Mechanisms for Usage Control

A. Pretschner, M. Hilty, D. Basin
Information Security, ETH Zurich, Switzerland
{pretscha,hiltym,basin}@inf.ethz.ch

C. Schaefer, T. Walter
DoCoMo Euro-Labs, Munich, Germany
{schaefer,walter}@docomolab-euro.com

ABSTRACT

Usage control is a generalization of access control that also addresses how data is used after it is released. This problem is particularly challenging in distributed settings, where servers, acting as data providers, release sensitive data to clients, acting as data consumers. We present a formal model for different mechanisms that can enforce usage control policies on the consumer side.

1. INTRODUCTION

Usage control [4] generalizes access control by controlling not only who may access which data, but also how the data may be used or distributed afterwards. We consider distributed settings, where processes act in the roles of data providers and data consumers. A data provider can give sensitive data to a data consumer based on conditions both on the past (which we ignore in this paper) and the future. The latter requirements come as *obligations* that restrict the future usage of data. When data providers release data, they would like mechanisms on the consumer's side to enforce their requirements. They would also like to check consistency of policies, and if mechanisms are capable of enforcing them. To this end, we present a model of usage control mechanisms that formalizes the problem domain at a realistic level of complexity. Mechanisms are modeled as trace transformers that map desired (attempted) events into actual usage-controlled events. Our model allows the specification of a wide range of usage control mechanisms — those based on inhibition, execution, delay, modification, and signaling — which includes all those found in practice. Moreover, it caters for concurrent and ongoing usages. The automated analysis part is, for reasons of space, not part of this paper [2]. In the following, we assume some familiarity with the Z language, the formalism employed in this paper.

2. SETUP AND POLICY LANGUAGE

Usage control requirements are negotiated between data providers and consumers, and enforced using consumer-side mechanisms. Data consumers request data. Using negotiators, the consumer and provider negotiate the usage request, a topic not treated in this paper. Upon successful negotiation, data is transferred from the provider to the consumer and the usage control requirements are activated. From this point onward, mechanisms on the consumer's side will enforce the requirements (which is, in general, not fully possible for all requirements: taking photographs of a monitor will always be an option). We assume the consumer possesses a secure data store and that, prior to usage, all data is routed through usage control mechanisms whenever it leaves the store. This paper's sole focus is on the mechanisms.

Semantic Model. Our model is based on classes of parameterized events. The event classes include *usage* and *other*, with the latter including activation events. An *event* consists of the event name and parameters, represented as a partial function from names to values. We will describe instantiated event parameters as $(name, value)$ pairs. An example is the event $(play, \{(obj, o)\})$, where $play$ is the name of the event and the parameter obj has the value o.

The definition of events in the Z language is shown below. $EName$, $PName$, and $PVal$ define disjoint basic types for event names, parameter names, and parameter values.

$[EName, PName, PVal];\ Ev == EName \times (PName \nrightarrow PVal)$
$EClass ::= usage \mid other;\quad getclass : EName \rightarrow EClass$

Events are ordered via a refinement relation *refinesEv*. Event e_2 refines event e_1 iff e_2 has the same event name as e_1 and all parameters of e_1 have the same value in e_2. e_2 can also have additional parameters specified; see [1] for a formalization. The idea is that when specifying usage control requirements, we do not want to specify all parameters. For instance, if the event $(play, \{(obj, o)\})$ is prohibited, then the event $(play, \{(obj, o), (device, d)\})$ should also be prohibited. The event $(nil, \varnothing)$ is reserved and denotes no event.

We need a language to define obligations. Its semantics will be defined over traces: mappings from abstract points in time—represented by the natural numbers—to possibly empty sets of *maximally refined* events. We cater for usage events that execute over a time interval, e.g., watching a movie. For example, if a time step lasts 1 minute and a user plays a movie for 3 minutes, there will be 3 consecutive events indexed with *start* and *ongoing*, respectively: $((play, \{(obj, mov)\}), start)$, and twice $((play, \{(obj, mov)\}), ongoing)$. The data type *IndEv* defines such indexed events. We also need to express that usage is *desired* by a user (*DesIndEv*): not every attempted usage is executed.

