A note on the formalisation of UCON

Helge Janicke, Antonio Cau, and Hussein Zedan
Software Technology Research Laboratory, De Montfort University
LE1 9BH Leicester, UK
heljanic@dmu.ac.uk, acau@dmu.ac.uk, hzedan@dmu.ac.uk

ABSTRACT

Usage Control (UCON) Models, similar to Access Control Models, control and govern the users' access to resources and services that are available in the system. One of the major improvements of UCON over traditional access control models is the continuity of the control and the concept of attribute mutability. In this paper we provide an alternative formalisation of the UCON model that relaxes many of the assumptions made in earlier formalisations of the model. We question the enforceability of UCON policies as described by previous formalisations and improve on it.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection – *Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection – *Unauthorized access*

General Terms: Security, Theory

Keywords: Access control, formal specification, security policy, usage control.

1. INTRODUCTION

The UCON model [7] describes the enforcement of a given policy on a session-based single usage process. The novelty of the approach is that it addresses mutable attributes [6] and the continuity of the enforcement. Mutable attributes are associated with the subjects, objects or the system and are updated as side-effects of usage processes. They can be used for example to count the number of times a resource has been accessed. The continuity of enforcement means that a UCON process can be revoked based on conditions that are expressed in terms of attributes.

The UCON model has been first formalised using an extension of the temporal logic of actions (TLA) [5] by Zhang et.al. [9, 10]. Here a single usage process is described in form of a state diagram. System and user actions represent the transitions in the diagram. UCON policies are then defined as logical formulae that postulate temporal relationships between system and user actions of a single usage process.

SACMAT'07, June 20-22, 2007, Sophia Antipolis, France.

As such the formalisation of the UCON model is good, as it provides a unambiguous semantics to the ideas. We believe that this is important for any policy model. However, a strong assumption is made in the formalisation presented in [10] in that only a single usage process is specified. It is assumed that the time-line is finite, viz. it starts with the beginning of the single usage request and ends with the subsequent usage request. This makes it difficult to reason about the interactions of several *concurrent* usage requests, or even *sequences* of usage requests. This complicates the formal analysis of policies. The (side-) effects of a usage process are captured in mutable attributes which are assumed to be persistent over usage processes and can influence subsequent usage control decisions.

In this paper we provide an alternative formalisation of UCON using Interval Temporal Logic (ITL) as the underlying formalism. The expressiveness of ITL allows us to keep the specification comparatively simple and also to drop many of the assumptions made in [10]. Our choice of ITL is based on our experience in the formal specification of dynamically changing security policies [4, 8]. ITL has the advantage over the version of TLA used in [10] that it allows us to naturally define sequences of actions using the "chop" operator. Our focus is to capture the informal requirements of UCON accurately and in a manner that allows for implementations that are correct w.r.t. the formal model.

We first introduce ITL in Section 2. This is followed by a short and concise informal overview of UCON in Section 3. In Section 4 we formalise a single UCON usage process. In Section 5 we define formally UCON authorisation coremodelsand conclude in Section 7.

2. PRELIMINARIES

The key notion of ITL is an *interval*. An interval σ is considered to be a (in)finite sequence of states σ_0, $\sigma_1 \ldots$, where a state σ_i is a mapping from the set of variables *Var* to the set of integer values $\mathbb{Z}$. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to n (one less than the number of states in the interval, so a one state interval has length 0). The syntax of ITL is defined as follows:

$$e ::= \mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \bigcirc v \mid \mathsf{fin}\ v$$
$$f ::= p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$$

where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol. The informal semantics of the most interesting constructs are as follows:

- skip: unit interval (length 1, i.e., an interval of two states).
- $f_1 ; f_2$: holds if the interval can be decomposed into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.
- fin v: value of v in the final state when evaluated on a finite interval, otherwise an arbitrary value.

The following is a list of some derived constructs that are used in the remainder of this paper.

$\Diamond f \mathrel{\widehat{=}} (\neg(\mathsf{true} ; \mathsf{false})) ; f$ sometimes f, i.e., any interval such that f holds over a suffix of that interval.

