FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

Model-Based Policy Derivation for Usage Control

Prachi Kumari

2015

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl XXII - Software Engineering

Model-Based Policy Derivation for Usage Control

Prachi Kumari

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ-Prof. Dr. Uwe Baumgarten
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. Alexander Pretschner
	2.	Reader Naranker Dulay, Ph.D.,
		Imperial College London, GB

Die Dissertation wurde am 13.05.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.07.2015 angenommen.

Acknowledgments

In every achievement is the contribution of many. And so is with this doctoral work. Thank you everybody who helped me in this endeavor.

I am indebted to my supervisor Prof Dr Alexander Pretschner for entertaining my ideas and giving me this opportunity to work on them. His invaluable guidance and persuasion at the right times coupled with moral support, is the reason I was able to hold on till the end. He was also immensely helpful at steering me through the tedious writing process.

I also thank my co-supervisor, Dr Naranker Dulay for his feedback and suggestions.

To Enrico Lovat and Florian Kelbert: thank you gentlemen for participating in all the discussions with your thoughtful comments, for reviewing the first draft of this thesis and for all the initial help with the formalisms.

I am also grateful to all my colleagues at the chair for the constructive criticism in my group talks and in one-on-one conversations and, for providing me the fun at work.

I also thank all my Bachelors and Masters thesis students and Hiwis who contributed to parts of this work.

To my family and my best friend Hasan: I cannot thank you enough for inculcating in me the ambition to pursue a PhD. I immensely value your contribution in all my successes.

Finally and most importantly, thank you Moritz for your endless patience, love and belief in my abilities. Without your unwavering support and motivation, this work would not have been possible.

Zusammenfassung

Datennutzungskontrolle ist eine Erweiterung der Zugriffskontrolle, die zusätzlich spezifiziert und durchsetzt was mit Daten passieren soll oder nicht passieren darf, sobald der Zugriff auf diese gewährt wurde. In bereits existierenden Formen der Durchsetzung von Datennutzungskontrolle lag der Fokus auf der Implementierung der Überwachung von Ereignissen. In diesen wurden Datennutzungsrichtlinien auf der Implementierungsebene manuell von technisch versierten Experten spezifiziert. Diese manuelle Spezifikation von Richtlinien auf der Implementierungsebene hat mehrere Nachteile. Erstens werden technisch unversierte Nutzer außen vor gelassen. Zweitens sind Richtlinien auf Implementierungsebene sehr komplex, was deren Spezifikation durch technisch versierte Experten aufwändig macht. Drittens bedingt die hohe Komplexität eine Spezifikation der Richtlinien mit absoluter Sorgfalt, die hierdurch fehleranfällig ist.

Zusätzlich kann die Durchsetzung von Datennutzungskontrollrichtlinien bezüglich Aufbewahrung, Verteilung und Löschung der Daten nicht die Anforderungen des Nutzers widerspiegeln, da kein universeller Ansatz existiert, der abstrakten Begriffen wie Kopieren und Löschen eine Bedeutung zuweist. Würde man nun eine ad-hoc Spezifikation der Semantik vornehmen, so könnte dies zur Blockierung von legitimen und Zulassung von unerwünschten Ereignissen führen.

Die vorliegende Arbeit bearbeitet diese Problemstellung durch die Erstellung eines Frameworks zur Ableitung von Richtlinien mit einer domänen-spezifischen formalen Semantik für Aktionen. Die Bedeutung der Aktionen ist auf Domänenebene durch die Instanzierung eines Metamodells definiert, das die hierarchische Verfeinerung von Aktionen erlaubt. Eine wohldefinierte Methodik zur Automatisierung dieser Ableitung wird ebenfalls beschrieben. Menschliche Eingriffe sind nur in zwei Rollen angedacht: Als technischer Experte, der die Richtlinienübersetzung konfiguriert, und als Endnutzer, der Richtlinien zum Datenschutz definiert.

Die Eingaben zur Ableitung von Richtlinien sind ein Domänenmodell, eine Richtlinie auf Spezifikationsebene und einige umgebungsabhängige Konfigurationsdetails der Infrastruktur. Die Ausgabe ist eine Menge an Richtlinien der Implementierungsebene, die Durchsetzungsmechanismen auf verschiedenen Abstraktionsniveaus des Systems konfigurieren. Der Ansatz wird in mehreren Fallstudien erklärt und evaluiert.

Abstract

Usage control is an extension of access control that additionally specifies and enforces what may or must not happen to data once access to it has been granted. In existing enforcements of usage control policies, the focus has been on the implementation of event monitors. Implementation-level policies have been manually specified by technical experts. Manual specification of policies at the implementation level has certain disadvantages. Firstly, users who are technically not proficient, are kept out of usage control specification. Secondly, implementation-level policies tend to be very complex, lengthy and tedious to specify, even for technical experts. Thirdly, because of the complexity of these policies, their specification requires meticulous care. This makes the manual specification of implementation-level policies error-prone.

In addition to this, as there is no universal way of giving meanings to high-level actions like copy and delete, enforcements of usage control policies regarding retention, distribution and deletion of data may not adequately reflect user requirements. An ad hoc specification of semantics might result in the inhibition of legitimate events during enforcement while unwanted events might be allowed.

This work addresses the aforementioned issues through a policy derivation framework with domain-specific formal semantics of actions. Meanings of actions are defined at the domain level by instantiating a metamodel that allows for a hierarchical refinement of actions. A well-defined methodology to automate the policy derivation is also described. Human intervention is suggested in two roles: a technical expert power user who configures the policy translation and, an end user who specifies policies for data protection.

The input of the policy derivation is a domain model, a specification-level usage control policy and some infrastructure configuration details that vary according to the environment. The output is a set of implementation-level policies that configure enforcement mechanisms at different layers of abstraction in a system. The approach is demonstrated and evaluated through case studies.

Contents

List of Figures

List of Listings

List of Tables

Part I.

Introduction and Background

Chapter 1

Introduction

The rapid growth in computing in the past decades [4, 5, 6] has ensured that data generation, storage, processing and sharing is no more confined to businesses, government agencies and other organizations. From shopping habits, banking details to health history, data can be available to creditors, employers, landlords, insurers, law enforcement agencies, and criminals. Recent developments in pervasive computing have made this data accessible on all sorts of hand-held devices including mobile phones, tablets, PDAs and various other smart equipments. A trivial example is the energy usage data generated by smart meters, which can now be accessed on a mobile phone and can be disseminated by users via online social networks [7]. This data is open to access and interpretation by individuals and agencies across the globe. In such scenarios, a usual approach in data security has been to limit data exposure to only those people who must get access to it. Several type of authentication mechanisms have been proposed and implemented to ensure *access control* [8, 9, 10, 11].

However, even if data is accessible at a smaller scale by limiting access, it can still be misused by the very people who legitimately access it for performing their tasks. Various incidents of deliberate and accidental data leakages [12, 13] are well-known examples of this. In this new paradigm, data security is no more limited to controlling access; it must also cover post-access aspects of data protection. In the recent years, access control has been extended to additionally specify and enforce what must or must not happen to data once access has been granted. This extension of access control is called *Usage Control* [14, 15, 16].

Usage control is relevant to a number of contexts. In the above example of sharing smart meter data on a social network, usage control enforcement can protect users' privacy by limiting the uncontrolled duplication and sharing of their data by other stakeholders. Further examples of privacy protection can be found in ambient assisted living systems, healthcare and video surveillance systems.

Usage control can also be used for the enforcement of several data protection laws that restrict collection and usage of personal data to specific purposes and enforce mandatory anonymization, encryption and deletion, e.g., the EU privacy directives 95/46/EC and 2002/58/EG, the Bundesdatenschutzgesetz of 1990 (with amendments of 2009 to accommodate digital data protection) in Germany, the Health Insurance Portability and Accountability Act of 1996 (HIPAA) in the United States and the Data Protection Act 1998 in the United Kingdom [17].

Similar requirements arise from copyright protection laws. One example of usage control enforcement for copyright and other intellectual property protection is the mandatory check by many cloud storage providers to actively prevent copyrighted data from being shared [18, 19]. Digital rights management is another example of usage control enforcement. Some other areas of relevance are data protection for business secrets, protection of administrative and intelligence data and military secrets.

Usage control is not only relevant for protecting data from active adversaries, but it can also be used to hinder passive attackers who unintentionally do malicious things such as accidental data leakage [12]. As recent reports show, in 2013 alone, various data breaches left over 800 million records lost with almost a third of the breaches taking place due to insider activities [13]. In fact, data leakage prevention is one of the top concerns for the organizations that collect, process and maintain data and they are therefore legally accountable to maintain the confidentiality and integrity of the data [20].

Therefore, usage control requirements come from public administrations, data protection officers, health care providers, military organizations, and numerous commercial enterprises [21]. In most of the cases, these users who have an understanding of 'what' they expect the system to do, neither have sufficient knowledge of the deployed technical systems nor are they concerned with 'how' these requirements should be technically met. A majority of such users would like to specify usage control requirements in form of policies that talk of user actions and data without technical details. For example, "data must be deleted after 30 days", "access logs must be stored for 2 years", "a movie may be played maximum two times without payment", "no non-anonymized data should leave the system" and "notify the owner upon each distribution of document". Such usage control requirements are called *specification-level policies*. As end users often only understand high-level terms, usage control policies are specified at the higher levels.

However, technical systems *only* understand low-level technical terms like files, system calls, methods, etc. Therefore, for enforcing specification-level policies, they must be transformed to *implementation-level policies* that consists of low-level technical terms equivalent to the high-level ones.

Typically, transformation of specification-level policies into implementation-specific constructs has been a manual activity in usage control. Although there exist several approaches for policy derivation in the related area of access control (see Chapter 8 for more details), in the absence of a generic way to precisely describe the refinement of high-level policy constructs in the usage control context, the behavior of concrete systems might deviate from the expected behavior. As a result, we might have implementations that, at different levels of abstraction in a system, permit executions that should be inhibited and/or forbid others that should be allowed. Also, in the absence of a systematic, technological and methodological framework, automated policy derivations cannot be realized.

This thesis addresses this twofold problem: the lack of a generic policy refinement approach and the absence of a technological framework to derive policies in an automated way in the context of usage control. The solution consists of (i) formulating the generic relationship between different policy constructs at the specification and the implementation levels, (ii) giving formal semantics to these relationships in the context of an existing usage control model, and (iii) using this relationship to derive usage control policies from the specification level to the implementation level. A generic methodology to achieve the policy derivation in an automated manner is also described.

The next section starts with the basics. At first, the distinction between access control and usage control in the context of this work is clarified; then, the fundamental usage control concepts relevant to this work are discussed.

1.1. Access Control vs. Usage Control

Access control provides means to specify and enforce policies about who can access data and how. Usage control [15] extends this to include constraints upon the future usage of data once access has been granted. Usage control policies talk about temporal conditions ("data must be deleted *after 30 days*"), cardinality constraints ("a movie may be played *maximum two times* without payment"), event-defined restrictions ("*if the data provider revokes the usage right*, any further usage must be inhibited"), purpose of use restrictions ("data must be used for *non-commercial purpose only*"), and environmental conditions ("data must be encrypted when used *outside the company premises*") [21].

When a request to access specific data is made, two type of conditions apply. One type of conditions are checked before the access is granted and are typically called the authorization constraints in an access control (AC) system. In this work, these conditions are called *Provisions*. At the moment in time when access is granted, provisions refer to the past and concern the conditions upon whose fulfillment a requester gets access to data. *Obligations*, on the other hand, concern future usage of the data item and are part of usage control (UC). Figure 1.1 (adapted from [16]) shows the demarcation of access and usage control control for a data 'd' in the context of this work. Access control rules (provisions) must be specified and activated (e.g., at time $-t_j$) before a request for data access is made at time $-t_i$, while the usage control rules (obligations) must be activated anytime before access is granted (at time t_0). Obligations must be checked for conformance in all future timesteps from time t_0 onwards, including any timestep t_n when the data is actually used.

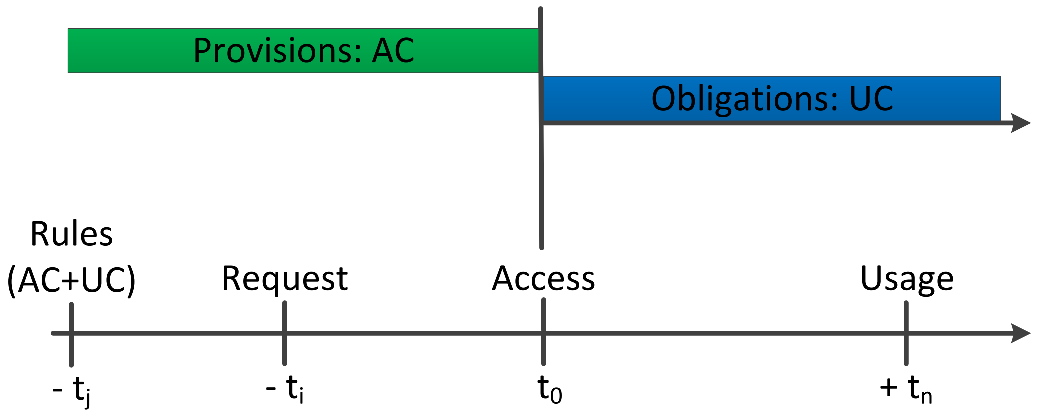


Figure 1.1.: Access control vs usage control in the context of this work

This work distinguishes between access and usage control as shown in Figure 1.1 and focuses on the post-access obligations. Hence, *policies are sets of obligations* that a data receiver must fulfill once access is granted. Further discussion on access control policies is out of the scope of this thesis.

1.2. Usage Control Basics

The acceptability of obligations, enforcement guarantees and penalties in cases of the violations, and the scheme to handle exceptional cases are negotiated before a *data provider* sends the requested data along with usage control policies [22, 23]. The *data consumer* thereafter, must enforce all the policies specified by the data provider, in order to use that data. The communication between the data provider and the data consumer must be protected so that policies cannot be changed on the fly.

The assignment of entities to the roles of data provider and consumer keeps changing in the process of data usage. For example, let us consider the scenario shown in Figure 1.2 (adapted from [16]): in a software company, a manager sends an email to the development team head with certain instructions about the latest project. As this mail contains information that is classified as business secret, the manager attaches a set of policies allowing only limited distribution and constraining printing on shared printers. The manager in this case is the provider while the development head is the consumer of data. However, when the development head forwards this mail along with other information to the relevant persons in his team, he becomes the data provider and others, the consumers of the data. Subjects assigned to the roles of data provider and consumer keep changing as email conversations proceed back and forth. In order to ensure adherence to policies in such a scenario, the usage control infrastructure keeps track of any copy or derivation of the data item (e.g., forwarded email). If any consumer ships the data item (or a copy or derivative of this data) to a further data consumer, the usage control infrastructure makes sure that a policy is attached to this data item that at most strengthens the original policy.

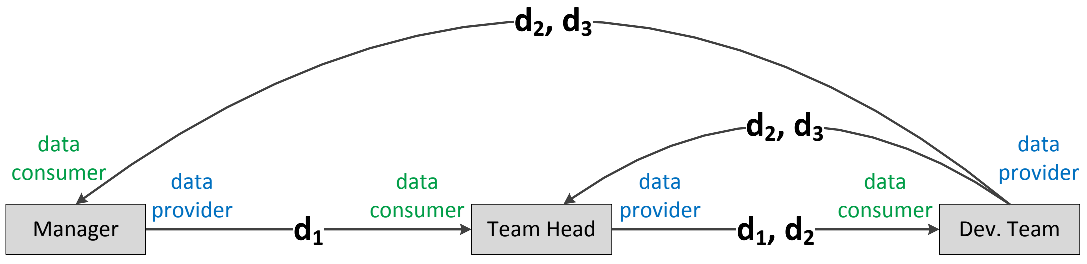


Figure 1.2.: Data providers and consumers keep changing in a typical UC scenario

Usage control policies stipulate both rights and duties and they can be enforced in two ways. *Detective enforcement*, on the lines of optimistic security [24], relies on the detection of policy violations. Policy violations are allowed but logged, followed by penalties or other appropriate actions. This is similar to law enforcement in the physical world where for example, drivers cannot be prevented from speeding but based on evidence, they can be fined for doing so. Detective enforcement of policies is particularly useful in situations in which imposing any kind of control on data consumer's end is not feasible, practical, or sensible. Companies, for example, are reluctant to let other parties control parts of their IT infrastructure. What they might rather be willing to do is provide evidence that their information processing indeed adheres to the stipulated policies. In these cases, deploying observation mechanisms is a more appropriate solution [25].

In contrast, *preventive enforcement* works by altogether not allowing unauthorized actions. This is done by conceptually distinguishing between an attempt to perform an action and the actual execution of that action (*intended* vs *actual* events, see Section 2.2). Technically, the signaler intercepts events before they are completed, queries the monitor and subsequently aborts or completes the event.