$Index ::= start \mid ongoing;\quad IndEv == Ev \times Index$
$DesIndEv ::= TRY \langle\langle IndEv \rangle\rangle;\ Trace : \mathbb{N} \rightarrow \mathbb{P}(IndEv \cup DesIndEv)$

We use the temporal logic OSL [1] as language for usage control requirements. Its syntax is provided by Φ^+ (+ for future; we slightly deviate from the Z syntax).

$$\begin{aligned}
\Phi^+ ::= {} & E_{fst}\langle\!\langle Ev\rangle\!\rangle \mid E_{all}\langle\!\langle Ev\rangle\!\rangle \mid T_{fst}\langle\!\langle Ev\rangle\!\rangle \mid T_{all}\langle\!\langle Ev\rangle\!\rangle \mid \\
& \neg\Phi^+ \mid \Phi^+ \wedge \Phi^+ \mid \Phi^+ \vee \Phi^+ \mid \Phi^+ \Rightarrow \Phi^+ \mid \underline{until}\langle\!\langle \Phi^+ \times \Phi^+\rangle\!\rangle \mid \\
& \underline{always}\langle\!\langle \Phi^+\rangle\!\rangle \mid \underline{after}\langle\!\langle \mathbb{N} \times \Phi^+\rangle\!\rangle \mid \underline{within}\langle\!\langle \mathbb{N} \times \Phi^+\rangle\!\rangle \mid \\
& \underline{during}\langle\!\langle \mathbb{N} \times \Phi^+\rangle\!\rangle \mid \underline{repmax}\langle\!\langle \mathbb{N} \times \Phi^+\rangle\!\rangle \mid \\
& \underline{replim}\langle\!\langle \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \Phi^+\rangle\!\rangle \mid \underline{repuntil}\langle\!\langle \mathbb{N} \times \Phi^+ \times \Phi^+\rangle\!\rangle
\end{aligned}$$

We distinguish between the start of an action (syntactically: E_{fst}; semantically: an indexed event with index *start*) and any lasting action (syntactically: E_{all}; semantically: indexed events with any index). T_{fst} and T_{all} refer to the respective *attempted* actions drawn from set *DesIndEv*. When specifying events in obligations, by virtue of the refinement relation *refinesEv*, there is an implicit universal quantification over unmentioned parameters. $\neg, \wedge, \vee, \Rightarrow$ have the usual semantics. Our $\underline{until}$ operator is the weak-until operator from LTL. $\underline{after}(n)$ refers to the time after n time steps. $\underline{during}$ specifies that something must constantly hold during a specified time interval and $\underline{within}$ requires something to hold at least once during a specified time interval.

Cardinality operators restrict the number of occurrences or the duration of an action. The $\underline{replim}$ operator specifies lower and upper bounds of time steps within a fixed time interval in which a given formula holds. The $\underline{repuntil}$ operator is independent of any time interval: it limits the maximal number of times a formula holds until another formula holds (e.g., the occurrence of some event). $\underline{repmax}$ defines the maximal number of times a formula may hold in the indefinite future. These cardinality operators are also used to express limits on the accumulated usage time, e.g., by using $E_{all}(e)$ as an argument for limiting the accumulated time of usage e. For instance, $\underline{replim}(20, 0, 5, E_{all}(play, \{(obj, mA)\}))$ specifies that movie mA may be played for at most five time units during the next twenty time units. Similarly, $\neg\underline{replim}(20, 0, 2, E_{fst}(play, \{(obj, mB)\})) \Rightarrow \underline{after}(25, E_{fst}(notify, \{(rcv, subA)\}))$ specifies that if the movie mB is started more than twice during the next 20 time units, then subject $subA$ will be notified after 25 time units. We omit a formalization of the semantics, $\models_f$, for brevity's sake [1, 2].

Obligations represent usage control requirements ("delete after 30 days"; "do not distribute"). Each obligation has a name that indexes respective activation events, and a formula from Φ^+ that must hold after its activation. A trace satisfies an obligation iff the obligations's formula holds at the moment of the obligation's activation, which is captured by a dedicated event.

3. MECHANISMS

Mechanisms are the means by which usage can be controlled. They are installed on the consumer's side and may be *configured* by the provider. Mechanisms consist of two parts: a description of when they are applicable, often including a triggering event, and the respective actions to be taken. We have already introduced the distinction between desired and actual usages. Mechanisms are triggered by desired events. If the mechanism's condition holds, the mechanism's actions are executed. Otherwise, the overall system guarantees that the desired usage is transformed into the respective actual usage. We will later model the effect of a set of mechanisms (not the mechanisms themselves) as a function that maps possible traces—including desired usages and activations only—to usage-controlled traces.

Since enforcement mechanisms can only make decisions based on their current knowledge, we use a temporal logic of the past, Φ^-, to describe the conditions under which mechanisms perform their tasks. This is done by dualizing the respective future operators in Φ^+. The straightforward semantics, $\models_{f^-}$, is defined elsewhere [2]. To express the past nature of all operators, we superscript them with a $-$.

