Model-Based Usage Control Policy Derivation

Prachi Kumari and Alexander Pretschner

Technische Universität München, Germany
{kumari, pretschn}@cs.tum.edu

Abstract. Usage control is concerned with how data is used after access to it has been granted. In existing usage control enforcement frameworks, policies are assumed to exist and the derivation of implementation-level policies from specification-level policies has not been looked into. This work fills this gap. One challenge in the derivation of policies is the absence of clear semantics of high-level domain-specific constructs like data and action. In this paper we present a model-based refinement of these constructs. Using this refinement, we translate usage control policies from the specification to the implementation level. We also provide methodological guidance to partially automate this translation.

1 Introduction

Usage control systems provide means to specify and enforce policies about the future usage of data. Usage control requirements have been enforced for various policy languages [1–8], at [9–18] and across [19] different layers of abstraction in various types of systems [20,21]. The focus there has been on the implementation of policy monitors. How policies are specified, translated or instantiated, has not been addressed. The challenge is that system implementations of usage control policies might not always adequately reflect end user requirements. This is due to several reasons, one of which is the problem of mapping concepts in the end user's domain to technical events and artifacts. For instance, the semantics of basic operators such as "copy" or "delete", which are fundamental for specifying usage control policies, tend to vary according to domain context and can be mapped to different sets of system events. This might wrongly allow events that should have been inhibited and block those that should have been allowed. Thus, in the absence of clear semantics of actions in an application context, it is impossible to define and enforce usage control requirements in a way that is unambiguous. This is the problem that we address in this paper.

We present a model-based policy derivation that combines usage control enforcement with data and action refinement. Policies are supposed to be specified by **end users** and translated using technical details provided by a more sophisticated user whom we call the **power user**. The translation process is **semi-automated** because it requires intervention from power users at specific points. One use case is from a web-based social network (WBSN) where an end user Alice would like to exercise control over copies of her data by other users. She would specify "do not copy my photos" in a user-friendly way. This policy

would then be translated, deployed and enforced at all client-side machines that access Alice's data. We show the step-by-step translation of this policy in the rest of this paper. It is organized along five steps:
Step 1: Specification of policies We start with an overview of a policy language [19] that is used to express constraints on the future usage of data ("don't copy photos," "delete document after 30 days," "play video at most 5 times," etc.) These requirements are called **specification-level policies**.
Step 2: Refinement of actions We express specification-level policies in terms of high-level actions like "delete" or "copy." For enforcement, we must refine these actions into their technical counterparts. Intuitively, the semantics of actions vary according to the domain context. Therefore any solution that caters to the semantics of actions must address the problem at the domain level. We recap a domain-specific meta-model from the literature [22] that distinguishes between abstract and concrete events and refine the former to the latter (no formal semantics have been given to the refinements in the foundational work).
Step 3: Semantics of action refinement We combine the usage control model and the domain meta-model to specify the formal semantics of action refinement.
Step 4: From specification-level policies to ECA rules **Implementation-level policies** are rules of event-condition-action (ECA) form that execute an *action* when a trigger *event* takes place and the respective *condition* evaluates to true. As real systems cannot look into the future, the condition part of the ECA rules must be expressed in past tense. We provide a methodological guidance for automated transformation of specification-level policies to ECA rules.
Step 5: Example translation We present the translation of our example policy "don't copy photos" for enforcement in multiple systems.
We have deliberately not considered the dynamic nature of systems in this paper; systems structures are assumed to be static. Though unrealistic, this assumption is reasonable to narrow the scope for initial results.
This work provides semantics to abstract constructs in end-user policies by modeling the basis for such semantics. It is not possible to check the correctness of the semantics that adhere to our meta-model if they indeed correspond to the idea in the end user's mind. Hence we do not discuss any theorems to check if the semantics given by the power user are indeed correct.
Problem. In sum, we tackle two problems in this paper. The first one is the fundamental problem of the lack of semantics of high-level actions in usage control policies. The second one concerns the problem of transforming specification-level policies to implementation-level policies in an automated manner.
Solution. We present a model-based translation schema for high-level actions, taking into account the different representations of data and the potential data flow through a concrete system.
Contribution. We are not aware of any work that provides a semi-automated translation of specification-level usage control policies into implementation-level policies in a generic, domain- and system-independent way.
Organization. In §2 we recap a usage control model and a domain meta-model

from the literature which we combine in §3 for action refinement. §4 backs our work with a detailed example. §5 puts our work in context and §6 concludes.

2 Background

Step 1: Specification of usage control policies. End user policies are expressed in OSL (originally described in [6]), a policy specification language that combines classical propositional operators with future-time temporal and cardinality operators. To specify and enforce policies on abstract data, the original usage control model was extended to distinguish between *data* (photo, song etc.) and its technical representations called *containers* (files, windows, records etc.) [19]. Enforcement of policies on data is done through data flow tracking. Possible data flows are defined by a transition relation on system states; actual data flows are monitored on the grounds of this relation. Formally, we consider systems $(P, \mathit{Data}, \mathit{Event}, \mathit{Container}, \Sigma, \sigma_i, \varrho)$ where P is a set of principals, *Data* is a set of data elements, *Event* is the set of events, *Container* is a set of data containers, Σ is the set of states of the system with σ_i being the initial state, and ϱ is the state transition function. System states are defined by a tuple of three mappings between data, containers and container identifiers: a *storage function* of type $\mathit{Container} \to \mathbb{P}(\mathit{Data})$ that reflects which container stores what data; an *alias function* of type $\mathit{Container} \to \mathbb{P}(\mathit{Container})$ that captures the fact that some containers may implicitly get updated whenever other containers do; and a *naming function* that provides names for containers and that is of type $F \to \mathit{Container}$, where F is a set of identifiers. The system's state space is defined as $\Sigma = (\mathit{Container} \to \mathbb{P}(\mathit{Data})) \times (\mathit{Container} \to \mathbb{P}(\mathit{Container})) \times (F \to \mathit{Container})$ with the initial state $\sigma_i = (\varnothing, \varnothing, \varnothing)$. $\mathit{Trace} = \mathbb{N} \to (\Sigma \times \mathbb{P}(\mathit{Event}))$ captures both events and the information state at a moment in time. Transitions between two states are given by $\varrho : \Sigma \times \mathbb{P}(\mathit{Event}) \to \Sigma$. At any given point of time, the state of the system is computed using a recursive function $\mathit{states} : (\mathit{Trace} \times \mathbb{N}) \to \Sigma$ which in turn is defined as $\mathit{states}(t, 0) = \sigma_i$ and $n > 0 \Rightarrow \mathit{states}(t, n) = \varrho(\mathit{states}(t, n-1), t(n-1))$.