As preventive enforcement aims to block policy violations, it may hinder certain functionality which in general is not desired but might turn out to be a critical requirement in exceptional cases. For example, in a hospital, every doctor *may get access to read and modify* the medical record and prescriptions of only those patients whom he treats. Additionally, doctors *must never send this information* to anybody who is not employed with the hospital. However, in case of a medical emergency outside the hospital, the medical history of the patient might be a critical information for the attending doctor who is not employed with the patient's hospital. To handle such exceptional cases, a hybrid enforcement scheme may be installed: the default being preventive enforcement with a fall-back to detective enforcement. In the aforementioned example, this would mean that the regular doctor is *allowed to send* the medical history to the attending doctor. The attending doctor may not only *read* the data but also perhaps *modify the data* to add the medicines he administered and prescribed.

Typically, usage control policies address *data*, irrespective of the various *concrete technical representations* in different systems. So, when the manager in the previous example specifies a policy about restricting the distribution of the *confidential data*, he means to apply this policy to all copies of it. Hence, recipients of the concerned *email* must not only be inhibited from forwarding the email to unauthorized recipients, they must also not be allowed to forward copies of the mail. Hence, all of the following (and many other) actions that result in the policy violation on the copies of the confidential data should be inhibited:

(a) compose a new *mail* with the protected content of the received mail

(b) save the mail or the confidential parts of it in a *file* on the disk and send this file as an attachment to another mail

(c) copy the file in (b) on an external storage device

(d) upload the file in (b) to a remote server in form of *network packets* to a remote server

(e) take screenshot of the confidential data rendered on the screen (viz. *pixmap*) and distribute it

However, it is not possible for an end user (the manager in this example) to enumerate all the possible technical representations of the data and the different technical ways to perform a high-level action addressed in a policy. So he would like to specify policies in high-level terms and expect an infrastructure to somehow translate and enforce these policies. In order to specify policies in abstract terms (data) and to enforce these policies without having to enumerate all the different possible respective technical representations (called containers), there must be a conceptual distinction between the two.

1.2.1. Data and Containers

In the context of this work, data is an abstract concept with an intuitive semantics. E.g., the German national anthem, a social network profile, a patient's medical history, an individual's personal data, a film called "Jurassic Park", the requirements document in a project, etc. Data is stored in different concrete representations at and across various *layers of abstraction* in a machine: as contents of files in a file system, pixmaps in a windowing system, memory regions in an operating system, network packets in a network communication etc. One concrete technical representation of data in a specific layer of abstraction is called a *Container* of data.

While there are some realistic usage control policies such as those about data integrity, that target specific containers, e.g., "don't edit /etc/passwd file" in the context of a Unix operating system or "/.config file must not be moved" in the context of a java application; it can be argued that usually it is the content of these containers and not the specific containers themselves that a data provider is interested in protecting. So when a policy states "do not forward or print this email", it is most likely that the data provider wishes to address all containers that store that data including files, sockets, email attachments, pixmaps, data structures and so on. Therefore, **policies are specified on data**. But as concrete systems only have containers, **policies are enforced on containers**. For this reason, there are at least two sets of policies *corresponding to two levels* in usage controlled setups: one class of policies that talk about constraining actions on data and the other that address the concrete representations of action and data in implementation-specific systems.

1.2.2. Specification vs. Implementation -Level Policies

Because data that has to be protected comes in different representations, within a machine, usage control policies must be enforced at and across different layers of abstraction. In principle, all these representations eventually boil down to some representation in memory and usage control could be as well enforced only at that one layer. However, it turns out to be generally more convenient and simpler to perform protection at higher levels of abstraction. For instance, disabling the print command is easily done at the word processor level; taking screenshots is easily inhibited at the X11 level; prohibiting dissemination via a network is most conveniently performed at the operating system level. Moreover, as usage control comes into picture after the data is at the consumer side, enforcement solutions must cater to specific implementations that are deployed at the consumer side. For example, a usage control implementation for one mail client, viz. Thunderbird does not work for another, viz. Outlook. Therefore, enforcements specific to the deployed systems are required. As a result, each policy specified in terms of an abstract data is enforced as multiple technical policies with implementation details of the respective concrete systems. The former set of policies are called the *specification-level policies* while the latter are called the *implementation-level policies*.

Specification-level Policies (SLPs) state *what* must or must not happen to data, while the Implementation-level Policies (ILPs) describe *how* the system should achieve that. SLPs are composed of abstract domain-level actions, data and future-time constraints. ILPs on the other hand can be formulated as sets of Event-Condition-Action (ECA) rules that configure enforcement mechanisms. An ECA rule executes an *action* (or a set of actions), when a trigger *event* takes place and the respective *condition* evaluates to true. One SLP may correspond to several ILPs at different layers of abstraction in place.

1.2.2.1. Enforcement strategies

One Specification-level Policy (SLP) can be enforced in several ways. For example, "a movie may be played maximum two times without payment" is a specification-level policy. One way to enforce it would be to *inhibit* all further usages when the movie has already been played two times and no payment has been made; another approach would be to allow playing the movie *at a lower quality* or *for a very limited time* (like a preview); yet another way to enforce the policy can be to allow playing the movie normally but *log the action* so that penalties could be issued or *delay* playing the movie until the data provider explicitly grants permission to do so. These various ways of enforcement are referred to as *enforcement strategies*.

Technically, during enforcement, an attempt to perform an event is intercepted, and depending upon the conditions in the policy, the attempted event is blocked, delayed, modified, or executed along with other actions. Responses triggered by the actions may as well be modified. Modification is used as a way to declassify data [26].

The choice of enforcement strategy may depend upon several factors including operational context, legal requirements, technical limitations and organizational goals and policies. E.g., a company enforces a policy "don't copy document" by inhibition because the company wants to prevent data leakage and its infrastructure supports it; or, as in the above example, a film producer enforces "don't play unpaid movies" by allowing corresponding events with lower quality videos because either it's technically not feasible to inhibit the events or he wants to give a limited preview to the prospective customers.

1.2.3. From SLPs to ILPs

Ideally, policies for preventive enforcement should be specified at a high-level and through a policy derivation process, the SLPs should be refined to enforceable ILPs. However, in the usage control literature, policies are directly specified at the implementation level because the focus of these work is the enforcement of policies. Derivation of implementation-level policies from specification-level policies has not been looked into. Manual policy derivation and their specification at the implementation levels imposes some fundamental limitations, one being the correct and consistent interpretation of high-level terms. The next sections highlight the problems with this approach and briefly summarize the solution and the contribution of this thesis.

1.3. Problem

In existing work on usage control, policies are specified directly at the implementation-level as sets of ECA rules. The drawback of this approach is that firstly, end users who are not technical experts cannot specify usage control policies to protect their data. Secondly, manual policy specification at the implementation-level or a manual derivation of implementation-level policies from the specification level is tedious and error-prone even for sophisticated users due to the complexity of the involved systems. Thirdly, as data protection requirements usually arise at a much higher level i.e., in the minds of the users, than at the levels where they are enforced, any ad-hoc policy derivation from the specification to the implementation levels might result in gaps between the two types of policies. This is possible due to several reasons, one of which is the problem of mapping concepts in the end user's domain to technical events and artifacts. Policies with user actions that have complex technical semantics might not be captured correctly in ad-hoc ways. For instance, semantics of basic operators such as "copy" or "delete", which are fundamental for specifying usage control policies, tend to vary according to the domain context. For this reason, in ad hoc policy derivation, they might be mapped to incomplete or incorrect sets of system events. This might wrongly allow events that should have been inhibited and block those that should have been allowed during the enforcement of the policies.[1]

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. As a result, system implementations of usage control policies might not always adequately reflect end user requirements. This thesis aims to fill this gap. The problem can be extended to the *systematic derivation of system requirements from user requirements*.

The goal of this work is to fill the gap between the user and the system requirements in the context of usage control.

In sum, two problems are tackled in this work. The first one is the *fundamental problem of the lack of clear 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. The solution addresses both of these aspects.

1.4. Solution

Action refinement is at the core of policy derivation in this thesis. It relies on the conceptual distinction between high-level user actions and their technical representations, called the *transformers*.

1.4.1. Actions and Transformers

High-level actions can have several technical interpretations. These depend upon the application context, the layer of abstraction and the concrete representation of data in that

[1]These drawbacks need not be an issue in the case of detective enforcement as SLPs can be directly monitored: see Section 2.5.

layer and, the system implementations in place. E.g., "copying" a photo might mean opening the file and saving it with another name (or location), taking a screenshot of the displayed image, uploading the file to a web server, sending the photo as an email attachment, cloning the java object that reads from the file and write the object into another file or, copying blob object from one table to another in a database. Similarly, "deleting" personal data might mean removing the FAT entry in a file system or drop the table in a database. Even within a file system, deleting can mean several things: unlink the file, overwrite the file, encrypt the file and erase the key, quarantine the file or erase a block of the disk at the firmware level or by resetting the bits on an SSD chip.

Each of these different technical ways of achieving a high-level goal is called a *transformer* that "transforms" the content of the containers that it targets. Transformers are composed of atomic or complex events. Each transformer can be recursively refined to correspond to a set of technical events in concrete systems.

It is important to understand that there is no single "correct" definition of what a high-level action means. Several transformers together correspond to one action. With the distinction between data and containers and, actions and transformers in place, policies concerning actions on data implicitly address all transformers that target containers where data is stored.

Summary of the Solution: In a nutshell, the problem of policy derivation is addressed in three steps:

1. A generic approach to systematically refine specification-level policy constructs viz. data and user actions into corresponding technical artifacts is described.

2. The semantics of these refinements are formalized to derive implementation-specific policies for usage control enforcement.

3. A methodological guidance to automate the derivation of implementation-level policies from specification-level policies is provided.

As the meanings of data and actions vary according to the domain context, any solution that aims to refine these constructs in a generic way must address the problem at the domain level. For this reason, a domain metamodel is described that allows to define models of domains where classes of data and actions are mapped to classes of respective technical representations. In order to formally define what "refining" a data or an action precisely means, the domain metamodel is combined with an existing usage control model.

In terms of the policy derivation methodology, the domain model of the application is defined by sophisticated users called the *power users* (system administrators, architects, etc.). Power users design and set up the policy specification and derivation infrastructure. Besides the domain model, they also define all types of policies that could be specified, translated and enforced and, the enforcement strategies for the SLPs. *End users* (managers, data protection officers, online social network users, etc.) specify usage control policies at a higher level addressing specific user actions on data. From these policies, implementation-level policies are derived using the already-defined domain model and enforcement strategies.

Policy enforcement, evolution of policies and policy lifecycle management is out of the scope of this work.

Policy enforcement in single and distributed systems is tackled in [27, 28, 2, 29, 30] etc. and the evolution of usage control policies is addressed in [31]. Though not in the context of usage control, policy lifecycle management is addressed in several other related work [32, 33, 34].

1.4.2. A Running Example

The following scenario will be revisited and elaborated as and when necessary for the explanation of the different concepts in this thesis:

> In an Online Social Network (OSN), user Alice wants to protect her data from being misused by other users. She wants that her friends should be able to view everything that she shares with them, but they should not be able to copy or distribute her data. For this, Alice wants to specify and enforce policies such as "friends must never copy this data" for her photos, videos, etc. without any technical knowledge. Alice's policies must be automatically translated and enforced at and across different layers of abstraction in her friends' machines.

This work describes the conceptual, technical and methodological aspects of the framework that helps Alice. For this use case, the attacker model considers attacks on a user's privacy that emerge from other users. The OSN provider is trusted.

This use case is just one instance of the generic scheme and is used in this thesis for demonstration purposes. The problem, the solution and the core of the implementation is generic and can be applied to a variety of other cases.

1.5. 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 in a generic, domain- and system-independent way. In light of this, the contribution of this thesis is threefold:

1. **Technical:** This work provides a generic way to model high-level user actions like copy and delete in terms of system events. The generic semantics of domain-specific actions has been formalized in the context of a system model. Using these two technical contributions, this work fills the gap between user and system requirements in the context of usage control.

2. **Methodological:** The second contribution is the definition of a methodology for automated policy derivation. Owing to the nature of the problem (policy enforcements can be achieved in several ways, depending upon the interpretation of the policy), a "one fits all in all situations" type of solution cannot be designed. The contribution therefore is the insight about the challenges and the hints to address them. Through

several instantiations, the feasibility of achieving an automated policy translation for usage control has also been demonstrated.

3. **Generalizable Solution:** This work proposes a generic way to define the semantics of high-level actions like copy and delete. Though these semantics are used in this thesis for automated policy derivation specific to usage control, the solution presented here can be easily generalized to other cases (Chapter 9). Hence the third contribution of this thesis.

Contribution of this Thesis with respect to Related Work: In the literature, one of the well-known approaches to policy refinement is based on modeling policies as goals and incrementally refining each of the high-level goals to those at the lower levels [35, 36, 37, 38]. Some other prominent work on policy derivation use resource hierarchies [39, 40]; ontology, where policy constructs are refined according to their meanings [41, 42, 43]; decomposition of actions in policies [44]; commitment/obligations analysis [45]; data classification [46] and patterns that define some kind of best practices [47] in order to refine policies.

All these approaches are similar to this work in spirit because of the refinement of policies based on hierarchical structures. But, *firstly*, they refine policies for different access control models, not for usage control. *Secondly*, most of the policies are refined from the abstract level to the logical levels; further technical representations of policy elements in concrete systems are only specific to the domain in consideration. They do not aim to provide generic ways to give semantics to actions like "copy" and "delete" in a domain and system -independent way. In this context, different ways of secure deletion and its meanings are covered in [48]. But this work also does not provide a comprehensive discussion on all types of deletion. Besides, other actions such as copy are not considered at all.

In the context of *usage control*, though [49] describes a policy derivation for usage control, it refines policy actions to business processes functions for policies to be enforced in a BPMN engine.

In a nutshell, although there has been a lot of work in the area of automated policy derivation in recent years, the focus has been on access control. In usage control, the related work [21, 50, 51, 27, 7, 52] has been devoted to policy enforcement and has not looked into the policy derivation. This has left a gap between specification and implementation -level policies in usage control context that this work fills.

A detailed discussion of the contribution of this work with respect to the state of the art is in Chapter 8.

1.6. Assumptions and Limitations

This work focuses on the derivation of usage control policies from specification to implementation levels. Enforcement of policies and related guarantees are not in the scope of this work. As the solution described in this thesis integrates into the usage control infrastructure, a fundamental assumption is that the enforcement components are correctly implemented, they are not tampered with and they are up and running.

One of the core contributions of this thesis is a generic language to model domain-specific semantics of high-level user actions. However, no theorems to prove the correctness of the such semantics are discussed here. The reason is that the meanings of abstract terms like data and actions are intuitively understood by the end users. That is, the semantics of high-level propositions is not precisely defined but rather exists in the user's mind. It is therefore hard to establish a notion of correctness between the semantics of low-level and high-level policies based on these semantics. Because of this, the domain model's correctness cannot be theoretically proven. This means that the solution relies on the power user's capability of correctly modeling domain-specific semantics.

For this reason, the assumption is that data only moves in a usage control environment where each layer of abstraction on every machine could be correctly modeled for the semantics of the actions. If the power user wrongly models the domain, the errors would propagate throughout the enforcement. Also, if data is stored at a layer of abstraction which is not included in the domain model or which could not be modeled for any reasons (business secret, lack of understanding of the functionality, closed source with no documentation, etc.), then no implementation-level policies would be generated for monitoring data usage in that layer of abstraction. Hence corresponding usages might be wrongly allowed during the enforcement of policies.

The second limitation is that policy distribution is not discussed. This work assumes the communication between the data provider and the data consumer to be protected so that policies cannot be changed on the fly. This is possible using end-to-end encryption with appropriate key management solutions: the data consumer receives an encrypted pair of data item and policy. The local usage control infrastructure of the consumer then decrypts this pair, deploys the policy (see Chapter 7), and makes the data item accessible to the consumer. This is the *moment of access* to data item, already shown in Figure 1.1 as time t_0. From this point onward, the usage control infrastructure of the consumer makes sure that the policy is enforced at the consumer's side, one assumption being that every access to the data item is mediated by the usage control infrastructure.

In addition to the above, this work does not address policy evolution, policy conflict resolution and policy lifecycle management. For example, cases when specification-level policies may also change from one receiver to another in a distributed setup have not been taken into account. Also, issues pertaining to different administrative domains (e.g., conflicts in action refinements and negotiations) have not been considered.

1.7. Organization

This dissertation is structured along the following lines:

Background: This part recaps the background information required for understanding this thesis. This includes a policy language and a system model that the core contribution of this thesis builds upon.

Chapter 2 describes a formal usage control model and an Obligation Specification Language (OSL) which is first-order linear temporal logic with macros for cardinality constraints. Policies are constraints upon events. The language description includes the syntax and the semantics of a future-time and a past-time flavor of OSL for specifying SLPs and ILPs respectively.

