Deriving Implementation-level Policies for Usage Control Enforcement[*]

Prachi Kumari
Certifiable Trustworthy IT Systems
Karlsruhe Institute of Technology
Germany

Alexander Pretschner
Certifiable Trustworthy IT Systems
Karlsruhe Institute of Technology
Germany

ABSTRACT

Usage control is concerned with how data is used after access to it has been granted. As such, it is particularly relevant to end users who own the data. System implementations of access and usage control enforcement mechanisms, however, do 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, semantics of basic operators such as "copy" or "delete", which are fundamental for specifying privacy policies, tend to vary according to context. For this reason they can be mapped to different sets of system events. The behaviour users expect from the system, therefore, may differ from the actual behaviour. In this paper we present a translation of specification-level usage control policies into implementation-level policies which takes into account the precise semantics of domain-specific abstractions. A tool for automating the translation has also been implemented.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Security

Keywords

Security and privacy, policy enforcement, usage control, semantics, user vs. system requirements

1. INTRODUCTION

Access and usage control systems provide means to specify and enforce policies about who can access data and how.

[*]Support by the Google Research Award "Towards Operational Privacy" is gratefully acknowledged.

At the level of *end users*, a policy is a set of rules specified using abstract vocabularies such as "this picture may not be printed" or "this address may not be distributed." We refer to these rules as specification-level policies. The set of actions, subjects and objects that are used in such policies differs according to the domain context. Within each domain, policies must be defined and enforced at different levels of abstraction, including the operating system [1], windows manager [2], virtual machine [3], etc. Every specification-level policy hence needs to be mapped to a set of implementation-level policies, usually one per layer of abstraction. This is because the data that is to be protected comes in different representations: as file, as window content, object attributes, etc. Eventually, all these representations boil down to some representation in memory, but it is often more convenient and simpler to perform protection at higher levels of abstraction [4].

Usage control policies can in general be enforced in two ways. *Detective enforcement* aims at detecting violations of a policy. In case of a violation, usually a compensating, correcting, or notifying action is taken. In contrast, *preventive enforcement* aims at avoiding policy violations [5]. The subject of this paper is the derivation of implementation-level policies from abstract specification-level policies for preventive enforcement of usage control requirements. There are two major challenges in enforcement: firstly, to keep track of all representations of the same data at different layers of abstractions. For instance, if a picture downloaded from a web-based social network site (WBSN) is to be protected, then the window content, the cache file, and the browser-internal representation, as well as their copies, need to be protected. The second challenge is to translate the policy specified for abstract data with abstract actions into implementation-specific policies at different layers of abstraction.

The problem of addressing different representations of the same data in the system without explicitly listing them has been tackled elsewhere [6]. For enforcement of usage control policies, a set of layer-specific enforcement and data flow tracking mechanisms is configured by a data-oriented policy language. The respective data flow tracking model keeps track of the connections between various representations of data across different levels of abstraction. In this line of research, the authors expect the policies to exist and have not addressed the problem of deriving them. In this paper, we tackle the derivation of implementation-level policies.

One major issue in the derivation of policies is that there is not one single "correct" semantics for actions like "copy" or "delete." To address this problem, two questions need to be answered: the first one concerns the domain-specific semantics of actions, that is, how an action on a data affects other associated data in the domain. For example, "does deleting a profile in a social network mean deleting all posts and links from other profiles also?" The second question is about the technical semantics of actions. That is, how is an action at the abstract level translated into technical events that are eventually performed, e.g., "does copying a picture mean taking screenshots also or only coping the corresponding jpeg files? Does deleting a file mean throwing away the key or deleting the FAT entry or randomly overwriting the hard disk's sectors several times?" Finding answers to both sets of questions is fundamental to the derivation of policies. This is because, in the absence of clear semantics of basic operators (viz. copy and delete), it is impossible to define policies for basic privacy requirements regarding dissemination, retention or deletion of data, that leave no room for misinterpretation. In general, it is not possible to directly intercept/check/detect/monitor if a copying or deletion event has happened or is currently happening.

The other challenge in the derivation of policies is methodological. Although there exist many methodologies for recording, analyzing and understanding user requirements, we do not know of a well-defined framework that facilitates the translation of these requirements into implementation-level policies in a systematic way. Typically, this translation is a manual exercise which can result in implementations that, at different levels of abstraction, permit events that should not be allowed to happen and/or forbid events that should be allowed. Also, in the absence of a well-defined translation methodology, this tedious and lengthy process cannot be automated. Similar problems arise whenever user requirements are transformed into system requirements.

In this paper, we address these issues in the context of usage control policies. We provide a framework for precisely defining the semantics of the actions specified in domain-specific policies that, at the same time, takes into account the underlying technology. We also present methodological guidance in order to automate the translation of policies. The complete process requires human intervention in two roles: the first one is an "end user" who is expected to be well-versed in the domain-specific terminology but is oblivious to the corresponding technical details, which are taken care of by the more sophisticated "power user." For this, we present a meta-model of the data-processing system that reflects the specification and translation of policy elements from abstract to technical levels. We do this in three steps: (1) define a domain model to describe obligations on future data usage; (2) map high-level domain-specific abstractions to low-level technical representations; (3) define enforcement mechanisms for every technical representation.

End users define policies using high level domain-specific terminology. The mapping from high level domain-specific abstractions to low-level technical representations and the definition of enforcement mechanisms [7,8] is provided by the second role, the power user. Policies are then automatically translated to rules that are sufficiently technical to configure enforcement mechanisms.

One example use is from the social network domain: Alice wants her friends to be able to view the pictures she posts in her profile. In contrast, they should not be able to make local copies of those pictures on their machines. Alice can specify the policy "never copy picture" using one of the policy templates described later in this paper, *without knowledge of any technical policy specification language*. By means of the domain model and the mappings provided by the power user, this policy is mapped to more technical policies at various levels of abstraction. The power user also decides on the mode of enforcement (inhibition, modification, execution). Finally, sets of implementation-specific executable rules are generated, one per layer of abstraction. The enforcement of the policy itself is tackled elsewhere [9] and is outside the scope of this paper.

One crucial assumption in this paper is the static system structure. In deriving implementation-level policies from specification-level usage control policies, we have deliberately not considered the adaptive nature of systems. Though unrealistic, this assumption is reasonable in order to simplify the problem and achieve initial results.

Problem. In sum, we tackle two problems in this paper. The first concerns the lack of semantics of events in usage control policies. The second concerns the problem of transforming abstract policies to their technical counterparts in an automated manner. In the larger context, through this work we try to fill the gap between user and system requirements for domain-specific applications.

Solution. We present a framework and a tool for defining the semantics of events that takes into account the different representations of data and its potential flow through a technical system. Our solution comprises both the technical and the methodological aspects of deriving technical policies from more abstract policies.

Contribution. We are not aware of methodologies for translating specification-level usage or access control policies into implementation-level policies that configure usage or access control mechanisms at different abstraction levels.

Organization. §2 puts our work in context. §3 presents the domain meta-model and, as an example, a WBSN instantiation of it. §4 describes the translation of policies. §5 describes the tool we implemented for automating policy translations. §6 backs our proposed framework through several examples. §7 concludes by discussing limitations, further refinements and future work. Appendix A describes the translation of future-time specification-level obligational formulas to the past-time conditions of implementation-level policies.

2. RELATED WORK

The subject of this paper is the derivation of implementation-level policies from specification-level policies using a domain meta-model that is instantiated to define the technical semantics of domain-specific abstractions in high-level usage control policies. The general problem addressed is the translation of user requirements into system requirements.