Policies are expressed in terms of *parameterized* events on data and container. Each event belongs to the set $\mathit{Event} \subseteq \mathit{EventName} \times (\mathit{ParamName} \to \mathit{ParamValue})$. Data and containers are parameter values, belonging to disjoint sets *Data* and *Container*. Events are classified as *dataUsage* when they apply to a data object (reserved parameter *obj*) and *containerUsage* if they apply to a container object. The specification language is Φ^+ (+ for future), distinguishing between purely propositional (Ψ) and temporal and cardinality operators:

$$\Psi ::= \underline{\mathit{true}} \mid \underline{\mathit{false}} \mid E(\mathit{Event}) \mid T(\mathit{Event}) \mid \underline{\mathit{not}}(\Psi) \mid \underline{\mathit{and}}(\Psi, \Psi) \mid \underline{\mathit{or}}(\Psi, \Psi) \mid \underline{\mathit{implies}}(\Psi, \Psi)$$
$$\Phi^+ ::= \Psi \mid \underline{\mathit{not}}(\Phi^+) \mid \underline{\mathit{and}}(\Phi^+, \Phi^+) \mid \underline{\mathit{or}}(\Phi^+, \Phi^+) \mid \underline{\mathit{implies}}(\Phi^+, \Phi^+) \mid$$
$$\underline{\mathit{until}}(\Phi^+, \Phi^+) \mid \underline{\mathit{after}}(\mathbb{N}, \Phi^+) \mid \underline{\mathit{within}}(\mathbb{N}, \Phi^+) \mid \underline{\mathit{during}}(\mathbb{N}, \Phi^+) \mid \underline{\mathit{always}}(\Phi^+) \mid$$
$$\underline{\mathit{repmax}}(\mathbb{N}, \Psi) \mid \underline{\mathit{replim}}(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi) \mid \underline{\mathit{repuntil}}(\mathbb{N}, \Psi, \Phi^+)$$

As events might be modified or blocked for enforcement, distinction between *attempted/desired* and *actual* events is needed. Formulas of the form $E(\cdot)$ and $T(\cdot)$ denote actual and desired events; $\underline{\mathit{not}}, \underline{\mathit{and}}, \underline{\mathit{or}}, \underline{\mathit{implies}}$ have their intuitive

semantics; $\underline{until}$ is the weak until from LTL; the $\underline{always}$ operator is intuitive; $\underline{after}(n, a)$ is true if a becomes true after n time steps; $\underline{within}(n, a)$ is true if a holds true at least once in n timepsteps, whereas $\underline{during}(n, a)$ is true only when a is constantly true in n timesteps. $\underline{repmax}(n, a)$ specifies that a must be true at most n times in the future; $\underline{replim}(l, m, n, a)$ specifies lower(l) and upper(m) bounds on repetitions of a in n timesteps and $\underline{repuntil}(n, a, b)$ limits the maximal number of times a holds until b holds.

Sometimes, it is convenient to specify policies not in terms of events but in terms of states a system must or must not enter. E.g., our example policy in §1, "don't copy photos," would mean that in an operating system, all sequences of system calls corresponding to "copy" actions must be inhibited. But infinitely many such sequences can achieve the effect of "copy," and it is infeasible to come up with a complete list of all of them. Instead, the same requirement can be expressed as, "data must not leave a specific set of containers." To allow this type of policies, three operators Φ_i have been added to Φ^+ [9, 19]:

$$\Phi_i ::= \underline{isNotIn}(Data, \mathbb{P}\ Container) \mid \underline{isCombinedWith}(Data, Data) \mid \underline{isOnlyIn}(Data, \mathbb{P}\ Container)$$

where $\underline{isNotIn}(Data, \mathbb{P}\ Container)$ is true if data is not in a specific set of containers; $\underline{isCombinedWith}(Data, Data)$ is true if two data items are stored in the same container; and $\underline{isOnlyIn}(Data, \mathbb{P}\ Container)$ is true if data is only in a specified set of containers. The extended language is $\Phi_i^+ = \Phi^+ \cup \Phi_i$ with semantics $\models_i^+ \subseteq (Trace \times \mathbb{N}) \times \Phi_i^+$.

ECA rules, that we need for system-level enforcement, are specified in the past temporal logic Φ^- with added state-based operators, Φ_i. The extended past-time OSL is $\Phi_i^- = \Phi^- \cup \Phi_i$ with semantics $\models_i^- \subseteq (Trace \times \mathbb{N}) \times \Phi_i^-$.

$$\Phi^- ::= \Psi \mid \underline{not}^-(\Phi^-) \mid \underline{and}^-(\Phi^-, \Phi^-) \mid \underline{or}^-(\Phi^-, \Phi^-) \mid \underline{implies}^-(\Phi^-, \Phi^-) \mid \underline{since}^-(\Phi^-, \Phi^-) \mid \underline{before}^-(\mathbb{N}, \Phi^-) \mid \underline{within}^-(\mathbb{N}, \Phi^-) \mid \underline{during}^-(\mathbb{N}, \Phi^-) \mid \underline{always}^-(\Phi^-) \mid \underline{repmax}^-(\mathbb{N}, \Psi) \mid \underline{replim}^-(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi) \mid \underline{repsince}^-(\mathbb{N}, \Psi, \Phi^-)$$

$\underline{since}^-(a, b)$ is true if b has been true ever since a happened; $\underline{before}^-(n, a)$ is true if a was true n time steps ago; $\underline{within}^-$, $\underline{during}^-$ and $\underline{always}^-$ are intuitive, given the semantics of their future-time duals. $\underline{repmax}^-(n, a)$ specifies that a has been true at most n times in the past; $\underline{replim}^-(l, m, n, a)$ specifies a lower(l) and an upper limit(m) upon repetitions of a in the last n timesteps; and $\underline{repsince}^-(n, a, b)$ specifies that b has been true at most n times since a became true.

Step 2: Refinement of actions. In the usage control model of step 1, both abstract actions and their technical counterparts are called *events*. But to refine actions, we must be able to distinguish between action (copy, delete etc.) and its technical representations (generic copy file or delete file; or more specifically, read, write, unlink systems calls in unix). Our domain meta-model [22], reproduced in Fig. 1, distinguishes among user-intelligible high-level actions on data like "copy photo" at the platform-independent (PIM) layer; corresponding implementation-independent technical representations (called **transformers**) like "take screenshot" at the platform-specific (PSM) layer; and the specific implementations of these transformers like "getImage()" function in the X11 windowing system at the implementation-specific (ISM) layer. Mappings between

various components at different layers in the model provide the semantics of a high level action in terms of a number of mapped transformers. As an example, the meta-model is instantiated for the refinement of copy in WBSN domain (Figure 2). The "copy photo" part of Alice's WBSN policy would be refined in this model as "copy&paste DOM element" and "screenshot of window" at the PSM layer; and at the ISM layer as "copy_cmd on HTML element" in Firefox web browser and as "getImage" function on a drawable in X11 windowing system.