Chapter 3 extends the usage control model of Chapter 2 with data flow concepts to distinguish between data and its technical representations called containers. 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.

The Core: Chapter 4 lays down the foundations of the core contribution of this thesis in form of a set of user requirements for policy derivation. Starting towards the fulfillment of those requirements, in this chapter, a domain metamodel that distinguishes between actions and technical transformers at the level of classes and refines the former to the latter, is described and instantiated for three use cases.

Chapter 5 combines the domain metamodel to the extended usage control model of Chapter 3 to give formal semantics to the refinement of actions in policies using OSL. A set of rules to derive the past equivalents of the future-time SLP formulas is also discussed. The approach is elaborated by showing the translation of several example policies.

Chapter 6 provides a methodological guidance to automate the policy derivation from SLPs to ECA rules. This includes the description of the steps in the technical process and a work flow for the involved human users.

Chapter 7 describes the overall software architecture of the usage control infrastructure. The policy derivation components facilitate the specification, derivation and deployment of policies to the enforcement core that monitors and intercepts events, evaluates them against the deployed policies and triggers further events according to the result of the evaluation.

Conclusion: Chapter 8 discusses the state of the art in policy derivation, especially the *automated* policy derivation approaches and their differences with this work. The sub-problems and their solutions presented in this thesis are also reviewed in the context of the related literature and the contribution of this work is argued in terms of the identified differences.

Chapter 9 concludes with a critical analysis of the results and a highlight of interesting future work directions.

Chapter 2

The Usage Control Policy Language

Major parts of this chapter are verbatim from our yet-to-be-published book on usage control [2] and have not been written by the author of this thesis.

In this chapter, the formal model for usage control policies is described where usage control policies are constraints on traces of events. This is extended to constraints over system states in Chapter 3. The language is called the Obligation Specification Language (OSL) which is essentially linear temporal logic with first-order constructs to express requirements that put time conditions ("data D must be deleted *after 30 days*"), cardinality constraints ("video V may be played *maximum twice* before being paid for"), event-defined conditions ("notify the author *when the document is edited*"), action conditions ("*watermarks must be inserted* before the photo is forwarded") and purpose of use constraints ("*for advertising purposes only*") [53]. The discussion of the language includes abstract syntax and semantics of the specification- and the implementation- level policies in Section 2.2 and Section 2.3 respectively. A concrete syntax of the policies is described in Appendix B.

2.1. Preliminaries

2.1.1. Events

Events are defined by a name (set $EName$) and a set of parameters. Parameters are defined by a name (set $PName$) and a value (set $PValue$). Formally,

$$\mathcal{E} \subseteq EName \times \mathbb{P}(PName \times PValue).$$

For an $e \in \mathcal{E}$, let $e.n$ denote the event's name, $e.p$ the set of its (parameter, value) pairs and $e.p.pn$ the value of the event's parameter pn where $pn \in PName$. Parameter name $obj \in PName$ denotes the primary object of an event. Similarly, $subject \in PName$ can be a reserved parameter to express the subject who executes the event. One example for an event and the notation adopted is $copy \mapsto \{obj \mapsto myphoto.jpg, quality \mapsto 100, subject \mapsto Alice\} \in \mathcal{E}$ that specifies that photo $myphoto.jpg$ is copied with 100% quality by $Alice$.

In policies, events may occur with variables that map names to possible values, $Var = VName \rightarrow VVal$ for $VVal = \mathbb{P}(PValue \cup EName)$. Given a set of variables, the set of possibly variable events is defined as

$$\mathcal{VE} \subseteq (EName \cup VName) \times \mathbb{P}(PName \times (PValue \cup VName))$$

where variables are assumed to respect the types, i.e., to range over event names or parameter values. For instance, for $Var = \{v_1 \mapsto \{photo_1, photo_2\}, v_2 \mapsto \{51, \ldots, 100\}\}$, the variable event $copy \mapsto \{obj \mapsto v_1, quality \mapsto v_2\} \in \mathcal{VE}$ represents all copy events of photos $photo_1$ and $photo_2$ with a quality greater than 50%.

2.1.2. Event Refinement

It is convenient to not have to specify all possible event parameters in policies but rather concentrate on those that are relevant in a specific context. As an example, the quality with which a photo p is copied may be irrelevant but not that it is photo $myphoto.jpg$ that is copied. To capture this, $refinesEv \subseteq \mathcal{E} \times \mathcal{E}$ defines a refinement relation on non-variable events. e_1 $refinesEv$ e_2 iff e_1 and e_2 have the same name and the parameters (names and values) provided for e_1 are a superset of those provided for e_2. In the above example, $copy \mapsto \{obj \mapsto myphoto.jpg, quality \mapsto 80\}$ $refinesEv$ $copy \mapsto \{obj \mapsto myphoto.jpg\}$ but $\neg(copy \mapsto \{obj \mapsto myphoto.jpg, quality \mapsto 80\}$ $refinesEv$ $copy \mapsto \{obj \mapsto myphoto.jpg, quality \mapsto 100\})$ because of the disagreement on the quality parameter. Formally,

$$\forall\, e_1, e_2 \in \mathcal{E} : e_1 \; refinesEv \; e_2 \Leftrightarrow e_1.n = e_2.n \wedge e_1.p \supseteq e_2.p.$$

The actual execution of a system produces maximally refined events only: all parameters are determined by the system run. Maximum refinement, $maxRefEv$, is defined as

$$maxRefEv = \mathcal{E} \setminus \{e \in \mathcal{E} \mid \exists\, e' \in \mathcal{E} : \; e' \neq e \wedge e' \; refinesEv \; e\}.$$

2.1.3. Semantic Model of Events and System Runs

In order to model realistic systems, the distinction between actual and intended events is useful. At some layers of the system, it is possible to make this distinction. Examples include system calls at the operating system layer, commands in a user interface, and API calls in a Java virtual machine: there is the intention to execute a system call/GUI command/API method; and the execution of the system call/command/method that leads to a state change and/or an output message. The distinction is necessary for preventive enforcement via inhibiting and modifying implementation-level policies: if an event has already happened, it is in many cases impossible to undo it. Syntactically, the distinction is relevant for the specification of implementation-level policies in Section 2.3 and irrelevant in specification-level policies.

Since all parameters are provided if an event takes place in a system run, the set of system events is defined as,

$$\mathcal{S} \subseteq maxRefEv \times \{intended, actual\}$$

System runs are modeled as traces,

$$Trace : \mathbb{N} \rightarrow \mathbb{P}(\mathcal{S}),$$

that map abstract moments in time to sets of system events. The next section uses these basic concepts of the formal model to describe the specification-level policy language.

2.2. Specification-Level Policies

Specification-level policies (SLP) are specified in the Obligation Specification Language (OSL) which is first-order linear temporal logic with cardinality operators to express requirements upon the number of times an action could be executed. E.g., "copy at most three times". As SLPs describe constraints upon future usage of data, they use the future-time OSL.

2.2.1. Propositions

The first order propositional part of policies is defined by

$$\begin{aligned}\Gamma ::= &\ \mathcal{VE} \mid \mathbb{N} \mid \mathit{String} \mid \Gamma\ \mathit{op}\ \Gamma \mid \ldots \\ \Psi ::= &\ (\Psi) \mid \underline{\mathit{false}} \mid \Psi\ \underline{\mathit{implies}}\ \Psi \mid E(\mathcal{VE}) \mid I(\mathcal{VE}) \mid \mathit{eval}(\Gamma) \mid \underline{\mathit{forall}}\ \mathit{VName}\ \underline{\mathit{in}}\ \mathit{VVal} : \Psi\end{aligned}$$

plus the usual shortcuts in Ψ: $\underline{\mathit{true}}$ for $\underline{\mathit{false}}\ \ \underline{\mathit{implies}}\ \ \underline{\mathit{false}}$, $\underline{\mathit{not}}(\psi)$ for $\psi\ \ \underline{\mathit{implies}}\ \ \underline{\mathit{false}}$, $\psi_1\ \ \underline{\mathit{or}}\ \ \psi_2$ for $(\underline{\mathit{not}}\ \psi_1)\ \underline{\mathit{implies}}\ \ \psi_2$, $\psi_1\ \ \underline{\mathit{and}}\ \ \psi_2$ for $\underline{\mathit{not}}(\psi_1\ \underline{\mathit{implies}}\ \underline{\mathit{not}}\ \psi_2)$, and $\underline{\mathit{exists}}\ \mathit{vn}\ \underline{\mathit{in}}\ \mathit{VS} : \psi$ for $\underline{\mathit{not}}(\underline{\mathit{forall}}\ \mathit{vn} \in \mathit{VS} : \underline{\mathit{not}}\ \psi)$. Whenever no confusion between semantics and syntax is possible, in the following, the symbols $\rightarrow, \neg, \wedge, \vee, \forall, \exists, \in$ will be used for $\underline{\mathit{implies}}$, $\underline{\mathit{not}}$, $\underline{\mathit{and}}$, $\underline{\mathit{or}}$, $\underline{\mathit{forall}}$, $\underline{\mathit{exists}}$, $\underline{\mathit{in}}$.

$I(\cdot)$ and $E(\cdot)$ syntactically capture the distinction between intended and actual events introduced in Section 2.1.3.

2.2.2. Quantification

Quantifications range over the values of the type of the respective variables. For instance,

$$\begin{aligned}&\forall\, \mathit{vn}_1 \in \mathit{PHNAME} : \forall\, \mathit{vn}_2 \in \mathit{TIME} : \\ &E(\mathit{copy} \mapsto \{\mathit{obj} \mapsto \mathit{vn}_1, \mathit{time} \mapsto \mathit{vn}_2\}) \ \rightarrow\ E(\mathit{log} \mapsto \{\mathit{obj} \mapsto \mathit{vn}_1, \mathit{time} \mapsto \mathit{vn}_2\})\end{aligned}$$

stipulates, for a set of photos PHNAME and time points TIME, that "the fact that a photo is copied must be recorded". In the following, with the exception of implementation-level policies as described in Section 2.3, let us assume that all variables are bound by a quantifier and that all bound variables are renamed in a way that makes sure which quantifier binds them. Related work [21] introduced the additional distinction between first and ongoing events to the end of distinguishing between the possible semantics of events that last longer than one timestep duration. For instance, if there is a policy about playing a video like "play video at most twice", this can be understood as (1) play the video twice (regardless of how long it is) and (2) play the video for at most two moments in time. For brevity's sake, however, this distinction is not made in this thesis.

2.2.3. Conditions outside Temporal and First-Order Logic

$eval$ is used for the specification of such conditions. A full language for the respective computation, Γ with possible operations op, is not provided here. One choice is XPath that is also used in the implementation-specific concrete syntax described in Appendix Section B.2. As an example, with $vn_1, vn_2 \in VName$,

$$\begin{aligned}
&\forall\, vn_1 \in TIME : \forall\, vn_2 \in \{EU, US, JP, AU\} :\\
&E(print \mapsto \{obj \mapsto myphoto.jpg, time \mapsto vn_1, location \mapsto vn_2\})\\
&\quad \rightarrow \big(\neg eval(9 \leq vn_1 \leq 17) \wedge eval(vn_2 \neq EU) \;\rightarrow\\
&\qquad\qquad E(log \mapsto \{obj \mapsto myphoto.jpg, time \mapsto vn_1, location \mapsto vn_2\})\big)
\end{aligned}$$

specifies that "the fact that the specific photo is printed outside Europe and the time of printing does not fall between 9 to 17 hours must be logged".

The use of $eval$ makes it possible to specify conditions on (absolute) time and location, one of the type of policies described in the beginning of this chapter. This has, for instance, been implemented for mobile phones to the end of enforcing requirements such as "do not use outside company premises" [52].

The semantics of $eval(\gamma)$ is left unspecified and referred to as

$$[\![eval(\gamma)]\!]_{eval}.$$

2.2.4. Formal Semantics of Events

The semantics of events is defined by $\models_e \subseteq \mathcal{S} \times \Psi$ as follows. For a variable event e that contains a variable $v \in Var$ with $vn \in \mathsf{dom}(v)$ and for $x \in Var(vn)$, let $e[vn \mapsto x]$ denote the result of replacing all occurrences of the variable's name vn in e by the value x. There may be more than one substitution in square brackets. For a term t, let $VarsIn(t)$ denote the set of variables in t. Then, let $Inst_\varepsilon : \mathcal{VE} \rightarrow \mathbb{P}(\mathcal{E})$ define a function that generates all ground substitutions of an event with variables, i.e.,

$$\begin{aligned}
&VarsIn(e) = \{vn_1 \mapsto VS_1, \ldots, vn_k \mapsto VS_k\}\\
&\quad \Rightarrow Inst_\varepsilon(e) = \{e[vn_1 \mapsto vv_1, \ldots, vn_k \mapsto vv_k] : \textstyle\bigwedge_{i=1}^{k} vv_i \in VS_i\}.
\end{aligned}$$

The semantics of possibly variable events is defined as

$$\begin{aligned}
&\forall\, e' \in maxRefEv;\;\; e \in \mathcal{VE}\;\; \exists\, e'' \in \mathcal{E}\\
&\qquad\qquad (e', actual) \models_e E(e) \;\;\Leftrightarrow\;\; e' \; refinesEv \; e'' \wedge e'' \in Inst_\varepsilon(e)\\
&\quad \wedge \;\; (e', intended) \models_e I(e) \;\;\Leftrightarrow\;\; e' \; refinesEv \; e'' \wedge e'' \in Inst_\varepsilon(e).
\end{aligned}$$

2.2.5. First-Order Future-Time Formulae

Overloading the propositional operators, the abstract syntax of the future temporal logic for specifying usage control requirements is defined as

$$\begin{aligned}
\Phi ::= \;\; & (\Phi) \mid \Psi \mid \underline{false} \mid \Phi \; \underline{implies} \; \Phi \mid \underline{forall} \; VName \; \underline{in} \; VVal : \Phi \mid\\
& \Phi \; \underline{until} \; \Phi \mid \Phi \; \underline{after} \; \mathbb{N} \mid \underline{replim}(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi) \mid \underline{repuntil}(\mathbb{N}, \Psi, \Phi)
\end{aligned}$$

with the macros $\underline{\mathit{true}}, \underline{\mathit{not}}(\varphi), \varphi_1\ \underline{\mathit{and}}\ \varphi_2, \varphi_1\ \underline{\mathit{or}}\ \varphi_2, \exists\, vn \in VV : \varphi$. Whenever no confusion between semantics and syntax is possible, in the following the symbols $\rightarrow, \neg, \wedge, \vee, \forall, \exists, \in$ will be used for $\underline{\mathit{implies}}$, $\underline{\mathit{not}}$, $\underline{\mathit{and}}$, $\underline{\mathit{or}}$, $\underline{\mathit{forall}}$, $\underline{\mathit{exists}}$, $\underline{\mathit{in}}$. The intuitive semantics of the propositional and first-order operators is as usual. $\underline{\mathit{until}}$ is the weak until operator of LTL: $\varphi_1\ \ \underline{\mathit{until}}\ \varphi_2$ is true iff φ_1 is true until φ_2 eventually becomes true, or φ_1 is true eternally. $\varphi\ \underline{\mathit{after}}\ n$ is true iff φ becomes true after n timesteps. $\underline{\mathit{replim}}(i, m, n, \psi)$ is true iff ψ is true in-between m and n times in the next i steps. Finally, $\underline{\mathit{repuntil}}(n, \psi, \chi)$ is true iff ψ is true at most n times until χ eventually becomes true, or until eternity if χ never becomes true. It is convenient to add four further shortcuts to the syntax: $\Box\varphi$ for $\varphi\ \underline{\mathit{until}}\ \underline{\mathit{false}}$ specifies that φ will always be true in the future. $i\ \underline{\mathit{repmax}}\ \psi$, defined as $\underline{\mathit{repuntil}}(i, \psi, \underline{\mathit{false}})$, specifies that ψ should be true at most i times in the future. $\psi\ \underline{\mathit{within}}\ i$ for $\underline{\mathit{replim}}(i, 1, i, \psi)$ and $\psi\ \underline{\mathit{during}}\ i$ for $\underline{\mathit{replim}}(i, i, i, \psi)$ for $i \in \mathbb{N}, \varphi \in \Phi, \psi \in \Psi$ describe that ψ will become true at least once or continuously in the next i steps.

Lifting the notation for substitutions of events, for a formula $\varphi \in \Phi$ that contains a variable $vn \in \mathit{Var}$, $\varphi[vn \mapsto vv]$ will denote the result of substituting all occurrences of vn in φ by value $vv \in \mathit{Var}(vn)$.