$\Box f \mathrel{\widehat{=}} \neg \Diamond \neg f$ always f, i.e., any interval such that f holds for all suffixes of that interval.

$\boxdot f \mathrel{\widehat{=}} \neg((\neg f) ; \mathsf{true})$ box-i, i.e., any interval such that f holds over all prefix sub-intervals.

$\boxtimes f \mathrel{\widehat{=}} \neg(\neg(\mathsf{true} ; \mathsf{false}) ; (\neg f) ; \mathsf{true})$ box-a, i.e., any interval such that f holds over all sub-intervals.

$\mathsf{keep}\ f \mathrel{\widehat{=}} \boxtimes (\mathsf{skip} \supset f)$ keep f, i.e., any interval such that f holds over all unit sub-intervals.

$v \leftarrow e \mathrel{\widehat{=}} \neg(\mathsf{true} ; \mathsf{false}) \wedge (\mathsf{fin}\ v) = e$ temporal assignment, i.e., the value of v in the final state will be the value of e.

3. USAGE CONTROL

UCON is a session-based model, viz. between the start of a usage request and its termination the user can perform a number of actions. These actions are not modelled in the original formalisation of UCON presented in [10]. We extend their work here by considering the behaviour of the subject *during* the access. The UCON model supports authorisation, obligation and conditions. Authorisation is concerned with the authorisation of a subject to exercise a specific right. Obligations are concerned with actions the user must perform. Conditions are somewhat similar to authorisations, as they also determine the access of a subject — however they depend on a specific class of attributes that are not modified as a part of the system execution. Conditions are described to depend on the environment of the usage process that can for example be influenced by administrative actions.

Based on the three supported mechanisms UCON_{ABC} is categorised in **A**uthorisation, o**B**ligation and **C**ondition core-models. A more detailed classification dependents on: (1) whether an authorisation, obligation, or condition[1] is checked. This is encoded by the letters A, B and C respectively; (2) whether the check is performed before or during the usage process. This is encoded by the prefixes *pre* and *on*; (3) whether mutable attributes have to be updated. This is encoded as a numeric subscript. An attribute can require update before (*1*), during (*2*) or after (*3*) the usage. If no attributes are updated this is indicated as (*0*).

This classification scheme of UCON core models leads to a large number of possible cases. For example the core authorisation model $preA_2$ defines that *"a usage control decision is determined by authorisations before the usage and one or more subject or object attributes are updated during this usage." [10].* When referring to a class of UCON core-models we use the asterisk notation. For example the class of UCON authorisation models that are checked before the usage process starts is denoted by $preA_*$.

[1] *"Conditions are environmental restrictions that have to be valid before or during a usage process"* [10]. They are defined in terms of system attributes, viz. attributes that can be changed by administrative actions and not by update actions that are performed as part of a usage process.

4. SINGLE USAGE PROCESS

A UCON usage process is characterised by the triplet (s, o, r), where s is the subject that exercises its right r on the object o. We denote in the following the universal set of subjects as $\mathcal{S}$, the universal set of objects as $\mathcal{O}$ and the universal set of rights as $\mathcal{R}$. The usage process can be in one of the following states: *initial*, *requesting*, *denied*, *accessing*, *revoked* or *end*. The current state is described by Zhang et.al. as a function *state* mapping from the triplet (s, o, r) to one of these states. A single usage process is defined by the state diagram in Figure 1.

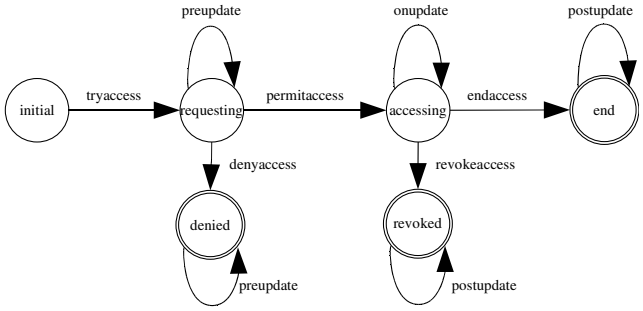

Figure 1: State Transitions adopted from [10]

