds and access is denied. We specify access denial as follows.

```
Hold_e(S,A,O,start(no_pre_obligations))
after Find_Obligations(S,A,O) if No_Obl_
For_Access(S,A,O)
deny_Access: Hold_e(S,A,O,start(no_pre_
obligations))
initiates Deny(S,A,O)
```

The fluent *No_Obl_For_Access(S,A,O)* as well as the context *no_pre_obligations* seize to hold when the access request is honored to allow the reevaluation of the authorization policy at subsequent access requests.

Enforcement of Pre-Obligation Sets

After the activation of a set of pre-obligations for an access (S,A,O), pre-obligations are enforced as follows.

Permission Activation: Whenever a permission is activated for (S,A,O) and there is a request to take (S,A,O), pre-obligations for (S,A,O) are deactivated and access is allowed. The following context *authorized_request* starts to hold when

Table 3. Context attributes

Context	*Weight*	*Violation*
d_paid_1$	*2*	*delay(3.minutes)*
d_paid_2$	*3*	*delay(4.minutes)*
d_in_WiFi_area	*4*	*delay(5.minutes)*

some requested access (S,A,O) become authorized.

```
Hold_e(S,A,O,start(authorized_request))
after Insert(Permitted(N,S,A,O,Ctx))
if Hold(S,A,O,access_req_ctx)
```

When an access request for (S,A,O) is authorized, the following rule deactivates pending pre-obligations for (S,A,O) (if any exists).

```
deactivate_Pre:
Hold_e(S,A,O,start(authorized_request))
initiates Remove(Obl_For_
Access(N,S',A',O',S,A,O)) if Obl_For_
Access(N,S',A',O',S,A,O)
```

We also accept the requested access by initiating the action *Allow* using the following active rule.

```
Allow_access*:
Hold_e(S,A,O,start(authorized_request))
initiates Allow(S,A,O)
```

Violation of Pre-obligations: When a pre-obligation is violated, the fulfillment of other pre-obligations becomes unnecessary since the access will not be allowed. Therefore, we deactivate in this case other related pre-obligations (*i.e.* pre-obligations for the same access request) and deny access. We define the context *pre_obl_violated* which holds when pre-obligations are violated as follows.

```
Hold_e(S,A,O,start(pre_obl_violated))
after Violate(N,S',A',O') if Obl_For_Ac-
cess (N,S',A',O',S,A,O)
```

When *pre_obl_violated* starts to hold, we deactivate pre-obligations and deny the access requested.

```
violate_Pre: Hold_e(S,A,O,start(pre_
obl_violated))
initiates Remove(Obl_For_
Access(N,S',A',O',S,A,O) if Obl_For_
Access(N,S',A',O',S,A,O)
deny_access*: Hold_e(S,A,O,start(pre_
obl_violated))
initiates Denys(S,A,O)
```

Pre-obligation Fulfillment: When pre-obligations are fulfilled, they are removed from the state using the following active rule.

```
violate_Pre: Hold_e(S,A,O,start(ctx_
fulfillment))
initiates Remove(Obl_For_
Access(N,S,A,O,S',A',O')) if Obl_For_
Access(N,S,A,O,S',A',O')
```

Application Example

To illustrate the concepts presented in this paper and discuss the evolution of the authorization and obligation policy states when pre-obligations are supported, we consider an example policy that includes the following permission rules.

```
Permission(p1,mobile_users,use,video_
on_demand, d_paid_2$)
Permission(p2,mobile_users,use,video_
on_demand,
d_in_WiFi_area & d_paid_1$)
```

Table 3 shows the values given to the attributes *Weight* and *Violation* of each context.

Figure 4. Policy state evolution

	Ctx_State				A_State		O_State			
	paid_2$	paid_1$	wifi_area	work_hours	p'_1	p_2	$o_{2\$}$	$o_{1\$}$	o_{wf}	Action
s_4	.	.	.	X	.	.	.	.	.	req(use_vod)
	.	.	.	X	.	.	X	.	.	Do(pay_2$)
	X	.	.	X	X	.	.	.	.	Allow(use_vod)
s_5	.	.	.	.	.	.	.	.	.	req(use_vod)
s'_5	.	.	.	.	.	.	.	X	X	Do(pay_1$)
s''_5	.	X	.	.	.	.	.	.	X	Do(enter_wifi_area)
	.	X	X	.	.	X	.	.	.	Allow(use_vod)
s'_5	.	.	.	.	.	.	.	X	X	Delay(3.minutes)
	.	.	.	.	.	.	.	.	.	Deny(use_vod)
s''_5	.	X	.	.	.	.	.	.	X	Delay(5.minutes)
	.	X	.	.	.	.	.	.	.	Deny(use_vod)

We first discuss the selection of pre-obligations after a user requests to use the VoD service in the following situations:

- (S1) The user has paid 2$: Access is directly granted since permission p1 is effective.
- (S2) The user has paid 1$ and is not in a WiFi covered area: the user is asked to pay 2$ since this pre-obligation is assigned lower weight than the weight given to the obligation to move to a WiFi covered area.
- (S3) The user has not paid and is in an area having WiFi: the user is asked to pay 1$.