Because mechanisms can be configured, we allow for variables in the definition of conditions and triggering actions. The latter are then events where either some parameter values or the name of the event is left unspecified. The respective language is called Φ_v^- which introduces a syntactic category for variables (and events with variables, *VarEv*). Elements from Φ_v^- can be straightforwardly instantiated to elements from Φ^- by means of a substitution function called $subst_f$. Finally, to describe the overall functionality of mechanisms, we will use mixed formulae, $\Phi^\pm$, which combine both future and past formulae in a restricted manner, as in schema *Mechanism* below (see [2] for the semantics, $\models_{f^\pm}$).

Mechanism

$$\begin{array}{l}
\varphi_p : \Phi_v^-; \quad \varphi : \Phi^-; \quad \alpha : \Phi^+; \quad \psi : \Phi^\pm; \quad m : Mode \\
ua_p : VarEv; \quad ua : Ev; \quad \sigma : Var \rightarrow\!\!\!\!\!+ (PVal \cup EName) \\
\hline
\varphi = subst_f(\varphi_p, \sigma) \wedge ua = subst(ua_p, \sigma) \\
\forall\, s : Trace;\ t : \mathbb{N} \bullet (s, t) \models_{f^-} \varphi \Leftrightarrow (rmTry(s), t) \models_{f^-} \varphi \\
\psi = \displaystyle\bigwedge_{e \in maxRefs(ua)} (T_m(e) \wedge^\pm (E_m(e) \Rightarrow^\pm \varphi)) \Rightarrow^\pm \alpha
\end{array}$$

The general form of mechanisms is defined by the schema *Mechanism*. Mechanisms are defined by formulae ψ of type $\Phi^\pm$ (abbreviated by $\psi : \Phi^\pm$). ψ references the triggering (desired) action *ua* (possibly *nil*, i.e., no trigger is provided) together with its *Mode m*. The latter indicates whether or not the triggering action refers to a first usage or to all usages, i.e., first and ongoing usages ($Mode ::= fst \mid all$). We do not require *ua* to be maximally refined as we often want to specify an entire class of triggering actions. For instance, we may wish to specify that playing a movie is prohibited, without enumerating all possible devices. ψ also references the condition $\varphi : \Phi^-$ under which the mechanism performs its task, and the effects $\alpha : \Phi^+$. The definition does not relate mechanisms to sets of traces of a usage-controlled system yet; this will be done below. *maxRefs* computes the set of maximally refined events for any event; this is needed because traces consist of maximally refined events only.

If the triggering action of a mechanism is executed—as indicated by a respective *TRY* event and syntactically captured by predicates T_{fst} and T_{all} (in the schema, we use variable $m : Mode$ as index)—then we require the following. Under the assumption that adding the actual usage satisfies φ, the effect α of the mechanism is "executed." This means that, in principle, a mechanism can itself invalidate its condition (and this can also be done by other mechanisms).

When applying a mechanism, the respective $TRY(\cdot)$ events are kept because more than one mechanism may be applicable. The remaining desired actions that are not controlled by any mechanism—either because they are not triggering events of any mechanism or because no condition was true—are transformed into actual usages in the end. We require φ not to depend on any *TRY* events. This is what the auxiliary function $rmTry : Trace \rightarrow Trace$ is needed for.

Mechanisms are parameterized. The respective variables are used in both the triggering actions (ua_p) and the conditions (φ_p). By providing substitutions σ, variables are instantiated. The process of instantiating variables models the *configuration of mechanisms.*

Four Classes of Mechanisms. There are four classes of control mechanisms [2]: inhibition, finite delay, modification of usage, and execution of actions. Inhibition reduces the set of possible executions. Delays postpone usages. Modifications change usages, for instance, by converting an editing usage into a reading usage only or by lowering the quality of an output signal. The execution of actions adds events to the execution, e.g., logging events, or sending signals to the provider. In principle, all usage control policies could be enforced by inhibitors and executors only; modifiers and delayers merely increase user convenience. For instance, a mechanism may not categorically forbid watching unpaid movies, but rather reduce the quality considerably.

Because we differentiate between desired and actual usages, the effect of a mechanism may *add* events to a trace, but never remove any. Inhibition is then modeled as *not adding* the actual usage corresponding to a desired usage. The effect of applying a set of mechanisms to a trace is the trace that satisfies all formulae of the mechanisms and that, in addition, exhibits, at each time step, a minimum set of events—mechanisms may add events that relate to their "task" but not arbitrary events. This is handled below.