In the *initial* state the subject s performs the action *tryaccess(s,o,r)* initiating the usage process. The enforcement mechanism (for example a reference monitor (RM)) either denies the access (*denyaccess(s,o,r)*) or proceeds by executing actions to update those attributes, that must be updated before the usage process commences. After the RM updated the relevant attributes (*preupdate(s,o,r)*) it permits the access (*permitaccess(s,o,r)*) and continues to perform all required update actions that must be performed during the ongoing usage process (*onupdate(s,o,r)*). Alternatively the RM may revoke the access if any of the constraints of an $on*_*$ core-model are violated. The subject may end the usage process using the *endaccess(s,o,r)* action. In both cases, the post update actions (*postupdate(s,o,r)*) are performed to update any mutable attributes that require updating.

In the following section we formalise a single usage request using ITL and use this as a starting point to address the specification of UCON policies; their enforceability; and the concurrency of the enforcement.

The behaviour during a single usage process can be naturally expressed in ITL. We extend the original model by additionally considering the behaviour of the subject during

the usage process. This means that once the usage process is authorised, the subject will continue to make requests as part of that usage process. This has not been addressed in [10], however we feel that the behaviour of the subject during the usage process is important and requires modelling. As stated in [10]: *"... an access is not a simple action, but consists of a sequence of actions and events not only from a subject, but also from the system."* An example of such a usage process is the use of a website, that requires a user log-in. Once logged in, the user can access the contents of that site (the access to individual items is assumed to be restricted and controlled by a security mechanism). The usage process ends when the user logs out. Logging in and out are modelled as *tryaccess* and *endaccess*, respectively. A usage process is interactive, i.e. the concrete choice of actions that are executed during a usage request is at the discretion of the user. This distinguished UCON from other transaction oriented access control models.

The user behaviour is described as a sequence of usage requests that are concurrent to the ongoing updates of the RM. We denote the subject's behaviour by $((usage(s, o', r') \oplus idle(s))^* \oplus do(s, o, r))$, where $do(s, o, r)$ is the execution of a primitive action and *idle(s)* denotes that the subject remains idle. The operator $\oplus$ denotes the logical exclusive-or. For the purposes of this paper we adopt a top-down approach for the specification. The following is the high-level specification of a usage process.

$$
\begin{aligned}
usage(s,o,r) \mathrel{\widehat{=}} & \; tryaccess(s,o,r); \\
& (\; denyaccess(s,o,r) \oplus \\
& \quad (\; preupdate(s,o,r) \wedge permitaccess(s,o,r); \\
& \quad\; (\; ((usage(s,o',r') \oplus idle(s))^* \oplus do(s,o,r)) \wedge \\
& \quad\quad onupdate(s,o,r) \;); \\
& \quad\; (revokeaccess(s,o,r) \oplus endaccess(s,o,r)); \\
& \quad\; postupdate(s,o,r) \;))
\end{aligned}
$$

The subject performs the action *tryaccess*(s,o,r) to initiate the usage process (s,o,r). Subsequently the RM either denies the access (*denyaccess*(s,o,r)) or continues executing all pre-update actions that are associated with this usage process. The association is defined by the UCON policy that, following the approach of [10], is expressed as a set of logical formulae that impose constraints on the described behaviour of a usage request. After all pre-update actions have been performed the RM permits the access (*permitaccess*(s,o,r))[2]. Once the access is permitted the subject can initiate other usage processes. We define this here recursively as a sequence of usage processes initiated by the same subject. The recursion anchor is the execution of a primitive action, that is assumed to be an atomic access. We take here the assumption that the choice whether a sequence of new usage processes or the execution of an atomic action is executed is determined by the concrete right r in the request. This means for example that a request like (`Alice,server,ssh`) would constitute the execution of a new usage process, during which other actions can be executed; a request like (`Alice,file,delete`) would constitute an atomic action, that does not allow for any other requests. Concurrent to the subject's behaviour the RM will perform those on-update actions that are associated with the current usage process. The usage process terminates either with the revocation (*revokeaccess*(s,o,r)) of the access or the explicit termination (*endaccess*(s,o,r)) by the subject itself. Subsequently all post-update actions associated with the usage process (s,o,r) are processed by the RM. We further make the reasonable assumption that all atomic actions and update actions terminate within finite time.