Often, user requirements are not adequately translated into system requirements which results in systems with functionalities that are not desired while missing out on the desired ones [10]. The gap between these two types of requirements has been a major problem in the domain of requirement engineering [11–13] and although a lot of work has been done on understanding, eliciting, analyzing user requirements [14] and translating them into system requirements for different application domains [15, 16], it remains one of the prominent topics in software engineering research [10, 17].

In the context of security, modeling vulnerabilities, failures and countermeasures [18–21], security requirements and their potential conflicts with other functional and non-functional requirements, application of standards [22, 23], access control policies and requirements [24,25] and policy description languages [7, 26–28] have been mainly addressed.

In terms of derivation/refinement of policies from the business to the technology level, the focus has been on the derivation of role-based access control policies. In this line of work, Yee and Korba have presented two approaches for semi-automatic derivation, one relies on third-party surveys of user perceptions of data privacy and the other on retrieval from a community of peers [29]; Young and Anton [30] have proposed a commitment (obligations) analysis methodology to derive software requirements from privacy policies; Su et al. have proposed a policy refinement methodology based on resource hierarchies [31]; Bandara et al. have presented a methodology based on system goals [32]; Guerrero et al. have proposed an ontology-based approach [33]; and Udupi et al. have presented an automated, domain independent approach based on data classification [34]. Lodderstedt et al. specify RBAC policies in UML and translate them to code or deployment files [35] and do not consider the semantics of atomic actions or resources. Aziz et al. [36] have proposed a resource hierarchy meta-model for translating domain-specific elements in XACML policies at the level of virtual organizations to generate corresponding XACML 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. Besides, the policy refinement literature concerns access control privacy policies only. To the best of our knowledge, the translation of usage control policies has not been investigated.

Usage control, which extends the concept of data protection beyond access control [37, 38], puts constraints upon further usage of data after access is granted. Usage control enforcement, for various policy languages [7, 26–28, 39], has been done at different layers of abstraction [1–4] in the system. However, in all these enforcement cases, policies are supposed to exist and their derivation from end user requirements to enforcement mechanisms has not been addressed.

Though all the work cited above makes some kind of distinction between high-level abstract and low-level concrete data and actions, they have not defined precise frameworks that can guide the derivation from top to bottom. In this context, many architecture frameworks have been proposed that differentiate among business, data, application, and technical infrastructure layers in enterprise settings; the prominent ones being the Zachman framework [40] and the Open Group Architecture Framework (TOGAF) [41]. Of these, the Zachman framework focuses on the classification of various architectural artifacts. The three initial perspectives of “what”, “how” and “where” for information systems architecture [42] are part of our domain meta-model at the second and third layers.

3. THE FRAMEWORK

Specification-level policies describe *what* must and must not happen throughout the execution of a system. Implementation-level policies refine them by stating *how* they will actually be enforced. This paper addresses the problem of automatically deriving the latter from the former.

Specification-level policies consist of domain-specific abstractions (that is, data and actions) and constraints. Examples include never-copy-picture, delete-document-after-30-days, etc. We need, firstly, to define the meaning of data and actions. Secondly, because specification-level policies tend to be formulated in terms of the *future* usage of data, we need to transform the constraints into enforceable conditions that are defined on the *past* (as explained elsewhere [7]; otherwise, the system would need to be able to look into the future. For a special subset of properties, it is in fact possible to detect the policy’s violation at the earliest moment in time [43]. However, since we are concerned with *preventive* enforcement, we do not rely on these approaches). To address the first challenge, we present a meta-model where the meaning of high-level abstractions of data and actions is provided by mapping them to their possible technical representations. The second challenge, the translation of constraints, is essentially the problem of deriving past-time rules to configure enforcement mechanisms. Our mechanisms consist of a description of when they are applicable (triggering event and a condition); and the respective actions to be taken in that case. Mechanisms are hence of the event-condition-action format. Different mechanisms may correspond to one obligation at different levels of abstraction.

We address all these issues step-by-step in the next sections. We start by addressing the problem of defining the semantics of data and actions in specification-level policies.

3.1 Data and Containers

In order to enforce usage control policies specified for data, we must be able to differentiate between data and actions, and their technical representations. At the end user’s level, data is an abstract concept with an intuitive set of actions relevant to it. The end user can therefore specify usage control policies only in those abstract terms. At the system level, data comes in different concrete representations called the containers (files, pixmaps, memory regions, network packets etc.). The mappings between data and containers tell which set of data is potentially stored in which container at runtime [6]. Extending this concept of distinction between data and its representation, we present a meta-model of the domain that includes the distinction among user-intelligible high-level actions on data like “delete profile” and “copy picture” (layer 1), corresponding technical events on containers (which we call transformers) (layer 2), and the specific implementations of these transformers and containers (layer 3). Mappings between various components at different layers in the meta-model provide the semantics of high level actions on data in terms of a set of corresponding technical events at various levels of abstraction.

3.2 The Domain Meta-Model

Our domain meta-model is shown in Figure 1. It consists of three layers: the platform-independent layer that is to be defined and used by the end user; the platform-specific layer that is to be defined and used by the power user; and the implementation-specific layer, also to be defined and used by the power user, that reflects the technicalities of the system on which the policies are to be deployed.

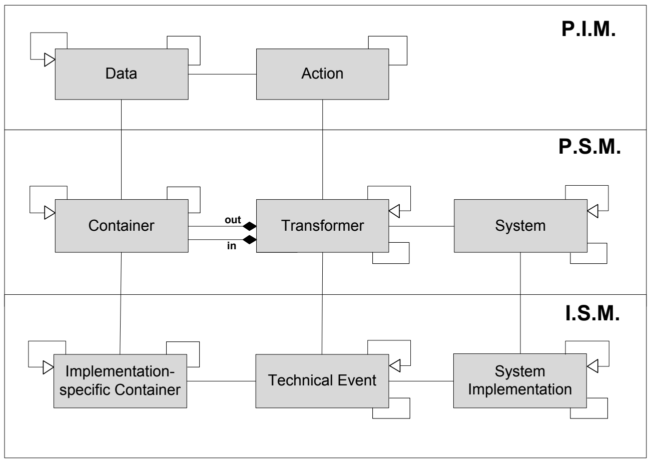


Figure 1: Domain meta-model

The platform-independent (PIM) layer corresponds to the abstract *data* and the set of intuitive *actions*. Associations and generalization-specialization relationships among data define the abstract data model of the application domain and capture the scenario when in a domain, certain data are seen as aggregation of or associated with (set of) other data or as generalization/specializations of other data. For example, a profile is associated with all blogposts that a specific user posts in a blog-domain, an album is an aggregation of all pictures posted by a specific user in a WBSN-domain etc. The specification of such relationships can also help define semantics of actions on one data in terms of a set of actions on other data. For example, deleting the profile means also deleting all associated posts, or denying copies of the album means that no picture in the album can be copied irrespective of the difference in the technical representations of the two. Generalization-specialization relationships among data at the PIM layer also define the semantics of an action on one data in terms of actions on another data at the platform-specific layer. For example, if a blog account is a *specialization of* a general account, then the technical semantics of actions like copy and delete for a blog account is the same as for the general account.

