Formal Analyses of Usage Control Policies

Alexander Pretschner[*]
Fraunhofer IESE and TU Kaiserslautern
Kaiserslautern, Germany
alexander.pretschner@iese.fhg.de

Judith Rüesch
BBP AG
Baden, Switzerland
judith.rueesch@bbp.ch

Christian Schaefer, Thomas 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 handled after it is released. Usage control requirements are specified in policies. We present tool support for the following analysis problems. Is a policy consistent, i.e., satisfiable? Is an abstractly specified usage control mechanism capable of enforcing a given policy? Can we configure such a mechanism by analyzing respective policies? In the context of propagation, where upon re-distribution of data duties may only be increased and rights decreased, can we check if a policy is only strengthened in this sense? — Our solution uses a model checker as theorem prover and is based on a translation of usage control policies into a Linear Time Logic (LTL) dialect. We provide evidence that even complex policies can be analyzed efficiently.**

I. Introduction

Usage control generalizes access control by controlling not only who may access which data, but also how the data may or may not be used or distributed afterwards. We concentrate on (but are not restricted to) usage control in a distributed setting, 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 (provisions, including access control requirements, which we ignore in this paper) and the future. The latter requirements come as *obligations* that restrict the future usage of data ("use for statistical purposes only" or "do not distribute"), or require certain actions to be taken in the future ("delete after 30 days" or "notify me whenever the document is accessed"). Obligations consist of both (restricted) rights on the future usage of data and of duties to be performed by the consumer.

This field gives rise to many fascinating research and implementation problems, including the following. Usage control policies must be specified. What are the requirements, and what are appropriate languages [10]? Secondly, when writing policies, one would like to make sure that they are consistent. How do we check if a (large) policy contains contradictions? Thirdly, once consistent policies have been specified, an utterly difficult problem is that of enforcing them. How can we implement mechanisms that, under specific assumptions, can guarantee that a policy is adhered to, or at least report the violation of a policy? Given generic mechanisms, can we configure them on the grounds of a policy? Fourthly, in distributed settings, where data items can be re-distributed, a problem is that of rights (and duties) propagation [19] in that a data provider likely wants to make sure that policies only change in a specific way upon re-distribution of data. Can we check that policies are at most strengthened?

a) Problem and Solution: We focus on analysis problems for usage control policies, including the configuration of mechanisms and the determination of whether or not policies are strengthened upon re-distribution. We show how to translate usage control policies [10], abstract mechanism descriptions [18], and propagation requirements [19] into a dialect of LTL [16]. By using the model checker NuSMV (nusmv.irst.itc.it) as a theorem prover, we can answer all of the above analysis questions, and provide a solution to the problem of configuring generic mechanisms. On the grounds of several experiments, we show that even complex policies can be handled. We pinpoint benefits and shortcomings of our approach.

b) Contributions: To the best of our knowledge, we are the first to implement an automatic analysis tool for general usage control policies, thus leveraging the more conceptual work in this area to the more practical implementation level.

c) Overview: In §II, we give some background. We introduce usage control and recapitulate earlier work on a formal system model for usage control, an obligation specification language (OSL), and its extension to ordered events and parameters for re-distribution purposes. In §III, we present several analysis problems, and show how to translate OSL, mechanism descriptions, and delegation requirements into a variant of LTL. We present experiments that we conducted with the model checker NuSMV. We present related work in §IV and conclude in §V.

We use the Z specification language [21] in this paper. This choice is arbitrary, yet convenient. We introduce the necessary constituents of this language as the need arises. Note that we specify algebraic properties only and do not specify transition systems, in contrast to how Z is often used. Furthermore, we assume some familiarity with future and past LTL.

II. A Formal Grasp at Usage Control

Figure 1 sketches the general setup for *usage-controlled systems*. Here, boxes are technical components and arrows represent the main data flows. Usage control requirements are negotiated between data providers and consumers, and enforced using consumer-side mechanisms. A *data consumer* requests data. Using *negotiators*, the consumer and provider negotiate the usage request. Upon successful negotiation, data is transferred from the provider to the consumer and the usage

[*]This work was supported by the FhG Internal Programs under Grant No. Attract 692166.

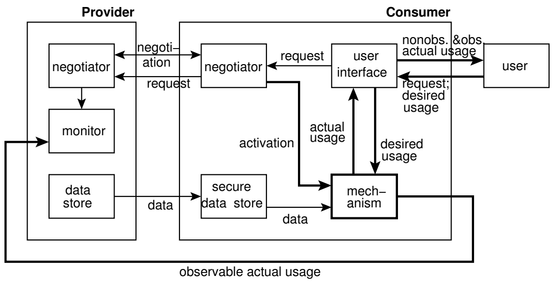


Fig. 1. Usage-Controlled Systems: Overview

control requirements are *activated*. From this point onward, the respective *mechanisms* on the consumer's side will (or at least should) enforce the requirements. This 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 these mechanisms whenever it leaves the store. Actual mechanisms usually implement the secure data store by cryptographic means. Examples of such mechanisms are given by current DRM solutions; other mechanisms simply send out information that a provider-side *monitor* can use to assess policy adherence [17]. Mechanisms transform desired usages into actual usages, which includes blocking an attempt.

Mechanisms are general-purpose devices: they are configured to protect specific resources. We model this by instantiating variables (§II-D). Given an abstract description of a parameterized mechanism, one additional question is which values have to be assigned to which variables to protect a given resource.

Distributed usage control means that data can be re-distributed. In §II-C, we show how the originating provider can make sure that policies are only strengthened in a well-defined way, by reducing rights and increasing duties. We will assume that whenever data is re-distributed, it is checked that policies are at most strengthened.

A. Semantic Model

Our model [10] is based on classes of parameterized events. The event classes include *usage* and *other*, with the latter including signaling and activation events. Parameters represent attributes. For example, a usage event must indicate on which data item it is performed, and a signaling event—one that is sent from the consumer to the outside—must name the recipient of the message. An *event* therefore consists of the event name and parameters, represented as a partial function ($\nrightarrow$) from names to values. We will describe event parameters as $(name, value)$ pairs. An example is the event $(play, \{(object, o)\})$, where *play* is the name of the event and the parameter $object$ has the value o.