Executors add events to a trace. If event ua is desired at time t and φ also holds at t, then the sequences of events in $exacts$ are executed. The composition of mechanisms (given below) ensures that the desired usage ua is also converted into an actual usage, provided that this would not affect any other mechanism. Function $subst$ substitutes variables w.r.t. the substitution provided as σ by schema *Mechanism.*

Executor

$$\begin{array}{l}
\underline{Mechanism}; \quad exacts_p : \mathbb{P}\,\mathrm{seq}\, VarEv; \quad exacts : \mathbb{P}\,\mathrm{seq}\, Ev \\
\hline
exacts = \mathrm{map}(\lambda\, t : \mathrm{seq}\ VarEv \bullet \mathrm{map_s}(\lambda\, x : VarEv \bullet \\
\qquad\qquad subst(x, \sigma), t), exacts_p) \\
\alpha = \bigwedge_{es \in exacts} \bigwedge_{i=1}^{\#es} \underline{after}(i-1, T_{all}(es(i)))
\end{array}$$

The *Eraser* deletes an object (variable $V(1)$) $V(2)$ days after it was stored. Note the absence of a triggering action.

Eraser

$$\begin{array}{l}
Executor[\underline{before}^-(V(2), E_{fst}((store, \{(obj, V(1))\})))\wedge^- \\
\qquad \neg^-\ \underline{within}^-(V(2), E_{fst}((delete, \{(obj, V(1))\})))/\varphi_p, \\
\qquad (nil, \varnothing)/ua_p, fst/m, \{\langle(delete, \{(obj, V(1))\})\rangle\}/exacts_p] \\
\hline
\end{array}$$

Modifiers replace an event ua by a set of events $modifyBy$ under certain conditions. Constraint 2 of schema *Modifier* states that, in contrast to executors, modifiers are always triggered by a desired usage and, furthermore, that events are replaced by different events. The specified effect, α, is similar to that of *Executors*, except that the events that are added are drawn from the set $modifyBy$ rather than $exacts$ and that ua is not added if the mechanism is applicable. An example of a modifier is given by the schema *NoPayNoGood*: songs not paid for will only be played in reduced quality.

Modifier

$$\begin{array}{l}
\underline{Mechanism}; \quad modifyBy_p : \mathbb{P}\ VarEv; \quad modifyBy : \mathbb{P}\ Ev \\
\hline
modifyBy = \mathrm{map_s}(\lambda\, x : VarEv \bullet subst(x, \sigma), modifyBy_p) \\
ua \neq (nil, \varnothing) \wedge ua \notin modifyBy \\
\alpha = \neg E_m(ua) \wedge \bigwedge_{e \in modifyBy} T_{all}(e)
\end{array}$$

NoPayNoGood

$$\begin{array}{l}
Modifier[\underline{always}^-(\neg^-\ E_{fst}((pay, \{(obj, V(1))\})))/\varphi_p, \\
\qquad (play, \{(obj, V(1), (qual, full))\})/ua_p, all/m, \\
\qquad \{(play, \{(obj, V(1)), (qual, red)\})\}/modifyBy_p] \\
\hline
\end{array}$$

Inhibitors prevent specified events from happening when given conditions are met. Hence, they are modifiers with an empty *modifyBy* set. An example for an inhibiting control mechanism is given by the schema *Subscription.* It ensures that each *play* is preceded by a *pay* that dates back at most $V(2)$ days.

Inhibitor

$$\begin{array}{l}
Modifier[\varnothing/modifyBy_p] \\
\hline
\end{array}$$

Subscription

$$\begin{array}{l}
Inhibitor[\neg^-\ \underline{within}^-(V(2), E_{fst}((pay, \{(obj, V(1))\})))/\varphi_p, \\
\qquad (play, \{(obj, V(1))\})/ua_p, all/m] \\
\hline
\end{array}$$

Delayers. Delaying mechanisms perform a sequence of events, $seqe$, if ua is desired and condition φ is true. Once this sequence of events has occurred, the request to execute ua is expressed again. Delayers always delay an attempted usage, which motivates the second constraint of schema *Delayer.* We omit an example for brevity's sake [2].