The platform-specific (PSM) layer corresponds to all possible representations of data, known as *containers*, and the set of *transformers* that read and write data between these containers at different layers of abstraction in the *system*. Transformers can be seen as functions that use sets of (atomic and/or complex) events in corresponding systems to transform the containers when certain actions are performed on data; e.g. screenshot in the windowing system, copy & paste in the web browser, copy file in the file system, etc. Containers, transformers and systems can be associated with and/or generalize sets of other containers, transformers and systems. For instance, a database can be seen as a specialization of a file; copy file transformers are associated with a set of transformers for opening, reading, writing and closing a file, and a WBSN system is spread across multiple layers of abstractions over many servers and client machines.

As there is more than one way of implementing different systems, multiple implementations of containers and transformers exist. This is shown in the third layer of our meta-model, the implementation-specific layer (ISM). Mappings between data and containers, and, actions and transformers specify the semantics of actions on data in terms of sets of transformers applied to sets of containers; e.g. semantics of “copy picture” in the context of a WBSN are given by screenshot in the windowing system, copy & paste in the web browser, copyfile in the operating system. Mapping containers and transformers to their implementations specifies the semantics of action on data in terms of low-level technical implementations; e.g. the semantics of “copy picture” is given by “getImage on a drawable object” in the X11 windowing system, “cmd_copy on an html image object” in the Mozilla Firefox web browser, and “sequence of open, read, write and close system calls on file” in a Unix operating system. For illustration, we instantiate our meta-model for the case of a WBSN in the next subsection.

3.3 A Social Network Example

For space reasons, only parts of the complete domain model are shown in Figure 2. The classes of the meta model (data, container) that are instantiated by a class of the WBSN domain model are indicated as stereotypes (« ») on top of the class name (WBSN data, WBSN container).

The PIM layer domain model contains copy, delete etc. actions for the classes of payload and traffic data. Profile information, pictures, posts, comments and other data posted by end users is payload data; all data generated by browsing activities of end users, e.g. links clicked, profiles visited etc. are traffic data. A detailed data model of WBSN is given in [44].

WBSN Containers are specialized into web browser container, windowing system container, operating system containers etc. which are further specialized as DOM element, window, file and other containers. Copy&paste, screenshot, CopyFile and other transformers transfer data between these containers. The system is a generalization of logical systems (i.e. layers of abstraction like web browser, windowing system, operating system etc.), physical hosts/locations (e.g. caches, backups and other servers, client machines etc.) and applications that span over both logical and physical systems (e.g. social network, mail systems, ERP systems etc.). These systems and their implementations evolve over time. However, in this paper we treat them as static. We reiterate that we have done this deliberately - in order to simplify the problem. The dynamic nature of distributed systems is part of our ongoing work.

At the ISM layer, specific implementations include a Firefox web browser, the X11 windowing system, an OpenBSD operating system with corresponding containers and transformers. For the Facebook implementation of social network, various logical systems shown at the ISM layer are deployed at hundreds of physical locations.

The definition of the domain models and the mappings provides the semantics of data (in terms of its structure and the containers that contain the data) and actions (in terms of technical events) in specification-level policies. The next step is the translation of constraints. We begin with an overview of the complete process of specifying and deriving the policies, followed by the details.

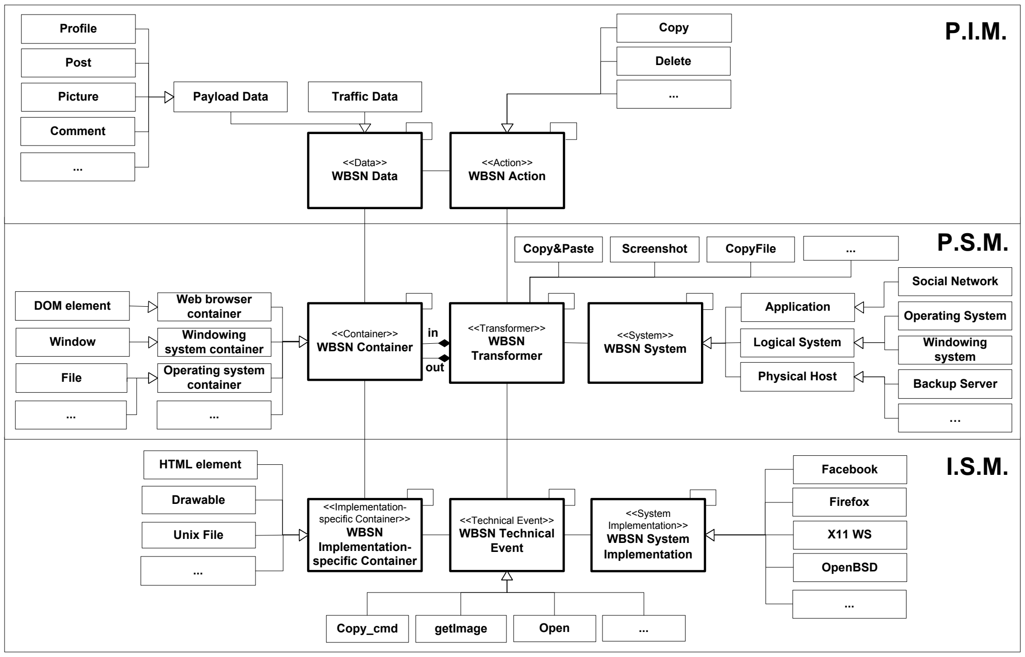


Figure 2: WBSN Domain Model, an instance of Fig. 1

3.4 Overview of Policy Specification and Translation

Figure 3 shows the sequences of steps for the systematic specification and translation of specification-level policies to implementation-level policies. The first flow chart describes the tasks performed by the power user for the definition of the domain model and the policy templates and the translation of policies specified using these templates. The second flow chart describes the end user tasks for specification of policies (and also their translation, in case he opts not to use the templates). In the complete process, end user tasks start only after all the tasks are completed by the power user. The initial steps are about defining the three domain mod-

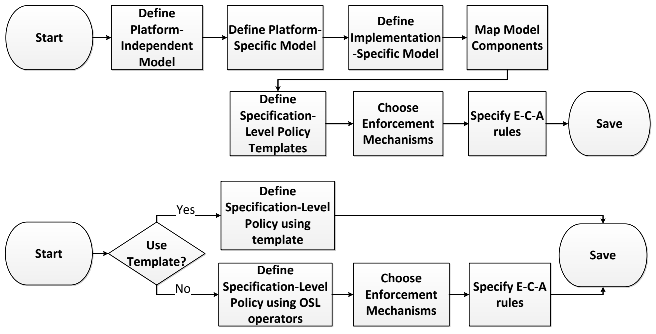

Start
Define Platform-Independent Model
Define Platform-Specific Model
Define Implementation-Specific Model
Map Model Components
Define Specification-Level Policy Templates
Choose Enforcement Mechanisms
Specify E-C-A rules
Save
Start
Use Template?
Yes
No
Define Specification-Level Policy using template
Define Specification-Level Policy using OSL operators
Choose Enforcement Mechanisms
Specify E-C-A rules
Save

Figure 3: Policies Specification and Translation

els because this provides the vocabulary for specifying PIM policies and their translations to the ISM layer: string representations of data, actions and their technical implementations are used in specification and implementation-level policies. The platform-independent domain model is defined in the ***first step***. In the ***second*** and the ***third steps***, the platform and implementation -specific models of the application are defined. In the ***fourth step***, high level components from the PIM level are mapped to PSM level technical representations (§5). As the implementation-specific model is a refinement of the platform-specific model, the definition of ISM components includes the mapping to PSM-level counterparts and is not a separate step in the process. In the ***fifth step***, when the power user creates specification-level policy templates (§5), he is also required to define the further translation of the resulting policies. In sum, when the PIM policy (or template) is saved, multiple XML representations of the PIM policy are generated, each corresponding to one specific layer of abstraction in the ISM model and containing implementation-specific representations of the abstract data and action specified in the PIM policy. In the ***sixth step***, enforcement strategies corresponding to every XML PIM-policy are chosen. Finally, executable event-condition-action rules are defined in the ***seventh step***. Usually, the power user defines classes of policies as templates. The end user specifies PIM policies using the available templates. As their translation is already defined by the power user, the policies are automatically translated when the end user saves them. The end user can also specify PIM policies without templates, using standard OSL operators. In that case, he must decide upon the enforcement mechanisms and define the corresponding ECA rules.