The reason to model the subject's behaviour during the usage process is that the effects of the actions may be a reason for a revocation. For example the use of a website that requires the acceptance of a usage licence. Entering and using the website constitutes a single usage process. Clicking on a "decline licence" link within the site constitutes a user action that is part of the usage process. Performing this action can be a reason to revoke the current usage process. User actions can also constitute new usage processes. An example is the use of `ssh` connections to remote Unix systems. It is possible to establish a second `ssh` connection from within an `ssh` session on a remote host. This constitutes a usage process within another usage process. Key advantage of modelling the behaviour is to show the difficulties of implementing UCON $on*_*$ core-models.

The uniform treatment of the user behaviour as part of the usage process has the advantage that obligations become more natural and there is no need to assume that *"an obligation action is always doable whenever required, so that an obligation is not dependent on other permissions."* [10]. Instead obligation actions can be treated as normal usage processes in their own right under the same level of control.

We distinguish between user and system actions. The user initiates and ends a usage using the dedicated actions *tryaccess(s,o,r)* and *endaccess(s,o,r)*. All other actions, are performed by the system, viz. in this case the RM.

In the definition of UCON policies [10], the *state*(s,o,r) function has been used. Using ITL as a formalism, the state of an access is modelled as a state variable *state*(s,o,r) that can assume any value in the set {*initial*, *requesting*, *denied*, *accessing*, *revoked*, *end*}. We can therefore provide high-level specifications of the different actions that constitute a usage process: *tryaccess*$(s,o,r) \mathrel{\widehat{=}}$ $\mathsf{keep}\,(state(s,o,r) = initial) \wedge state(s,o,r) \leftarrow requesting$. Meaning that the usage process (s,o,r) is in its *initial* state when the execution of *tryaccess*(s,o,r) starts and the final state of *tryaccess*(s,o,r) is equal to *requesting*. Recall from the definition of "chop" that the final state of *tryaccess*(s,o,r) coincides with the initial state of *denyaccess*(s,o,r) and *preupdate*(s,o,r). This specification rules out that a usage process is initiated recursively. If recursive usage processes are to be considered a parent-child relationship between usage processes has to be introduced. This results in extending the triplet denoting the usage process by the parent usage process to (p, s,o,r), where p is either a quadruple denoting the parent usage process or $\emptyset$, denoting that there is no parent process. For simplicity we will not consider the parent usage process in this paper, but hint at its application where appropriate. We include here the post-update action to ensure that the

[2]It is not clear in the UCON model whether pre-updates can be performed before the access is denied. Given the state-diagram shown in Figure 1 they could. We take here the viewpoint that this is not possible as a pre-update has been used in [10] to model the costing of an access — here it would not be sensible if the cost of a usage is deducted before the access is rejected. Another of the state-diagrams in [10] also seems to make this assumption. However, if this was the intention of the original model, it can be included without much difficulty.

state of the usage process is reset to the initial state, after all post-updates have been processed:

$$postupdate(s,o,r) \supset \mathsf{keep}\,(state(s,o,r) = end) \wedge state(s,o,r) \leftarrow initial$$

This allows the subject to initiate the usage process again.

5. UCON$_A$ MODELS

Policies for UCON *preA* models define the condition under which the access is permitted. UCON [10] makes the assumption that policies are closed policies and that only positive authorisations are defined. Conflict detection and the resolution of conflicts that is important for hybrid policies, viz. policies that can contain positive and negative authorisation rules, are not addressed. This limits the flexibility of UCON in comparison to other policy models such as [3, 4, 1]. Following the approach of Zhang et.al. we will write a UCON policy as a conjunction of logical formulae. However, we would like to point out that we see the UCON model not as a model for policy specification, but rather as a model of an enforcement mechanism. Policy languages in our view should allow for more structured and compositional approach to specification that allows for the analysis of conflicts, information-flow and other high-level properties. However, for studying the effect that the enforcement of a policy has on mutable attributes the abstraction level is indeed suitable. For UCON authorisation core-models $preA_*$ the policy rules (or rule templates) are as follows:

Authorisation Rules: A usage process (s,o,a) is only permitted if the state formula $C_{autho}(s,o,r)$ is true. A state formula is a formula that does not contain temporal operators. This can be expressed by the following UCON *preA* policy template:

$$\bigwedge_i \boxdot_a\, (permitaccess(s,o,r) \supset C_{autho,i}(s,o,r))$$

Where i is an index distinguishing conditions for all invariants expressed in the policy. This means that in all subintervals where $permitaccess(s,o,r)$ is true, the condition $C_{autho,i}(s,o,r)$ must be true in the initial state. We define the normal form of a UCON *preA* policy to be a single invariant of the form: $\boxdot_a\, (permitaccess(s,o,r) \supset C_{autho}(s,o,r))$, where $C_{autho}(s,o,r) \mathrel{\widehat{=}} \bigwedge_i C_{autho,i}(s,o,r)$.

In [10] the informal assumption is made that when an access in not allowed, it is denied by default. We make this assumption explicit by defining the invariant: $\boxdot_a\, (denyaccess(s,o,r) \supset \neg\, C_{autho}(s,o,r))$. This means that the access is only denied if at least one of the conditions $C_{autho,i}(s,o,r)$ is violated. Using these two invariants it is obvious that the operator exclusive-or between the $denyaccess(s,o,r)$ action and the permission case must be refined by a conditional choice with $C_{autho}(s,o,r)$ as the condition. This represents the choice-point (often called Policy Enforcement Point (PEP)) that is typical for any access control mechanism implementation using RMs.

onA models are one of the key novelties of UCON. They allow for the definition of conditions under which an ongoing usage process is revoked. The formalisation in [10] uses the negated case, where the condition for the usage process to continue is expressed as a conjunction of predicates. We chose here to use the revocation condition to simplify the presentation.

Revocation Rule: When the condition for revocation is observed, the usage process is revoked:

$$\Box\Big(\begin{pmatrix} state(s,o,r) = accessing \\ \wedge\ C_{revoke} \end{pmatrix} \supset revokeaccess(s,o,r);\mathsf{true}\Big)$$

The state-formula C_{revoke}[3] denotes the condition under which the usage process is to be revoked. This means that whenever during an ongoing usage process the revocation condition C_{revoke} is observed, the usage process is revoked *immediately*. This is a strong requirement. The definition in [10] is similar. As they assume that all actions are atomic (i.e. length one, for example assignments) and as they do not explicitly model the behaviour during a usage process they can maintain this strict notion. However, we feel that these assumptions are hindering the applicability of the approach in real system implementations, where actions are likely to be more complex. Under these considerations the requirement for *immediate* revocation seems to be overly strong as it effectively requires to interrupt all actions that the user is currently executing as part of the usage process. Recall that we defined the behaviour of the subject and the RM during the usage process as:

$$((usage(s,o',r') \oplus idle(s))^* \oplus do(s,o,r)) \wedge onupdate(s,o,r))$$

To be able to interrupt the current execution of child-usage processes, viz. those that have been initiated by the user as part of the current process, they must be atomic. However, during a usage session the user can perform a series of different activities, which are most likely not atomic. Indeed, assuming they were atomic, the concept of revocation and ongoing update functions seems to be irrelevant. Consequently we find the revocation rule that was given above is too strong to be implementable/enforceable in real systems. Let us consider a less strict variant.

$$\Box((state(s,o,r) = accessing \wedge C_{revoke}) \supset \bigwedge_{o'\in\mathcal{O},r'\in\mathcal{R}} \boxdot_a \neg\, tryaccess(s,o',r')\,;\, revokeaccess(s,o,r)\,;\,\mathsf{true})$$

This means that once the revocation condition is true during the access the user may not initiate any other usage process. We denote here by $\mathcal{O}$ and $\mathcal{R}$ the universal sets of objects and rights. W.r.t. the behaviour of the user it means that the current atomic request is allowed to complete, the user may remain idle, but no new usage process can be initiated. We assume here that any atomic action terminates in finite time. The advantage of this less strict notion of revocation over the previously presented and the one in [10] is that it is implementable/enforceable without making the assumption that actions are atomic (i.e. of length one).