The definition of events in the Z language is shown below. *EventName*, *ParamName*, and *ParamValue* define basic types for event names, parameter names, and parameter values, respectively. In Z, such definitions are made by listing the types in square brackets. All such basic types are disjoint. EBNF-style definitions are also possible.

$$[\mathit{EventName}, \mathit{ParamName}, \mathit{ParamValue}]$$
$$\mathit{EventClass} ::= \mathit{usage} \mid \mathit{other}; \quad \mathit{getclass} : \mathit{EventName} \to \mathit{EventClass}$$
$$\mathit{Params} : \mathit{ParamName} \nrightarrow \mathit{ParamValue}; \quad \mathit{Event} == \mathit{EventName} \times \mathit{Params}$$

$$\begin{aligned}
&\Phi^+ ::= \underline{\mathit{true}} \mid \underline{\mathit{false}} \mid E_{\mathit{fst}}\langle\langle \mathit{Event}\rangle\rangle \mid E_{\mathit{all}}\langle\langle \mathit{Event}\rangle\rangle \mid T_{\mathit{fst}}\langle\langle \mathit{Event}\rangle\rangle \mid T_{\mathit{all}}\langle\langle \mathit{Event}\rangle\rangle \mid \\
&\underline{\mathit{not}}\langle\langle \Phi^+\rangle\rangle \mid \underline{\mathit{and}}\langle\langle \Phi^+ \times \Phi^+\rangle\rangle \mid \underline{\mathit{or}}\langle\langle \Phi^+ \times \Phi^+\rangle\rangle \mid \underline{\mathit{implies}}\langle\langle \Phi^+ \times \Phi^+\rangle\rangle \mid \underline{\mathit{until}}\langle\langle \Phi^+ \times \Phi^+\rangle\rangle \mid \\
&\underline{\mathit{always}}\langle\langle \Phi^+\rangle\rangle \mid \underline{\mathit{after}}\langle\langle \mathbb{N} \times \Phi^+\rangle\rangle \mid \underline{\mathit{within}}\langle\langle \mathbb{N} \times \Phi^+\rangle\rangle \mid \underline{\mathit{during}}\langle\langle \mathbb{N} \times \Phi^+\rangle\rangle \mid \\
&\underline{\mathit{repmax}}\langle\langle \mathbb{N} \times \Phi^+\rangle\rangle \mid \underline{\mathit{replim}}\langle\langle \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \Phi^+\rangle\rangle \mid \underline{\mathit{repuntil}}\langle\langle \mathbb{N} \times \Phi^+ \times \Phi^+\rangle\rangle \\
&\underline{\mathit{permitonlyevname}}\langle\langle \mathbb{P}\,\mathit{EventName} \times \mathit{Params}\rangle\rangle \mid \\
&\underline{\mathit{permitonlyparam}}\langle\langle \mathbb{P}\,\mathit{ParamValue} \times \mathit{ParamName} \times \mathit{EventName} \times \mathit{Params}\rangle\rangle
\end{aligned}$$

Events are ordered with respect to 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 which explains the $\subseteq$ relation in the definition below. (In such *axiomatic definitions*, the defined mathematical object is named and typed above the line and its properties are given below the line.) *x.i* identifies the *i*-th component of a tuple *x*. $\leftrightarrow$ introduces a binary relation.

$$\begin{array}{|l}
\_\ \mathit{refinesEv}\ \_ : \mathit{Event} \leftrightarrow \mathit{Event} \\
\hline
\forall e_1, e_2 : \mathit{Event} \bullet e_2\ \mathit{refinesEv}\ e_1 \Leftrightarrow e_1.1 = e_2.1 \land e_1.2 \subseteq e_2.2
\end{array}$$

The rationale is that when specifying usage control requirements, we do not want to specify all parameters. For instance, if the event $(play, \{(object, o)\})$ is prohibited, then the event $(play, \{(object, o), (device, d)\})$ should also be prohibited. *nil* of type *EventName* is reserved and denotes no event.

d) Indexed Events and Traces: To define obligations, we need a language for usage control (§II-B). Its semantics is defined over traces: mappings from abstract points in time—represented by the natural numbers—to possibly empty sets of 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: $((play, \{(object, movie)\}), start)$; $((play, \{(object, movie)\}), ongoing)$; then $((play, \{(object, movie)\}), ongoing)$. The data type *IndEvent* defines such indexed events. We also need to express that usage is attempted by a user (see above). This is important for the discussion of mechanisms because not every attempted usage is actually executed in the end. This concept is covered by desired indexed events (*DesiredIndEvent* in the definition below). In Z, records can be defined by stating the name of a constructor and its arguments in angular brackets. If there is more than one argument, the Cartesian product is used.

$$\mathit{Index} ::= \mathit{start} \mid \mathit{ongoing}; \quad \mathit{IndEvent} == \mathit{Event} \times \mathit{Index}$$
$$\mathit{DesiredIndEvent} ::= \mathit{TRY}\langle\langle \mathit{IndEvent}\rangle\rangle$$
$$\mathit{Trace} : \mathbb{N} \to \mathbb{P}(\mathit{IndEvent} \cup \mathit{DesiredIndEvent})$$

B. The Obligation Specification Language

Φ^+ (+ for future) defines the building blocks for describing obligations in OSL [10]. It is a temporal logic with explicit operators for cardinality and permissions. We will later extend OSL with ordered events and intervals which are necessary to express re-distribution requirements.

We distinguish between the start of an action (syntactically: E_{fst}; semantically: an indexed event with index *start*) and lasting actions (syntactically: E_{all}; semantically: indexed events with any index). T_{fst} and T_{all} refer to the respective *desired* or *attempted* actions. When specifying events in obligations, by virtue of the refinement relation, there is an implicit universal quantification over unmentioned parameters. $\underline{not}$, $\underline{and}$, $\underline{or}$, $\underline{implies}$ have the usual semantics. We will use the infix operators $\neg, \land, \lor, \Rightarrow$ as shorthand. Our $\underline{until}$ operator is the weak-until operator from LTL. Using $\underline{after}(n)$, which refers to the time after n time steps, we can express concepts like $\underline{during}$ (something must constantly hold during a specified time interval) and $\underline{within}$ (something must 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 does the same, but independent of any time interval. Instead, it limits the maximal number of times a formula holds until another formula holds (e.g., the occurrence of some event). With the help of $\underline{repuntil}$, we can also define $\underline{repmax}$, which 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, $\underline{replim}(20, 0, 5, E_{all}((play, \{(obj, movieA)\})))$ specifies that *movieA* may be played for at most five time units during the next twenty time units. Similarly, $\neg\underline{replim}(20, 0, 2, E_{fst}((play, \{(obj, movieB)\}))) \Rightarrow \underline{after}(25, E_{fst}((notify, \{(rcv, subA)\})))$ specifies that if *movieB* is started more than twice during the next 20 time units, then subject *subA* will be notified after 25 time units.

OSL supports both the *must* and the *may* modalities. The former is the "standard" semantics of linear time logics, and the latter is supported by two designated operators: these operators allow one to specify that out of a given set of usage events, only selected usage events are allowed. The operator $\underline{permitonlyevname}$ defines the names of the usage events that are exclusively allowed with a set of given parameters. For example, the expression $\underline{permitonlyevname}(\{play, print\}, \{(object, oid)\})$ states that the only usages permitted on the object *oid* are *play* and *print*. It does not say anything about non-usage events or events with different parameters (e.g., if a usage is applied to a different data object). Similarly, $\underline{permitonlyparam}$ only allows certain values for a given parameter of an event. It prohibits all other values for this parameter. For example, the expression $\underline{permitonlyparam}(\{s_1, s_2\}, recipient, send, \{(object, doc)\})$ specifies that out of all *send* events with *doc* as the "object" parameter, only those with the "recipient" parameter being s_1 or s_2 are allowed. In other words, *doc* may only be sent to s_1 or s_2. The first argument of $\underline{permitonlyparam}$ is the set of allowed parameter values, the second is the name of the parameter whose values should be restricted, and the third and fourth arguments define an underspecified event.