In a nutshell, the meta-model from Figure 1 is instantiated to three-layered domain models and the mappings between the layers for the application context (as in Figure 2). In the subsequent steps, the specification-level policy, the enforcement strategies and the event-condition-action rules for enforcement are defined; they can be generated for template classes of policies. In the next section we go into the details of policy specification and translation.

4. POLICY SPECIFICATION AND TRANSLATION

Specification-level policies stipulate what must or must not happen to data in the future. Hence they contain obligations in future time temporal logic. In our framework, we specify obligations in OSL, a policy specification language that combines the classical propositional operators with future time temporal and cardinality operators such as $\underline{until}$, $\underline{always}$, $\underline{after}$, $\underline{within}$, $\underline{during}$, $\underline{repmax}$ etc. Intuitively, $\underline{until}(a, b)$ is true if a is true until b eventually becomes true or b never becomes true; the $\underline{always}$ operator is intuitive, $\underline{after}(n, a)$ is true if a becomes true after n time steps; $\underline{within}$ and $\underline{during}$ are intuitive. The cardinality operator $\underline{repmax}(n, a)$ specifies that a must be true at most n times in the future. For space reasons, we do not describe all OSL operators here. For the complete syntax and semantics, see [6, 7].

Usually, a high-level action in a policy is expressed in terms of a sequence of corresponding low level events. In certain cases, this might not be the best approach. Let us for instance consider the policy of denying picture copies at the operating system level; this means that a sequence of system calls that result in a copy of the picture are denied. But infinitely many sequences of system calls can achieve the same result and coming up with a list of them has a high chance that we miss out a few of them. Alternatively, if all such sequences might start with a particular system call (e.g. in any case, a file must be opened to be copied), blocking this particular system call (the first in the sequence) might seem a solution. The problem with this approach is that this particular system call starts many other sequences, some being required features of the system. Blocking any other "main" system call of the sequence requires the sequence to be precisely identified and enumerated. Thus it turns out that expressing the semantics of high level action in terms of a sequence of low level events might not be the best approach in some cases. Especially when actions of the form copy and delete are concerned, the user wants to make sure that specific situations must or must not occur in the system, that is, the system must or must not be in specific states. For this, state-based policies have been introduced in [6]. State-based policies refine actions in policies in terms of states that the system must or must not enter. As the mappings between data and containers define the states of a system, this means that the data must be restricted to or must never enter specific containers. For this, operators like $\underline{isNotIn}$, $\underline{isCombinedWith}$ and $\underline{isOnlyIn}$ are used. Thus specification level policies can be enforced both as event-based policies and state-based policies. In Section 6 we show example policies for both cases.

Figure 4 shows how the meta-model fits into the big picture of usage control policies and how components from one relate to the other. An OSL policy consists of a set of obligational *formulas*. An obligational formula consists of a logical expression which is a set of state-based and event-based operators. Complex policies contain nested formulas; hence, formulas can also be operands in a policy. The other operands include action with data as its parameter, both of which are string representations of the classes in the PIM domain model. End user policies are specified using templates which contain these string representations, hence they apply to classes and not specific instances of data. Note that

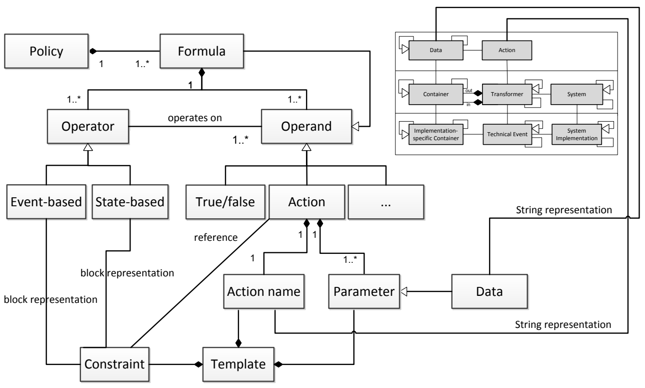


Figure 4: The big picture

graphical templates are defined in order to hide OSL from the end user; end users may, but are not expected to express policies in OSL. So the templates are designed to consist of data, action and constraints as representations of OSL operators. Constraints can also be expressed in terms of actions that trigger the policy; hence actions are shown to be referenced in constraints (in concrete syntax these are XPath constructs containing reference to trigger event).

The specific kind of preventive enforcement (inhibition, modification, execution) needs to be prescribed by the power user as one specification-level policy can be enforced in many ways. For instance, let us consider the requirement "picture must not be copied without notification" in a WBSN domain. At the abstract level, preventive enforcement by *inhibition* of this requirement means that all actions of the type copy are blocked; preventive enforcement by *modification* means that the picture is allowed to be copied but the resulting image at the destination is for instance, replaced by another picture (for instance, showing an error); and preventive enforcement by *execution* means that the picture is allowed to be copied but the owner is notified about the action. Abstractly, these enforcement mechanisms are parameterized event-condition-action (ECA) rules where the rule is triggered provided that the *Event* takes place and the *Condition* evaluates to true. The *Action* specifies how precisely inhibition, delay, modification, or execution take place [8]. For instance, let us consider again the above requirement. To show the ECA rules corresponding to different enforcement strategies, we use these propositions with picture as p: *attemptCopy(p)* denotes that an attempt to copy picture has been made, *copy(p)* denotes that a picture is copied, *notifyAdmin(p)* states that a notification about the picture should be sent to the administrator, *adminNotified(p)* means that the administrator has already been notified about the picture.

Enforcement by execution means that whenever the picture is copied, a notification is sent to the administrator.

```
Event: copy(p)
Condition: true
Action: EXECUTE notifyAdmin(p)
```

Enforcement by modification means that whenever there is an attempt to copy the picture and a notification has not been sent, the picture is replaced by another predefined picture "error.jpg"(using function *replaceByError*(p)).

```
Event: attemptCopy(p)
Condition: not(adminNotified(p))
Action: MODIFY replaceByError(p)
```

Enforcement by inhibition means that whenever there is an attempt to copy the picture and a notification has not been sent, the attempt is blocked.

```
Event: attemptCopy(p)
Condition: not(adminNotified(p))
Action: INHIBIT
```

Implementation-level policies are encoded in XML with an Event declaration part, a Condition part, and an Action part (ECA pattern). The event declaration part is the trigger of the implementation-level policy, and represents the request for the execution of an action. The trigger event can be underspecified, that is, all the parameters need not be declared here. It is possible to reference the trigger event of implementation-level policies in the conditions and action parts using XPath expressions. When a trigger event matching the event declaration part of an implementation-level policy is received, and the condition part evaluates to true, the action part of the mechanism is evaluated. Our language allows the specification of implementation-level policies that inhibit/prevent the intended action and might also execute arbitrary actions before the requested action is executed. In case the action allow is specified, it is possible to specify modifications in the parameters values or to delay the execution of the action.