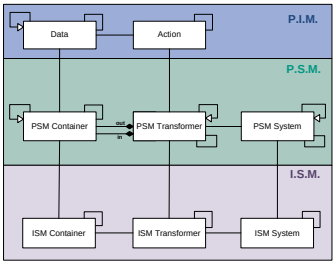

Fig. 1. The Domain Meta-model

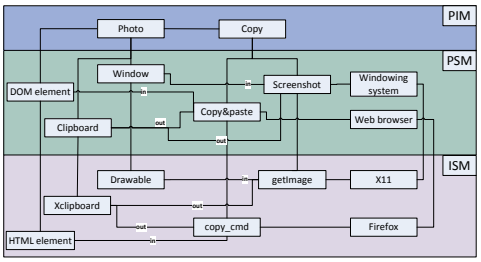

Fig. 2. A WBSN instance of Fig 1

The concepts of data and action (PIM layer) and containers and transformers (PSM and ISM layers) are also present in the usage control model, with some differences. Firstly, the above usage control model uses the term *event* to refer to both actions and transformers. In the domain meta-model, there is a clear distinction between the two. Secondly, in the domain meta-model, these constructs have been grouped according to the level of technical detail they encompass. Thus data and action form the PIM part whereas container and transformer form the ISM part in the meta-model. Thirdly, to systematically reach from elements of PIM to ISM, another layer of detail that maps the two, called the PSM layer, is introduced in the meta-model. This is motivated by the systematic translation requirement that the platform-specific result of a transformer on a container must remain the same, irrespective of the implementations. For example, deleting a file can be achieved in many ways. But by defining it as "overwrite file with random bytes OR remove file" at the PSM level narrows down the interpretation of deleting a file irrespective of the file system implementations.

The presentation of the domain meta-model [22] also contains a model-based, semi-automated approach to policy translation. However, that work discusses initial results at a high level and does not explain the exact relationship between actions and transformers. Actions are mapped to transformers using UML associations, but no semantics has been given to these associations. So we do not know what high-level actions like copy mean in terms of transformers. For example, looking at the system calls executed in a Unix operating system, we cannot know if a copy action has indeed taken place because we do not know if copy corresponds to the set or the sequence of these system calls. Even if we know that it is the sequence, we do not know how the sequence is to be interpreted: if the system calls must happen one after the other or if some other executions can take place between any two of them. Reference [22] also does not relate

high-level actions to system states: the authors only mention a refinement of the former in terms of the latter for cases where transformer-based refinements are not sufficient; they do not explain how this refinement is achieved (step 3b below). To address these issues, we combine this domain meta-model with the usage control model of step 1 to use the concept of system states and the semantics of language in terms of traces to formally refine actions and translate specification-level policies.

3 A Combined Model

Data is the set of all data, *Action* is the set of all actions, *PSMContainer* is the set of all PSM containers, *PSMTransformer* is the set of all PSM transformers and so on. *Event* is the set of all actions and transformers at all levels in the domain model: $Event = (Action \cup PSMTransformer \cup ISMTransformer)$. Associations between the elements of these sets are functions. So, $dataPotentiallyIn : Data \rightarrow \mathbb{P}(PSMContainer)$ maps data to a set of PSM containers that potentially store that data and $containerImplementedAs : PSMContainer \rightarrow \mathbb{P}(ISMContainer)$ gives a further refinement of PSM container in terms of a set of ISM containers that actually store data. Additionally, transformers are functions that modify respective containers:

$PSMTransformer : \mathbb{P}\ PSMContainer \rightarrow \mathbb{P}\ PSMContainer$
$ISMTransformer : \mathbb{P}\ ISMContainer \rightarrow \mathbb{P}\ ISMContainer$

Function $inputContainer : PSMTransformer \nrightarrow \mathbb{P}\ PSMContainer$ gives all containers modified by a PSM transformer. *inputContainer* is overloaded to get input containers of ISM transformers. While the refinement of data is straightforward, actions can be refined in two ways: *SETrefmnt* maps an action to a set of PSM transformers with the intuitive semantics that any one of the mapped transformers corresponds to the high-level action; *SEQrefmnt* maps an action to a sequence of PSM transformers: all of the specified transformers in the particular sequence correspond to the high-level action. As PSM and ISM transformers can be further refined, both within and across their respective levels in the meta-model, their refinement functions are overloaded to express both of these refinements. *SETrefmnt* and *SEQrefmnt* express intra-level refinements

$SETrefmnt : PSMTransformer \rightarrow \mathbb{P}\ PSMTransformer$
$SETrefmnt : ISMTransformer \rightarrow \mathbb{P}\ ISMTransformer$
$SEQrefmnt : PSMTransformer \rightarrow \text{seq}\ PSMTransformer$
$SEQrefmnt : ISMTransformer \rightarrow \text{seq}\ ISMTransformer$

and, *crossSETrefmnt* and *crossSEQrefmnt* express inter-level refinements.

$crossSETrefmnt : Action \rightarrow \mathbb{P}\ PSMTransformer$
$crossSETrefmnt : PSMTransformer \rightarrow \mathbb{P}\ ISMTransformer$
$crossSEQrefmnt : Action \rightarrow \text{seq}\ PSMTransformer$
$crossSEQrefmnt : PSMTransformer \rightarrow \text{seq}\ ISMTransformer$

In a specific domain model, the PSM level talks about the static, design-time system while the ISM level talks about the concrete system at the runtime.

Data Storage. The storage function in the usage control model tells which container stores what data. For the translation of actions on specific data, we need the reverse relationship; we need to know where a specific data is stored in a particular moment in time. Function $dataActuallyIn : Data \times \Sigma \rightarrow \mathbb{P}\, ISMContainer$ gives this information:

$$\forall\, d \in Data;\ t \in Trace;\ n \in \mathbb{N};\ \sigma \in \Sigma \bullet \sigma = states(t, n) \wedge dataActuallyIn(d, \sigma) = \{c \in ISMContainer \mid d \in \sigma.1(c)\}$$

where $\sigma.1$ denotes the projection on the first component of σ.

Remember that formulas of the form $E(\cdot)$ and $T(\cdot)$ denote actual and desired events in the OSL. Therefore, refinement of actions corresponds to the translation of OSL formulas of the form $E(\cdot)$ and $T(\cdot)$. A high-level action is refined in two ways: **firstly**, in terms of sets/sequences of transformers using function τ_{ev}; **secondly**, in terms of system states using function τ_{state}. We combine both refinements to get the **complete** refinement of a high-level action.

Step 3a: Action Refinement using Transformers. As it is impossible to predict the length of executions between any two members of a sequence of transformers in real systems, we allow arbitrary executions between any two members of a sequence of applicable transformers in *SEQrefmnt*. This introduces liveness in our action refinement definitions. Because indefinite past can be checked in a running system as opposed to indefinite future, we first translate a specification-level policy from future to past tense and then execute action refinement. For this reason, our action refinement functions act on, and are formalized, using past-time OSL operators. A translation function $\tau_p : \Phi^+ \rightarrow \Phi^-$ that works along the lines of the methodological guidance provided in [22], translates a formula in Φ^+ to another in Φ^-. To express indefinite past, we use $\underline{eventually}^-$, semantically equivalent to $\underline{not}^-(\underline{always}^-(\underline{not}^-))$ in the language Φ_i^-. Intuitively, $\underline{eventually}^-(\varphi)$ is true if the formula φ was true at least once in the past.