The semantics of Φ^+ is formally defined by the binary

$$\begin{array}{|l}
\_\ \models_e\ \_ : \mathit{IndEvent} \leftrightarrow \Phi^+ \\
\hline
\forall\, ie : \mathit{IndEvent};\ \varphi : \Phi^+ \bullet ie \models_e \varphi \Leftrightarrow \exists\, e : \mathit{Event} \bullet \\
\qquad ie.1\ \mathit{refinesEv}\ e \land ((\varphi = E_{fst}(e) \land ie.2 = \mathit{start}) \lor \varphi = E_{all}(e))
\end{array}$$

$$\begin{array}{|l}
_\ \models_f\ _ : (\mathit{Trace} \times \mathbb{N}) \leftrightarrow \Phi^+ \\
\hline
\forall\, s : \mathit{Trace};\ t : \mathbb{N};\ \varphi : \Phi^+ \bullet (s, t) \models_f \varphi \Leftrightarrow \\
\qquad \varphi = \underline{true} \lor \varphi = E_{fst}((nil, \varnothing)) \lor \varphi = E_{all}((nil, \varnothing)) \\
\qquad \lor \exists\, e : \mathit{Event} \setminus \{(nil, \varnothing)\};\ ie : \mathit{IndEvent} \bullet \\
\qquad\qquad\qquad (\varphi = E_{fst}(e) \lor \varphi = E_{all}(e)) \land ie \in s(t) \land ie \models_e \varphi \\
\qquad \lor \exists\, \psi : \Phi^+ \bullet \varphi = \underline{not}(\psi) \land \neg\, ((s, t) \models_f \psi) \\
\qquad \lor \exists\, \psi, \chi : \Phi^+ \bullet \varphi = \underline{or}(\psi, \chi) \land ((s, t) \models_f \psi \lor (s, t) \models_f \chi) \\
\qquad \lor \exists\, \psi, \chi : \Phi^+ \bullet \varphi = \underline{until}(\psi, \chi) \\
\qquad\qquad \land (\exists\, u : \mathbb{N} \mid t < u \bullet \Big((s, u) \models_f \chi \land (\forall\, v : \mathbb{N} \mid t < v < u \bullet (s, v) \models_f \psi)\Big) \\
\qquad\qquad\qquad\qquad \lor (\forall\, v : \mathbb{N} \mid t < v \bullet (s, v) \models_f \psi)) \\
\qquad \lor \exists\, i : \mathbb{N};\ \psi : \Phi^+ \bullet \varphi = \underline{after}(i, \psi) \land (s, t + i) \models_f \psi \\
\qquad \lor \exists\, i : \mathbb{N}_1;\ m, n : \mathbb{N};\ \psi : \Phi^+;\ e : \mathit{Event} \bullet \\
\qquad\qquad \varphi = \underline{replim}(i, m, n, \psi) \land (\psi = E_{fst}(e) \lor \psi = E_{all}(e)) \land \\
\qquad\qquad\qquad m \leq (\sum_{j=1}^{i} \#\{ie : \mathit{IndEvent} \mid ie \in s(t + j) \land ie \models_e \psi\} \leq n \\
\qquad \lor \exists\, n : \mathbb{N};\ \psi, \chi : \Phi^+;\ e : \mathit{Event} \bullet \varphi = \underline{repuntil}(n, \psi, \chi) \land (\psi = E_{fst}(e) \lor \psi = E_{all}(e)) \\
\qquad\qquad \land \Big((\exists\, u : \mathbb{N}_1 \bullet (s, t + u) \models_f \chi \land (\forall\, v : \mathbb{N}_1 \mid v < u \bullet \neg((s, t + v) \models_f \chi)) \\
\qquad\qquad\qquad \land (\sum_{j=1}^{u} \#\{ie : \mathit{IndEvent} \mid ie \in s(t + j) \land ie \models_e \psi\}) \leq n) \\
\qquad\qquad \lor (\sum_{j=1}^{\infty} \#\{ie : \mathit{IndEvent} \mid ie \in s(t + j) \land ie \models_e \psi\}) \leq n\Big) \\
\qquad \lor \exists\, ex : \mathbb{P}\,\mathit{EventName};\ ps : \mathit{Params} \bullet \varphi = \underline{permitonlyevname}(ex, ps) \\
\qquad\qquad \land \forall\, en : \mathit{EventName} \mid getclass(en) = usage \land en \notin ex \bullet \\
\qquad\qquad\qquad (t, n) \models_f \underline{always}(\underline{not}(E_{all}((en, ps)))) \\
\qquad \lor \exists\, ex : \mathbb{P}\,\mathit{ParamValue};\ pn : \mathit{ParamName};\ en : \mathit{EventName};\ ps : \mathit{Params} \bullet \\
\qquad\qquad \varphi = \underline{permitonlyparam}(ex, pn, en, ps) \land pn \notin \mathrm{dom}\, ps \\
\qquad\qquad \land \forall\, pv : \mathit{ParamValue} \mid pv \notin ex \bullet \\
\qquad\qquad\qquad (t, n) \models_f \underline{always}(\underline{not}(E_{all}((en, ps \cup \{(pn, pv)\})))) \\
\qquad \lor \varphi = \underline{false} \land (s, t) \models \underline{not}(\underline{true}) \\
\qquad \lor \exists\, \psi, \chi : \Phi^+ \bullet \varphi = \underline{and}(\psi, \chi) \land (s, t) \models_f \underline{not}(\underline{or}(\underline{not}(\psi), \underline{not}(\chi))) \\
\qquad \lor \exists\, \psi, \chi : \Phi^+ \bullet \varphi = \underline{implies}(\psi, \chi) \land (s, t) \models_f \underline{or}(\underline{not}(\psi), \chi) \\
\qquad \lor \exists\, \psi : \Phi^+ \bullet \varphi = \underline{always}(\psi) \land (s, t) \models_f \underline{until}(\psi, \underline{false}) \\
\qquad \lor \exists\, i : \mathbb{N};\ \psi : \Phi^+ \bullet \varphi = \underline{within}(i, \psi) \land (s, t) \models_f \underline{replim}(i, 1, i, \psi) \\
\qquad \lor \exists\, i : \mathbb{N};\ \psi : \Phi^+ \bullet \varphi = \underline{during}(i, \psi) \land (s, t) \models_f \underline{replim}(i, i, i, \psi) \\
\qquad \lor \exists\, n : \mathbb{N};\ \psi : \Phi^+ \bullet \varphi = \underline{repmax}(n, \psi) \land (s, t) \models_f \underline{repuntil}(n, \psi, \underline{false})
\end{array}$$

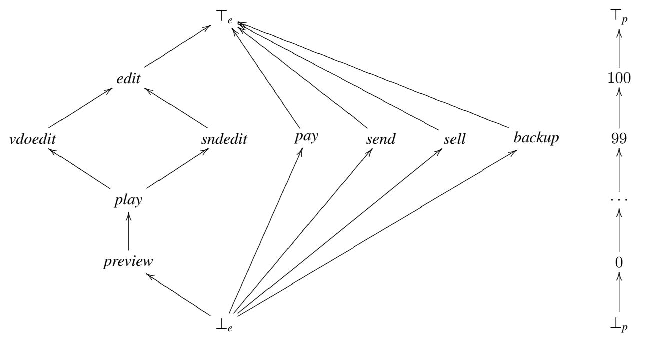
relation $\models_f$, reproduced verbatim from [10]. It makes use of a shorthand, $\models_e$, to relate single indexed events (rather than traces) to formulae of the form $E_{fst}(\cdot)$ or $E_{all}(\cdot)$.