Deciding about the enforcement strategy and the translation of future time obligations to their past forms are the two basic requirements for deriving implementation-level policies from specification-level policies. We present the translation of future time obligations to past time conditions in Appendix A. In the next section we give an overview of the automation of the translation and the implemented tools.

5. AUTOMATION OF POLICY SPECIFICATION AND TRANSLATION

In Section 3.4, we have introduced the methodology for a systematic derivation of implementation-level policies from specification-level policies. The first four tasks of the power user define the semantics of domain specific abstractions in technical terms. The last three tasks describe the translation of an OSL policy into ECA rules. Based on this, policies are translated from the PIM to ISM layer in the meta-model when the end user specifies them using templates. If the end user specifies policies without predefined templates, the translation is semi-automatic. The complete process cannot be fully automated in this case because of the following reasons:

1. Definition of enforcement mechanisms needs human intervention for at least every class of specification-level policy, as one specification-level policy can be enforced in many different ways.

2. Complex obligations may need to be decomposed into subconditions each of which is then mapped to the *condition* part of a separate ECA rule. Decomposition may be required to get rid of nested temporal formulas, whose translation cannot be automatically achieved. A trivial (and fully automatic) solution of this problem is to turn each subformula of the specification-level policy into one ECA rule.

We have implemented two editors, to be used by two roles of users (the end user and the power user), for specification and translation of policies according to the meta-model described in Section 3. In this section, we describe these tools through examples from the domain of WBSN applications. However, *the tools are generic and can be used for any domain-specific application* (in the next section we also show some examples for Android-based smart phone applications) as all domain data is stored separately in databases and XML files and the domain can be changed by modifying the contents of them.

At the PIM level, end users are not supposed to be adept in any technical policy specification language. Yet, they should be able to specify usage control policies. We address this problem by means of our *end user tool* which enables the user to specify PIM policies by editing templates in a drag and drop interface. *Templates* consist of constraints that abstract the syntax of the different OSL operators. Additionally, they contain placeholders for string representations of data and actions. Thus in a template, a policy is made up of data, action and constraint blocks. Constraints are represented by rounded rectangles, actions by rectangles and data by double ellipses. The solid arrow is from the operator to the operand and the dotted arrow is from action to its object parameter value. The specific actions and data are specified by the end user by dragging and dropping these values from the palette. Additional parameter values for actions (sender, receiver, location, purpose, currency etc.) are specified by choosing the option from the right-click context menu. To begin with, we have specified five templates for describing usage control policies, as most of the policies relevant for end users of WBSN applications can be specified through them. Creating new templates is not difficult and can be done later also.

The five templates can specify policies of the form "**never do X;**" "**no X until Y;**" "**X always implies Y;**" "**do X within N duration;**" "**X can be repeated maximum N times.**" Figure 5 shows an example template for a policy of the form "**never do X**" on the left and an example policy specified using that template on its right.

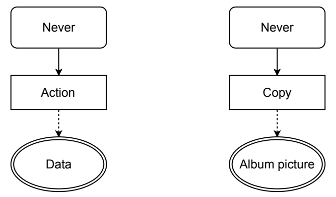


Figure 5: Pictures in albums must not be copied

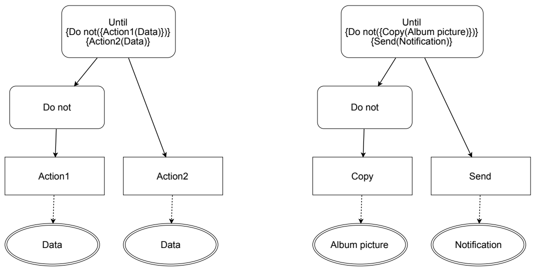
It appears more convenient for an end user to state "album picture must not be copied" using the template than to formally express it in OSL as the obligational formula:

$$\underline{always}(\underline{not}(E_{fst}(Copy, \{(object, albumpicture)\}))).$$

Thus, templates enable less sophisticated end users to specify usage control policies without the knowledge of any policy specification language. We strengthen this argument by help of further examples of templates and corresponding policies.

Figure 6 shows the graphical representation of the policy mentioned earlier in Section 4, "picture must not be copied without notification" which is of the form "**no X until Y**" and can be formally expressed in OSL as the obligation

$$\underline{until}(\underline{not}(E_{fst}(Copy, \{(object, Albumpicture)\})), E_{fst}(Send, \{(object, Notification)\}))$$



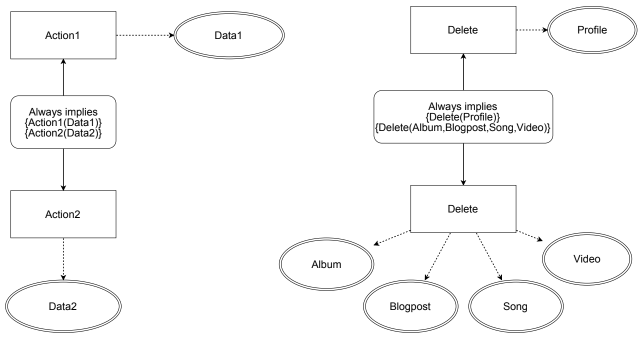
Until {Do not({Action1(Data)})} {Action2(Data)}
Do not
Action1
Action2
Data
Data
Until {Do not({Copy(Album picture)})} {Send(Notification)}
Do not
Copy
Send
Album picture
Notification

Figure 6: No copy without notification

Figure 7 shows the graphical representation of the obligation

$$\underline{always}(\underline{implies}(E_{fst}(Delete, \{(object, Profile)\}), E_{fst}(Delete, \{(object, \{Album, Blogpost, Song, Video\})\})))$$

which is of the form "**X always implies Y**" and in general defines the meaning of action on one data in terms of same or other action(s) on (other) data; the specific example states that deleting a profile also means deleting all albums, blogposts, songs and videos posted in the profile.



Action1
Data1
Always implies {Action1(Data1)} {Action2(Data2)}
Action2
Data2
Delete
Profile
Always implies {Delete(Profile)} {Delete(Album,Blogpost,Song,Video)}
Delete
Album
Video
Blogpost
Song

Figure 7: Deleting profile means deleting all other data

The simplification of specifying OSL policies is obvious in the above examples. If the end user is knowledgable about OSL, he can also write these policies using our editor. In that case, further definition of the enforcement mechanisms is also taken care of by the end user.

The *power user tool* is used to define the abstract (PIM) and concrete (PSM & ISM) domain models and the respective mappings. It provides two sets of functionalities: one for the definition of different domain models and the other for mapping the elements across models. Domain-specific models for PIM, PSM and ISM layers can be defined, modified and deleted using this tool and their components are mapped to each other by links; for instance, by connecting "picture" from the PIM layer to "file, DOM element, window content" from the PSM layer. The abstract model provides domain-specific vocabulary for specifying PIM policies and the mappings achieve the automatic translation of abstract policy components to their concrete counterparts in the specified policies. Syntactical transformation from OSL to ECA rules is specified by the power user using predefined templates .

In the next section we present some examples of implementation-level policies derived from specification-level policies. These policies have been defined and shown to be enforced in [4, 6, 9] and other related work where they are assumed to exist and have not been derived from user requirements. We show their derivation here as a proof of concept of our work.