Updating Mutable Attributes: Mutable attributes are introduced in [6]. They provide a structured way to maintain (meta-) information on subjects and objects in the settings of usage control. Most interesting for the purposes of this paper are attributes that are associated with objects and subjects and that can be changed by the system as part of a usage process. The update functions $preupdate(s,o,r)$, $onupdate(s,o,r)$ and $postupdate(s,o,r)$ represent the (side-) effects that the usage process (s,o,r) has on mutable object and subject attributes.

[3]The condition used here is equivalent to $\neg(p_1 \wedge \cdots \wedge p_i)$ as it is used in [10].

We denote here by $preupdate(s,o,r)$ **all** update actions that must be performed before the usage (s,o,r). Similarly for $onupdate(s,o,r)$ and $postupdate(s,o,r)$. The concrete update actions are defined in the UCON policy. For simplicity we assume that a separate policy is defined for each individual usage process (s,o,r)[4].

Pre-update Rule: The pre-update must perform the update action: $preupdate(s,o,r) \supset \Diamond update(A)$. Here $update(A)$ denotes the update action of a specific attribute A. An update is the assignment of that attribute to a new value, for example: $update(s.credit) \mathrel{\widehat{=}} s.credit \leftarrow s.credit - o.value$. The definitions of update actions are part of the policy. It is important to note that the temporal operator sometimes ($\Diamond$) in this specification does mean at some point in the interval over which the formula $preupdate(s,o,r)$ holds. The use of the sequential composition in the overall definition of a usage process bounds the scope of this temporal operator. This allows for a relatively simple formal specification of the behaviours without making assumptions on the time-line (i.e. intervals) as in [10].

On-update Rule: The on-update must perform the update action: $onupdate(s,o,r) \supset \Diamond update(A)$. This rule ensures that the update action is executed. As noted by [10] stronger update rules can be useful. For example the following rule ensures that the update is performed in every state[5]:
$onupdate(s,o,r) \supset \mathsf{keep}\ update(A)$. Varying from [10] we exclude the final state before the action is ended or revoked from the update. For example when counting usage duration the $endaccess(s,o,r)$ action would not be counted. The ongoing updates also have difficulties in concurrent settings, as they do not allow for any interleaving implementation. A RM that is enforcing such an on-update requirement cannot engage in any other activity, for example also not process any request the user makes as part of the usage process. To model examples that rely on usage-time we therefore suggest to assume that a global clock T is available with the specification: $T = 0 \wedge \mathsf{keep}\,(T \leftarrow T + 1)$. This corresponds for example to the system's hardware clock.

Example: Alice's Yahoo-Session should be revoked after 20 min of usage. We record the usage start time in the subject attribute *Alice.StartTime* using a pre-update.

$$preupdate(Alice, Yahoo, email) \supset \Diamond update(Alice.StartTime)$$
$$update(Alice.StartTime) \mathrel{\widehat{=}} Alice.StartTime \leftarrow T$$

The revocation of the usage process should take place when 20 min have been elapsed. This can be captured by the revocation rule:

$$\begin{aligned}\Box((&state(Alice, Yahoo, email) = accessing\\ &\wedge T - Alice.StartTime > 20min) \supset\\ &((\bigwedge_{o\in\mathcal{O}, r\in\mathcal{R}} \boxdot \neg\, tryaccess(Alice, o, r);\\ &revokeaccess(Alice, Yahoo, email)\,;\mathsf{true}))\end{aligned}$$

To distinguish between actual usage time during a usage session and idle time, we could alternatively accumulate the duration of all child usage-processes that are initiated within the usage session. This would require to model the parent usage process as described earlier. The example can be classified as a onA_1 core model, viz. the check is performed during the usage, it is an authorisation check and attributes are updated before the access.