C. Ordered Events and Parameters for Re-Distribution

Upon re-distribution of usage-controlled data items, a provider may want to make sure that policies are at most strengthened. In earlier work, we have provided a simple role-based re-distribution model that allows an originating data provider — the first in a distribution chain — to define policies for different roles [19]. When data is re-distributed, policies can only be strengthened in that rights are reduced and duties increased. A role hierarchy governs who may strengthen whose policies. This role hierarchy is hence not a hierarchy of rights, in contrast to what is usually the case in role-based access control.

Strengthening policies boils down to logically strengthen the respective policies. As OSL essentially is an LTL dialect, strengthening along the temporal and propositional "dimensions" comes for free. However, if a policy states that a movie may be played at a video quality of 75%, then one would assume that this really means "at most 75%." The provider probably would not object to playing it at 60%. Allowing to play it at a quality of "at most 50%" would hence be strengthening. To accomodate for this, we include ordered parameters. Parameter values are ordered w.r.t. a user-defined partial order, $\leq_p$, in fact a lattice with $\top$ and $\bot$ elements. In policies, parameters are then given by left and right boundaries of the desired interval. For rights and duties, left and right boundaries can be identical. Otherwise, for duties, we require the upper bound to be $\top$ which means that "pay USD 10" is given as "pay USD (10,$\top$)." This can be strengthened to, for instance, "pay USD (15,$\top$)." Conversely, rights must have $\bot$ as left boundary so that "permission to play with 75% video quality" is given as "permission to play with ($\bot$,75%) video quality." This can be strengthened to "permission to play with ($\bot$,50%) quality."

Similarly, events can be ordered. We hence introduce lattices for event names, $\leq_e$, so that in a movie application, *preview* may be smaller (fewer rights) than both *vdoedit* (video) and *sndedit* (sound) which, in turn, are both smaller (fewer rights) than (full) *edit*. An example is given in the following (left: ordered events, right: ordered parameters).



Similar to ordered parameters, events are then specified by pairs. Both boundaries can be identical. Otherwise, when specifying rights, the left boundary must be $\bot$, and duties must have $\top$ as right boundaries. Formally, we introduce ordered events with the data type *OrdEvent* that consists of a pair of event names — with one boundary being either $\top$ or $\bot$ if both components are not identical — and a set of ordered parameters. From now on, we will assume that OSL uses ordered events.

$$\begin{array}{l}
OrdEventName == EventName \times EventName \\
OrdParam == ParamName \nrightarrow (ParamValue \times ParamValue) \\
OrdEvent == OrdEventName \times OrdParam
\end{array}$$

The intuitive semantics of an ordered event is that the respective (desired or actual) indexed event is in-between the two boundaries, and the same holds true for all actual parameters. The formal semantics of *ordered events* is defined by simply adjusting $\models_e$: an *IndexedEvent* consists of one event name and a (full) set of parameter names and parameter values. For each ordered (desired) event that is given in an OSL formula, we check if the actual event name and the actual parameters are in-between the specified boundaries. In the formal definition, this amounts to checking membership in a set or in a disjunction over all possible parameter names. Similarly, the permission operators have to be slightly adjusted.

More specifically, the semantics of events is adjusted by re-defining $\models_e$ as follows.

$$\begin{array}{|l}
\_ \models_e \_ : IndEvent \leftrightarrow \Phi^+ \\
\hline
\forall ie : IndEvent;\ \varphi : \Phi^+ \bullet ie \models_e \varphi \Leftrightarrow \\
\quad \exists en, en_l, en_u : EventName;\ ps : \mathbb{P}\, Params;\ ops : OrdParam \bullet \\
\quad\quad ie = (en, ps) \\
\quad\quad \wedge ((\varphi = E_{fst}(((en_l, en_u), ops)) \wedge ie.2 = start) \vee \varphi = E_{all}(((en_l, en_u), ops))) \\
\quad\quad \wedge en_l \leq_e en \wedge en \leq_e en_u \\
\quad\quad \wedge \forall pn : ParamName;\ pv, pv_l, pv_u : ParamValue \mid \\
\quad\quad\quad (pn, pv) \in ps \wedge (pn, (pv_l, pv_u)) \in ops \bullet pv_l \leq_p pv \wedge pv \leq_p pv_u
\end{array}$$

As a consequence, a few operators have to be redefined as well. We show the formal semantics of the permission operators only:

$$\begin{array}{|l}
\_ \models_f \_ : (Trace \times \mathbb{N}) \leftrightarrow \Phi^+ \\
\hline
\forall s : Trace;\ t : \mathbb{N};\ \varphi : \Phi^+ \bullet (s, t) \models_f \varphi \Leftrightarrow \\
\quad \ldots \\
\quad \vee \exists ex : \mathbb{P}\, OrdEventName;\ ps : OrdParam \bullet \\
\quad\; \varphi = \underline{permitonlyevname}\ (ex, ps) \wedge \\
\quad\;\; \forall en, en_u, en_l : EventName;\ pn : ParamName;\ pv_l, pv_u : ParamValue \mid \\
\quad\;\;\; getclass(en) = usage \wedge (en_l, en_u) \in ex \wedge en_u \leq_e en \vee en \leq en_l \bullet \\
\quad\;\;\; (s, t) \models_f \underline{always}\ (\underline{not}\ (E_{all}((en, ps)))) \\
\quad \vee \exists ex : \mathbb{P}(ParamValue \times ParamValue);\ oen : OrdEventName; \\
\quad\; ps : OrdParam;\ pn : ParamName \bullet \\
\quad\; \varphi = \underline{permitonlyparam}\ (ex, pn, oen, ps) \wedge pn \notin \mathrm{dom}\, ps \wedge \\
\quad\;\; \forall pv, pv_l, pv_u : ParamValue \mid (pv_l, pv_u) \in ex \wedge pv_u \leq_p pv \vee pv \leq_p pv_l) \bullet \\
\quad\;\;\; (s, t) \models_f \underline{always}\ (\underline{not}\ (E_{all}((oen, ps \cup (pn, (pv, pv)))))) \\
\quad \ldots
\end{array}$$

D. Mechanisms

Mechanisms are the means by which usage control requirements can be enforced, either by making sure policies are adhered to, or by spelling out penalties if policy violations are detected. In earlier work [18], we have shown that usage control mechanisms can be classified as executors, signalers, modifiers, inhibitors, and delayers. Executors make sure something happens – for instance, send out a notification to data owners if their data is accessed. Signalers are a special case that report policy violations. Modifiers transform attempted usages into different actual usages. For instance, if a song cannot be played at a quality of 100%, a modifier may transform it to 75% in order to satisfy a policy. Inhibitors make sure specific attempted usages are not transformed into actual usages. For instance, if a policy stipulates that a song must not be played more than twice, an inhibitor will make sure nothing happens if the third request for playing the song is received. Finally, delayers re-try to perform a specific usage after some time, hoping that certain conditions may have changed in the meantime.