6. DERIVATION OF POLICIES BY EXAMPLES

We start with the example policy from the use case in Section 1, "picture must not be copied" whose graphical specification is shown in Figure 5. Figure 2 shows the semantics of data and action of this policy, where the concrete model of the WBSN domain includes three layers of abstraction in client machines: the web browser, the windowing system and the operating system. Implementation-level policies for the three levels of abstraction are shown below.

```
<!-- For Firefox Web Browser -->
<controlMechanism>
 <id>Browser_CopyPaste</id>
 <triggerEvent>
  <id>copy</id>
   <parameter name="obj" value="img_profile" type="dataUsage"/>
   <parameter name="isTry" value="true"/>
 </triggerEvent>
 <condition> <true /> </condition>
 <actions> <inhibit/> </actions>
</controlMechanism>

<!-- For X11 Windowing System -->
<controlMechanism>
 <id>X11_Screenshot</id>
 <triggerEvent>
  <id>GetImage</id>
   <parameter name="obj" value="0x1a00005" type="dataUsage"/>
   <parameter name="isTry" value="true"/>
 </triggerEvent>
 <condition> <true /> </condition>
 <actions>
  <allow>
    <modify>
     <parameter name="planeMask" value="0x0" />
    </modify>
  </allow>
 </actions>
</controlMechanism>

<!-- For Windows Operating System -->
<controlMechanism>
 <id>OS_Restrict_File_Usage</id>
 <triggerEvent>
  <id>open</id>
   <parameter name="obj" value="cacheFile" type="dataUsage"/>
   <parameter name="isTry" value="true"/>
 </triggerEvent>
 <condition>
  <XPathEval>
   /triggerEvent/parameter[@name='PNAME']/@value!='c:\\Firefox\\firefox.exe
        '
  </XPathEval>
 </condition>
 <actions> <inhibit/> </actions>
</controlMechanism>
```

The policy is enforced at the web browser and the operating system levels by inhibition (we make an exception to our simple inhibition rule at the OS level by allowing the web browser (specifically, Mozilla Firefox) to open the picture so that the end user can (only) view the picture in a web browser) and at the windowing system level by modification. Here, the choice of the enforcement strategy is implementation-driven. For the X11 windowing system, inhibiting screenshots altogether (sending an empty response) results in an error message from the server. So it is better to enforce the policy as modifying the trigger event instead of inhibiting it: we modify the request for screenshot on a drawable by changing the *planeMask* value from 0xffff (meaning every plane is included) to 0x0 (no plane included) resulting in a black rectangle at the destination.

We have already discussed the problem and the solution of expressing high-level actions like copy and delete in terms of sequences of low level events (Section 4). So in the next example, we show a high-level policy translated into a state-based low level policy. In the use case from the WBSN, the end user wants that his friends should be able to play his songs, make local copies, but they should not be able to distribute it further. The abstract policy is, "song must not be distributed." At the specification level, this requirement is expressed using the policy template of Figure 5 with action as distribute and data as song. Figure 8 shows an instantiation of the domain meta-model for the corresponding concrete representations at the OS-level.

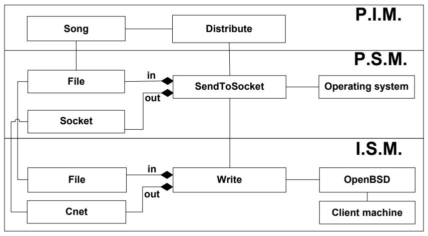


Figure 8: Semantics of "distribute song"

At the OS level, distributing data over the network is interpreted by the power user as writing data in a specific container to the socket descriptor. "song.mp3" and "Net" are names for containers at runtime. In particular, "Net" is a reserved name for the container "Cnet" that stands for "the network". The requirement can be formally expressed in OSL as

$\underline{isNotIn}(song.mp3, Net)$

We choose to enforce the policy by inhibition. Hence the mechanism is triggered whenever a write system call sends the song (the data) stored in "song.mp3" over the network, i.e. to a socket descriptor.

```
<controlMechanism>
 <id>OS_Disclosure_example</id>
 <triggerEvent>
  <id>write</id>
   <parameter name="isTry" value="true"/>
 </triggerEvent>
 <condition>
  <not>
   <isNotIn data="song.mp3">
    <containers>
     <container>Net</container>
    </containers>
   </isNotIn>
  </not>
 </condition>
 <actions> <inhibit/> </actions>
</controlMechanism>
```

Our third example is from the domain of smart phone applications for Android phones: pictures taken at certain locations must be blurred when viewed outside the premises. This requires that the concrete representations of pictures taken from the camera must be identified when saved in the system, these representations must be tainted when the picture is taken at the specified location and, when the location of viewing the pictures is not the same, the representations must be temporarily blurred before being displayed. Therefore location is a key parameter of action in this case. At the PIM layer, tainting is captured by "flagging" a picture. Figure 9 shows the specification of two policies in the end user tool.

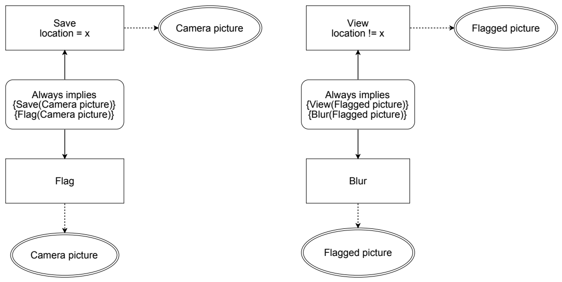


Figure 9: Blur picture when viewed outside the original location

Both the policies are specified using the template shown in Figure 7. The first policy (on left) stipulates that when a camera picture taken at location x is saved, it should be flagged:

$\underline{always}(\underline{implies}(E_{fst}(Save, \{(object, Camerapicture), (location, x)\}),$
$E_{fst}(Flag, \{(object, Camerapicture)\})))$

The second policy stipulates that when there is an attempt to view the flagged picture 'outside' a specific location ('negateLocation' parameter value set to true), the picture must be blurred:

$\underline{always}(\underline{implies}(T_{fst}(View, \{(location, x), (negateLocation, true),$
$(object, Flaggedpicture)\}), E_{fst}(Blur, \{(object, Flaggedpicture)\})))$

with T_{fst} denoting an 'intended' action. Figures 10 and 11 show the semantics of data and action in the two policies using domain models. At the implementation layer, pictures taken from the camera are identified by a special taint mark (0x00080 in our case) and are tainted with another taint mark (0x10000) when they are saved as files at the OS level.

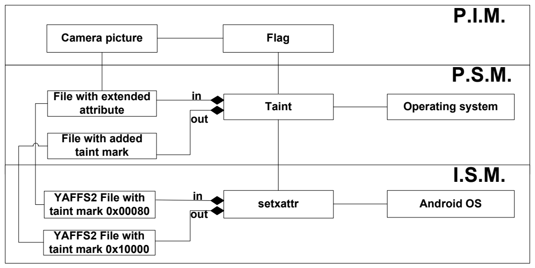

P.I.M.
Camera picture
Flag
P.S.M.
File with extended attribute
in
out
Taint
Operating system
File with added taint mark
I.S.M.
YAFFS2 File with taint mark 0x00080
in
out
setxattr
Android OS
YAFFS2 File with taint mark 0x10000

Figure 10: Semantics of "flag camera picture"

Viewing a flagged picture means reading the file with taint marking 0x10000 at the OS level.