Delayer

$$\begin{array}{l}
\underline{Mechanism}; \quad seqe_p : \mathrm{seq_1}\ VarEv; \quad seqe : \mathrm{seq_1}\ Ev \\
\hline
ua \neq (nil, \varnothing) \wedge seqe = \mathrm{map}(\lambda\, x : VarEv \bullet subst(x, \sigma), seqe_p) \\
\alpha = \neg E_m(ua) \wedge \bigwedge_{i=1}^{\#seqe} \underline{after}(i-1, T_{all}(seqe(i))) \wedge \\
\qquad \underline{after}(\#seqe, T_m(ua))
\end{array}$$

Composition and Semantics. In this paper, we use trace transformers to express properties of usage-controlled systems, as opposed to simply using sets of traces. If the semantics of a set of mechanisms and the system specification were both given by sets of traces, then we could simply define the composition of the two as the intersection of the sets of traces. However, this imposes constraints on the system: it must be liberal enough, i.e., under-specified, to allow for the effects of applying mechanisms (because we do not confine ourselves to inhibitors, applying mechanisms is not simply a trace refinement). In contrast, specifying mechanisms as trace transformers allows us to apply them to arbitrary systems that, when specified, must not take into account the possibility of future mechanisms being applied.

Simultaneously applied mechanisms may interfere with each other: two mechanisms may be triggered by the same event; the effect of one mechanism may trigger another (and this may lead to loops); and the effect of one mechanism may invalidate the condition of a mechanism that was applied before. We simply forbid such cases here [2].

The schema *CombinedMechanism* above defines the composition of mechanisms in terms of functions μ and μ^*. The trace transformer μ applies all mechanisms and possibly adds usages that are allowed by the mechanisms (e.g., action ua of executors). The function μ^* afterwards transforms *TRY* events that are not controlled by any mechanism into the respective actual usage, and then removes all remaining *TRY* events. We use an auxiliary function, $addEv : Trace \times \mathbb{N} \times Ev \times Mode \rightarrow Trace$, that adds an event with a given mode to a trace at a specified time.

Constraint 2 of schema *CombinedMechanism* specifies three

CombinedMechanism

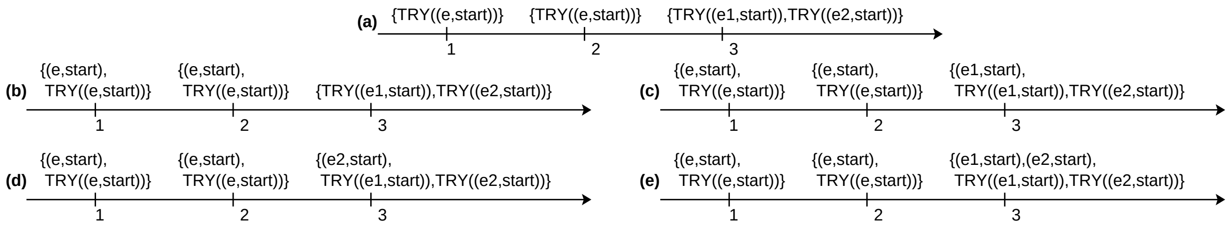
$M : \mathbb{P}\,\mathit{Modifier};\quad I : \mathbb{P}\,\mathit{Inhibitor};\quad D : \mathbb{P}\,\mathit{Delayer};\quad E : \mathbb{P}\,\mathit{Executor}$

$\mu : \mathit{Trace} \rightarrow \mathit{Trace};\quad \mu^* : \mathit{Trace} \rightarrow \mathit{Trace};\quad \mathit{ucmechs} : \mathbb{P}(\mathit{Modifier} \cup \mathit{Inhibitor} \cup \mathit{Delayer} \cup \mathit{Executor})$

$\mathit{ucmechs} = M \cup I \cup D \cup E$

$$\forall\, i : \mathit{Trace};\ t : \mathbb{N};\ e : \mathit{Ev};\ \mathit{md} : \mathit{Mode} \bullet i(t) \subseteq (\mu(i))(t) \wedge (\mu(i), t) \models_{f^\pm} \bigwedge_{u \in \mathit{ucmechs}} u.\psi$$
$$\wedge \Big((\mathit{addEv}(\mu(i), t, e, \mathit{md}), t) \models_{f^-} T_{\mathit{md}}(e) \wedge^- \bigwedge_{u \in \{v:\mathit{ucmechs} \mid (e\ \mathit{refinesEv}\ v.ua \vee v.ua = (\mathit{nil}, \varnothing)) \wedge v.m = \mathit{md}\}} \neg^- u.\varphi\Big)$$
$$\Rightarrow (\mu(i), t) \models_f E_{\mathit{md}}(e)$$
$$\forall\, i : \mathit{Trace};\ t : \mathbb{N};\ e : \mathit{Ev};\ \mathit{md} : \mathit{Mode} \bullet \neg \exists\, o : \mathit{Trace} \bullet (o, t) \models_{f^\pm} \bigwedge_{u \in \mathit{ucmechs}} u.\psi$$
$$\wedge \Big((\mathit{addEv}(o, t, e, \mathit{md}), t) \models_{f^-} T_{\mathit{md}}(e) \wedge^- \bigwedge_{u \in \{v:\mathit{ucmechs} \mid (e\ \mathit{refinesEv}\ v.ua \vee v.ua = (\mathit{nil}, \varnothing)) \wedge v.m = \mathit{md}\}} \neg^- u.\varphi\Big) \Rightarrow (o, t) \models_f E_{\mathit{md}}(e)$$
$$\wedge\, i(t) \subseteq o(t) \wedge o(t) \subseteq (\mu(i))(t) \wedge \exists\, t' : \mathbb{N} \bullet i(t') \subset o(t') \wedge o(t') \subset (\mu(i))(t')$$
$$\forall\, i : \mathit{Trace};\ t : \mathbb{N} \bullet (\mu^*(i))(t) = ((\mu(i))(t) \setminus \mathit{DesIndEv}) \cup \{e : \mathit{Ev};\ j : \mathit{Index} \mid$$
$$\mathit{TRY}((e, j)) \in (\mu(i))(t) \wedge e \in \mathit{Ev} \setminus \{m : \mathit{Mechanism} \mid m \in \mathit{ucmechs} \bullet m.ua\} \bullet (e, j)\}$$