Formal Semantics The formal semantics is defined by $\models_f \subseteq (\mathit{Trace} \times \mathbb{N}) \times \Phi$ as follows.[1]

$$
\begin{array}{l}
\forall\, s \in \mathit{Trace};\ t \in \mathbb{N};\ \varphi \in \Phi \bullet (s,t) \models_f \varphi \Leftrightarrow \varphi \neq \underline{\mathit{false}} \wedge \\
\quad \big(\exists\, e \in \mathcal{VE} \bullet (\varphi = E(e) \vee \varphi = I(e)) \wedge \exists\, e' \in s(t) : e' \models_e \varphi \\
\quad \vee \exists\, \psi, \chi \in \Phi \bullet \varphi = \psi\ \underline{\mathit{implies}}\ \chi \wedge \neg((s,t) \models_f \psi) \vee (s,t) \models_f \chi \\
\quad \vee \exists\, \gamma \in \Gamma \bullet \varphi = \mathit{eval}(\gamma) \wedge [\![\varphi]\!]_{\mathit{eval}} = \mathit{true} \\
\quad \vee \exists\, vn \in \mathit{VName};\ vs \in \mathit{VVal};\ \psi \in \Phi \bullet \\
\qquad \varphi = (\underline{\mathit{forall}}\ vn\ \underline{\mathit{in}}\ vs : \psi) \wedge \forall\, vv \in vs \bullet (s,t) \models_f \psi[vn \mapsto vv] \\
\quad \vee \exists\, \psi, \chi \in \Phi \bullet \varphi = \psi\ \underline{\mathit{until}}\ \chi \wedge \big(\forall\, v \in \mathbb{N} \bullet t \leq v \Rightarrow (s,v) \models_f \psi \\
\qquad \vee \exists\, u \in \mathbb{N} \bullet t < u \wedge (s,u) \models_f \chi \wedge \forall\, v \in \mathbb{N} \bullet t \leq v < u \Rightarrow (s,v) \models_f \psi\big) \\
\quad \vee \exists\, n \in \mathbb{N};\ \psi \in \Phi \bullet \varphi = \psi\ \underline{\mathit{after}}\ n \wedge (s, t+n) \models_f \psi \\
\quad \vee \exists\, n \in \mathbb{N}_1;\ l, r \in \mathbb{N};\ \psi \in \Psi \bullet \\
\qquad \varphi = \underline{\mathit{replim}}(n, l, r, \psi) \wedge \\
\qquad\quad l \leq \#\{j \in \mathbb{N}_1 \mid j \leq n \wedge (s, t+j) \models_f \psi\} \leq r \\
\quad \vee \exists\, n \in \mathbb{N};\ \psi \in \Psi, \chi \in \Phi \bullet \varphi = \underline{\mathit{repuntil}}(n, \psi, \chi) \\
\qquad \wedge \big((\exists\, u \in \mathbb{N}_1 \bullet (s, t+u) \models_f \chi \wedge (\forall\, v \in \mathbb{N}_1 \bullet v < u \Rightarrow \neg((s, t+v) \models_f \chi)) \\
\qquad\quad \wedge (\#\{j \in \mathbb{N}_1 \mid j \leq u \wedge s(t+j) \models_f \psi\}) \leq n) \\
\qquad\quad \vee (\#\{j \in \mathbb{N}_1 \mid s(t+j) \models_f \psi\}) \leq n\big)\big)
\end{array}
$$

This section described a future-time policy language that will be used in this thesis for the specification of policies. The next section describes a past-time variant of this language for the implementation-level policies.

[1]Counting has been simplified for just one event per timestep. E.g., in the semantics of $\underline{\mathit{replim}}$, we evaluate in how many timesteps the specified formula is true. The advantage of such semantics is that ψ is not limited to intended and actual events, but may contain any proposition. The disadvantage is that this way, we are not able to count how often the same event happened within one timestep. For real implementations, this is not a limitation because there is usually only one event per timestep.

2.3. Implementation-Level Policies

Specification-level policies specify what may or must not be done with data. Implementation-level policies (ILP), specify how this should be done. In general, this can be achieved by inhibition, modification, and execution, as introduced with examples in Section 1.2.2.1.

ILPs for preventive enforcement (detective enforcement is discussed in Section 2.5) can be described as event-condition-action rules [54]. The trigger event is an intended event, $I(x)$; the condition is a formula in a past variant of the specification language ($\varphi \in \Phi^-$, introduced and described in Section 2.3.1); and the action inhibits or modifies the trigger event or executes additional events [25]. Formally, an ILP is defined as follows. In case the trigger is detected—an intended event $I(x)$—and if turning the intended event into an actual event made the condition φ true, then one of the possible actions is performed.

- Inhibition is specified as $\neg E(x)$.
- Modification means $E(x)$ is inhibited, but further events, $\bigwedge_{i=1}^{n_k} I(x_i)$, may be performed. Since these events themselves can be subject to treatment by ILPs, they are specified as intended and not actual events.
- Finally, execution means the trigger may be executed if no other ILPs prohibit this, and a further set of events, $\bigwedge_{i=1}^{n_k} I(x_i)$, is executed in addition.

2.3.1. Past-Time Conditions

Conditions for ILPs are specified in a past temporal logic, Φ^- defined as

$$\begin{aligned}\Phi^- ::= \; & (\Phi^-) \mid \Psi \mid \underline{\mathit{false}}^- \mid \Phi^- \; \underline{\mathit{implies}}^- \; \Phi^- \mid \underline{\mathit{forall}} \; \mathit{VName} \; \underline{\mathit{in}} \; \mathit{VVal} : \Phi^- \mid \\ & \Phi^- \; \underline{\mathit{since}}^- \; \Phi^- \mid \Phi^- \; \underline{\mathit{before}}^- \; \mathbb{N} \mid \underline{\mathit{replim}}^-(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi) \mid \\ & \underline{\mathit{repsince}}^-(\mathbb{N}, \Psi, \Phi^-)\end{aligned}$$

plus the shortcuts $\underline{\mathit{true}}^-$, $\underline{\mathit{not}}^-$, $\underline{\mathit{and}}^-$, $\underline{\mathit{or}}^-$, $\underline{\mathit{exists}}$. Once again, if no confusion between semantics and syntax is possible, in the following the symbols $\rightarrow, \neg, \wedge, \vee, \forall, \exists, \in$ will be used for $\underline{\mathit{implies}}^-$, $\underline{\mathit{not}}^-$, $\underline{\mathit{and}}^-$, $\underline{\mathit{or}}^-$, $\underline{\mathit{forall}}$, $\underline{\mathit{exists}}$, $\underline{\mathit{in}}$. $\boxminus\varphi$ is true if φ is true in all timesteps before and including the current timestep. As in the future case, it is convenient to add three further shortcuts to the syntax: for $i \in \mathbb{N}, \varphi \in \Phi$ and $\psi \in \Psi$, $\psi \; \underline{\mathit{repmax}}^- \; i \leftrightarrow \underline{\mathit{repsince}}^-(i, \psi, \underline{\mathit{false}}^-)$ stipulates that ψ must have happened at most i times in the past; $\psi \; \underline{\mathit{within}}^- \; i \leftrightarrow \underline{\mathit{replim}}^-(i, 1, i, \psi)$ stipulates that ψ has happened at least once in the past i steps; and $\psi \; \underline{\mathit{during}}^- \; i \leftrightarrow \underline{\mathit{replim}}^-(i, i, i, \psi)$ stipulates that ψ must have been true in each of the past i steps.

Lifting the notation for substitutions of events, for a formula $\varphi \in \Phi^-$ that contains a variable $vn \in \mathit{Var}$, $\varphi[vn \mapsto vv]$ will denote the result of substituting all occurrences of vn in φ by value $vv \in \mathit{Var}(vn)$. The semantics of Φ^- is defined by $\models_{f^-} \subseteq (\mathit{Trace} \times \mathbb{N}) \times \Phi^-$ that will be used in an infix manner:

$$\begin{array}{l}
\forall\, s \in \mathit{Trace};\ t \in \mathbb{N};\ \pi \in \Phi_i^- \bullet (s,t) \models_{f^-} \pi \Leftrightarrow (\pi \neq \underline{\mathit{false}}^-) \wedge \\
\quad \big(\exists\, e \in \mathcal{VE} \bullet (\pi = E(e) \vee \pi = I(e)) \wedge \exists\, e' \in s(t) : e' \models_e \pi \\
\vee \quad \exists\, \psi, \chi \in \Phi_i^- \bullet \pi = \psi\ \underline{\mathit{implies}}^-\ \chi \wedge \neg((s,t) \models_{f^-} \psi) \vee (s,t) \models_{f^-} \chi \\
\vee \quad \exists\, \gamma \in \Gamma \bullet \pi = \mathit{eval}(\gamma) \wedge [\![\pi]\!]_{\mathit{eval}} = \mathit{true} \\
\vee \quad \exists\, \mathit{vn} \in \mathit{VName};\ \mathit{vs} \in \mathit{VVal};\ \psi \in \Phi_i^- \bullet \\
\qquad \pi = (\underline{\mathit{forall}}\ \mathit{vn}\ \underline{\mathit{in}}\ \mathit{vs} : \psi) \wedge \forall\, \mathit{vv} \in \mathit{vs} \bullet (s,t) \models_{f^-} \psi[\mathit{vn} \mapsto \mathit{vv}] \\
\vee \quad \exists\, \psi, \chi \in \Phi_i^- \bullet \pi = \psi\ \underline{\mathit{since}}^-\ \chi \wedge \big((\forall\, v \in \mathbb{N} \bullet v \leq t \Rightarrow (s,v) \models_{f^-} \psi) \\
\qquad \vee (\exists\, u \in \mathbb{N} \bullet u \leq t \wedge (s,u) \models_{f^-} \chi \wedge \forall\, v \in \mathbb{N} \bullet u < v \leq t \Rightarrow (s,v) \models_{f^-} \psi)\big) \\
\vee \quad \exists\, n \in \mathbb{N};\ \psi \in \Phi_i^- \bullet \pi = \psi\ \underline{\mathit{before}}^-\ n \wedge t \geq n \wedge (s, t-n) \models_{f^-} \psi \\
\vee \quad \exists\, n, l, r \in \mathbb{N};\ \psi \in \Psi;\ \bullet \varphi = \underline{\mathit{replim}}^-(n, l, r, \psi) \\
\qquad \wedge\ l \leq (\#\{j \in \mathbb{N} \mid j \leq \mathit{min}(n,t) \wedge s(t-j) \models_{f^-} \psi\}) \leq r \\
\vee \quad \exists\, n \in \mathbb{N};\ \psi \in \Psi;\ \chi \in \Phi;\ e \in \mathcal{E} \bullet \varphi = \underline{\mathit{repsince}}^-(n, \psi, \chi) \\
\quad \wedge \big((\exists\, u \in \mathbb{N}_1 \bullet t \geq u \wedge (s, t-u) \models_{f^-} \chi \wedge (\forall\, v \in \mathbb{N} \bullet v < u \Rightarrow \neg((s, t-v) \models_{f^-} \chi)) \\
\qquad \wedge (\#\{j \in \mathbb{N} \mid j \leq u \wedge s(t-j) \models_{f^-} \psi\} \leq n)) \\
\quad \vee (\#\{j \in \mathbb{N} \mid j \leq t \wedge s(t-j) \models_{f^-} \psi\} \leq n)\big)\big)
\end{array}$$

As already mentioned, implementation-level policies are of the form event-condition-action (ECA) rules described next:

2.3.2. ECA Rules

Let $t \subseteq \mathcal{VE} \times \Phi^-$, defined as

$$t(\mathit{ve}, \varphi) \Leftrightarrow \big(I(\mathit{ve}) \wedge (E(\mathit{ve}) \rightarrow \varphi)\big) \tag{2.1}$$

denote the situation (trigger event ve and condition $\varphi \in \Phi^-$) under which a specific ILP is supposed to fire. Three different kinds of ILPs are typed as $m_{\mathit{inh}} \subseteq \mathcal{VE} \times \Phi^-$, $m_{\mathit{mod}} \subseteq \mathcal{VE} \times \Phi^- \times \mathbb{P}(\mathcal{VE})$, $m_{\mathit{exc}} \subseteq \mathcal{VE} \times \Phi^- \times \mathbb{P}(\mathcal{VE})$. Using $\forall\, \mathit{VarsIn}(\varphi)$ as shortcut for $\forall\, \mathit{vn}_1 \in \mathit{VV}_1 : \ldots : \forall\, \mathit{vn}_k \in \mathit{VV}_k$ if $\mathit{VarsIn}(\varphi) = \{\mathit{vn}_1, \ldots, \mathit{vn}_k\}$ and $\mathit{Var}(\mathit{vn}_i) = \mathit{VV}_i$ for all $i \leq k$, ILPs are defined as

$$\begin{array}{lcll}
m_{\mathit{inh}}(\mathit{ve}, \varphi) & \Leftrightarrow & \forall\, \mathit{VarsIn}(\mathit{ve}) : & t(\mathit{ve}, \varphi) \rightarrow \neg E(\mathit{ve}) \\
m_{\mathit{mod}}(\mathit{ve}, \varphi, \mathit{Mod}) & \Leftrightarrow & \forall\, \mathit{VarsIn}(\mathit{ve}) : & t(\mathit{ve}, \varphi) \rightarrow (\neg E(\mathit{ve}) \wedge m_{\mathit{exc}}(\mathit{ve}, \varphi, \mathit{Mod})) \\
m_{\mathit{exc}}(\mathit{ve}, \varphi, \mathit{Exc}) & \Leftrightarrow & \forall\, \mathit{VarsIn}(\mathit{ve}) : & t(\mathit{ve}, \varphi) \rightarrow \bigwedge_{x_i \in \mathit{Exc}} I(x_i).
\end{array}$$

Note that inhibitors can be expressed as modifiers via $m_{\mathit{inh}}(\mathit{ve}, \varphi) \leftrightarrow m_{\mathit{mod}}(\mathit{ve}, \varphi, \varnothing)$. For the sake of readability, a slightly redundant notation has been adopted here. For fixed sets of n_1 inhibiting, n_2 modifying and n_3 executing ILPs, their composition computes to

$$M \leftrightarrow \bigwedge_{i=1}^{n_1} m_{\mathit{inh}}(\mathit{ve}_i^{\mathit{inh}}, \varphi_i^{\mathit{inh}}) \wedge \bigwedge_{i=1}^{n_2} m_{\mathit{mod}}(\mathit{ve}_i^{\mathit{mod}}, \varphi_i^{\mathit{mod}}, \mathit{Mod}_i) \wedge \bigwedge_{i=1}^{n_3} m_{\mathit{exc}}(\mathit{ve}_i^{\mathit{exc}}, \varphi_i^{\mathit{exc}}, \mathit{Exc}_i). \tag{2.2}$$

2.3.3. Default Behavior

The final step is the conversion of intended to actual events in case no other ILP forbids them (in other words, the default policy is allow on all events). If there is an intended event $I(e)$, in a real system, it is always maximally refined, $e \in \mathit{maxRefEv}$. It should be allowed, i.e., $E(e)$ should be executed, if no modifying or inhibiting ILP fires. Such an ILP fires if e refines any ground substitution of its trigger event and, at the same time, makes the ILP's condition true. Taken together, the default rule is expressed as follows. If there is a maximally refined intended event, then there either must be a corresponding actual event; or a modifying or inhibiting ILP is triggered, which would prohibit the corresponding actual event. Formally,

$$M_{\mathit{default}} \leftrightarrow \bigwedge_{e \in \mathit{maxRefEv}} I(e) \rightarrow \Big(E(e) \vee \bigvee_{\substack{(ve,\varphi):\; M \rightarrow m_{\mathit{inh}}(ve,\varphi) \\ \vee\; M \rightarrow m_{\mathit{mod}}(ve,\varphi,\mathit{Mod})}} \exists\, \mathit{VarsIn}(ve) : e \ \mathit{refinesEv}\ ve \wedge \varphi\Big) \tag{2.3}$$

where the ILP definitions are assumed to contain pairwise mutually disjoint set of variables.

2.3.4. Composition and Mechanisms

The composition of a set of ILPs is defined by

$$M_{\mathit{complete}} \leftrightarrow M \wedge M_{\mathit{default}}. \tag{2.4}$$

In terms of the architecture described in Chapter 7, the Policy Decision Point (PDP) evaluates the instantiations of Equation 2.1 (which in part also occur in Equation 2.3), and the Policy Enforcement Point (PEP) implements the right hand sides of the instances of Equation 2.3 and the "allow" case—where an intended event is transformed into an actual event—in Equation 2.3.

The formula $\Box(M_{\mathit{complete}})$ defines the semantics of all combined ILPs in a system.

ILPs and default rules can be implemented using runtime verification or complex event processing technology. When deployed in a usage control context, such monitoring technology will be referred to as "mechanism" in the remainder of this article. While strictly speaking the composition of ILPs (predicates M_{complete}) configures mechanisms, we will use the two terms interchangeably in the following. This also explains why we denote ILPs by M and m.

For the sake of conciseness, the above definition of ILPs slightly simplifies the original definition [23]: The original definition allows for the specification of multiple subsequent events to be performed by modifying or executing ILPs.