Formal specifications of mechanisms essentially consist of a condition – under which the mechanism is supposed to do something, and an effect – execution, modification, etc. One part of the condition may be a triggering event (try to play a song a third time), but this is not necessarily the case (e.g., to satisfy the policy that a data item must be deleted after thirty days). The condition is a formula that relates to the past and is hence specified in a past version of OSL called Φ^-. In contrast, the effect is a rather simple (future) OSL formula. For brevity's sake, we do not define Φ^- here but refer to related work [18].

Mechanisms can be parameterized. For instance, a modifier is likely to be deployed not just for one particular song but rather for an entire class of songs. This is achieved by introducing variables into OSL and Φ^{-}.

As an example, we show how modifiers are defined [18]. Parameterized conditions are given as φ_p; the variables are instantiated by means of a substitution σ, with function *subst* actually performing the substitution. The effect of a modifier is provided by the set $\mathit{modifyBy}_p$; it replaces the triggering event, ua_p, by an actual usage. *m* is the mode (beginning or ongoing) of ua_p. With *maxRefs* computing all maximum instantiations of an event w.r.t. relation *refinesEv*, the overall specification of a modifier is provided by ψ. Given that a maximum instantiation of ua_p is desired, and given that actually executing *ua* would make the condition φ_p true, a modifier makes sure that ua_p is *not* executed, but instead all elements of the set *modifyBy* are attempted (not executed, because there may be other applicable mechanisms). The following shows a Z schema that, in the upper half, defines the different elements of the defined data type *Modifier* and, in the lower half, formulates constraints on these elements.

Modifier

$$\mathit{modifyBy}_p : \mathbb{P}\,\mathit{VarEvent}; \quad \mathit{modifyBy} : \mathbb{P}\,\mathit{Event}$$
$$\varphi_p : \Phi_v^{-}; \quad \varphi : \Phi^{-}; \quad \psi : \Phi^{\pm}; \ \mathit{ua}_p : \mathit{VarEvent}; \quad \mathit{ua} : \mathit{Event}; \quad m : \mathit{Mode}$$
$$\sigma : \mathit{Var} \nrightarrow (\mathit{ParamValue} \cup \mathit{EventName})$$

$$\mathit{modifyBy} = \mathrm{map_s}(\lambda\, x : \mathit{VarEvent} \bullet \mathit{subst}(x, \sigma), \mathit{modifyBy}_p)$$
$$\varphi = \mathit{subst}_f(\varphi_p, \sigma) \wedge \mathit{ua} = \mathit{subst}(\mathit{ua}_p, \sigma) \wedge \mathit{ua} \neq (\mathit{nil}, \varnothing) \wedge \mathit{ua} \notin \mathit{modifyBy}$$
$$\psi = \bigwedge_{e \in \mathit{maxRefs}(\mathit{ua})} \Big(T_m(e) \wedge^{\pm} (E_m(e) \Rightarrow^{\pm} \varphi)\Big) \Rightarrow^{\pm} \neg E_m(\mathit{ua}) \wedge \bigwedge_{e \in \mathit{modifyBy}} T_{\mathit{all}}(e)$$

NoPayNoGood is a modifier: if a song, given by variable V(1), is not paid for, then it is played only with reduced quality. The configuration of this mechanism is modeled by substituting variable $V(1)$ with an actual object. This schema makes use of the schema *Modifier* by instantiating specific values in *Modifier*.

NoPayNoGood

$$\begin{aligned}&\mathit{Modifier}[\underline{\mathit{always}}^{-}(\neg^{-} E_{\mathit{fst}}((\mathit{pay}, \{(\mathit{obj}, V(1))\})))/\varphi_p,\\ &\quad (\mathit{play}, \{(\mathit{obj}, V(1), (\mathit{qual}, \mathit{full}))\})/\mathit{ua}_p, \mathit{all}/m,\\ &\quad \{(\mathit{play}, \{(\mathit{obj}, V(1)), (\mathit{qual}, \mathit{red})\})\}/\mathit{modifyBy}_p]\end{aligned}$$

III. TRANSLATION INTO LTL

A. OSL

e) Application: The first application that we consider is that of consistency. A policy is consistent if it has a model, i.e., if there is at least one execution of a system that satisfies it. For instance, using the concrete OSL syntax, the following two requirements are consistent when combined by conjunction: `PermitEvNames{listen,download} For {(music=songX)}` and `Always{If{download(music=songX)} Then {After[7] {delete(music =songX)}}}`. In contrast, the following two requirements are inconsistent when conjoined into one policy: `PermitEvNames{listen} For {(subject=TestUser)}` and `After[1]{download(subject=TestUser,music=songX)}`. We can check this with the model checker NuSMV as described below.

f) Translation: For each event *e*, we introduce a bit field `ename` of length four, where `ename` is the name of the event, i.e. *e*.1. `ename[0]` corresponds to the indexed event $(\mathit{TRY}(e), \mathit{start})$, `ename[1]` to $(\mathit{TRY}(e), \mathit{ongoing})$, `ename[2]` to (e, start) and `ename[3]` to $(e, \mathit{ongoing})$. In this paper, we do not consider simultaneous events with the same name which is why we could restrict ourselves to three bits (beginning/ongoing, attempted usage, actual usage), but we ignore this optimization here. Ignoring multiple simultaneous events of the same name allows us to handle parameters as follows. For each parameter *pname* of event *e*, we simply introduce a variable `e_pname`. A specification $T_{\mathit{fst}}((\mathit{download}, \{(\mathit{object}, \mathit{songX})\}))$ then translates to `download[0] & download_object=songX`, and an ongoing event specification $E_{\mathit{all}}((\mathit{play}, \{(\mathit{object}, \mathit{songY})\}))$ translates to `(play[2]|play[3]) & play_object=songY`. Intervals for event names and parameters are naturally unfolded into disjunctions. For an example, see www.inf.ethz.ch/personal/pretscha/ares09/exp.pdf. There is no need to translate the *refinesEv* relation: a model checker will automatically instantiate unspecified parameters to all possible values.