We would recommend to exclude the continuous on-update rules from the UCON model, as the implementation or enforcement in the form $\mathsf{keep}\ update(A)$ seems to be infeasible in most real systems. Weaker continuous on-update requirements could for example be expressed as $(update(A) \oplus idle)^*$ — which would be implementable. However, this form is not suitable for the implementation of a usage time counter as suggested in [10], because no guarantees on the frequency of updates can be provided. To be able to implement a reliable continuous update functionality the underlying system must exhibit real-time properties, viz. guarantee the execution of an action within a deterministic, guaranteed time-span.

Post-update Rule: The post-update must perform the update action: $postupdate(s,o,r) \supset \Diamond update(A)$. A post-update is performed after the access has been ended by the user or revoked by the system. An example usage would be to store accumulated usage times in a subject attribute for billing purposes or in an object attribute to allow for resource usage rankings.

We modelled here the semantics of update actions to match closely the semantics provided in [10]. However, it is a rather loose semantics, as an implementation is only required to execute the update action *at least once*, compared to the arguably more typical requirement of *exactly once*. In [10] this may not be that obvious as the operator that is used to describe this property is called "once" ($\blacklozenge$) with the meaning at some point in the past.

The formula $\Diamond update(A)$ describing the update rules still holds when the update is performed more than once. In the example given for the pre-update rule an enforcer that deducts the cost of the resource n times ($n > 0$) from the subject's credit would correctly enforce the policy — which is counter-intuitive.

Another point of concern is that in [10] it is assumed that *"... without loss of generality, we assume that in each logical formula there is at most one update for an attribute, as multiple updates on the same attribute have the same effect as the last one"*.

We would argue the generality of the claim – especially in concurrent settings. Consider the case, where the updates are defined as $update_a(A) \mathrel{\widehat{=}} A \leftarrow A + 1$ and $update_b(A) \mathrel{\widehat{=}} A \leftarrow A + 2$. Analogously to [10] multiple updates would be defined as: $preupdate(s,o,r) \supset \Diamond update_a(A) \wedge \Diamond update_b(A)$.

Given this specification both actions can occur sequentially in any order. The result would in any case be that the value of the attribute A is increased by 3 — which is different from the effect of the last one (either $update_a(A)$ or $update_b(A)$).

Even if we would assume as in [10] that every attribute is updated only once, there is still an issue of inter-dependencies of updates. Consider the case of three mutable attributes A,B and C that are initially 0. A usage process could require to update all three attributes using the update actions: $update(A) \mathrel{\widehat{=}} A \leftarrow 1$, $update(B) \mathrel{\widehat{=}} B \leftarrow 2$, $update(C) \mathrel{\widehat{=}} C \leftarrow A + B$, together with the pre-update rule:
$preupdate(s,o,r) \mathrel{\widehat{=}} \Diamond update(A) \wedge \Diamond update(B) \wedge \Diamond update(C)$
In this case the outcome of the policy enforcement could

[4]More general policies that define constraints on sets of subjects or actions can be transformed into this normal form. See for example [8] for details.

[5]This assumes that the update actions are of length one, for example a single assignment.

yield the attribute values: $A = 1, B = 2, C = 0$ if the update of C happens to be performed first. $A = 1, B = 2, C = 1$ if first A happens before the update of C and B is updated last. Other combinations are possible. This means that the outcome of the pre-update rule is not determined by the policy. This makes the structure of the update rules difficult to use in real applications. We noted that the formal specification of scheme rules in [10] does not provide support for multiple attribute updates, although it has been used in several examples. However, the scheme rules for $preO_*$ of models (CR1) allow for the specification of several obligations for which similar conflicting situations can be found. We propose two alternative forms of update rules in the following.

Alternative Semantics for Update Rules: We have described previously that the semantic of update rules defines updates to happen *at least once*. The following definition would capture *exactly once*:

$$preupdate(s,o,r) \supset (\boxdot \neg update(A)); update(A); (\boxdot \neg update(A))$$

This states that the interval over which $preupdate(s,o,r)$ holds can be decomposed into three parts. In the first part the update is never performed, in the second part the update is performed exactly once, and in the last part the update is never performed. Although more complex than the original definition, it captures the informal requirement more precisely. This alternative formalisation does only address the problem of an enforcement mechanism executing the update action more than once. The problem of inter-dependent updates is more general — and more difficult to solve.