2.3.5. Free Variables

In contrast to specification-level policies, ILP specifications may contain free variables. When containing free variables, their semantics is undefined. However, they can then be *configured* by replacing all free variables by constants which makes sure a semantics is defined. ILPs with free variables act as templates for an entire set of ILPs that can be configured, among other things, at runtime.

The next section shows by example how one SLP corresponds to several ILPs with different enforcement strategies.

2.3.6. Policies by Examples

The following subsections demonstrate the different enforcement strategies for the use case of the running example introduced in Section 1.4.2.

2.3.6.1. Example 1

In the first example, Alice specifies a policy for controlling the copies of her personal data by her friends. The specification-level policy is "don't copy my personal data", formally expressed as

$$\forall\ v \in \mathit{PERSDATA} : \Box(\neg(E(\mathit{copy} \mapsto \{\mathit{obj} \mapsto v\})))$$

- Enforcement by inhibition can be performed by blindly blocking all attempts to copy personal data. Formally, for $v \in \mathit{VN}$,

$$m_{\mathit{inh}}(\mathit{copy} \mapsto \{\mathit{obj} \mapsto v\})$$

- For enforcement by modification, let us assume for simplicity's sake that personal data contain two fields, email and birth date, and that anonymization means that these fields have been replaced by XXX, as expressed by proposition $\mathit{anonymized}(\cdot)$. Enforcement by modification can be performed by transforming the email and birth date fields of non-anonymized data into XXX. Formally, with $v \in \mathit{VN}$,

$$\begin{aligned} m_{\mathit{mod}}(\quad & \mathit{copy} \mapsto \{\mathit{obj} \mapsto v\}, \mathit{eval}(\mathit{anonymized}(v) == \mathit{false}), \\ & \{\mathit{copy} \mapsto \{\mathit{obj} \mapsto \mathit{eval}(\mathit{removeEmailBirthday}(v))\}\}) \end{aligned}$$

  where proposition $\mathit{removeEmailBirthday}$ performs the anonymization. Note that the condition could also be set to true which would make the ILP anonymize data that may well have been anonymized before.

- Enforcement by execution can be performed by sending a notification message to Alice whenever data is to be copied. Formally, with $v \in \mathit{VN}$,

$$\begin{aligned} m_{\mathit{exc}}(\quad & \mathit{copy} \mapsto \{\mathit{obj} \mapsto v\}, \\ & \{\mathit{notify} \mapsto \{\mathit{rcvr} \mapsto \mathit{Alice}, \mathit{obj} \mapsto v\}\}). \end{aligned}$$

2.3.6.2. Example 2

In the second example, Alice wants to stipulate that a video shared by her, $\mathit{myvid.mp}4$, must be paid for in order to be played with good quality. A respective specification-level policy is

$$\begin{aligned} \big(\forall\, q \in \{\bot_p, \ldots, \top_p\} : \quad & \neg(E(\mathit{play} \mapsto \{\mathit{obj} \mapsto \mathit{myvid.mp}4, \mathit{qual} \mapsto q\}) \wedge \mathit{eval}(q > 50))\big) \\ & \underline{\mathit{until}}\ E(\mathit{pay} \mapsto \{\mathit{obj} \mapsto \mathit{myvid.mp}4, \mathit{amount} \mapsto 10\}) \end{aligned}$$

where $\bot_p$ and $\top_p$ are respectively the bottom and top elements of the lattice $(PValue, \leq_p)$ imposed on parameter values.

- An *inhibiting* ILP that makes sure an unpaid video is not played with good quality is

$$\begin{aligned} m_{inh}(\quad & play \mapsto \{obj \mapsto myvid.mp4, qual \mapsto q\}, \\ & eval(q > 50) \wedge (\boxdot(\neg E(pay \mapsto \{obj \mapsto myvid.mp4\}))\ \underline{before}^{-}\ 1)). \end{aligned}$$

- A *modifying* ILP that reduces quality to 50% is defined by

$$\begin{aligned} m_{mod}(\quad & play \mapsto \{obj \mapsto myvid.mp4, qual \mapsto q\}, \\ & eval(q > 50) \wedge \boxdot(\neg E(pay \mapsto \{obj \mapsto myvid.mp4\}))\ \underline{before}^{-}\ 1, \\ & \{play \mapsto \{obj \mapsto myvid.mp4, qual \mapsto 50\}\}). \end{aligned}$$

- Finally, an *executing* ILP could trigger the automated payment of the video:

$$\begin{aligned} m_{exc}(\quad & play \mapsto \{obj \mapsto myvid.mp4\}, \\ & \boxdot(\neg E(pay \mapsto \{obj \mapsto myvid.mp4\}))\ \underline{before}^{-}\ 1, \\ & \{pay \mapsto \{obj \mapsto myvid.mp4, amount \mapsto 10\}\}). \end{aligned}$$

2.4. Policy Activation and Violation

Defining the semantics of combined ILPs as $\Box(M_{complete})$, as suggested in Section 2.3.4, does not state from which point onwards this formula should hold. A simplistic approach is to assume that all ILPs are active from time point 0 onwards. In reality, with policy livecycle management, for an ILP π with name $\overline{\pi}$, there would be activation and deactivation events parameterized by the name of the ILP and an abstract moment in time, t. The semantics of a ILP would then amount to

$$\Box(activate(\overline{\pi}, t) \implies \pi\ \underline{until}\ deactivate(\overline{\pi})),$$

starting at activation time t, and the operators of Φ^{-} would have to be augmented by a further parameter t and adjusted to "look back" only until $\max(0, t)$. In deriving past-time rules from future-time SLPs, a construct $\mathcal{S}tart$ is used to represent policy activation in Section 5.2.

In terms of policy violation, it must be stated what should happen if a policy has been violated for the first time. For instance, we could assume that once a policy is violated (that is, the formula evaluates to *false*), it remains violated forever; or we could issue penalties and upon payment, reset the state of the formula to start "fresh" monitoring and allow further violations; or we could specify meta-policies that would specify constraints upon the maximum number of times a policy could be violated before either the violation becomes a norm (create new usage control rules learning from policy violation patterns) or the data consumer is blacklisted.

2.5. Detective and Preventive Enforcement

Inhibitors and modifiers are used for preventive enforcement: their goal is to ensure that a policy is adhered to. Executors can be used for both preventive and detective enforcement. Examples for preventive enforcement have been provided in Section 2.3.6. Executors can be instrumental for detective enforcement as well: the event to be executed then is a mere logging event. There also is a second possibility to implement detective enforcement that directly uses specification-level, i.e., future-time policies rather than the ILPs. For safety properties—and all realistic usage control properties are safety properties because in realistic systems all duties are time-bounded—there exists an earliest moment in time when a policy is violated or satisfied forever. The respective shortest "good" and "bad" prefixes can be precisely defined and detected at runtime [55, 56]. If the violation of a specification-level policy is detected by using this technology, then any kind of compensating action can be undertaken. These include undoing an action, lowering trust ratings, implementing penalties, etc. Formal details are provided elsewhere [53].

2.6. Discussion

In this work, though the policies are specified in the future variant of OSL, it is also possible to express usage control requirements in past-time OSL. Let us consider an SLP "do not use outdated data" where *outdated data* means that the data was not updated in the last 3 days. This SLP can be expressed in future OSL as:

$$\Box(E(\mathit{update} \mapsto \{\mathit{obj} \mapsto d\}) \wedge \underline{\mathit{within}}(3, E(\mathit{use} \mapsto \{\mathit{obj} \mapsto d\})) \vee \neg E(\mathit{use} \mapsto \{\mathit{obj} \mapsto d\}))$$

The same policy can be expressed in past OSL as:

$$\boxdot(E(\mathit{use} \mapsto \{\mathit{obj} \mapsto d\}) \to \underline{\mathit{within}}^{-}(3, E(\mathit{update} \mapsto \{\mathit{obj} \mapsto d\})))$$

Specification-level policies are described in future-time OSL because intuitively it makes more sense to use future tense to talk about events that will take place only later. However, a decision to use future over past temporal logic for the specification of policies is driven by intuition and taste. We know that the two (the future logic and the past logic) are equivalent in the sense that everything that can be expressed in one can also be expressed in the other [57, 58]. Therefore, the specification of usage control policies is not limited to using future-time OSL.

Chapter 3

Data Flow Tracking for Usage Control

Major parts of this chapter are verbatim from our yet-to-be-published book on usage control [2] and have not been written by the author of this thesis.

As introduced in Chapter 1, *data* (e.g., a photo) exists within a machine in form of several representations that possibly reside at different layers of abstraction (e.g., a file at the operating system layer, a set of pixels at the windowing manager layer, a browser object at the application layer etc.). These representations are called *containers*. However, from an end user's point of view, data is an abstract concept with an intuitive set of actions relevant to it (copy, print, delete, etc. a photo). Therefore, it is natural for end users to specify usage control policies that address data ("delete this *photo*") rather than events on layer-specific representations ("unlink *file*$_1$, *file*$_2$,..., *file*$_n$, empty clipboard, delete $mail_{xyz}$ from archive, erase $backup_{abc}$, etc."). For enforcement of policies on data, a system must distinguish between data and containers. Usage control decisions should then take into account the dissemination of data among different containers. For instance, "delete this photo" should be enforced by deleting every container that contains this photo.

In the previous chapter, usage control policies were specified as constraints on events that target objects without distinguishing between the abstract data and its concrete representations. In this chapter, a formal distinction between the two is introduced (in Section 3.2) and the usage control model is augmented with data flow tracking capabilities at one arbitrary layer of abstraction in a machine (in Section 3.3). Building upon this distinction, the policy language is extended to also specify usage control policies as constraints upon system states (in Section 3.4).

3.1. Data Flow Preliminaries

Data flows are considered in systems that are described as tuples $(\mathcal{P}, \mathcal{D}, \mathcal{S}, \mathcal{C}, F, \Sigma, \sigma_i, \mathcal{R})$ where $\mathcal{P}$ is a set of principals triggering system events, $\mathcal{D}$ is a set of data elements, $\mathcal{S}$ is the set of system events introduced in Section 2.1.2, and $\mathcal{C}$ is a set of containers.

Data flow states Σ are defined by three mappings: a *storage function* of type $\mathcal{C} \to \mathbb{P}(\mathcal{D})$ that describes which set of data is potentially stored in which container; an *alias function* of type $\mathcal{C} \to \mathbb{P}(\mathcal{C})$ 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 $F \to \mathcal{C}$ for a set of identifiers, F. We need identifiers to correctly model renaming activities. Thus the data flow state of an arbitrary system is defined as:

$$\Sigma = (\mathcal{C} \to \mathbb{P}(\mathcal{D})) \times (\mathcal{C} \to \mathbb{P}(\mathcal{C})) \times (F \to \mathcal{C}).$$

σ_i is the initial state of the system where containers are mapped to empty sets of data, containers and identifiers.

The transition relation $\mathcal{R} \subseteq \Sigma \times \mathbb{P}(\mathcal{S}) \to \Sigma$ is the core of the data-flow tracking model. It encodes how the execution of events affects the dissemination of data in the system. At runtime, the usage control infrastructure maintains the data state Σ. Events are intercepted, and the information state is updated according to $\mathcal{R}$. In the following, it is assumed that the principals executing events are provided as a parameter of the event itself.

For simplicity's sake, let us assume that the events happening at the same timestep are *independent* of each other such that any possible serialization of them would lead to the same state:

$$\begin{aligned}
&\forall\, \sigma \in \Sigma : \mathcal{R}(\sigma, \varnothing) = \sigma \\
&\forall\, \sigma \in \Sigma;\ t \in \mathit{Trace}, n \in \mathbb{N};\ \mathit{Es} \subseteq t(n);\ e \in t(n) : \mathcal{R}(\sigma, \mathit{Es}) = \mathcal{R}(\mathcal{R}(\sigma, \{e\}), \mathit{Es} \setminus \{e\})
\end{aligned}$$

This is to say that if the order in which some events take place matters (e.g., "copy A to B" and "delete A"), it is assumed that such events take place in different subsequent timesteps.[1]

In the usage control model of Chapter 2, data is addressed by referring to specific representations of it as event parameters. For instance, the policy described in Example 2, Section 2.3.6.2 stipulates that if a file (a specific representation and a specific container) called *myvid.mp4* has not been paid for, it cannot be played with good quality. With data flow concepts, the situation where a copy of that file, *myvid2.mp4*, should not be played either is addressed. To this end, the semantic model is extended by *data usages* that allow to specify protection requirements for all representations rather than just one. Using the data flow tracking model, at each moment in time t, the current data state of the system is computed as follows: the usage control model's system trace is considered until t, the respective events in each step are extracted, the successor data states for each data state is iteratively computed and eventually we get the data state at time t. In an implementation, of course, the recursive computation is done using state machines to record the data state of a system at each moment in time (Chapter 7).

[1]This assumption may sound restrictive, but when it comes to concrete implementations, it is seldom the case that multiple events take place at the same time; usually each event has a unique timestep that allow us to establish a clear order between them.

3.2. Data, Containers and Events

We need to distinguish between *data* and *containers for data*. At the specification level, this leads to the distinction between two classes of events according to the "type" of the *obj* parameter: events of class *dataUsage* define actions on data objects. The intuition is that these pertain to *every representation* of the data. In contrast, events of class *containerUsage* refer to one single container. In a real system, only events of class containerUsage can happen. This is because each monitored event in a trace is related to a specific representation of the data (a file, a memory region, etc). dataUsage events are used only in the specification of policies (SLPs), where it is possible to define a rule abstracting from the specific representation of a data item. There also a third class of events, $noUsage$ events, that do not define usages of containers or data. $noUsage$ events include notifications, penalties, etc. A function $getEclass$ that extracts if an event is a data or a container usage or neither is defined as:[2]

$$\begin{aligned}
&EventClass = \{dataUsage, containerUsage, noUsage\}\\
&getEclass : \mathcal{E} \rightarrow EventClass\\
&\mathcal{D} \cup \mathcal{C} \subseteq PValue \quad \wedge \quad \mathcal{C} \cap \mathcal{D} = \varnothing
\end{aligned}$$

The system must satisfy the following conditions:

$$\begin{aligned}
&\forall\, e \in \mathcal{E} : getEclass(e) = dataUsage \iff \exists\, par \in \mathcal{D} : (obj \mapsto par) \in e.p\\
&\forall\, e \in \mathcal{E} : getEclass(e) = containerUsage \iff \exists\, par \in \mathcal{C} : (obj \mapsto par) \in e.p.
\end{aligned}$$

We have seen in Section 2.1.2 that unmentioned parameters are implicitly included in events specified in policies by using relation $refinesEv$. For example, if an obligation prohibits the event $copy \mapsto \{obj \mapsto objB\}$, then the event $copy \mapsto \{obj \mapsto objB, device \mapsto dev123\}$ is prohibited as well. Now this definition of event refinement can be extended as follows: an event of class dataUsage can be refined by an event of class containerUsage if the latter is related to a specific representation of the data the former refers to. As in the original definition, in both cases the more refined event can have more parameters than the more abstract event. An event e_2 refines an event e_1 iff

- e_1 and e_2 both have the same class (containerUsage or dataUsage) and
- e_2 $refinesEv$ e_1;

or

- if e_1 is a dataUsage and e_2 a containerUsage event,
- e_1 and e_2 have the same event name,

[2] Note that a *dataUsage* event addresses all the different representations of data at once. Therefore, a dataUsage event could be seen as a special case of a variable event with the condition that the variable object can take only those values from the set of containers where the concerned data resides. On the other hand, the set of values over which the codomain of a variable ranges in a variable event, is fixed by definition. Whereas, in case of a dataUsage event, the set of containers from which the object can take its value varies according to the data and container mappings.

- in the current data flow state, there exists a data item *d* stored in a container *c* such that $(obj \mapsto d) \in e_1.p$ and $(obj \mapsto c) \in e_2.p$, and
- all parameters (except for *obj*) of e_1 have the same value in e_2, and e_2 may possibly have additional parameters.

Formally, relation

$$\mathit{refinesEv}_i \subseteq (\mathcal{E} \times \Sigma) \times \mathcal{E}$$

checks whether one event e_2 refines another event e_1 also w.r.t. data and containers. Σ is needed to access the current data state:

$$\begin{aligned}
&\forall\, e_1, e_2 \in \mathcal{E}, \forall\, \sigma \in \Sigma : (e_2, \sigma)\ \mathit{refinesEv}_i\ e_1 \iff \\
&\quad (\mathit{getEclass}(e_1) = \mathit{getEclass}(e_2) \wedge e_2\ \mathit{refinesEv}\ e_1) \vee \\
&\quad (e_1.n = e_2.n \wedge \mathit{getEclass}(e_1) = \mathit{dataUsage} \wedge \mathit{getEclass}(e_2) = \mathit{containerUsage} \wedge \\
&\qquad (\exists\, d \in \mathcal{D}, \exists\, c \in \mathcal{C} :\ d \in \sigma.s(c) \wedge \\
&\qquad obj \mapsto d \in e_1.p \wedge obj \mapsto c \in e_2.p \wedge e_1.p \setminus \{obj \mapsto d\} \subseteq e_2.p \setminus \{obj \mapsto c\})).
\end{aligned}$$

where $\sigma.s$ denotes the storage function of state σ.