$\tau_{ev} : \Phi^- \times \Sigma \rightarrow \Phi^-$ translates an action into sets/sequences of transformers that are further refined using $\pi_{ev} : \Phi^- \times \Sigma \rightarrow \Phi^-$. The system state ($\Sigma$) provides the knowledge of data storage in specific containers and is filled in by the "higher" translation function τ_{action}, defined later in this paper.

We refine high-level actions by taking into account all representations of data in a concrete system. Therefore, only those transformers that modify containers where data *may* reside, refine the corresponding high-level action. If the data on which an action operates, cannot be stored in a particular container, all transformers that operate on this container are left out of the refinement process. For example, a copy action is refined into the set {copyFile(file), takeScreenshot(window)} at the PSM level. When the data is a song and the policy addresses copy(song), the action is refined into set {copyFile(file)} rather than {copyFile(file), takeScreenshot(window)}. This is because the other transformer operates on windows where a song cannot be stored. In this example, {copyFile(file)} is the set of **applicable transformers**. The set of applicable transformers for a data ($appTransformer : Data \nrightarrow \mathbb{P}\, PSMTransformer$) is computed as follows:

$$\forall\, d \in Data \bullet appTransformer(d) = \{t \in PSMTransformer \mid inputContainer(t) \subseteq dataPotentiallyIn(d)\}$$

Using set and sequence mappings from action to PSM transformers that modify potential storage of data object of the action (via $\mathsf{ran}\ SEQrefmnt(e) \cap appTransformer(d)$ etc.), we compute action refinement upto the lowest level in the platform-specific model (ran is the standard operation for sequences [23] that gives the set of objects which are elements of the sequence)

$$
\begin{aligned}
&\forall\, s \in Trace;\ x \in \mathbb{N};\ \sigma \in \Sigma \bullet \sigma = states(s, x) \Rightarrow \\
&\forall\, d \in Data;\ e \in Event;\ \{t_1, .., t_n\} \in \mathbb{P}(PSMTransformer);\ \varphi \in \Phi^- \bullet \\
&\tau_{ev}(E(e, \{(obj, d)\}), \sigma) = \varphi \Leftrightarrow \\
&\varphi = \underline{and}^-(\tau_{ev}(E(t_n, \{(obj, d)\}), \sigma), \underline{eventually}^-(\underline{and}^-(\tau_{ev}(E(t_{n-1}, \{(obj, d)\}), \sigma), \\
&\qquad ...\underline{eventually}^-(\tau_{ev}(E(t_1, \{(obj, d)\}), \sigma))...))) \wedge \\
&\quad (\{t_1, .., t_n\} = \mathsf{ran}\ SEQrefmnt(e) \cap appTransformer(d) \vee \\
&\quad \{t_1, .., t_n\} = \mathsf{ran}\ crossSEQrefmnt(e) \cap appTransformer(d)) \\
&\vee \varphi = \underline{or}^-(\tau_{ev}(E(t_1, \{(obj, d)\}), \sigma), \underline{or}^-(\tau_{ev}(E(t_2, \{(obj, d)\}), \sigma), \\
&\qquad ..., \tau_{ev}(E(t_n, \{(obj, d)\}), \sigma))) \wedge \\
&\quad (\{t_1, ..., t_n\} = SETrefmnt(e) \cap appTransformer(d) \vee \\
&\quad \{t_1, ..., t_n\} = crossSETrefmnt(e) \cap appTransformer(d)) \\
&\vee \varphi = \pi_{ev}(E(e, \{(obj, d)\}), \sigma)
\end{aligned}
$$

π_{ev} further refines these transformers till the ISM level. Meanings of sequence and set refinement remain the same. From all the possible input containers, mapped transformers act on only those containers that indeed store the specific data object ($c \in inputContainer(t) \cap dataActuallyIn(d, \sigma)$):

$$
\begin{aligned}
&\forall\, s \in Trace;\ x \in \mathbb{N};\ \sigma \in \Sigma \bullet \sigma = states(s, x) \wedge \\
&\forall\, d \in Data;\ e \in Event;\ \{t_1, .., t_n\} \in \mathbb{P}(ISMTransformer);\ t \in \{t_1, .., t_n\}; \\
&c \in (inputContainer(t) \cap dataActuallyIn(d, \sigma));\ \varphi \in \Phi^- \bullet \\
&\pi_{ev}(E(e, \{(obj, d)\}), \sigma) = \varphi \Leftrightarrow \\
&\varphi = \underline{and}^-(E(t_n, \{(obj, c)\}), \underline{eventually}^-(\underline{and}^-(E(t_{n-1}, \{(obj, c)\}), \\
&\qquad ...\underline{eventually}^-(E(t_1, \{(obj, c)\}))...))) \wedge \langle t_1, .., t_n\rangle = crossSEQrefmnt(e) \\
&\vee \varphi = \underline{or}^-(E(t_1, \{(obj, c)\}), \underline{or}^-(E(t_2, \{(obj, c)\}), ..., E(t_n, \{(obj, c)\}))) \wedge \\
&\qquad \{t_1, ..., t_n\} = crossSETrefmnt(e) \\
&\vee \varphi = false
\end{aligned}
$$

Step 3b: Action Refinement using State. We have seen in §2 that expressing the semantics of high level actions in terms of sets/sequences of transformers might not be the best approach in many cases because one high-level action can be refined to infinitely many sequences of transformers at the system level. To address this issue, we define another refinement of action, τ_{state}, using state-based operators of Φ_i. This translation captures the state a system reaches when a high-level action is executed on some data. The power user models the execution of each action and defines the resultant state as a *StateFormula* for each high-level action. Intuitively, when no resultant state is defined for an action, its state-based translation is *false*.

To define the set of all possible events that can occur in a concrete system, event declarations were introduced in [6]. These event declarations are purely syntactic and are given by $EventDecl == EventName \times EventClass \times (ParamName \nrightarrow \mathbb{P}\ ParamValue)$. To express the state-based refinement of an action, we modify this event declaration by adding *StateFormula* to it. Thus, an event declaration is given by the event name, the event class, a partial function

that defines the name and possible values of each possible parameter and, the resultant state formula that gives the state-based refinement of the event.

$$EventDecl == EventName \times EventClass \times (ParamName \nrightarrow \mathbb{P}\, ParamValue) \times StateFormula$$

The relationship between an action and its declaration is **bijective**. For the state-based translation of action, a function *getStateFormula* fetches the resultant state from the declaration of the specific action:

$$getStateFormula : Action \nrightarrow StateFormula$$
$$\forall\, a \in Action;\ ed \in EventDecl \bullet getStateFormula(a) = ed.4 \Leftrightarrow a.1 = ed.1$$

The resultant state formula for an action is statically defined, before the action is used to specify policies. Actual data objects and their containers are known only when a policy is deployed in a concrete system. So resultant state formulas must address all potential data and their containers. To specify potential data in the state-based formula, we use variables which are substituted by actual data when a policy is specified. To specify containers that potentially store data, we extend the language Φ_i to include state-based operators on PSM containers. At runtime, respective ISM containers are extracted via function *containerImplementedAs*, introduced in the beginning of §3.