− − −plus noninterference constraints − − −



(a)
{TRY((e,start))}
{TRY((e,start))}
{TRY((e1,start)),TRY((e2,start))}
1
2
3
(b)
{(e,start), TRY((e,start))}
{(e,start), TRY((e,start))}
{TRY((e1,start)),TRY((e2,start))}
(c)
{(e,start), TRY((e,start))}
{(e,start), TRY((e,start))}
{(e1,start), TRY((e1,start)),TRY((e2,start))}
(d)
{(e,start), TRY((e,start))}
{(e,start), TRY((e,start))}
{(e2,start), TRY((e1,start)),TRY((e2,start))}
(e)
{(e,start), TRY((e,start))}
{(e,start), TRY((e,start))}
{(e1,start),(e2,start), TRY((e1,start)),TRY((e2,start))}

Figure 1: Possible effects of an inhibitor

conditions, one per conjunct. First, the effect of applying mechanisms at most adds events. Second, the resulting trace satisfies $\underline{\mathit{always}}(u.\psi)$ for all mechanisms u in the system. Third, those desired usages for which no mechanism was applicable are transformed into actual usages, *unless their transformation would render a mechanism applicable.* The intuition behind the formalization is that if a desired usage e occurs and no mechanism with a trigger that is refined by e is applicable (because all conditions φ do not hold), then either the actual usage is part of the trace, or adding this event would make one of the mechanisms applicable. We quantify over all potentially applicable mechanisms even though this is forbidden by the interference conditions described below; this is done to cater for relaxations of those conditions.

The motivation for the third case, and its complexity, is as follows. Consider an inhibitor i with condition $i.\varphi = \neg^- \underline{\mathit{repmax}}^-(3, E_{\mathit{fst}}(e))$, for some event e. This inhibitor ensures that e happens at most three times. Consequently, $i.\psi = (T_{\mathit{fst}}(e) \wedge i.\varphi) \Rightarrow \neg E_{\mathit{fst}}(e)$. Let $e = (\mathit{play}, \{(\mathit{name}, \mathit{123})\})$ and both $e_1 = (\mathit{play}, \{(\mathit{name}, \mathit{123}), (\mathit{device}, \mathit{1})\})$ and $e_2 = (\mathit{play}, (\mathit{name}, \mathit{123}), (\mathit{device}, \mathit{2})\})$ be refinements of e. Furthermore, let s denote a trace with $(s, 1) \models_f T_{\mathit{fst}}(e)$, $(s, 2) \models_f T_{\mathit{fst}}(e)$, and $(s, 3) \models_f T_{\mathit{fst}}(e_1) \wedge T_{\mathit{fst}}(e_2)$ (Fig. 1 (a); for simplicity's sake, we neither consider maximum refinements here nor do we show the $(\mathit{nil}, \varnothing)$ events). In other words, at time 3, there are two concurrent desired *play* events on different devices. Applying i will add (e, start) to both $s(1)$ and $s(2)$ because $\neg i.\varphi$ holds at both times, and adding the events does not invalidate $\neg i.\varphi$. Furthermore, $(s, 3) \models_{f^-} \neg\, i.\varphi$, i.e., the inhibitor is not applicable at time 3 either.