3.3. Computing the Data State

We use the function $\mathit{states} : (\mathit{Trace} \times \mathbb{N}) \rightarrow \Sigma$ to compute the data flow state σ at a given moment in time.

$$\begin{aligned}
&\mathit{states}(t, 0) = \sigma_i \\
&n > 0 \implies \mathit{states}(t, n) = \mathcal{R}(\mathit{states}(t, n-1), t(n-1))
\end{aligned}$$

Note that $\mathit{states}(t, n)$ does not consider events in $t(n)$ because the policy decision of events at time n requires the data-flow state of the system after the execution of all events in t *before* time n. With the help of *refinesEv*$_i$ and *states*, the satisfaction relation for event expressions in the context of data and container usages is defined by adding one argument to $\models_e$ and obtaining $\models_{e,i} \subseteq (\mathcal{S} \times \Sigma) \times \Psi$ as follows:

$$\begin{aligned}
&\forall\, e' \in \mathit{maxRefEv};\ e \in \mathcal{VE};\ \sigma \in \Sigma;\ \exists\, e'' \in \mathcal{E} : \\
&\qquad ((e', \mathit{actual}), \sigma) \models_{e,i} E(e) \iff (e', \sigma)\ \mathit{refinesEv}_i\ e'' \wedge e'' \in \mathit{Inst}_\varepsilon(e) \\
&\quad \wedge\ ((e', \mathit{intended}), \sigma) \models_{e,i} I(e) \iff (e', \sigma)\ \mathit{refinesEv}_i\ e'' \wedge e'' \in \mathit{Inst}_\varepsilon(e).
\end{aligned}$$

3.4. State-Based Operators

In the semantic model, policies are defined on sequences of events. The motive is to describe certain situations that must be avoided or enforced. However, in realistic scenarios, there might be infinitely many sequences of events that lead to the same situation, e.g., the copy or the deletion of a file. Instead of listing all these sequences, it appears more convenient in situations of this kind to define a policy based on the description of the (data flow state of the) system at that specific moment. To define such type of policies, a new set of *state-based operators*, Φ_s are introduced:

$$\Phi_s ::= \underline{isNotIn}(\mathcal{D}, \mathbb{P}(\mathcal{C})) \mid \underline{isCombinedWith}(\mathcal{D}, \mathcal{D})$$

plus the macro $\underline{isOnlyIn}(\mathcal{D}, \mathbb{P}(\mathcal{C}))$ with $\underline{isOnlyIn}(d, Cs) \Leftrightarrow \underline{isNotIn}(d, \mathcal{C} \setminus Cs)$, and define

$$\Phi_i ::= \Phi \mid \Phi_s.$$

Intuitively, $\underline{isNotIn}(d, Cs)$ is true if data *d* is not present in any of the containers in set Cs. This is useful to express constraints such as "video v must not be distributed over the network," which becomes $\Box(\underline{isNotIn}(v, \{c_{net}\}))$ for a network container (any socket) c_{net}. $\underline{isCombinedWith}(d_1, d_2)$ checks if data items d_1 and d_2 are combined in any container. This is useful to express simple Chinese Wall policies. $\underline{isOnlyIn}$, the dual of $\underline{isNotIn}$, is used to express concepts such as "data *d* has been deleted:" $\underline{isOnlyIn}(d, \varnothing)$.

Leveraging the *states* function defined before, the semantics of the specific data usage operators in Φ_s is defined with semantics $\models_s \subseteq (Trace \times \mathbb{N}) \times \Phi_s$:

$$\begin{aligned}
&\forall\, t \in Trace;\ n \in \mathbb{N};\ \varphi \in \Phi_s;\ \sigma \in \Sigma :\ (t, n) \models_s \varphi \iff \sigma = states(t, n) \wedge \\
&\quad \exists\, d \in \mathcal{D}, Cs \subseteq \mathcal{C} : \varphi = \underline{isNotIn}(d, Cs) \wedge \forall\, c' \in \mathcal{C} : \\
&\qquad d \in \sigma.s(c') \implies c' \notin Cs \\
&\quad \vee \exists\, d_1, d_2 \in \mathcal{D} : \varphi = \underline{isCombinedWith}(d_1, d_2) \wedge \exists\, c' \in \mathcal{C} : \\
&\qquad d_1 \in \sigma.s(c') \wedge d_2 \in \sigma.s(c').
\end{aligned}$$

The language Φ is now augmented with the state based operators in Φ_s, obtaining the new language Φ_i

$$\begin{aligned}
\Phi_i ::=\ & (\Phi_i) \mid \Psi \mid \underline{false} \mid \Phi_i\ \underline{implies}\ \Phi_i \mid \underline{forall}\ VName\ \underline{in}\ VVal : \Phi \mid \\
& \Phi_i\ \underline{until}\ \Phi_i \mid \Phi_i\ \underline{after}\ \mathbb{N} \mid \underline{replim}(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi) \mid \underline{repuntil}(\mathbb{N}, \Psi, \Phi_i) \mid \\
& \Phi_s
\end{aligned}$$

and its semantics $\models_{f,s} \subseteq (Trace \times \mathbb{N}) \times \Phi_i$

$$\begin{aligned}
&\forall\, s \in Trace;\ t \in \mathbb{N};\ \varphi \in \Phi_i \bullet (s, t) \models_{f,s} \varphi \Leftrightarrow \varphi \neq \underline{false}\ \wedge \\
&\big(\exists\, e \in \mathcal{VE} \bullet (\varphi = E(e) \vee \varphi = I(e)) \wedge \exists\, e' \in s(t) : e' \models_e \varphi \\
&\vee \exists\, \psi, \chi \in \Phi_i \bullet \varphi = \psi\ \underline{implies}\ \chi \wedge \neg((s, t) \models_f \psi) \vee (s, t) \models_{f,s} \chi \\
&\vee \exists\, \gamma \in \Gamma \bullet \varphi = eval(\gamma) \wedge [\![\varphi]\!]_{eval} = true \\
&\vee \exists\, vn \in VName;\ vs \in VVal;\ \psi \in \Phi_i \bullet \\
&\quad \varphi = (\underline{forall}\ vn\ \underline{in}\ vs : \psi) \wedge \forall\, vv \in vs \bullet (s, t) \models_{f,s} \psi[vn \mapsto vv] \\
&\vee \exists\, \psi, \chi \in \Phi_i \bullet \varphi = \psi\ \underline{until}\ \chi \wedge \big(\forall\, v \in \mathbb{N} \bullet t \leq v \Rightarrow (s, v) \models_{f,s} \psi \\
&\quad \vee \exists\, u \in \mathbb{N} \bullet t < u \wedge (s, u) \models_{f,s} \chi \wedge \forall\, v \in \mathbb{N} \bullet t \leq v < u \Rightarrow (s, v) \models_{f,s} \psi\big) \\
&\vee \exists\, n \in \mathbb{N};\ \psi \in \Phi_i \bullet \varphi = \psi\ \underline{after}\ n \wedge (s, t + n) \models_{f,s} \psi \\
&\vee \exists\, n \in \mathbb{N}_1;\ l, r \in \mathbb{N};\ \psi \in \Psi \bullet \varphi = \underline{replim}(n, l, r, \psi) \wedge \\
&\quad l \leq \#\{j \in \mathbb{N}_1 \mid j \leq n \wedge (s, t + j) \models_{f,s} \psi\} \leq r \\
&\vee \exists\, n \in \mathbb{N};\ \psi \in \Psi, \chi \in \Phi_i \bullet \varphi = \underline{repuntil}(n, \psi, \chi) \\
&\quad \wedge \big((\exists\, u \in \mathbb{N}_1 \bullet (s, t + u) \models_{f,s} \chi \wedge (\forall\, v \in \mathbb{N}_1 \bullet v < u \Rightarrow \neg((s, t + v) \models_{f,s} \chi)) \\
&\qquad \wedge (\#\{j \in \mathbb{N}_1 \mid j \leq u \wedge s(t + j) \models_{f,s} \psi\}) \leq n) \\
&\qquad \vee (\#\{j \in \mathbb{N}_1 \mid s(t + j) \models_{f,s} \psi\}) \leq n\big)\big) \\
&\vee \varphi \in \Phi_s \wedge (s, t) \models_s \varphi
\end{aligned}$$

Similarly, a past variant of the language (Φ_i^-) and its semantics $\models_{f,s}^- \subseteq (Trace \times \mathbb{N}) \times \Phi_i^-$ is also defined.

$$\begin{aligned}\Phi_i^- ::= \;& (\Phi_i^-) \mid \Psi \mid \underline{false}^- \mid \Phi_i^- \;\underline{implies}^-\; \Phi_i^- \mid \underline{forall}\; VName\; \underline{in}\; VVal : \Phi_i^- \mid \\ & \Phi_i^- \;\underline{since}^-\; \Phi_i^- \mid \Phi_i^- \;\underline{before}^-\; \mathbb{N} \mid \underline{replim}^-(\mathbb{N},\mathbb{N},\mathbb{N},\Psi) \mid \underline{repsince}^-(\mathbb{N},\Psi,\Phi_i^-) \mid \\ & \Phi_s\end{aligned}$$

$$\begin{aligned}
&\forall\, s \in Trace;\ t \in \mathbb{N};\ \pi \in \Phi_i^- \bullet (s,t) \models_{f,s}^- \pi \Leftrightarrow (\pi \neq \underline{false}^-) \wedge \\
&\quad \big(\exists\, e \in \mathcal{VE} \bullet (\pi = E(e) \vee \pi = I(e)) \wedge \exists\, e' \in s(t) : e' \models_e \pi \\
&\vee\ \ \exists\, \psi, \chi \in \Phi_i^- \bullet \pi = \psi\ \underline{implies}^-\ \chi \wedge \neg((s,t) \models_{f,s}^- \psi) \vee (s,t) \models_{f,s}^- \chi \\
&\vee\ \ \exists\, \gamma \in \Gamma \bullet \pi = eval(\gamma) \wedge [\![\pi]\!]_{eval} = true \\
&\vee\ \ \exists\, vn \in VName;\ vs \in VVal;\ \psi \in \Phi_i^- \bullet \\
&\qquad \pi = (\underline{forall}\ vn\ \underline{in}\ vs : \psi) \wedge \forall\, vv \in vs \bullet (s,t) \models_{f,s}^- \psi[vn \mapsto vv] \\
&\vee\ \ \exists\, \psi, \chi \in \Phi_i^- \bullet \pi = \psi\ \underline{since}^-\ \chi \wedge \big((\forall\, v \in \mathbb{N} \bullet v \leq t \Rightarrow (s,v) \models_{f,s}^- \psi) \\
&\qquad \vee (\exists\, u \in \mathbb{N} \bullet u \leq t \wedge (s,u) \models_{f,s}^- \chi \wedge \forall\, v \in \mathbb{N} \bullet u < v \leq t \Rightarrow (s,v) \models_{f,s}^- \psi)\big) \\
&\vee\ \ \exists\, n \in \mathbb{N};\ \psi \in \Phi_i^- \bullet \pi = \psi\ \underline{before}^-\ n \wedge t \geq n \wedge (s, t-n) \models_{f,s}^- \psi \\
&\vee\ \ \exists\, n, l, r \in \mathbb{N};\ \psi \in \Psi;\ \bullet \varphi = \underline{replim}^-(n,l,r,\psi) \\
&\qquad \wedge\ \ l \leq (\#\{j \in \mathbb{N} \mid j \leq min(n,t) \wedge s(t-j) \models_{f,s}^- \psi\}) \leq r \\
&\vee\ \ \exists\, n \in \mathbb{N};\ \psi \in \Psi;\ \chi \in \Phi;\ e \in \mathcal{E} \bullet \varphi = \underline{repsince}^-(n,\psi,\chi) \\
&\qquad \wedge \big((\exists\, u \in \mathbb{N}_1 \bullet t \geq u \wedge (s, t-u) \models_{f,s}^- \chi \wedge (\forall\, v \in \mathbb{N} \bullet v < u \Rightarrow \neg((s,t-v) \models_{f,s}^- \chi)) \\
&\qquad\quad \wedge (\#\{j \in \mathbb{N} \mid j \leq u \wedge s(t-j) \models_{f,s}^- \psi\} \leq n)) \\
&\qquad \vee (\#\{j \in \mathbb{N} \mid j \leq t \wedge s(t-j) \models_{f,s}^- \psi\} \leq n)\big)\big) \\
&\vee\ \ \varphi \in \Phi_s \wedge (s,t) \models_s \varphi
\end{aligned}$$

Leveraging $\models_{f,s}^-$ the ILP definitions of Section 2.3.2 are augmented to allow for state-based operators in the conditions ($\varphi \in \Phi_i^-$).

This chapter together with Chapter 2 recaps a usage control model that specifies and enforces policies on data and their concrete representations using data flow tracking concepts. With this background, the following chapters present an in-depth description of the core contribution of this thesis.

Part II.

The Core

Chapter 4

Generic Modeling of Abstract Data and Actions

The results discussed in this chapter have already been published in [54], co-authored by the author of this thesis.

Specification-level policies (SLP) describe *what* must and must not happen to data throughout the execution of a system. Implementation-level policies (ILP) refine the SLPs by stating *how* they will actually be enforced. One major challenge in deriving the latter from the former is that there is not one correct meaning of data and action in SLPs. Instead, they have several meanings depending upon the context and it is not sufficient to capture their semantics in an ad hoc manner.

Let us consider a simple usage control policy that Alice wants to enforce: "Friends must not *copy* my *photo*". A *photo* can be a pixmap in a windowing system, a file in an operating system, a memory area that an application writes to, a java object, a blob in a database, a stream of packets in a network connection, an email attachment and so on. Similarly, *copy* might be understood as taking a screenshot, copying the photo via context menu or file menu items, copying the file to another location on the disk or external storage, reading the file via a java process and serializing it after cloning the java object, uploading the file to an Internet location and downloading it on another machine, cloning database tables, sending the photo as an attachment in an email etc. Which events and related containers capture "copy photo" correctly, depends upon the domain and technical systems in place.

This simple example shows the need of a generic and systematic approach to refinement of data and actions for policy derivation in the usage control context. The following section describes the requirements for policy derivation at a high-level. These requirements will act as a benchmark to assess the achievements of this work. The rest of this chapter and the following chapters are structured along the fulfillment of these requirements.

4.1. Policy Derivation Requirements

The overall goal of this work is to *derive implementation-level policies from specification-level policies in the context of usage control*. Intuitively, this high-level goal can be decomposed to three sub-goals, each of which could be achieved by meeting a corresponding set of requirements.

R1⟩ *Refinement of constructs:* There must be a generic way to refine the basic policy constructs viz. data and action into technical equivalents.

R2⟩ *Policy derivation:* SLPs must be translated to ILPs using the refinement of data and action.

R3⟩ *Automation of policy derivation:* Policy derivation from the specification to the implementation levels must be automated.

Refinement of constructs: The intuitive meanings of data and action are domain-specific. For instance, in the above example, which events and related containers capture "copy photo" correctly, depends upon the domain structure. So, the first user requirement entails the definition of a language for modeling domains with at least two levels of details: one that represents the end user's view and is used in the SLPs; the other which describes in detail the technical systems where the SLPs are actually enforced after being translated to ILPs (*R1.1*). As technical systems can be described in various ways depending upon the level of details captured in the model, the language must allow for hierarchical refinement (*R1.2*). For refinement purposes, the language must have mechanisms to connect the different levels of details in the domain model that describes data and action (*R1.3*).

Policy derivation: As the motive of this work is the derivation of usage control policies, there must be a generic way to *use the data and action refinements in the policy derivation*. For this, the domain modeling language from *R1.1 – R1.3* needs to be brought to the usage control context (*R2.1*). The semantics of policy derivation must be defined in this combined scenario (*R2.2*).

As the preventive enforcement mechanisms in the usage control infrastructure make decisions based on the current state of the system, and, as specification-level policies tend to be formulated in terms of the *future* usage of data (see Chapter 2), we need to *transform the constraints into enforceable conditions that are defined on the **past*** (*R2.3*). Otherwise, the system would need to be able to look into the future. Moreover, enforcement of some policies might require system information that can be known only at the runtime, e.g., process ID at the OS layer, user session identification at the web browser layer and subject attributes at different layers in a distributed system. It must be possible to add this type of context-specific information in the derived policies (*R2.4*).