We enforce both policies by modification. At runtime, when a picture taken by camera (identified with the taint mark 0x00080=128) at the location x (location=49.445626, 7.760339; checked in the trigger event part)is written to the

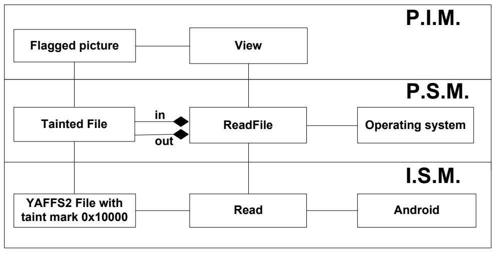


Figure 11: Semantics of "view flagged picture"

file system, the trigger is modified with parameter value 0x10000 (=65536). When there is an attempt to read the file with taint marking 65536 and the location is *not equal to* x (checked by the last parameter of location in trigger event part; when this flag is set to true, the evaluation is negated), the event is blocked.

```
 <!-- Adds taint 0x10000 for pictures taken at given location -->
 <controlMechanism>
  <id>TaintPictures</id>
  <triggerEvent>
   <id>OSFileSystem.write</id>
   <parameter name="taint" value="128" />
   <parameter name="location" value="49.445626;7.760339;50;true;false"/>
  </triggerEvent>
  <condition> <true /> </condition>
  <actions>
   <allow>
    <modify>
     <parameter name="data" value="taint$65536" />
    </modify>
   </allow>
  </actions>
 </controlMechanism>

<!-- When not at the given location, files with taint marking 0x10000 =
      65536 are blurred upon access-->
 <controlMechanism id="BlurPictures">
  <triggerEvent id="OSFileSystem.read" />
   <parameter name="taint" value="65536" />
   <parameter name="location" value="49.445626;7.760339;50;true;true" />
  </trigger>
  <condition> <true /> </condition>
  <actions>
   <allow>
    <modify>
     <parameter name="data" value="blur$5" />
    </modify>
   </allow>
  </actions>
 </controlMechanism>
```

7. CONCLUSIONS AND FUTURE WORK

The problem we address in this paper is the systematic derivation of usage control policies from specification-level to implementation-level taking into account the technical semantics of high level actions like copy and delete for all the different representations of data. Specification-level policies are specified using abstract terminology and hence must be translated into more technical policies that can be enforced at different layers of abstraction in the system. In this direction, our contribution is twofold: on one hand we present a framework that allows us to define the semantics of actions in terms of elements of application-specific domain models and translates the policies from their specification level syntax to event-condition-action (ECA) format; and on the other hand, we provide methodological guidance for the specification and translation of policies so that the complete process, which is cumbersome when manually done, can be automated.

For the precise semantics of actions in usage control policies, we present a domain meta-model that has three layers which correspond to abstract data and actions (PIM layer) and their various representations (PSM layer) and implementations of these representations in the real world (ISM layer) (§3). The idea is to map abstract entities from the specification-level policies to their technical counterparts to define the semantics of actions on data in terms of transformers that write/read to/from the containers in which data is stored. After this, enforcement strategies are decided upon and the specification-level policy, which is expressed in OSL in our framework, is transformed into sets of ECA rules (§4). As a result of translation, a set of implementation level policies is generated for one specification level policy: usually, one implementation-level policy per layer of abstraction. By means of several examples, we have shown the applicability of our methodology (§6).

To automate the process, we have two roles: an end user and a more sophisticated power user. We have implemented two editors to be used by them. In one editor, the power user defines the three domain models and maps the components from these models to each other (§5). The end user then uses the string representations of the classes of the abstract domain model to specify policies in the second editor. The end user specifies policies in an interactive drag and drop user interface using templates, which abstract standard OSL operators and thus make it possible for an end user to describe requirements without knowing any technical policy specification language.

This is however a first step towards the complete solution of deriving implementation-level policies from specification-level policies that configure usage control mechanisms at different levels of abstraction. There is a deliberately introduced limitation of our work - we have not considered the dynamic structure of systems while translating policies. We have assumed that the system architecture is static and has been specified in the very beginning. This means that the sets of all possible containers and the transformers and their implementations are known beforehand. In the first step of our work this assumption is reasonable as it helps us in reducing the complexity of the problem. However, in the real world, systems change over time. Services and applications are added/ removed and physical hosts change location. In that case, many of the mappings between the components of different models might become invalid and new mappings need to be specified. To address this issue, we need a policy management system where manager components take care of registering and de-registering details to handle the dynamic case. This is work in progress and so we do not discuss details of it here.

Another limitation on the solution is posed by the evolution of specification-level policies in a distributed setup where receivers of data transform themselves into senders of data over time, and so the policies that they attach with data might evolve with this change [45]. In our present solution, we have not considered this aspect of the problem.

Finally, a fundamental concern is about the usability of policy-specification tools in general: though we have introduced templates to abstract both technical semantics and syntax of usage control policies, these requirements tend to be complex and end users might not be capable of understanding and specifying them at all. An approach in that case would be to let data protection officers or any other trusted authority specify these policies for particular domains.

8. REFERENCES

[1] 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.

[2] A. Pretschner, M. Buechler, M. Harvan, C. Schaefer, and T. Walter. Usage control enforcement with data flow tracking for x11. In *Proc. 5th Intl. Workshop on Security and Trust Management*, pages 124–137, 2009.

[3] 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.

[4] 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.

[5] D. Povey. Optimistic security: a new access control paradigm. In *Proc. 1999 workshop on New security paradigms*, NSPW '99, pages 40–45. ACM, 2000.

[6] A. Pretschner, E. Lovat, and M. Buechler. Representation-independent data usage control. In *Proc. 6th Intl. Workshop on Data Privacy Management*, 2011.

[7] 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.

[8] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for Usage Control. In *Proc. ACM Symp. on Information, Computer & Communication Security*, pages 240–245, 2008.

[9] E. Lovat and A. Pretschner. Data-centric multi-layer usage control enforcement: A social network example. In *Proc. ACM Symp. on Access Control Models and Technologies*, 2011.

[10] J. Beatty and J. Hulgan. Experiences with a requirements object model. Lecture Notes in Comput. Sci., pages 104–117. Springer Berlin / Heidelberg, 2009.

[11] M.E.C. Hull, K. Jackson, and J. Dick. *Requirements Engineering, 2nd Ed.* Springer, 2005.

[12] G. Kotonya and I. Sommerville. *Requirements engineering: processes and techniques.* Worldwide series in computer science. 1998.

[13] I. Bray. *An Introduction to Requirements Engineering.* Addison Wesley, aug 2002.

[14] M. Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices.* ACM Press/Addison-Wesley, New York, USA, 1995.

[15] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications - extended abstract. In *Proc. 4th Intl. Conf. on Requirements Engineering, 2000*, 2000.

[16] K.L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. on Software Engg.*, SE-6(1):2 – 13, jan. 1980.

[17] B.H.C. Cheng and J.M. Atlee. Research directions in requirements engineering. *Future of Software Engineering*, pages 285–303, 2007.

[18] J. McDermott and C. Fox. Using abuse case models for security requirements analysis. ACSAC '99, 1999.

[19] Bruce Schneier. Attack trees, 1999.

[20] G. Sindre and A.L. Opdahl. Eliciting security requirements by misuse cases. In *Proc. 37th Intl Conf. on Technology of Object-Oriented Languages and Systems*, pages 120 –131, 2000.