PSM Containers in state-based operators. To specify PSM containers in state-based operators, we classify ISM containers according to the PSM containers they implement. So each ISM container belongs to a *container class* that is a PSM container. Function $getContainerClass : ISMContainer \rightarrow PSMContainer$ extracts the class of a container using function *containerImplementedAs*:

$$\forall\, c \in ISMContainer;\ cl \in PSMContainer \bullet$$
$$getContainerClass(c) = cl \Leftrightarrow c \in containerImplementedAs(cl)$$

We extend the language by overloading two operators with PSM containers: $isNotIn(Data, \mathbb{P}\, PSMContainer)$ and $isOnlyIn(Data, \mathbb{P}\, PSMContainer)$. Intuitively, $isNotIn(d, Cl)$ is true if data d is not in any container whose class is in set Cl. This operator is useful for defining state-based refinement of actions like copy or print. For example, if print is refined as $not(isNotIn(d, \{print_{cont}\}))$ where $print_{cont}$ represents the class of printer containers. When data d flows into any container that belongs to this class, the enforcement infrastructure would recognize a print. Similarly, $isOnlyIn(d, Cl)$ is true when data is restricted to specific classes of containers. This is useful to express semantics of *weak deletion* where data is not actually deleted but only quarantined. We did not find any use case where the semantics of $isCombinedWith(d, d)$ need to be specified using container classes. The new operators are added to the language Φ_i:

$$\Phi_i ::= isNotIn(Data, \mathbb{P}\, ISMContainer) \mid isNotIn(Data, \mathbb{P}\, PSMContainer) \mid$$
$$isOnlyIn(Data, \mathbb{P}\, ISMContainer) \mid isOnlyIn(Data, \mathbb{P}\, PSMContainer) \mid$$
$$isCombinedWith(Data, Data)$$

The semantics of Φ_i is $\models_i \subseteq (Trace \times \mathbb{N}) \times \Phi_i$, shown in Figure 3.

Variables in OSL. Resultant state formulas are expressed in terms of potential data and their containers because actual data and their containers are known only when a policy is deployed in a concrete system. To specify state formulas with potential data items, we introduce variables in the language. Only variable

data is needed; potential containers are specified using PSM containers. In case of *isCombinedWith*, the first data is variable, the second data is given by the power user. The respective language is Φ_{iv}.

$$
\begin{aligned}
&Var ::= V(\mathbb{N}_1)\\
&VarData == Var \cup Data\\
&\Phi_{iv} ::= isNotIn(VarData, \mathbb{P}\, ISMContainer) \mid isNotIn(VarData, \mathbb{P}\, PSMContainer) \mid\\
&\quad isOnlyIn(VarData, \mathbb{P}\, ISMContainer) \mid isOnlyIn(VarData, \mathbb{P}\, PSMContainer) \mid\\
&\quad isCombinedWith(VarData, Data)
\end{aligned}
$$

Elements from Φ_{iv} are instantiated into elements from Φ_i using **substitution**.

Finally, the refinement of actions in terms of states is achieved using $\tau_{state} : \Phi^- \rightarrow \Phi_i^-$. When an action is refined, the variable in the respective state formula is substituted by the value of the *obj* parameter of the action.

$$
\forall\, a \in Action;\ d \in Data;\ \varphi \in \Phi_{iv};\ vd \in VarData \bullet
$$

$$
\tau_{state}(E(a, \{(obj, d)\})) = \begin{cases} \varphi[d/vd] & \text{if}(\varphi = getStateFormula(a)) \\ false & \text{otherwise} \end{cases}
$$

We have defined the refinement of a high-level action in terms of sets/sequences of transformers (using function τ_{ev}) and in terms of system states (using function τ_{state}). We now combine both functions to express the "complete" refinement of a high-level action, given by $\tau_{action} : (\Phi^- \times \Sigma) \rightarrow \Phi_i^-$. Intuitively, at least one of the refinements is needed to express a high-level action in a concrete system. Hence the disjunction ($\underline{or}^-$) over the refinements (Figure 4).

Step 4: From specification-level policies to enforcement mechanisms Policy specification and translation is semi-automated with two roles of users: the end user specifies usage control policies with constructs and templates defined by the more sophisticated power user §1.

In the **first step**, τ_p translates a future-time formula into another past-time formula [22]. In the **second step**, action refinement takes place. After action refinement, we get a complex, nested formula that is broken down to subformulas (Fischer Ladner closure) in the **third step** and each subformula is then mapped to the condition part of one ECA rule in the **fourth step**. Thus we get a set of ECA rules corresponding to one specification-level policy. One high-level policy can be enforced in many ways (allow/modify/inhibit/delay). For example, Alice's policy "don't copy photo" can be enforced by inhibiting every copy event;

$$
\begin{aligned}
&\forall\, t \in Trace;\ n \in \mathbb{N};\ \varphi \in \Phi_i;\ \sigma \in \Sigma \bullet\ (t, n) \models_i \varphi \Leftrightarrow \sigma = states(t, n) \wedge\\
&\quad \exists\, d \in Data, C \in \mathbb{P}\, ISMContainer \bullet \varphi = \underline{isNotIn}(d, C) \wedge\\
&\qquad \forall\, c' \in ISMContainer \bullet d \in \sigma.1(c') \Rightarrow (c' \notin C)\\
&\quad \exists\, d \in Data, Cl \in \mathbb{P}\, PSMContainer \bullet \varphi = \underline{isNotIn}(d, Cl) \wedge\\
&\qquad \forall\, c' \in ISMContainer \bullet d \in \sigma.1(c') \Rightarrow (getContainerClass(c') \notin Cl)\\
&\quad \vee\, \exists\, d \in Data, C \in \mathbb{P}\, ISMContainer \bullet \varphi = \underline{isOnlyIn}(d, C) \wedge\\
&\qquad \forall\, c' \in ISMContainer \bullet d \in \sigma.1(c') \Rightarrow \overline{(c' \in C)}\\
&\quad \vee\, \exists\, d \in Data, Cl \in \mathbb{P}\, PSMContainer \bullet \varphi = \underline{isOnlyIn}(d, Cl) \wedge\\
&\qquad \forall\, c' \in ISMContainer \bullet d \in \sigma.1(c') \Rightarrow \overline{(getContainerClass(c') \in Cl)}\\
&\quad \vee\, \exists\, d_1, d_2 \in Data \bullet \varphi = \underline{isCombinedWith}(d_1, d_2) \wedge\\
&\qquad \exists\, c' \in ISMContainer \bullet d_1 \in \sigma.1(c') \wedge d_2 \in \sigma.1(c')
\end{aligned}
$$