If the ILPs contain events that take place in a machine at different layers of abstraction (as is expected from the fulfillment of the aforementioned requirements), then some of these events might be interdependent in the sense that one high-level goal can be achieved in the system through several events. E.g., saving a webpage triggered at the web browser layer includes writing a file at the OS layer. If such interdependent events are not connected to each other either during policy derivation or enforcement, then, policies related to counting high-level actions will be wrongly enforced. For instance, in this example, a single attempt to copy data via saving a webpage would be recognized as two copy events: one, because copy is refined as save page event and the other, because copy is refined also as write file events. Therefore, policy derivation must somehow include the connection of relevant events across different layers of abstraction (*R2.5*).

Automation of policy derivation: The disadvantages of manual policy refinement are discussed in Section 1.3. In light of this, we need an automated policy derivation framework. So a well-defined methodology to *automate the policy derivation* must be described. This must include identifying and putting all the different activities of the involved users (*R3.1*) and relevant system processes in a well-defined order (*R3.2*).

Additionally, policy derivation must include the transformation of the policies to the Event-Condition-Action (ECA) format (*R3.3*) so that the derived policies could be automatically deployed. This includes the specification of enforcement strategy in each policy. Intuitively, this can be non-trivial because one SLP can be enforced in several ways as shown by examples in chapters 1 and 2.

As SLPs are to be specified by end users who are not supposed to know the syntax and the semantics of the policy language OSL, there must be a solution for OSL-agnostic specification of usage control policies (*R3.4*). Policy specification also needs to address the concern that it is not trivial for end users to fully understand and specify security policies in general [59, 60] (*R3.5*).

The above requirements will serve as a guideline for the design and implementation of the solution and the evaluation of the achievements of this thesis (Chapter 9). The next sections and the following chapters describe in detail how these requirements are met. To start with the first high-level requirement, a domain metamodel that distinguishes between abstract *actions* and technical *transformers* and refines the former to the latter, is described and instantiated in the rest of this chapter.

4.2. The Domain Metamodel

As the meanings of data and actions vary according to the domain context, any solution that caters to the semantics of data and actions must address the problem at the domain level. This chapter describes a domain metamodel that distinguishes the abstract data and actions from their technical representations and refines the former to the latter in a systematic way. This approach of modeling domains is evaluated by instantiating the metamodel in several example scenarios in Section 4.3.

The domain metamodel, shown in Figure 4.1 is defined using a UML class diagram. It consists of three levels: the *platform-independent (PIM)* level that captures the end user's view of the domain and is independent of any technical details; the *platform-specific (PSM)* level that represents the technicalities of the corresponding systems from a conceptual viewpoint; and the *implementation-specific (ISM)* level, that reflects the implementation details of the systems in which the policies are to be enforced.

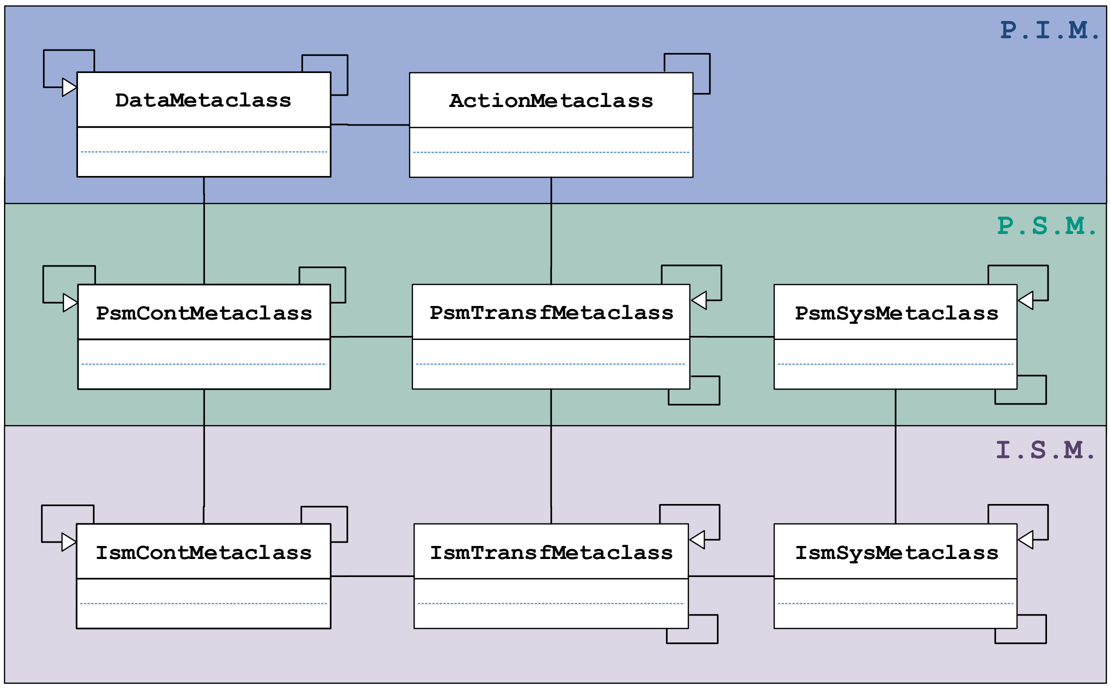


Figure 4.1.: The domain metamodel

4.2.1. The Platform-Independent Model

The Platform-Independent Model (PIM) has `DataMetaclass` and `ActionMetaclass` which are, as the names suggest, metaclasses of data and actions. Instances of these metaclasses are classes of data and actions. E.g., `Photo`, an instance of `DataMetaclass`, is a class of data; further instance of `Photo` is the specific data *d* stored in mypic.jpg and it is referred to as data item. An action *copy data d* is an instance of the class of actions `copy photo`, which is modeled as instance of `ActionMetaclass`. Each action class is *associated* with at least one data class that represents the data that is the target of the corresponding action: actions target specific data while classes of actions target data classes.

Associations and generalization-specialization relationships among data classes define the abstract data-model of the domain (an example of data model can be seen in a case study of online social networks in [61]). They represent the fact that in a domain, certain data are seen as aggregation/ composition of, or associated with (set of) other data, or as generalization/specializations of other data. In the OSN context, some examples of such relationships are as follows: a profile photo is a *special* type of photo; an album is a *UML composition* of photos (deleting the last photo deletes the album automatically in some OSNs where an empty album is not allowed); a profile is an *aggregation* of personal data, photos and posts; posts are *associated* to each other through links and tags, a user's profile is *associated* with all blog posts that the user writes and so on.

Inheritance and association relationships among data classes also suggest if certain user actions on one data have cascading effects on the other data. For example, deleting a profile means also deleting all associated posts, or denying copies of the album means that no picture in the album can be copied individually irrespective of the difference in the technical representations of the two.

If one data class specializes another data class, all actions associated with the *general data* are also valid for the *specific data*. E.g., if `Comment` is specialized as `Post` and all posts can be edited, then comments are also editable. The technical semantics of actions on the generic data are included in the technical semantics of those actions on the specific data. For example, if `BlogAccount` is a specialization of a `GeneralAccount`, then the technical semantics of actions like copy and delete for `BlogAccount` is the same as for the `GeneralAccount` if no semantics of those actions is specified for the blogs.

4.2.2. The Platform-Specific Model

The Platform-Specific Model (PSM) captures the technical details of systems at a conceptual level without any implementation details. `PsmContMetaclass` is the metaclass of all containers that are platform-specific. `PsmTransfMetaclass` is instantiated to model all classes of *transformers* that read and write data among containers at different layers of abstraction in the system at the platform-specific level.

Conceptually, transformers are functions that use sets of (atomic and/or complex) events in corresponding systems to *transform* the *content of containers*. Transformers can be recursively decomposed till each transformer corresponds to a single event. Some examples of domain-specific classes of transformers that target classes of containers are: `TakeScreen-shot` in `WindowingSystem`, `Copy&Paste` in `WebBrowser` and `CopyFile` in `File-System`. A screenshot on a specific window, a copy & paste on a selected text and one specific command to copy a file from a given filepath are instances of the respective transformer classes. Transformers target containers while the classes of transformers target classes of containers. This relationship is shown via an *association* between `PsmTransf-Metaclass` and `PsmContMetaclass` in the metamodel.

`PsmSysMetaclass` is the metaclass of all the systems at the PSM level. *Systems are seen as collections of transformers*, e.g., operating system is a collection of transformers that read and write data among files and memory locations. Systems denote the *locations* of the different containers which are read, written and modified by the transformers, e.g., a specific write system call targets a file myphoto.jpg in one running Linux distribution. Without the system part, it would not be possible to model the location where the system call is executed. As the domain model will be used in refining policies (in the following chapters), the system information in the domain model would help in the enforcement of policies that mandate execution of certain events in systems, e.g., for enforcing "data must be deleted from all systems".

Classes of containers, transformers and systems at the PSM level can be associated with and/or generalize sets of other classes of containers, transformers and systems at the PSM level. For instance, `Database` is a *specialization* of `File`: this means that databases are seen as *special* files. Similarly, copy file transformers are *associated* with a set of transformers for opening, reading, writing and closing files, and an OSN system is a *collection* of multiple layers of abstractions over many servers and client machines.

Because PSM classes model technical concepts without implementation details, we cannot directly instantiate these classes into concrete entities. For example, a concrete file does not exist in a system without its implementation details. Without implementation details, we can only refer to the *notion* of a file. Similarly, transformers cannot exist in concrete systems without implementation details. In the earlier example of a screenshot on a specific window, a copy & paste on a selected text and one specific command to copy a file from a given filepath as instances of the respective transformer classes implicitly subsume the implementation details.

Therefore, for instantiating a PSM class into concrete entities, it must be specialized with implementation details. E.g., transformer class `CopyFile` is specialized to `cp`, a class of concrete commands in Unix-based operating systems that read data from one file and write it to another file using system calls. An instance of the `cp` class is *cp source.txt dest.txt*. Similarly, a *generic* class `FileSystem` at the PSM level is modeled with the set of transformer classes that are common in all file systems without implementation details. Class `UnixFilesystem` *specializes* it with implementation details of all those transformer classes and might have additional classes of transformers.

4.2.3. The Implementation-Specific Model

As there is more than one way of implementing different systems, multiple implementations of containers and transformers exist. This is shown at the third level in the metamodel, the Implementation-Specific Model (ISM).

`IsmContMetaclass`, `IsmTransfMetaclass` and `IsmSysMetaclass` are instantiated to define domain-specific classes of *containers* that are modified by classes of *transformers* in classes of *system implementations*. Running systems with concrete containers and transformers are instances of these classes of system implementations.

Classes of systems represent all the following at the PSM and the ISM levels: the physical host (e.g., server hardware), the logical layers of abstraction in a machine (e.g., database, operating system and word processor) and the applications that comprises of several logical systems and physical hosts (e.g., file sharing application which is a part of OSN).

ISM transformer classes in the domain model and their instances in the concrete systems in the domain have the same name; the only difference is that transformer classes target container classes while the instances of the ISM transformers target specific containers in running system. For example, the class of write system calls targets the class of files while a specific write system call targets a concrete myphoto.jpg file. The association between `IsmTransfMetaclass` and `IsmContMetaclass` in the metamodel represents this relationship.

As mentioned earlier, PSM classes are specialized with implementation details into classes at the ISM level. We also see in Figure 4.1 that different ISM classes can be further refined using associations and inheritance. Therefore, a platform-specific class can be specialized at both the PSM and the ISM level, depending upon the type of detail added. For example, class `File` can be specialized as `Ext3File` at the PSM level and `Ext3File` can be specialized as `LinuxFile` and `BSDFile` classes at the ISM level. Alternatively, all specializations of `File` can be at the ISM level (as in Figure 4.2). Actually, technical systems can be modeled with recursive refinements of classes both at the PSM and the ISM levels and the demarcating lines between the refinements across the levels are not crisp. In the first case, all the intermediate refinements of `File` are at the PSM level and the last level of implementation details are added at the ISM level. In the second case, the refinements are split across the levels. This blurring of lines between the layers is unavoidable and well-recognized. However, it is not a limitation: it adds flexibility and makes it possible for a power user to model a domain in several ways.

The data, containers and the events described in chapters 2 and 3 are the instances of the classes of data, containers and transformers in any *domain model* which is drawn as an instance of the metamodel shown in Figure 4.1. Refinement of data and action is captured by the associations among classes at and across levels in the domain model. Mapping data classes and action classes to classes of containers and transformers respectively at the PSM level gives the conceptual meanings to user actions on data in terms of transformers applied to containers. The semantics are extended to the ISM level with the recursive refinement of classes of containers, transformers and systems from the PSM to the ISM level. Altogether, this tells us which transformers are executed on what containers when certain actions are performed on data.

For illustration, let us consider an instantiation of our metamodel shown in Figure 4.4. The semantics of "Copy Photo" in the context of an OSN are given by classes Screenshot in the Windowing System, Copy&Paste in the Web Browser and CopyFile in the Operating System at the PSM level. Mapping classes of containers and transformers to their implementation-specific forms gives the semantics of classes of action on data classes in terms of low-level technical implementations. Instantiating the classes to specific data, containers etc. results in the semantics of specific actions in terms of concrete transformers in running systems. E.g., in this example, the semantics of "copying a photo" would be "getImage on a Drawable object" in a X11 Windowing System, "cmd_copy on an HTML image object" in a Mozilla Firefox Web Browser, and executing "open, read, write and close system calls on a file" in a Ubuntu Unix distribution.

The next section discusses several instantiations of the proposed domain metamodel. Let's start with a metamodel instantiation for the case of our running example (OSN), introduced in Section 1.4.2.[1]

4.3. Evaluation

The approach of modeling domains using the proposed metamodel is evaluated using case studies. In this section, the instantiation of the domain metamodel is depicted at two levels. At first, we see from a high level, how each metaclasses could be instantiated to model a domain (Figure 4.2). The example domain is of the running example. For space reasons, only parts of the complete OSN model could be shown in Figure 4.2.

Thereafter, meaningful refinements of two actions namely copy and delete are modeled in the OSN context. For simplicity, only a few layers of abstraction are included in these refinements. The examples could be easily extended with further refinements at other layers of abstraction. Two more scenarios are used for the illustration of action refinement: Third-party Platform Applications in OSNs and Mobile Applications domain.[2]

[1]All instantiations of the domain metamodel are UML class diagrams. For space reasons, here onwards, all classes throughout this thesis are depicted as rectangles (without the UML class diagram notation).

[2]Intuitively, the result of a high-level action, as understood by an end user, cannot alone be captured in terms of corresponding transformers. One reason can be that the high-level action corresponds to infinitely many sequences of transformers. For this reason, another type of action refinement, that captures how system states are affected by the high-level user action, is needed. More details are in Section 5.1.2.2.

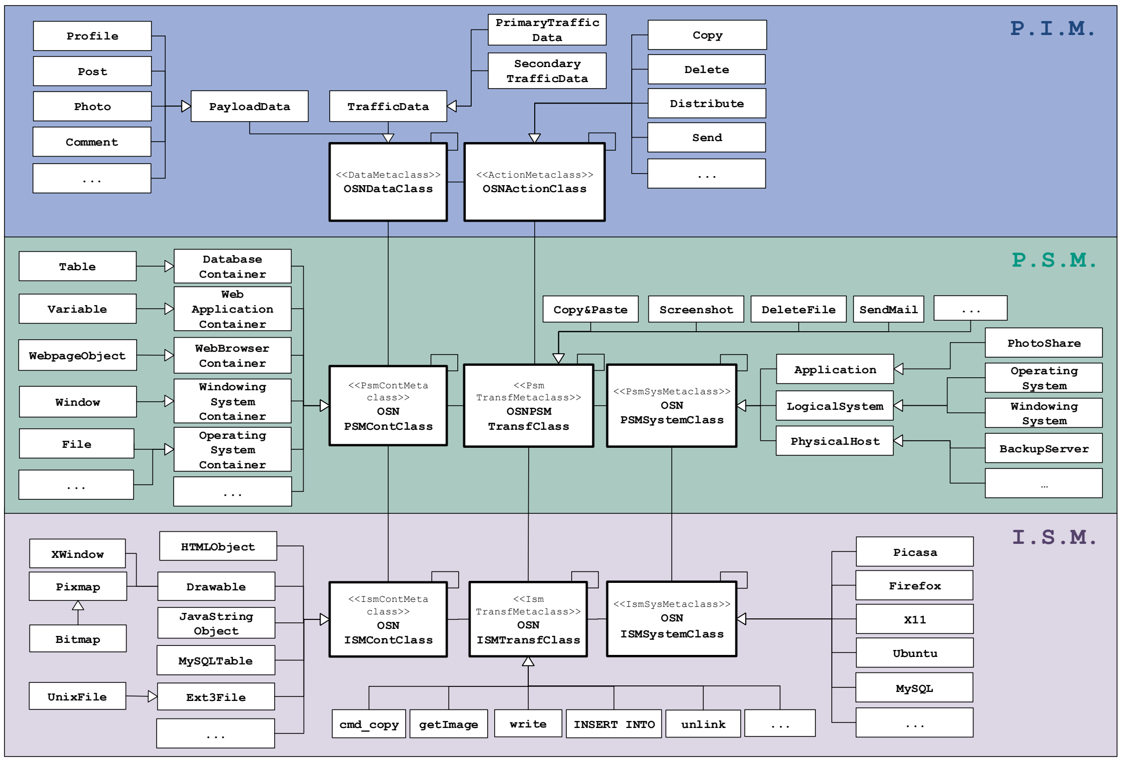


Figure 4.2.: An OSN instance of the metamodel of Figure 4.1

4.3.1. Example One: Online Social Network

A partial model of an OSN is shown in Figure 4.2. The metaclasses (`DataMetaclass`, `ActionMetaclass`, `PsmContMetaclass` etc.) that are instantiated by a class of the OSN domain model are indicated as stereotypes with guillemets (`<< >>`) on top of the basic class name (`OSNDataClass`, `OSNActionClass`, `OSNPSMContClass` etc).