The translation of the temporal and propositional operators is straightforward. We omit a definition of the respective parts of function τ_{OSL} here for brevity's sake. Permissions also unfold into huge disjunctions. The cardinality operators need some special attention though. Let us assume that LTL^{+} defines the future LTL part of the NuSMV input language. The translation of $\underline{\mathit{repmax}}$ can be expanded via $\underline{\mathit{repuntil}}$. For $\underline{\mathit{repuntil}}$, we need the release operator `V` from *LTL* (with $\varphi \mathtt{V} \psi$ having the intuitive meaning that either ψ holds forever, or ψ holds until some moment in time where both φ and ψ hold). To simplify the definition, we use the macro $X\varphi$ for $\underline{\mathit{after}}(1, \varphi)$, even though *X* is not part of OSL. We also use `X` to denote the next step operator in the NuSMV language; i.e., $\mathtt{X}\varphi$ is true at time *t* if φ is true at time $t+1$. $\underline{\mathit{repuntil}}(0, \varphi, \psi)$ is true if, starting from the next time step, φ is false until and including a moment in time where ψ becomes true and φ remains false, or if φ is false forever. For $n > 0$, we distinguish three cases. $\underline{\mathit{repuntil}}(n, \varphi, \psi)$ is true at a time *t* if

1) ψ is true at $t+1$ — there is at most one true instance of φ until ψ becomes true; or if
2) both φ and $\underline{\mathit{repuntil}}(n-1, \varphi, \psi)$ are true at $t+1$ — φ becomes immediately true, and we can perform a recursive evaluation; or if
3) φ is false at $t+1$ and remains false until (and including) a time point t' with the following properties. If ψ is true at t', then we are done. Otherwise, if φ is true at $t'+1$, then we check if ψ is also true at $t'+1$. If that is the case, we are done. If not, then we recurse with $\underline{\mathit{repuntil}}(n-1, \varphi, \psi)$.

The translation of the $\underline{\mathit{replim}}$ operator is a result of another rather straightforward recursive characterization. The base case is a direct translation of the formal semantics. For $n > 0$, $\underline{\mathit{replim}}(i, m, n, \varphi)$ is true if $\varphi \wedge \underline{\mathit{replim}}(i-1, m-1, n-1, \varphi)$

$$
\begin{array}{|l}
\tau_{OSL} : \Phi^{+} \rightarrow LTL^{+} \\
\hline
\forall \varphi, \psi, \varepsilon : \Phi^{+};\ i, m, n : \mathbb{N} \bullet \\
\quad \ldots \wedge \\
\wedge\ (\varepsilon = \underline{repmax}(n, \varphi) \Rightarrow \tau_{OSL}(\varepsilon) = \tau_{OSL}(\underline{repuntil}(n, \varphi, false))) \\
\wedge\ (\varepsilon = \underline{repuntil}(0, \varphi, \psi) \Rightarrow \tau_{OSL}(\varepsilon) = \mathtt{X}\,(\tau_{OSL}(\psi)\ \mathtt{V}\ \tau_{OSL}(\neg\varphi))\,) \\
\wedge\ ((n > 0 \wedge \varepsilon = \underline{repuntil}(n, \varphi, \psi)) \Rightarrow \tau_{OSL}(\varepsilon) = \\
\quad \tau_{OSL}(\underline{after}(1, \psi \vee (\varphi \wedge \underline{repuntil}(n-1, \varphi, \psi))) \\
\quad \mathtt{|}\ \ \mathtt{X}(\tau_{OSL}((\psi \vee (\underline{after}(1, \varphi) \wedge (\underline{after}(1, \psi) \vee repuntil(n-1, \varphi, \psi)))))\ \mathtt{V}\ \tau_{OSL}(\neg\varphi))) \\
\wedge\ ((\varepsilon = \underline{replim}(i, m, n, \varphi) \wedge (i = 0 \vee m > n \vee (i > 0 \wedge i < m))) \Rightarrow \tau_{OSL}(\varepsilon) = \mathtt{0}) \\
\wedge\ ((\varepsilon = \underline{replim}(i, 0, 0, \varphi) \wedge i > 1) \Rightarrow \tau_{OSL}(\varepsilon) =\ \tau_{OSL}(during(i, \neg\varphi))) \\
\wedge\ ((\varepsilon = \underline{replim}(1, 0, n, \varphi)) \Rightarrow \tau_{OSL}(\varepsilon) = \mathtt{1}) \\
\wedge\ ((\varepsilon = \underline{replim}(1, 1, n, \varphi) \wedge n \geq 1) \Rightarrow \tau_{OSL}(\varepsilon) = \mathtt{X}\tau_{OSL}(\varphi)) \\
\wedge\ ((\varepsilon = \underline{replim}(i, m, n, \varphi) \wedge (i > 1 \wedge i \geq m \wedge n \geq m \wedge m > 0)) \Rightarrow \tau_{OSL}(\varepsilon) = \\
\ \mathtt{X}((\tau_{OSL}(\varphi \wedge \underline{replim}(i-1, m-1, n-1, \varphi)))\ \ \mathtt{|}\ \ (\,!\,\tau_{OSL}(\varphi \wedge \underline{replim}(i-1, m, n, \varphi))))) \\
\wedge\ ((\varepsilon = \underline{replim}(i, 0, n, \varphi) \wedge (i > 1 \wedge n > 0)) \Rightarrow \tau_{OSL}(\varepsilon) = \\
\ \mathtt{X}((\tau_{OSL}(\varphi \wedge \underline{replim}(i-1, 0, n-1, \varphi)))\ \ \mathtt{|}\ \ (\tau_{OSL}(\neg\varphi \wedge \underline{replim}(i-1, 0, n, \varphi)))))
\end{array}
$$

or $\neg\varphi \wedge \underline{replim}(i-1, m, n, \varphi)$ is true in the next step. This definition needs to take into account several special cases.

For space reasons, we provide only the part of τ_{OSL} that defines the future cardinality operators:

Before we can check consistency of a Φ^{+} formula, we need to define two sanity constraints. Firstly, there only is an actual event if there is a corresponding desired event: $C_1 = \underline{always}(\bigwedge_{e \in Event} E_{fst}(e) \Rightarrow T_{fst}(e) \wedge E_{all}(e) \Rightarrow T_{all}(e))$. Secondly, every ongoing usage is eventually preceded by a beginning usage. We have seen that a logic of the past was used for the definition of mechanisms; we will also use it now (the NuSMV input language includes past LTL, and we omit the translation of Φ^{-} to past LTL because it is very similar to the future case). $Y\varphi$ is true at time $t > 0$ iff φ is true at $t-1$. The second sanity constraint then reads as $C_2 = \underline{always}(\bigwedge_{e \in Event}(E_{all}(e) \wedge \neg E_{fst}(e) \wedge \neg Y(E_{all}(e) \wedge \neg E_{fst}(e)) \Rightarrow YE_{fst}(e)))$ which translates into `G((ename[3]&!Y ename[3])->Y ename[2])` where we omit the symmetrical case for desired events.

Our model $\mathcal{M}$ simply consists of all the arrays and parameters that we used to model events. Initially, we set all array values to zero, thus modeling that no event takes place. Assuming that τ_{OSL} also handles past formulas, checking consistency of a conjunction of policies Ψ then amounts to model checking $\mathcal{M} \models \tau_{OSL}((C_1 \wedge C_2) \Rightarrow \neg\Psi)$. If the model checker returns *false*, then Ψ is consistent.

g) Example: We provide a simple translation example, again using the concrete OSL syntax. The policy `PermitEvParams {songA} For {obj} In {listen (subject=TestUser)}` specifies that `TestUser` may only listen to `songA`. Under the assumption that the system is aware of four songs A,X,Y, and Z, it translates into the following NuSMV formula: `G(((listen[2]|listen[3]) & listen_subject= TestUser) -> (!( listen_obj=songX | listen_obj =songY | listen_obj=songZ)))` where we omit the translation of the sanity constraints for brevity's sake. An example of the translated sanity constraints can be found at www.inf.ethz.ch/personal/pretscha/ares09/exp.pdf.