Now, to satisfy $\neg i.\varphi$, an inhibitor could simply prohibit both (e_1, start) and (e_2, start) at time 3 (Fig. 1 (b)). However, this seems undesirable from the consumer's perspective who has agreed to a policy stating that a movie must not be played more than three (rather than two) times. The inhibitor could also allow either (e_1, start) or (e_2, start) at time 3, but not both, also satisfying $\neg i.\varphi$ (Figs. 1 (c) and (d)). When adding both (e_1, start) and (e_2, start) to $s(3)$, then the resulting trace would violate $\neg i.\varphi$ (Fig. 1 (e)). Our definition of μ ensures that either one from Figs. 1 (c) and (d) is chosen non-deterministically (choosing *none* would invalidate constraint 2). In essence, the problem is that we have to express that "if, under a given condition, an event was added, some formula must be true", where adding the event may in itself invalidate the condition. The problem is thus not bound to the fact that we allow for parallel occurrences of events that are refinements of the same event.

Mechanisms are underspecified in that they can add *arbitrary* events, as long as the respective properties are not violated. That means that for a trace i, the set $\{o : \mathit{Trace} \bullet \forall\, t : \mathbb{N} \bullet i(t) \subseteq o(t) \wedge (o, t) \models_{f^\pm} \bigwedge_{u \in \mathit{ucmechs}} u.\psi \wedge \ldots\}$ contains those traces that look like i, but may contain additional events and yet satisfy the formula ψ of each mechanism. We hence require the effect of applying a set of mechanisms to be minimal, ensured by *constraint 3.* Constraints 2 and 3 together define the function μ. Minimal traces need not be unique. *Constraint 4* removes the remaining $\mathit{TRY}(\cdot)$ events and transforms the desired usages that are *not* controlled by any mechanism into the respective actual usages. This defines function μ^*.

Usage-Controlled Systems. Thus far, the "original" traces of a system have not been related to the usage-controlled ones. We also have not provided basic constraints such as the uniqueness of names, activation signals, etc. This is now done in the schema *UCSystem*, which completes our description of usage control mechanisms. *Traces* is the set of traces that are possible in the unprotected system, i.e., without any mechanisms in place; we assume all usages to be desired usages. *UCTraces* is the set of traces after application of the mechanisms given by $\mathit{Es} \cup \mathit{Ms} \cup \mathit{Is} \cup \mathit{Ds}$—each of which is equipped with a substitution for the variables oc-

UCSystem

$$\begin{array}{l}
\mathit{Traces} : \mathbb{P}\,\mathit{Trace}; \quad \mathit{UCTraces} : \mathbb{P}\,\mathit{Trace}; \quad \mathit{Es} : \mathbb{P}\,\mathit{Executor}; \quad \mathit{Ms} : \mathbb{P}\,\mathit{Modifier}; \quad \mathit{Is} : \mathbb{P}\,\mathit{Inhibitor}; \quad \mathit{Ds} : \mathbb{P}\,\mathit{Delayer}; \quad \mathit{Obls} : \mathbb{P}\,\mathit{Obligation} \\
\mathit{Mechs} : \mathit{CombinedMechanism}[\mathit{Es}/E, \mathit{Ms}/M, \mathit{Is}/I, \mathit{Ds}/D] \\
\hline
\forall s : \mathit{Trace};\ e : \mathit{IndEv} \mid s \in \mathit{Traces} \cup \mathit{UCTraces} \land e \in \bigcup \mathsf{ran}(s) \bullet \forall f : \mathit{Ev} \bullet f\ \mathit{refinesEv}\ e.1 \Rightarrow f = e.1 \\
\forall s : \mathit{Trace};\ e : \mathit{DesIndEv} \mid s \in \mathit{Traces} \cup \mathit{UCTraces} \land e \in \bigcup \mathsf{ran}(s) \bullet \forall f : \mathit{Ev} \bullet f\ \mathit{refinesEv}\ (\mathit{TRY}^{\sim}(e)).1 \Rightarrow f = (\mathit{TRY}^{\sim}(e)).1 \\
\forall s : \mathit{Trace};\ t : \mathbb{N};\ e : \mathit{IndEv} \mid s \in \mathit{UCTraces} \land \mathit{getclass}(e) = \mathit{usage} \bullet e \in s(t) \Rightarrow \mathit{TRY}(e) \in s(t) \\
\mathit{Traces} \subseteq \mathit{ValidOriginalTraces} \cap \mathit{TUniqueActivations} \cap \mathit{ValidNames}(\mathit{Obls}); \quad \mathit{Obls} \in \mathit{UniqueObls} \\
\mathit{UCTraces} = \{ s : \mathit{Trace} \mid s \in \mathit{Traces} \bullet \mathit{Mechs}.\mu^*(s) \}
\end{array}$$