Fig. 3. Semantics of Φ_i

$$
\begin{aligned}
&\forall\, t \in \mathit{Trace};\ n \in \mathbb{N};\ \sigma \in \Sigma \bullet \sigma = \mathit{states}(t, n) \wedge \\
&\forall\, d \in \mathit{Data};\ a \in \mathit{Action};\ \psi \in \Phi^{-};\ \varphi \in \Phi_i^{-} \bullet \tau_{action}(\psi, \sigma) = \varphi \Leftrightarrow \\
&\psi \in \{\mathit{true}, \mathit{false}\} \wedge (\varphi = \psi) \\
&\vee\ \psi = E(a) \wedge (\varphi = \underline{or}^{-}(\tau_{state}(E(a)), \tau_{ev}(E(a), \sigma))) \\
&\vee\ \psi = T(a) \wedge (\varphi = \underline{or}^{-}(\tau_{state}(T(a)), \tau_{ev}(T(a), \sigma))) \\
&\vee\ \exists\, \chi \in \Phi^{-} \bullet \psi \in \{\underline{not}(\chi), \underline{not}^{-}(\chi)\} \wedge (\varphi = \underline{not}^{-}(\tau_{action}(\chi, \sigma))) \\
&\vee\ \exists\, \chi, \xi \in \Phi^{-} \bullet \psi \in \{\underline{or}(\chi, \xi), \underline{or}^{-}(\chi, \xi)\} \wedge (\varphi = \underline{or}^{-}(\tau_{action}(\chi, \sigma), \tau_{action}(\xi, \sigma))) \\
&\vee\ \exists\, \chi, \xi \in \Phi^{-} \bullet \psi \in \{\underline{and}(\chi, \xi), \underline{and}^{-}(\chi, \xi)\} \wedge (\varphi = \underline{and}^{-}(\tau_{action}(\chi, \sigma), \tau_{action}(\xi, \sigma))) \\
&\vee\ \exists\, \chi, \xi \in \Phi^{-} \bullet \psi \in \{\underline{implies}(\chi, \xi), \underline{implies}^{-}(\chi, \xi)\} \wedge (\varphi = \underline{implies}^{-}(\tau_{action}(\chi, \sigma), \tau_{action}(\xi, \sigma))) \\
&\vee\ \exists\, \chi, \xi \in \Phi^{-} \bullet \psi = \underline{since}^{-}(\chi, \xi) \wedge (\varphi = \underline{since}^{-}(\tau_{action}(\chi, \sigma), \tau_{action}(\xi, \sigma))) \\
&\vee\ \exists\, i \in \mathbb{N};\ \chi \in \Phi^{-} \bullet \psi = \underline{before}^{-}(i, \chi) \wedge (\varphi = \underline{before}^{-}(i, \tau_{action}(\chi, \sigma))) \\
&\vee\ \exists\, \chi \in \Phi^{-} \bullet \psi = \underline{always}^{-}(\chi) \wedge (\varphi = \underline{always}^{-}(\tau_{action}(\chi, \sigma))) \\
&\vee\ \exists\, i \in \mathbb{N};\ \chi \in \Phi^{-} \bullet \psi = \underline{within}^{-}(i, \chi) \wedge (\varphi = \underline{within}^{-}(i, \tau_{action}(\chi, \sigma))) \\
&\vee\ \exists\, i \in \mathbb{N};\ \chi \in \Phi^{-} \bullet \psi = \underline{during}^{-}(i, \chi) \wedge (\varphi = \underline{during}^{-}(i, \tau_{action}(\chi, \sigma))) \\
&\vee\ \exists\, i \in \mathbb{N};\ \chi \in \Phi^{-} \bullet \psi = \underline{repmax}^{-}(i, \chi) \wedge (\varphi = \underline{repmax}^{-}(i, \tau_{action}(\chi, \sigma))) \\
&\vee\ \exists\, l, x, y \in \mathbb{N};\ \chi \in \Phi^{-} \bullet \psi = \underline{replim}^{-}(l, x, y, \chi) \wedge (\varphi = \underline{replim}^{-}(l, x, y, \tau_{action}(\chi, \sigma))) \\
&\vee\ \exists\, i \in \mathbb{N};\ \chi, \xi \in \Phi^{-} \bullet \psi = \underline{repsince}^{-}(i, \chi, \xi) \wedge (\varphi = \underline{repsince}^{-}(i, \tau_{action}(\chi, \sigma), \tau_{action}(\xi, \sigma)))
\end{aligned}
$$

Fig. 4. Definition of τ_{action}

it can be enforced by modifying the original photo with one that shows an error message; it can also be enforced by delaying the event until a permission for copying has been granted by Alice. For this reason, the action part of ECA rules cannot be specified automatically. The generic format of ECA rules at the end of step 4 is as follows (where c is one subformula)

```
Event: any
Condition: c
Action: ALLOW/MODIFY/INHIBIT/DELAY
```

Intuitively, (later configured) action takes place when the corresponding condition c is true, irrespective of the trigger event. To limit the set of trigger events for each rule, whenever c is of the form $\underline{and}^{-}(E(e), x)$ or $\underline{and}^{-}(T(e), x)$ where x is an OSL formula, we move e to the trigger event part and only x is checked in the condition part of the ECA rule.

```
Event: e
Condition: x
Action: ALLOW/MODIFY/INHIBIT/DELAY
```

All the steps described above are automated. In the **fifth step**, the power user manually specifies the enforcement mechanism. We now describe in detail the translation of the example policy introduced in §1.

4 Example Translation

Step 5: The partial domain model with transformer-based refinement of "copy photo" is shown Figure 5. The distinction between set and sequence refinements of events is shown via links with arrowheads representing SETrefmnts and links with AND(S) gate -head representing SEQrefmnts. For state-based action refinement, state formula is defined in the event declaration. *Copy* in this context means data flows in clipboard containers; hence the respective state formula is

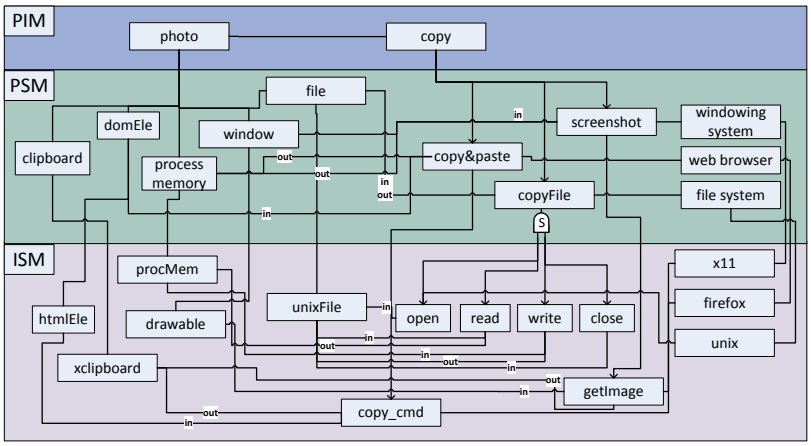

Fig. 5. Example domain model

$\underline{not}^-(\underline{isNotIn}(x, clipboard))$ where x is variable data and *clipboard* is the class of clipboard containers.