B. Re-Distribution Requirements

h) Application: Upon re-distribution of data, data providers may only strengthen the policies specified by the originating data provider. Strengthening means reducing rights and increasing duties. In a distributed usage control setting, checking whether a policy was at most strengthened must be done whenever data is distributed.

i) Translation: Distributed usage control policies consist of one subpolicy for each role plus one default policy that applies if no policy is specified for a role. Strengthening policies with ordered events amounts to logically strengthening the respective formulas [19]. Checking if a policy was strengthened then amounts to checking implication. Assuming that we know which subpolicies $\pi_1, \ldots, \pi_n$ for a data item d can be strengthened by the current data provider, we check if the policies defined for redistribution of d, $\pi'_1, \ldots, \pi'_n$, are stronger than the original ones. Using the model $\mathcal{M}$ and the constraints C_1 and C_2 defined above, we use the model checker to decide $\mathcal{M} \models \tau_{OSL}((C_1 \wedge C_2) \Rightarrow \bigwedge_{i=1}^{n}(\pi'_i \Rightarrow \pi_i))$. If the model checker returns *true*, all subpolicies have at most been strengthened.

j) Example: We consider the following two policies that we state without the necessary declarations (www.inf.ethz.ch/personal/pretscha/ares09/exp.pdf).

```
// stronger:
Obl[Peter]{Always{If{download(obj=(BOT,TOP))}
   Then {Within[1]{play(obj=(trailer,trailer),dev=RCVR)}}}}
Obl[Marta]{RepUntil[2]{display(obj=movie),pay(recipient=
                  Bob,currency=CHF,amount=(10,TOP))}}
// weaker:
Obl[Peter]{Always{If{download(obj=(BOT,TOP))}
   Then {Within[3]{play(obj=(trailer,trailer),dev=RCVR)}}}}
Obl[Marta]{RepUntil[3]{display(obj=movie),
   pay(recipient=Bob,currency=CHF,amount=(5,TOP))}}
```

For Peter, the first policy is stronger because he has less time to play the trailer—this is traditional temporal strengthening. Marta's second policy is weaker because (1) she can watch the move three times rather than two times before paying and (2) she needs to pay less.

C. Mechanisms

k) Application 1: We assume scenarios where a data provider wants to transmit data to a consumer provided that the consumer is capable of enforcing an applicable policy. If the consumer sent a description of its enforcement capabilities (along the lines of the classes of mechanisms described in §II-D), then the provider could check if these mechanisms were capable of enforcing the policy. This of course requires the received capability description to be trustworthy.

l) Example: We consider the obligation `Until{Not{play(object=cd)},pay(amount=30)}` that specifies that `cd` must not be played until an amount of 30 was paid. We would like to know if the following inhibitor – once more, in concrete syntax – is able to enforce it.

```
<mechanismDeclaration>
 <inhibitor name="NoPayNoPlay">
  <triggerEvent eventName="play">
    <value attributeName="object" val="cd"/>
  </triggerEvent>
  <condition>Historically{Not{pay(amount=30)}}</condition>
 </inhibitor>
</mechanismDeclaration>
```

m) Application 2: As mentioned in §II-D, mechanisms usually are configurable. Given a configurable mechanism in the same scenario as described in application 1, the provider may want to compute values for the variables in the mechanism description so that the mechanism enforces the policy with these variable instantiations. The provider would then send the data object together with a configuration object for the consumer's mechanisms.

n) Example: We consider the same obligation as above but a parameterized mechanism now. A configuration of the mechanism that enforces the obligation is `amountA=30` together with an enumeration of all possible subjects, i.e., instantiations of variable `subjectA`. Using concrete syntax, a respective mechanism description looks as follows (the full description is provided at www.inf.ethz.ch/personal/pretscha/ares09/exp.pdf).

```
<variableDeclaration>
 <parameterVariable name="subjectA" attributeName="subject"/>
 <parameterVariable name="amountA" attributeName="amount"/>
</variableDeclaration>
<mechanismDeclaration>
 <inhibitor name="PayToDownload">
  <triggerEvent lower="download" upper="download">
   <variables lowerVar="subjectA" upperVar="subjectA"/>
   <value attributeName="object" lower="songX" upper="songY"/>
  </triggerEvent>
  <condition>Historically{Not{pay(amount=amountA,
     subject=subjectA)}} </condition>
 </inhibitor>
</mechanismDeclaration>
```

o) Translation: As shown in §II-D, mechanisms essentially consist of a condition that relates to the past (and that is translated into past LTL), possibly a triggering attempted usage, and an effect (that is translated to future LTL). If *Mech* is the set of all mechanisms in a system, or the set of mechanisms that are available at the consumer's side, respectively, let us assume that $m.\psi$ specifies the mechanism's functionality for each $m \in Mech$, as described in §II-D. Checking if a policy φ is enforced by *Mech* then boils down to model checking $\mathcal{M} \models \tau_{OSL}((\bigwedge_{m \in Mech} \underline{always}(m.\psi) \wedge C_1 \wedge C_2) \Rightarrow \varphi)$. As to the second application, provided that the cardinality of the respective types is not too large, we can simply enumerate all possible variable instantiations, including values for ordered parameters, and then perform a check as described for application 1. An example of the generated NuSMV code is provided at www.inf.ethz.ch/personal/pretscha/ares09/exp.pdf.

D. Experiments

LTL model checking is PSPACE complete. We now study if this theoretical result carries over to practical applications, and if we can realistically expect to use a model checker for the stated purposes. To this end, we conducted a series of experiments. We do not state all the policies and mechanism descriptions here (available online at www.inf.ethz.ch/personal/pretscha/fast08/exp.pdf).

Several dimensions influence the performance of our tool. Since we use a model checker, the number of events, parameters, and parameter values can be expected to have an influence on the performance. The structure and complexity of policies and mechanism descriptions will also matter (the model is trivial, as explained in §III-A). When computing configurations for mechanisms, the number of variables and potential mechanisms also likely have an impact on performance.

The *number* of events and parameter values clearly matters. However, for policies that do not include $\underline{replim}$ or $\underline{repuntil}$ operators with too large parameters (where 12 is an upper bound that is handled in less than 20 minutes on a typical laptop; the reason is the release operator), several parameter values with 100 values each can be handled within seconds. The *structure* of policies tends to be rather simple, which is why this parameter — again with the exception of the $\underline{replim}$ and $\underline{repuntil}$ operators — does not turn out to be too problematic in practice. In terms of checking the consistency of policies, among other things, we analyzed 2 policies with 19 OSL formulas each. One policy is consistent (6 seconds); the other policy is inconsistent which is detected in 4 seconds.

In terms of checking policy subsumption for the delegation of rights and duties, we used the same two policies with 19 OSL formulas each. Subsumption for all of them could be shown in 24 seconds, with certain formulas taking more time than others. One formula contains the $\underline{repuntil}$ operator (parameter value 2); checking subsumption here takes 5 seconds.