curring in formulae and triggering actions. *Obls* denotes the obligations relevant to the system, i.e, those that will potentially be activated. This schema specifies that a *UCSystem* has the following properties. (i) If an event occurs in a trace, all its parameters have been specified ($e.1$ denotes the first component of the *IndEv* e, hence the *Ev*; see constraint 1). (ii) If a desired event occurs in a trace, all the parameters of the respective event have been specified. (iii) For all actual usages, there is a respective desired usage (constraint 3). (iv) A desired *nil* event takes place in each step, in order to allow for the activation of mechanisms without triggering actions. Each *ongoing* event must be preceded by a corresponding *start* event. In the original traces, all usages are desired usages. All this is achieved by the definition of set *ValidOriginalTraces*. (v) Each activation occurs at most once per trace (*TUniqueActivations*). (vi) Activation events must relate to the sets of relevant obligations of the *UCSystem*, namely *Obls* (function *ValidNames*). Items (iv)-(vi) motivate the fourth constraint; we omit a definition of the conditions for brevity's sake. (vii) Obligation names must be unique (*UniqueObls*). (viii) The set of usage-controlled traces, *UCTraces*, consists of the original traces, with all mechanisms being applied (constraint 6).

4. RELATED WORK

UCON [4] adds the notion of *ongoing* usage to access control. UCON assumes that the data never leaves the data provider's realm. This facilitates control as there is no explicit and consequential distinction between providers and consumers; one consequence is that UCON policies are device-dependent. This is in contrast to our distributed approach where data is given away. Possibly closest to our work is that of Zhang et al. [5] who define an obligation language for UCON where ongoing access models can be specified. Since UCON works with (only) one reference monitor, the fundamental abstraction of mechanisms is not introduced (even though their *onA1* model essentially corresponds to our delayers, the *onA2* and *onA3* models correspond to our executors, and their *pre* models do what our inhibitors do). Modifiers are not handled. Furthermore, because of the TLA-based declarative perspective, the authors specify properties of usage-controlled traces but not how to enforce them; there is no notion of "active" mechanism like ours. All cited models have not been applied to any decision or synthesis problems, and they do not cater for the configuration of mechanisms and the notion of event refinement.

Ligatti et al. propose edit automata for enforcing security policies [3]. Edit automata are similar to our work in that they represent trace transformers; our inhibitors have the same effect as their suppression automata, and our executors and modifiers are similar to their insertion automata. Delayers are not treated in the theoretical model. The main difference with our work is that our approach is logic-based rather than automaton-based. Edit automata neither cater for refinements of events nor for concurrent events.

5. CONCLUSIONS

We have presented a model of consumer-side mechanisms for distributed usage control. This model can be used to formally check if a set of mechanisms is able to enforce a given obligation and to check interference of mechanisms [2]. Our specification of mechanisms as trace transformers is intuitive and allows systems to be specified independently of any usage control mechanisms: the system's definition need not encompass the possible effects of any mechanisms.

We believe that the complexity of our formalization of mechanisms is of an essential, rather than an accidental, nature and reflects the complexity inherent in the problem domain in its full generality. Restricting usage control to mere inhibition, omitting parameters in mechanisms, not catering for usages that extend over more than one time step, ignoring interference of mechanisms, and defining mechanisms as trace properties rather than trace transformers would significantly simplify our model. However, this would be an over-simplification, preventing answers to many relevant and interesting research and development problems.

The notion of a secure data store is of course critical and may seem like a strong assumption. However, current trends in operating systems, hardware, trusted platform technology, and approaches to DRM are clear steps towards the existence of such secure data stores. A second assumption relates to the "dynamic" scenario: if a provider applies the decision procedures as described in this paper, then he must possess trustworthy information about the consumer's mechanisms. It seems likely that this could also be achieved using trusted platform technology and attestation mechanisms. We are currently working on the problem of rights delegation.

6. REFERENCES

[1] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A Policy Language for Usage Control. In *Proc. ESORICS*, pages 531–546, 2007.

[2] M. Hilty, A. Pretschner, C. Schaefer, and T. Walter. Enforcement for Usage Control: A System Model and a Policy Language for Distributed Usage Control. Technical Report I-ST-20, DoCoMo Euro-Labs, 2006.

[3] J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *Intl. J. of Inf. Security*, 4(1-2):2–16, 2 2005.

[4] J. Park and R. Sandhu. The UCON ABC Usage Control Model. *ACM Transactions on Information and Systems Security*, 7:128–174, 2004.

[5] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *Proc. SACMAT*, pages 1–10, 2004.