[21] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook.* U.S. Nuclear Regulatory Commission, 1981.

[22] G. Elahi. Security requirements engineering: State of the art and practice and challenges, 2008.

[23] J. Wilander and J. Gustavsson. Security requirements - a field study of current practice. In *E-Proc. Symposium on Requirements Engineering for Information Security*, 2005.

[24] Q. He and A.I. Antón. Requirements-based access control analysis and policy specification. *Information & Software Technology*, 51:993–1009, June 2009.

[25] G. Neumann and M. Strembeck. A scenario-driven role engineering process for functional rbac roles. In *Proc. 7th ACM symp. on Access control models and technologies*, SACMAT '02, pages 33–42, 2002.

[26] R. Iannella (ed.). Open Digital Rights Language v1.1, 2008. `http://odrl.net/1.1/ODRL-11.pdf`.

[27] Multimedia framework (MPEG-21) – Part 5: Rights Expression Language, 2004. ISO/IEC standard 21000-5:2004.

[28] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proc. Workshop on Policies for Distributed Systems and Networks*, pages 18–39, 1995.

[29] G. Yee and L. Korba. Semiautomatic derivation and use of personal privacy policies in e-business. *IJEBR*, 1(1):54–69, 2005.

[30] J. Young. Commitment analysis to operationalize software requirements from privacy policies. *Requirements Engineering*, 16:33–46, 2011.

[31] 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.

[32] 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.

[33] 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.

[34] Y.B. Udupi, A. Sahai, and S. Singhal. A classification-based approach to policy refinement. In *Proc. 10th IFIP/IEEE Intl Symp. on Integrated Network Management*, 2007.

[35] T. Lodderstedt, D. Basin, and J. Doser. Secureuml: A uml-based modeling language for model-driven security. In *UML*, pages 426–441, 2002.

[36] B. Aziz, A.E. Arenas, and M. Wilson. Model-based refinement of security policies in collaborative virtual organisations. ESSoS, pages 1–14, 2011.

[37] A. Pretschner, M. Hilty, and D. Basin. Distributed usage control. *Commun. ACM*, 49(9):39–44, 2006.

[38] J. Park and R. Sandhu. The UCON ABC usage control model. *ACM Trans. on Information and System Security*, 7(1):128–174, 2004.

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

[40] C. O'Rourke, N. Fishman, and W. Selkow. *Enterprise architecture using the Zachman Framework.* Course Technology, 2003.

[41] The Open Group. TOGAF Version 9. The Open Group Architecture Framework. 2009.

[42] J. A. Zachman. A framework for information systems architecture. *IBM Syst. J.*, 26:276–292, September 1987.

[43] O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[44] P. Kumari. Requirements analysis for privacy in social networks. In *Proc. 8th Intl. Workshop for Technical, Economic and Legal Aspects of Business Models for Virtual Goods*, 2010.

[45] A. Pretschner, F. Schütz, C. Schaefer, and T. Walter. Policy evolution in distributed usage control. *Electr. Notes Theor. Comput. Sci.*, 244:109–123, 2009.

APPENDIX

A. FUTURE TO PAST TRANSLATION

As mentioned in the start of Section 3, in order to be evaluated in the condition part of an ECA rule, every obligation expressed in future time OSL formula must be translated to its past form. Specification-level usage control obligations are described in language Φ^+ (+ for future). It is a temporal logic with explicit operators for cardinality and permissions. We distinguish between purely propositional (Ψ) and temporal and cardinality operators (Φ^+) [6]. The intuitive semantics of these opeartors are given in Section 4.

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

Mechanisms, or ECA rules, are specified in a past temporal logic Φ^-.

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

Informal semantics of these operators are as follows: $\underline{since}^-(a,b)$ is true if a has been true ever since b happened; $\underline{before}^-(n,a)$ is true if a was true n time steps ago; $\underline{within}^-$, $\underline{during}^-$ and $\underline{always}^-$ are intuitive. The cardinal operator $\underline{repmax}^-(n,a)$ 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 a has been true at most n times since b became true. For formal semantics of both future and past operators, see [6].

For translating specification-level obligations into conditions of implementation-level policies, we assume that there is a special proposition $START$ that denotes the moment in time when the future-time formula has to hold, that is, when the policy is deployed. This can be an activation event, the universally valid proposition *true*, or any other propositional formula described in OSL.[1] The translation function τ from specification-level obligational formulas to the conditions of implementation-level policies is inductively defined as follows:

1. Propositions $\phi \in \Psi$ remain unchanged: $\tau(\phi)$ is transformed into ϕ.
2. $\tau(\phi \wedge \psi)$ is transformed into $\tau(\phi) \wedge \tau(\psi)$; $\tau(\neg\phi)$ is transformed into $\neg\tau(\phi)$; all other propositional operators can be expressed by virtue of conjunction and negation.
3. $\tau(\underline{always}(\phi))$ for $\phi \in \Psi$ is transformed into the condition $\underline{before}^-(1, \tau(\phi)\underline{since}^- START) \wedge \neg\tau(\phi)$; the rule is applicable if ϕ has been true at every time-step since START except in the current time step.
4. $\tau(\phi\underline{until}\psi)$ for $\phi, \psi \in \Psi$ is transformed into the condition $\underline{before}^-(1, \tau(\phi \wedge \neg\psi)\underline{since}^- START) \wedge \tau(\neg\phi \wedge \neg\psi)$; the rule is applicable if, ϕ has been true (and ψ false) at every time step since START but in the current time step, both ϕ and ψ are false.
5. $\tau(\underline{within}(n,\phi))$ for $\phi \in \Psi$ is transformed into the condition $\underline{before}^-(n, START) \wedge \underline{during}^-(n, \tau(\neg\phi))$; the rule is applicable if START was true n time steps ago and in the past n time steps, ϕ has been false at every time step.
6. $\tau(\underline{during}^-(n,\phi))$ for $\phi \in \Psi$ is transformed into the condition $\underline{before}^-(n, START) \wedge \neg\underline{during}^-(n, \tau(\phi))$; the rule is applicable if START was true n time steps ago and in the past n time steps, ϕ has been false at least once.
7. $\tau(\underline{after}(n,\phi))$ for $\phi \in \Psi$ is transformed into the condition $\underline{before}^-(n, START) \wedge \neg\tau(\phi)$; the rule is applicable if START was true n time steps ago and in the current time step, ϕ is false.
8. $\tau(\underline{repuntil}(n,\psi,\phi))$ for $\phi \in \Psi$ is transformed into the condition $(\neg\tau(\phi)\underline{since}^- START) \wedge \neg\underline{repsince}^-(n-1, \psi, START) \wedge \psi$; the rule is applicable if ϕ has not been true since $START$ (via $\neg\tau(\phi)\underline{since}^- START$); if furthermore, there have been at least n occurrences of the propositional formula ψ (via $\neg\underline{repsince}^-(n-1, \psi, START)$); and if there is another occurrence of the propositional formula ψ in the current step (where usually one distinguished proposition will correspond to the triggering event; this translates into a respective desired event in the transformation of ψ). Note that the semantics of $\underline{since}^-$ includes the current time step.
9. $\tau(\underline{repmax}(n,\psi))$ is transformed via the equivalence with the respective $\underline{repuntil}$ operator:
$\underline{repmax}(n,\psi) = \underline{repuntil}(n,\psi,\underline{false})$

[1]The rule is assumed to be applicable at the *first* violation of the obligational formula; further violations are not considered.