Assume we replace the permission *p1* in the policy above with the following permission.

```
Permission(p'1,mobile_users,use,video_
on_demand,
working_hours & d_paid_2$)
```

The permission *p'1* specifies that users may use the VoD service during working hours provided that they have paid 2$. In this case, pre-obligations are selected as follows:

- (S4) During working hours, the user has not paid nor is in a WiFi covered area and is requesting to use the VoD service: the user is asked to pay 2$.
- (S5) Outside of working hours, the user has not paid nor is in a WiFi area and is requesting to use the VoD service: the user is asked to pay 1$ and to move to an area with WiFi since only *p2* can be activated.

Table 4 and Figure 4 show the selection of pre-obligations and discusses the evolution of the state of authorizations and obligations in the different situations just described. Each table row represents the state obtained by the execution of the action appearing in the right-most column of the row above. The obligations to pay 2$, to pay 1$ and to move to a WiFi covered area are denoted $o_{2\$}$, $o_{1\$}$ and o_{wf} respectively. We only give identifiers for situations when it is necessary. We will now consider the evolution of the authorization and obligation policy states for the situation (S3) where a user is asked to pay 1$ (within 3 minutes). In this scenario, the following may occur.

- The user pays 1$ successfully and access is granted.
- The user pays 2$. In this case, the permission *p1* is activated and the obligation to pay 1$ is deactivated.
- The user fails to pay within 3 minutes and access is denied.

We now consider the situation (S5) where a user is asked to pay 1$ (within 3 minutes) and to move to a WiFi covered area within (5 minutes). The following may happen.

Table 4. Policy state evolution

	Ctx_State			A_State		O_State			
	Paid_2$	Paid_1$	Wifi_ area	p_1	p_2	$o_{2\$}$	$o_{1\$}$	o_{wf}	Action
s_1	X X	. .	. .	X X	. .	. .	. .	. .	Req(use_vod) Allow(use_ vod)
s_2	. . X	X X X	. . .	. . X	. . .	. . .	. . .	. . .	Req(use_vod) Do(pay_2$) Allow(use_ vod)
s_3 s'_3 s''_3	. . .	. . X	X X X	. . .	. . X	. X ,	. X .	. . .	Req(use_vod) Do(pay_1$) Allow(use_ vod)
s'_3	. .	. X	X X	. .	. X	X .	X .	. .	Do(pay_2$) Allow(use_ vod)
s'_3	. .	. .	X X	. .	. .	X .	X .	. .	Delay(3. minutes) Deny(use_vod)

- The user successfully fulfills the two pre-obligations. In this case, the permission *p2* will be activated and access is granted.
- The user fails to pay within 3 minutes. In this case, the second pre-obligation to move to a WiFi covered area is deactivated and access is denied.
- The user pays within 3 minutes but fails to move to a WiFi covered area within 5 minutes. In this case, access is denied.

Related Work

Other models have been proposed to support preobligations in access control policies. To our knowledge, the notion of provisional actions was first introduced by Kudo and Hada (2000) to enable the association of access control security rules for XML documents with actions that should be triggered by access requests. In Kudo (2002), multiple hierarchies and property propagation are studied. In contrast, we study provisional actions in the form of user actions which are monitored for fulfillment/violation and formalize policy enforcement and evolution.

In (Jajodia, Kudo, & Subrahmanian, 2001), the ASL access control language (Jajodia et al., 1997) is extended to allow the association of security rules with provisional actions. An architecture for the enforcement of these provisional actions is proposed. Bettini et al. (2002, 2003) study the association of access control rules with provisions and obligations and propose algorithms for the computation a minimal provisions and obligations set. Obligation definition and monitoring in the framework is discussed in (Bettini, Wang, Jajodia, & Wijesekera, 2002). In comparison, the main advantage of our work is that we consider a formal description of change in state using the concepts of action specification languages. This enables us to formalize the activation, deactivation, violation and fulfillment of pre-obligations and the effects of these operations on the authorization and obligation states. Consequently, we clarify the semantics of pre-obligations by giving their enforcement declarative semantics. In addition, our formal model for pre-obligations is given operational semantics using ECA rules.