Checking policy enforcement is similar to checking policy subsumption. Depending on whether or not cardinality operators are used, the result is provided within fractions of a second, or, for larger values (< 10), up to 15 minutes.

Finally, in terms of mechanism configuration, the performance depends on the *number of variables*, m, and the *cardinalities of their types*, n_i. $\prod_{i=1}^{m} 3n_i$ combinations must be checked (recall that for ordered events and parameters, either both boundaries are identical, or one bound is $\top$ or the other $\bot$). This determines the number of model checking runs. When it is multiplied with the time needed for checking policy enforcement (possibly divided by two because on average, we can expect a solution to be found after one half of the runs), this gives the overall time that is needed. In our experiments, with up to 4 variables, results are obtained after as few as 10 seconds, but also after two hours.

We are aware that we need more policies and more mechanism descriptions before generalizing our results. However, our experiments very clearly suggest that model checking usage control policies is efficiently and practically feasible which, given the huge data types we used, surprised us.

IV. Related Work

While the focus of this paper is on machine support for analyzing usage control policies and their interplay with mechanisms, usage control itself has been discussed by several authors [15], [5], [4], [9]. UCON [15] 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. Work in the privacy area mainly focuses on the specification of privacy policies [2], [22] but not on their enforcement. Formal semantics have been provided for fragments of XrML [8] and ODRL [20], [11]. Possibly closest to our system model is that of Zhang et al. [24] who define an obligation language for UCON [15] where ongoing access models can be specified. Bandara's PhD thesis [3] is close to our work in many respects. In contrast to our approach, however, obligations in his work are implemented by mechanisms only (i.e., as event-condition-action triples), which means that no declarative "specification" policy that we use to relate policies to mechanisms exists.

Much work exists in the area of digital rights management, including rights expression languages [23], [1], interoperability frameworks [6], and architectures [13], [14] for concrete DRM mechanisms. Gunter et al. [7] present a semantics for DRM licenses based on sequences of events but do not explicitly discuss mechanisms. All the cited work has not been applied to any decision or synthesis problems, and it does not cater for the configuration of mechanisms and the notion of event refinement. One notable example is the work of Irwin et al. [12] who consider and analyze duties in the context of accountability. Rights, mechanisms, consistency, and re-distribution requirements are not considered. Finally, it is worth mentioning that many formal analyses of *access* control policies have been proposed. Since that work does not cater to the future usage of data, which is the subject of this paper, we do not provide the numerous related articles.

V. Conclusions

We have been concerned with the problem of automatically checking consistency of usage control policies, enforceability of policies by abstractly specified mechanisms, and subsumption of policies in the context of re-distribution. By enumerating possible solutions, we also want to configure mechanisms with the help of these analyses. We have shown how to formally analyze usage control policies with a model checker by translating OSL policies and abstract mechanism descriptions into a variant of LTL. As it turns out, rather complex policies can be handled very efficiently, i.e., in a matter of seconds. It is, however, easy to conceive policies or mechanisms that cannot be analyzed, a result of complex formulas that take into account cardinality operators or enormous data type domains. In other words, it is likely (and was expected) that our approach does not scale well; however, we were surprised at the comparably high complexity of policies that we could analyze within seconds. As far as we know, we are the first to provide tool support for this kind of problems.

Having analysis as the subject of this paper does not mean that other problems in the domain are less relevant. We did not discuss the negotiation of usage control policies (where we would argue that except for establishing trust relationships, it eventually boils down to bilateral take-it-or-leave-it scenarios). We also did not look into the remote enforcement of usage control policies, which, given the heterogeneity of the involved systems, of course is a far more challenging technical problem. We are currently working on such technology.

In the domain of distributed usage control, many hard problems remain, including policy management, the mentioned "physical" and tamper-proof enforcement of policies and the inherently difficult problem of relating events that are used in policies ("delete data") to the machine level (which file to delete? wipe or just delete a FAT entry? delete all copies as well?). In terms of analysis problems, the typically rather simple structure of formulas could be exploited, and constraint solvers could improve the performance when intervals are treated.

References

[1] Open Digital Rights Language - Version 1.1, August 2002. odrl.net/1.1/ODRL-11.pdf.

[2] M. Backes, B. Pfitzmann, and M. Schunter. A toolkit for managing enterprise privacy policies. In *8th European Symposium on Research in Computer Security*, Springer LNCS 2808, pages 162–180. 2003.

[3] A. Bandara. *A Formal Approach to Analysis and Refinement of Policies*. PhD thesis, Imperial College, University of London, 2005.

[4] E. Bertino, C. Bettini, and P. Samarati. A temporal authorization model. In *Proc. CCS*, pages 126–135, 1994.

[5] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy rule management. *J. Network and System Mgmt.*, 11(3):351–372, 2003.

[6] Coral Consortium Corporation. Coral Consortium Core Architecture Overview, June 2006. Version 3.0.

[7] C. A. Gunter, S. T. Weeks, and A. K. Wright. Models and languages for digital rights. In *Proc. 34th Annual Hawaii Intl. Conf. on System Sciences*, 2001.

[8] J. Halpern and V. Weissman. A Formal Foundation for XrML. In *Proc. 17th CSFW*, pages 251–265, 2004.

[9] M. Hilty, D. Basin, and A. Pretschner. On obligations. In *Proc. ESORICS*, Springer LNCS 3679, pages 98–117, 2005.

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

[11] M. Holzer, S. Katzenbeisser, and C. Schallhart. Towards a Formal Semantics for ODRL. In *Proc. 1st Intl. workshop on ODRL*, pages 137–148, 2004.

[12] K. Irwin, T. Yu, and W. Winsborough. On the Modeling and Analysis of Obligations, 2006.

[13] Marlin Engineering Workgroup. Marlin—Core System Specification, April 2006. Version 1.2.

[14] Open Mobile Alliance. DRM Architecture, March 2006. Available at www.openmobilealliance.org/release_program/drm_v2_0.html.

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

[16] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.

[17] D. Povey. Optimistic security: a new access control paradigm. In *Proc. workshop on new security paradigms*, pages 40–45, 1999.

[18] A. Pretschner, M. Hilty, C. Schaefer, T. Walter, and D. Basin. Mechanisms for Usage Control. In *Proc. ASIACCS*, pages 240–245, 2008.

[19] A. Pretschner, F. Schütz, C. Schaefer, and T. Walter. Policy evolution in distributed usage control. In *Proc. 4th Intl. Workshop on Security and Trust Management*, pages 97–110, 2008.

[20] R. Pucella and V. Weissman. A Formal Foundation for ODRL. In *Proc. Workshop on Issues in the Theory of Security*, 2004.

[21] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

[22] W3C. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification, Working Draft, 2005.

[23] X. Wang, G. Lao, T. DeMartini, H. Reddy, M. Nguyen, and E. Valenzuela. XrML – eXtensible rights Markup Language. In *ACM workshop on XML security*, pages 71–79. ACM Press, 2002.

[24] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *Proc. 9th ACM symp. on Access control models and technologies*, pages 1–10. ACM Press, 2004.