In our implementation, "don't copy photos" is specified by Alice in a block editor that uses the Open Blocks Java library [24]. When the data is sent to another user Bob, the respective policy is delivered to the policy translation point (PTP) which immediately translates and deploys the policy.

In the runtime, when policies are deployed, only concrete containers exist. So in our implementation, data is identified by the initial container in which it appears in the concrete system. With our usage control infrastructure, it is possible to track multiple representations of the same data at and across different abstract layers in a system. Hence, the PTP knows that Alice's photo is received by Bob at the web browser level in the initial container "img_profile" in Firefox; is stored in "myphoto.jpg" in the cache folder and rendered in window "0x1a00005" in X11.

The policy is $\underline{always}(\underline{not}(E(copy, \{(obj, img\_profile)\})))$ in OSL. It is of the form $\underline{always}(\varphi)$ where $\varphi = \underline{not}(E(copy, \{(obj, img\_profile)\}))$. τ_p gives us the past-time condition to be checked in the respective ECA rules. $\tau_p(\underline{always}(\varphi)) = \underline{and}^-(\underline{before}^-(1, \tau_p(\varphi)\underline{since}^- START), \underline{not}^-(\tau_p(\varphi)))$ where $START$ denotes the policy activation event [22]. This means that the respective ECA rule is triggered when φ has always been true since the policy was activated, except the current time-step. As $\varphi = \underline{not}(E(copy, \{(obj, img\_profile)\}))$, the ECA rule is triggered when $E(copy, \{(obj, img\_profile)\})$ is true.

The next step is action refinement as described in §3. $\tau_{action}(E(copy, \{(obj, img\_profile)\}), \sigma)$, where σ is the current state, works as follows: State-based refinement is achieved by substituting variable x with img_profile in the state formula:
$\tau_{state}(E(copy, \{(obj, img\_profile)\})) = \underline{not}^-(\underline{isNotIn}(img\_profile, clipboard))$
Applying τ_{ev}, copy is refined to {copy&paste,screenshot,copyFile} because these transformers operate on {domEle,window,file} where photo is potentially stored $(crossSETrefmnt(copy) \cap appTransformer(photo) = \{copy\&paste, screenshot,$

copyFile}); π_{ev} refines each of these transformers till the ISM level. Note that, of the sequence $\langle open, read, write, close\rangle$, write is not included in action refinement because it does not operate on the file that stores photo ($inputContainer(write) \cap dataActuallyIn(myphoto.jpg, \sigma) = \varnothing$). Finally,

$$\begin{aligned}
&\tau_{action}(E(copy, \{(obj, img_profile)\}), \sigma)\\
&= \underline{or}^{-}(\underline{not}^{-}(\underline{isNotIn}(img_profile, clipboard)),\\
&\quad \underline{or}^{-}(E(copy_cmd, \{(obj, img_profile)\}), \underline{or}^{-}(E(getImage, \{(obj, 0x1a00005)\}),\\
&\quad (\underline{and}^{-}(E(close, \{(obj, myphoto.jpg)\}), \underline{eventually}^{-}(\underline{and}^{-}(E(read, \{(obj,\\
&\qquad myphoto.jpg)\}), \underline{eventually}^{-}(E(open, \{(obj, myphoto.jpg)\}))))))))))
\end{aligned}$$

In the **third step**, following subformulas are computed:

$$\begin{aligned}
\varphi_1 &= \underline{not}^{-}(\underline{isNotIn}(img_profile, clipboard))\\
\varphi_2 &= E(copy_cmd, \{(obj, img_profile)\})\\
\varphi_3 &= E(getImage, \{(obj, 0x1a00005)\})\\
\varphi_4 &= \underline{and}^{-}(E(close, \{(obj, myphoto.jpg)\}), \underline{eventually}^{-}(\underline{and}^{-}(E(read, \{(obj,\\
&\qquad myphoto.jpg)\}), \underline{eventually}^{-}(E(open, \{(obj, myphoto.jpg)\}))))))
\end{aligned}$$

In the **fourth step**, generic ECA rules, as described above, are generated for each subformula. φ_2 and φ_3 are of the form $\underline{and}^{-}(E(e), true)$. So respective e becomes the trigger event as described above, and the condition part of the respective ECA rules is *true*. The specific action to be taken in each ECA rule is manually specified in the **fifth step**.

5 Related Work

The goal of this work is to automate the refinement of policies in the context of usage control. Policy refinement has been the focus of research since quite some time [25] and in the recent years, there have been various attempts towards automating it. Solutions have been based on refining policies using resource hierarchies [26], commitment (obligations) analysis [27], goal decomposition [28], data classification [29] and also from different perspectives viz. conflict prevention, where the focus has more been on the translation of constraints [30]. In [31] and [32], ontology-based refinement techniques are described for semi-automated translation of access control policies. In our work, such ontologies could be used at each level of the meta-model. In [33], authors have proposed a resource hierarchy meta-model for translating domain-specific elements in XACML policies for virtual organizations to generate corresponding resource-level policies. This is similar to our work in terms of the approach. However, the policies are refined from the abstract level (users, resources and applications) to the logical level (user ids, resource addresses and computational commands like read/write); further technical representations of policy elements in concrete systems are not considered. Another work which is quite similar to ours in terms of approach is described in [34]. This paper focuses on action decomposition in a policy refinement framework. Subjects perform operations on targets (services and devices) which are specified at a high level. Using a system model and a set of refinement rules, actions are decomposed and one higher level policy is refined into multiple policies. However, all elements (both abstract and concrete) of the system model

are at the same level; which makes this approach similar to the ontology-based refinement. Also, in the last stage of refinement, policies are transformed into ECA rules. How this transformation is achieved is however not specified.

In almost all of the work on policy refinement, there has been some kind of distinction between the abstract entities at high level and the corresponding technical entities at lower levels. This approach of capturing details of a system with several levels of abstraction has been addressed in many architecture frameworks [35–37] and is also common in the embedded systems domain [38–40]. We have adopted a model-based approach which is analogous to the MDA viewpoints [41] with varying level of details at the computation-independent, platform-independent and platform-specific levels. A minor difference with the MDA approach is in the naming of the different layers. We have combined this approach with usage control concepts to refine policies.

The contribution w.r.t. to reference [22] is detailed out in §2, step 2.

6 Conclusion and Future Work

This paper describes a model-based policy refinement for usage control enforcement. Through this work, we have addressed the fundamental problem of the lack of semantics of actions like copy or delete. Additionally, we have provided a methodological guidance for transforming specification-level policies into implementation-level policies that configure enforcement mechanisms at different layers of abstraction. This helps translate policies in an automated manner.