An approach to this problem is that all update actions are defined using only the temporal assignment operator. For the above example of the dependent updates of A,B and C this would mean that:

$$preupdate(s,o,r) \ \supset \ update(A) \wedge update(B) \wedge update(C)$$

Using this specification the result is deterministic: $A = 1$, $B = 2$ and $C = 0$. The expressions on the right-hand side are evaluated in the first state of the interval over which $preupdate(s,o,r)$ holds. Assuming that all UCON update actions can be expressed as simple assignments of expressions, this provides a suitable semantics. Another advantage of this approach is that multiple updates of the same attribute lead to logical conflicts that can be identified using formal analysis or model-checking.

6. UCON$_\mathbf{B}$ AND UCON$_\mathbf{C}$ MODELS

Due to space limitations we cannot discuss UCON$_\mathbf{B}$ models in this paper. W.r.t. UCON$_\mathbf{C}$: Conditions express environmental constraints on usage processes. They are defined in terms of immutable[6], administrative attributes. Their formalisation is therefore identical with those presented in Section 5.

7. CONCLUSION

In this paper we have presented an alternative formalisation of the UCON model [7]. We used Interval Temporal Logic (ITL) for the formalisation, as we feel that this is a more natural logic to express the model than the extended Temporal Logic of Actions (TLA) presented in [10].

[6]immutable in the sense that they can not be modified as a side-effect of usage-processes.

We improved upon the original formalisation in [10] by explicitly modelling the user behaviour during an ongoing usage process. This allowed us to identify shortcomings in the original specification with respect to the models enforceability. In addition we significantly reduced the number of informal assumptions that were made in [10]: (1) We described UCON policies as constraints on the overall system behaviour and dropped the assumption of a time-line (interval) that is limited to one single usage process. (2) We also formalised the exact condition under which a usage request is denied and dropped the informal assumption, that a usage process is denied if not allowed. (3) By relaxing the condition for revocation rules, we dropped the assumption that all actions that are performed as part of a usage process are atomic. As a consequence the actions that form part of a usage process can now be controlled by the RM and also constitute whole usage processes. (4) The multiple update assumption has shown to be not general. We provided two examples that contradict the claim and proposed a new specification of multiple updates that does not suffer from this problem.

We highlighted the difficulty of continuous on-update implementations in non-real-time systems. We proposed the introduction of a globally accessible clock to capture protection requirements that make use of a subject's usage time. We pointed out the specification of update rules in [10] is weak, as it allows the RM to perform the update *more* than once. We provided an alternative rule representation that defines that an update action is performed *exactly* once.

UCON policies define the enforcement of protection requirements at a relatively low level of abstraction. Other approaches such as [8, 4, 2] have the benefit of addressing the specification of policies at a higher level of abstraction and also provide mechanisms to derive concrete enforcement mechanisms (such as update actions) from these specifications.

8. REFERENCES

[1] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorisation Language. Technical report, Microsoft Research, 2006.

[2] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[3] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[4] H. Janicke, A. Cau, F. Siewe, H. Zedan, and K. Jones. A Compositional Event & Time-based Policy Model. In *Procceedings of POLICY2006*. IEEE, 2006.

[5] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, 1994.

[6] J. Park, X. Zhang, and R. S. Sandhu. Attribute mutability in usage control. In C. Farkas and P. Samarati, editors, *DBSec*, pages 15–29. Kluwer, 2004.

[7] R. Sandhu and J. Park. The UCON$_{ABC}$ usage control model. In *Proceeding of the Second International Workshop on Mathematical Method, Models and Architectures for Computer Networks Security*, 2003.

[8] F. Siewe. *A Compositional Framework for the Development of Secure Access Control Systems*. PhD thesis, De Montfort University, 2005.

[9] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *ACM Proceedings of SACMAT '04*, pages 1–10, 2004.

[10] X. Zhang, F. Rarisi-Presicce, J. Park, and R. Sandhu. Formal Model and Policy Specification of Usage Control. *ACM TISSEC*, 2005.