In Ni et al. (2008), an obligation model supporting the specification of pre- and post-obligations is presented. The paper studies two interactions between permissions and obligations namely invalid permission due to obligation cascading and the dominance of obligations. With respect to our work, the model subordinates obligations to permissions, only considers temporal obligation deadlines and does not consider the selection of pre-obligations after access requests. Furthermore, in our policy language, obligations are specified separately from access control rules and provisions are specified in the form of contexts in permission rules (as opposed to the specification of obligations embedded in access control rules). This simplifies the representation of the access control policy and, additionally, enables us to support general obligations which do not depend of access requests. Moreover, we provide formal declarative semantics for the enforcement of pre-obligations.

The UCON model (Park & Sandhu, 2004) introduces obligations to deal with usage control requirements and introduces the notion of attribute mutability. The model is formalized in Zhang, Parisi-Presicce, Sandhu, and Park (2005). Pre-obligations in UCON are evaluated using the functional predicate "preB" which checks, when an access request is made, whether pre-obligations required for this access have been fulfilled. Formally, checking pre-obligation fulfillment in UCON is similar to checking regular permission conditions. By contrast, in our framework, pre-obligations are activated just after an access request if their fulfillment is required to enable the access. This is an important advantage since, whenever necessary, subjects may be assisted by the system in accessing resources. Additionally, the UCON model does not support the specification of general or global obligations since obligations are always associated with resource usage.

Other works on trust management (Becker & Nanz, 2008 ; Bonatti, Olmedilla, & Peer, 2006 ; Koshutanski & Massacci, 2004) studied the use of abduction in explaining access denials to users by searching for missing facts or credentials which would allow the requested access. The work that is most relevant to ours is Becker and Nanz (2007) where a logic and an inference system for reasoning about sequences of user-actions and their effects on the authorization policy state are presented. This work is complementary to ours since we essentially study the enforcement and management of pre-obligations as opposed to how to derive the actions that should be taken to obtain particular permissions.

CONCLUSION

In this paper, we studied the specification, selection and enforcement of pre-obligations. First, we have proposed to specify pre-obligations in access control rules in the form of permission contexts to simplify both the specification and interpretation of the access control policy. We have also considered the notion of dynamic context attributes to allow the association of dynamic contexts (denoting preobligations) with different weights and deadlines. We have then studied the selection of pre-obligations after access requests and formalized the enforcement of pre obligations and its effects on the policy state.

Future work consists of modeling consent requirements in the form of special pre-obligations and the integration of group pre-obligations (Elrakaiby, Cuppens, & Cuppens-Boulahia, 2009a) to enable the specification of pre-obligations which may be fulfilled in different ways.

REFERENCES

Abou El Kalam, A., Benferhat, S., Balbiani, P., Miège, A., El Baida, R., Cuppens, F., et al. (2003). Organization based access control. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks* (pp. 120-131). Washington, DC: IEEE Computer Society.

Baral, C., & Lobo, J. (1996). Formal characterization of active databases. In *Proceedings of the International Workshop on Logic in Databases* (pp. 175-195).

Becker, M. Y., & Nanz, S. (2007). A logic for state modifying authorization policies. *ACM Transactions on Information and System Security*, *13*(3), 20.

Becker, M. Y., & Nanz, S. (2008). The role of abduction in declarative authorization policies. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Language* (pp. 84-99).

Bettini, C., Jajodia, S., Wang, X. S., & Wijesekera, D. (2002). Provisions and obligations in policy management and security applications. In *Proceeding of the 28th International Conference on Very Large Data Bases* (pp. 502-513). Washington, DC: IEEE Computer Society.

Bettini, C., Jajodia, S., Wang, X. S., & Wijesekera, D. (2003). Provisions and obligations in policy rule management. *Journal of Network and Systems Management*, *11*(3). doi:10.1023/A:1025711105609

Bettini, C., Wang, X., Jajodia, S., & Wijesekera, D. (2002). Obligation monitoring in policy management. In *Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks* (pp. 2-12). Washington, DC: IEEE Computer Society.

Bonatti, P. A., Olmedilla, D., & Peer, J. (2006). Advanced policy explanations on the web. In *Proceeding of the 17th European Conference on Artificial Intelligence* (pp. 200-204). Amsterdam, The Netherlands: IOS Press.

Craven, R., Lobo, J., Ma, J., Russo, A., Lupu, E., & Bandara, A. (2009). Expressive policy analysis with enhanced system dynamicity. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (pp. 239-250). New York, NY: ACM Press.

Cuppens, F., & Cuppens-Boulahia, N. (2008). Modeling contextual security policies. *International Journal of Information Security*, *7*(4), 285–305. doi:10.1007/s10207-007-0051-9

Cuppens, F., Cuppens-Boulahia, N., & Sans, T. (2005). Nomad: A security model with non atomic actions and deadlines. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations* (pp. 186-196). Washington, DC: IEEE Computer Society.