For precise semantics of action refinement, we have combined an existing domain meta-model with a usage control model from the literature. The combined model captures both the static (all possible cases) and dynamic (one particular case with runtime information) aspects of concrete systems. The refinement of actions in this combined model is twofold: actions are refined to sets/sequences of low-level transformers and also to state-based formulas that describe the storage of data in containers. Refinement of actions is used to give semantics to specification-level policies in terms of a set of system traces. We have also provided methodological guidance to automate the policy translation: when future-time policies are translated to their past-time equivalents, the complex formula with all action refinements is decomposed into subformulas and mapped to the condition part of ECA rules.

It is hard to establish a notion of correctness between the semantics of low-level and high-level policies because the semantics of high-level propositions is not precisely defined but rather exists in the (end) user's mind. In fact, we see our translation procedure as a way to *define* the semantics of high-level policies by assigning machine-level events and state changes to high-level actions.

We have deliberately introduced a limitation in this paper: we have not considered the dynamic nature of systems. Adaptive policy translation is a topic of ongoing work. Another topic of current investigation is the evolution of policies [42] since we have not considered the fact that specification-level policies may also change from one receiver to another in a distributed setup.

References

1. R. Iannella (ed.). Open Digital Rights Language v1.1, 2008. http://odrl.net/1.1/ODRL-11.pdf.
2. Multimedia framework (MPEG-21) – Part 5: Rights Expression Language, 2004. ISO/IEC standard 21000-5:2004.
3. P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise Privacy Authorization Language (EPAL 1.2). IBM Technical Report, 2003.
4. Open Mobile Alliance. DRM Rights Expression Language V2.1, 2008. http://www.openmobilealliance.org/Technical/release_program/drm_v2_1.aspx.
5. X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *Proc. SACMAT*, pages 1–10, 2004.
6. M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In *Proc. ESORICS*, pages 531–546, 2008.
7. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Workshop on Policies for Distributed Systems and Networks '95*.
8. W3C. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification, 2005. http://www.w3.org/TR/2005/WD-P3P11-20050104/.
9. M. Harvan and A. Pretschner. State-based Usage Control Enforcement with Data Flow Tracking using System Call Interposition. In *Proc. 3rd Intl. Conf. on Network and System Security*, pages 373–380, 2009.
10. A. Pretschner, M. Buechler, M. Harvan, C. Schaefer, and T. Walter. Usage control enforcement with data flow tracking for x11. In *Proc. STM '09*, pages 124–137.
11. M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded java. In *Proc. ECOOP*, pages pp. 546–569, 2009.
12. I. Ion, B. Dragovic, and B. Crispo. Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. In *Proc. Annual Computer Security Applications Conference*, pages 233–242. IEEE Computer Society, 2007.
13. L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The S3MS.NET Run Time Monitor: Tool Demonstration. *ENTCS*, 253(5):153–159, 2009.
14. U. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop*, pages 87–95, 1999.
15. B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc IEEE Symposium on Security and Privacy*, pages 79–93, 2009.
16. G. Gheorghe, S. Neuhaus, and B. Crispo. xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In *Proc. ICTM*, 2010.
17. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of USENIX Annual Technical Conference*, June 2007.
18. P. Kumari, A. Pretschner, J. Peschla, and J. Kuhn. Distributed data usage control for web applications: a social network implementation. In *Proc. 1st ACM Conf. on Data and application security and privacy*, pages 85–96, 2011.
19. A. Pretschner, E. Lovat, and M. Buechler. Representation-independent data usage control. In *Proc. 6th Intl. Workshop on Data Privacy Management*, 2011.
20. Denis Feth and Alexander Pretschner. Flexible Data-Driven Security for Android. In *SERE '12*, pages 41–50, June 2012.
21. Prachi Kumari, Florian Kelbert, and Alexander Pretschner. Data Protection in Heterogeneous Distributed Systems: A Smart Meter Example. In *INFORMATIK 2011 - Dependable Software for Critical Infrastructures*, 2011.

22. Prachi Kumari and Alexander Pretschner. Deriving implementation-level policies for usage control enforcement. In *Proc. 2nd ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 83–94. ACM, 2012.
23. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall (UK), 1998.
24. Ricarose Roque. Open Blocks, 2009. `http://education.mit.edu/openblocks`.
25. M. Abadi and L. Lamport. The existence of refinement mappings. In *LICS '88*.
26. L. Su, D. Chadwick, A. Basden, and J. Cunningham. Automated decomposition of access control policies. In *Proc. 6th IEEE Intl. Workshop on Policies for Distributed Systems and Networks*, pages 6–8, 2005.
27. J. Young. Commitment analysis to operationalize software requirements from privacy policies. *Requirements Engineering*, 16:33–46, 2011.
28. A.K. Bandara, E.C. Lupu, J. Moffett, and A. Russo. A goal-based approach to policy refinement. In *Proc. 5th IEEE Workshop on Policies for Distributed Systems and Networks*, pages 229–239, 2004.
29. Y.B. Udupi, A. Sahai, and S. Singhal. A classification-based approach to policy refinement. In *Proc. 10th Intl Symp. on Integrated Network Management*, 2007.
30. Steven Davy, Brendan Jennings, and John Strassner. Policy conflict prevention via model-driven policy refinement. In *Proc DSOM '06*. Springer-Verlag.
31. Cataldo Basile, Antonio Lioy, Salvatore Scozzi, and Marco Vallini. Ontology-based policy translation. In *Computational Intelligence in Security for Information Systems*, volume 63, pages 117–126. 2009.
32. A. Guerrero, V.A. Villagrá, J.E. López de Vergara, A. Sánchez-Macián, and J. Berrocal. Ontology-based policy refinement using swrl rules for management information definitions in owl. In *DSOM*, pages 227–232, 2006.
33. B. Aziz, A.E. Arenas, and M. Wilson. Model-based refinement of security policies in collaborative virtual organisations. ESSoS, pages 1–14, 2011.
34. R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman. Decomposition techniques for policy refinement. In *Proc CNSM '10*, pages 72 –79, 2010.
35. C. O'Rourke, N. Fishman, and W. Selkow. *Enterprise architecture using the Zachman Framework*. Course Technology, 2003.
36. J. A. Zachman. A framework for information systems architecture. *IBM Syst. J.*, 26:276–292, September 1987.
37. The Open Group. TOGAF Version 9. 2009.
38. Alexander Gruler, Alexander Harhurin, and Judith Hartmann. Modeling the functionality of multi-functional software systems. In *Proc. ECBS '07*, pages 349–358. IEEE Computer Society, 2007.
39. Dirk Ziegenbein, Peter Braun, Ulrich Freund, Andreas Bauer, Jan Romberg, and Bernhard Schatz. Automode - model-based development of automotive software. In *Proc. DATE '05*, pages 171–177. IEEE Computer Society, 2005.
40. Birgit Penzenstadler. Tackling automotive challenges with an integrated re & design artifact model. In *Proc. OTM '08*, pages 426–431. Springer-Verlag, 2008.
41. J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical Report omg/03-06-01, Object Management Group (OMG), June 2003.
42. A. Pretschner, F. Schütz, C. Schaefer, and T. Walter. Policy evolution in distributed usage control. *Electr. Notes Theor. Comput. Sci.*, 244:109–123, 2009.