Elrakaiby, Y., Cuppens, F., & Cuppens-Boulahia, N. (2009a). Formalization and management of group obligations. In *Proceedings of the 10th IEEE International Conference on Policies for Distributed Systems and Networks* (pp. 158-165). Los Alamitos, CA: IEEE Press.

Elrakaiby, Y., Cuppens, F., & Cuppens-Boulahia, N. (2009b). From state-based to event-based contextual security policies. In *Proceedings of the Fourth International Conference on Digital Information Management*, Ann Arbor, MI (pp. 1-7). Washington, DC: IEEE Computer Society.

Elrakaiby, Y., Cuppens, F., & Cuppens-Boulahia, N. (2010). From contextual permission to dynamic preobligation: An integrated approach. In *Proceedings of the International Conference on Availability, Reliability, and Security* (p. 70). Washington, DC: IEEE Computer Society.

Ferraiolo, D. F., & Kuhn, D. R. (1992). Role-based access control. In *Proceedings of the 15th National Computer Security Conference* (pp. 554-563).

Gelfond, M., & Lifschitz, V. (1993). Representing action and change by logic programs. *The Journal of Logic Programming*, *17*, 301–322. doi:10.1016/0743-1066(93)90035-F

Jajodia, S., Kudo, M., & Subrahmanian, V. (2001). Provisional authorizations. In *E-commerce security and privacy* (pp. 133–159). Amsterdam, The Netherlands: Kluwer Academic.

Jajodia, S., Samarati, P., & Subrahmanian, V. S. (1997). A logical language for expressing authorizations In *Proceedings of the IEEE Symposium on Security and Privacy* (p. 0031). Washington, DC: IEEE Computer Society.

Jordan, C. (1987). *A guide to understanding discretionary access control in trusted systems*. Retrieved from http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA392813

Koshutanski, H., & Massacci, F. (2004). Interactive access control for web services. In *Proceedings of the 19th International Conference on Information Security* (pp. 151-166).

Kudo, M. (2002). PBAC: Provision-based access control model. *International Journal of Information Security*, *1*(2), 116–130. doi:10.1007/s102070100010

Kudo, M., & Hada, S. (2000). Xml document security based on provisional authorization. In *Proceedings of the 7th ACM Conference on Computer and Communications Security* (pp. 87-96). New York, NY: ACM Press.

Ni, Q., Bertino, E., & Lobo, J. (2008). An obligation model bridging access control policies and privacy policies. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies* (pp. 133-142). New York, NY: ACM Press.

Park, J., & Sandhu, R. (2004, February). The UCONABC usage control model. *ACM Transactions on Information and System Security*, *7*(1), 128–174. doi:10.1145/984334.984339

Zhang, X., Parisi-Presicce, F., Sandhu, R., & Park, J. (2005). Formal model and policy specification of usage control. *ACM Transactions on Information and System Security*, *8*(4), 351–387. doi:10.1145/1108906.1108908

Yehia Elrakaiby holds a Ph.D. degree from the National School of Telecommunication (TELECOM-Bretagne), France in 2010. He received his Engineering degree, in 2003, from the Department of Electronics and Communications Engineering, Cairo University, Egypt. He pursued his studies at the National School of Telecommunications (TELECOM-Bretagne) where he obtained an Engineering degree, in 2005. His research interests include the formal modeling of security concepts and the specification and application of security policies to network and information systems.

Frédéric Cuppens is a full professor at the TELECOM Bretagne LUSSI department. He holds an engineering degree in computer science, a PhD and an HDR. He has been working for more 20 years on various topics of computer security including definition of formal models of security policies, access control to network and information systems, intrusion detection, reaction and counter-measures, and formal techniques to refine security policies and prove security properties. He has published more than 120 technical papers in refereed journals and conference proceedings. He served on several conference program committees and was the Program Committee Chair of ESORICS 2000, IFIP SEC 2004, of SARSSI 2006 and general chair of ESORICS 2009, DPM 2010 and SETOP 2010.

Nora Cuppens-Boulahia is an associate researcher at the TELECOM Bretagne LUSSI department. She holds an engineering degree in computer science and a PhD from SupAero and an HDR from University Rennes 1. Her research interest includes formalization of security properties and policies, cryptographic protocol analysis, formal validation of security properties and thread and reaction risk assessment. She has published more than 80 technical papers in refereed journals and conference proceedings. She has been member of several international program commitees in information security system domain and the PC Chair of Setop 2008, Setop2009, SAR-SSI 2008, CRISIS 2010 and the co-general chair of ESORICS 2009 and SETOP 2010. She is the French representative of IFIP TC11 "Information Security" and she is he co-responsible of the information system security axis of SEE.