The PIM level contains the data model of the OSN and related action. The respective classes model the following: all data is classified into two: payload data and traffic data. Payload data is all data posted by end users: personal profile data, videos, photos, messages, etc. Traffic data is all data generated by users' activities in the social network: user's IP address, browser and OS specifications, search terms, etc. Traffic data is classified as primary and secondary data, according to perspectives. Primary traffic data is all traffic data generated by the user while surfing. E.g., the profiles visited by the user. Secondary traffic data is all traffic data generated when other users visit this user's profile, view his pictures, videos etc. Payload data can also be classified as primary and secondary. Primary payload data would be all data posted about a user by himself. E.g., the data posted by user while creating his profile on the website. Secondary payload data is all data posted about the user, by other users. E.g., another user can post a picture of this user in his album and link it to him. A detailed data model of an OSN is given in [61]. Some OSN actions are copy, delete, distribute and send, shown via respective action classes.

At the PSM level, the respective classes model containers such as web browser container, windowing system container, operating system container, etc. The corresponding classes are specialized as `WebpageObject`, `Window`, `File`, etc classes. `Copy&paste`, `Screenshot`, `DeleteFile` and other transformer classes model the transformers at the PSM level. Systems can be logical systems (i.e., layers of abstraction like web browser, windowing system and operating system), physical hosts/locations (e.g., caches, backups and other servers and client machines) and applications that comprise of both logical and physical systems (e.g., photo sharing application and mail system). This is shown via an inheritance hierarchy among classes of systems.

At the ISM level, classes of system implementations include `Picasa` (logical systems + physical hosts), `Firefox`, `X11`, `Ubuntu`, `MySQL` etc. with connections to ISM transformer classes to model concrete systems with corresponding containers and transformers. Some examples of implementation-specific transformers are *cmd_copy* command in Firefox web browser, *getImage* function in X11, *write* and *unlink* system calls in Ubuntu operating system and *INSERT INTO* statement in MySQL. These are shown by similarly-named classes in the model diagram.

A multi-level refinement of containers at the ISM level is also shown. Drawable *is-a* OSN container at the ISM level which is a collection of windows and pixmaps. So the corresponding classes are shown in an inheritance hierarchy with associations. Another example shown is a bitmap, a 1-bit monochrome pixmap that is commonly used as a stencil or mask. Therefore `Bitmap` is shown specializing the class `Pixmap`.

The definition of the domain models and the mappings provides the refinement of data classes in terms of structure and the containers that store the data. They also refine actions to transformers. The mappings in Figure 4.2 are shown collectively for clarity purposes: in order to focus on the classes of the domain model at the PIM, PSM and ISM levels. In the next set of examples, the refinement associations are separated from each other to show how model components are connected to each other at and across the three levels. Two actions from the above model of the OSN are refined for illustration. For simplicity, only a few layers of abstraction are considered.

4.3.1.1. Refining *Copy Photo*

A simplified refinement of copy photo at two layers of abstraction, X11 windowing system and Firefox web browser, is shown via a class diagram in Figure 4.3. It represents the refinement of copying a photo as executing cmd_copy command on an HTML object at the Firefox level and via getImage function call for a drawable from an X11 client.

Adding another layer of abstraction to this refinement, we see in Figure 4.4 how multiple transformers corresponding together to a high-level action can be modeled.

All the nodes in these models are classes and the fact that certain transformers *together* correspond to a higher-level transformer or action is expressed using class attributes. In Figure 4.4, as simple rectangles are used instead of UML classes, this is shown using an AND-gate that groups the transformers together. Intuitively, there should be an order over the transformers (parallel system events can be modeled as transformers with the same order number attribute in the respective classes); this is depicted in Figure 4.4 with an *S* (for sequence) inside the AND-gate notation and the spatial layout denoting the order.

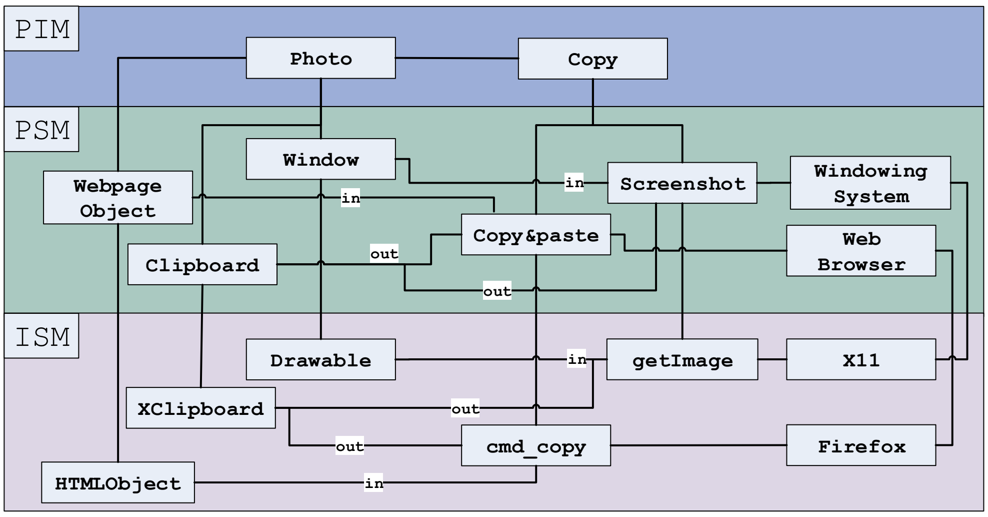


Figure 4.3.: Refinement of *copy photo* in Firefox and X11

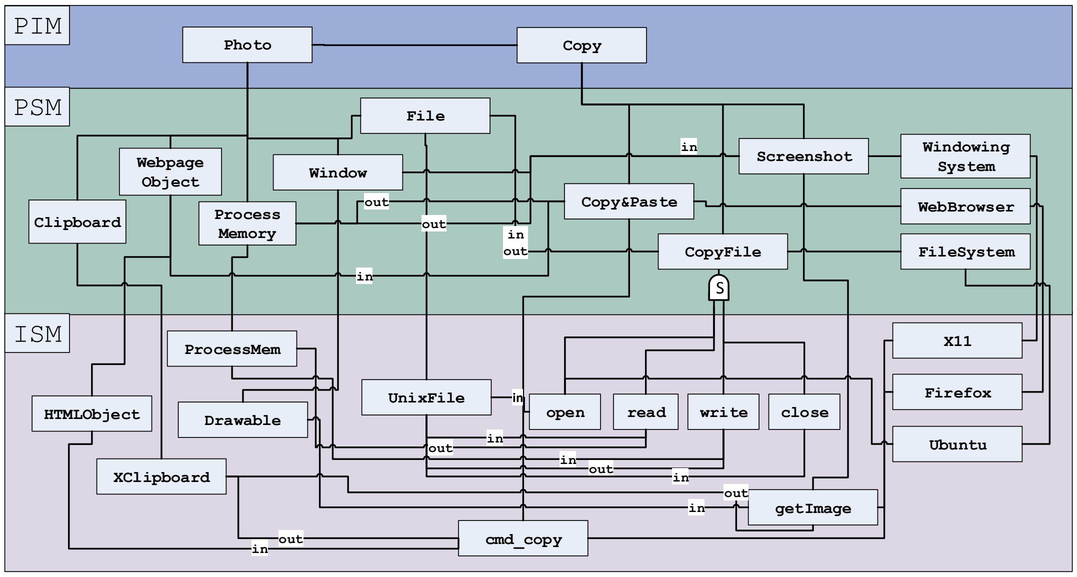


Figure 4.4.: Refining *copy photo*: multiple ISM transformers refine one PSM transformer

4.3.1.2. Refining *Delete Data*

Another example action is “delete” that targets songs and videos. Figure 4.5 shows the refinements of copy and delete actions together. It also gives an impression of how the domain model quickly grows with new elements.

Delete is first refined to transformer DeleteFile (shown by an association between `Delete` and `DeleteFile` classes). Further refinements of delete are shown by associating `DeleteFile` to `Remove` and `Overwrite` classes capturing the fact that deletion of files can be

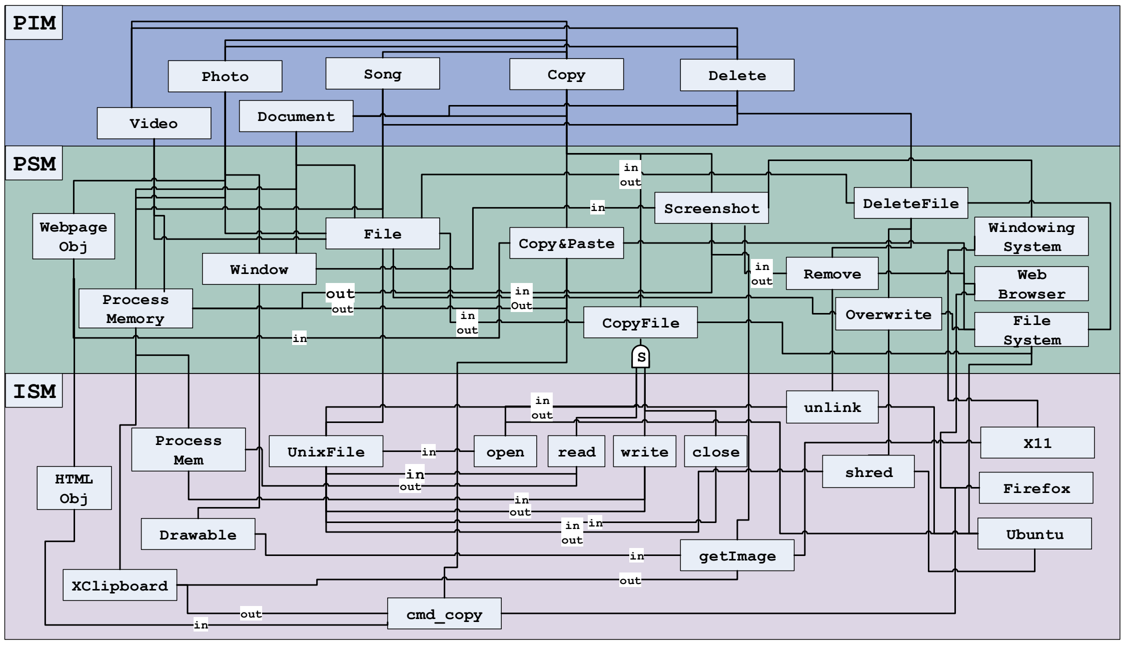


Figure 4.5.: Refinement of copy and *delete* in OSN

achieved in general by either removing the file from the file system or by overwriting the existing file. Classes of system call `unlink` and Unix command `shred` (the instances would be specific executions of the unlink and shred) respectively refine the two classes via associations at the ISM level.

4.3.2. Example Two: Mobile Applications Domain

The second use case is from mobile applications domain. Typical actions at the end user level are: share photos, edit contacts, send SMS, allow request for phone ID, share location, etc. Refinements of three user actions: "send SMS", "allow location requests" and "allow phone identification" are shown in Figure 4.6 via the association of `Send`, `AllowRequest`, `SMS`, `Location` and `PhoneID` classes to classes at the lower levels.

The mobile platform is Android OS. SMS is represented as textMessage at the ISM level, while the location and identification of the phone and is stored as GPS_DATA and IMEI_DATA respectively. Thus, "sending an SMS" means executing "sendTextMessage" while "allowing location and phone ID requests" corresponds to "httpRequest on GPS _DATA and IMEI_DATA" in the Android application.

4.3.3. Example Three: A Java Application

The last example is from the domain of third-party platform applications in OSNs. The example third-party application is a Birthday Reminder that requests the friendlist of Alice in order to notify her of the current and upcoming birthdays of her friends. In order to do so, the application must obtain the latest list of Alice's friends periodically so that she does not receive irrelevant notifications or that she does not miss any of the birthdays. From

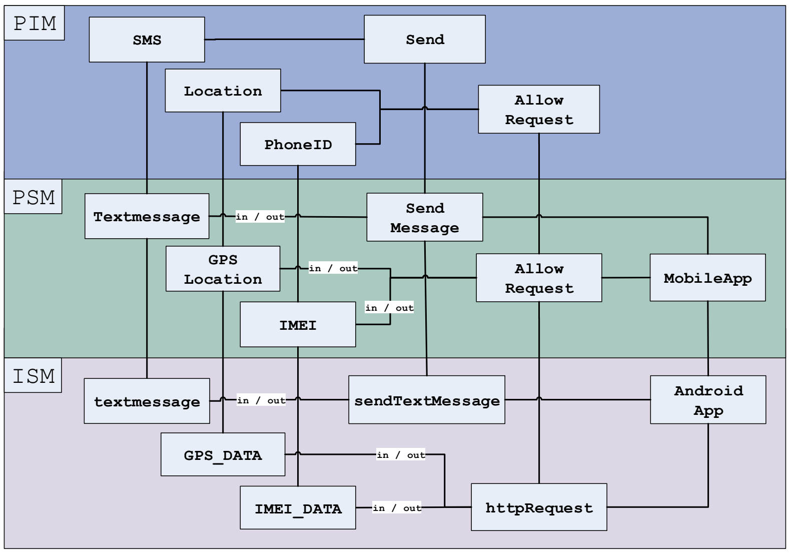


Figure 4.6.: Refining *send SMS* and *allow requests* in mobile applications

an end user's perspective (PIM), two actions in the application periodically update Alice's friendlist. Figure 4.7 shows a straightforward refinement of these user actions "create text data" and "update text data".

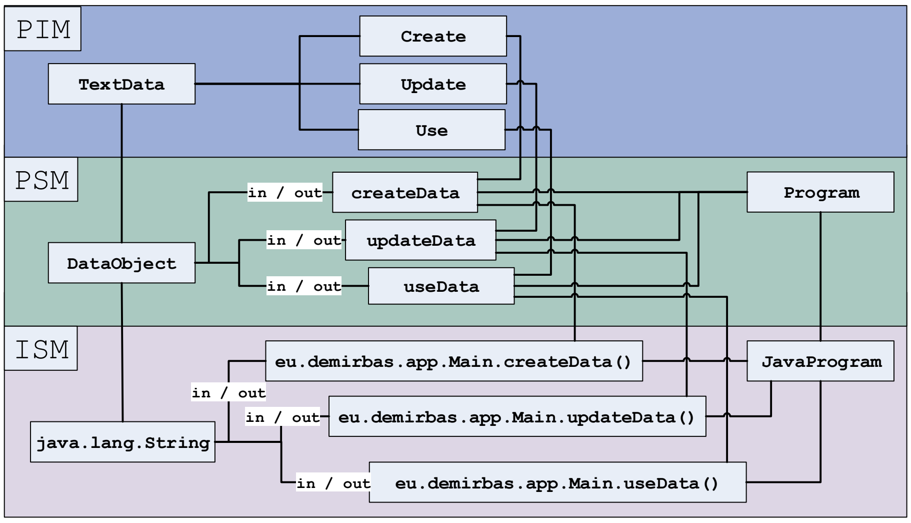


Figure 4.7.: Refining *create data*, *update data* and *use data* in a Java application

All text data are stored as String objects. Containers and transformers at the PSM and the ISM levels are modeled based on the following signatures of the methods `createData()`, `updateData()` and `useData()`:

```
java.lang.String createData(java.lang.String str)
java.lang.String updateData(java.lang.String str)
java.lang.String useData(java.lang.String str)
```

Instead of "application", the term "program" is used in the domain model in Figure 4.7 to avoid confusion with the "application domain". The domain model is simplified for illustration and can be easily extended for refining other user actions.[3]

As part of automating the policy derivation, a framework was designed with an intuition of the needed infrastructure components. In the first step towards the realization of the concepts presented in this thesis, the next section describes a prototypical implementation of a tool for modeling domains. The output of this tool, a domain model in concrete syntax, is used to configure the infrastructure for action refinement in policy derivation. The implementation is independent of the modeled domain and demonstrates the domain- and system- independence of the approach of data and action refinement.

4.4. Implementation

Eclipse model-driven development technologies Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF) were used to develop a graphical tool for modeling domains. The implemented tool is called the *Domain Model Editor*.

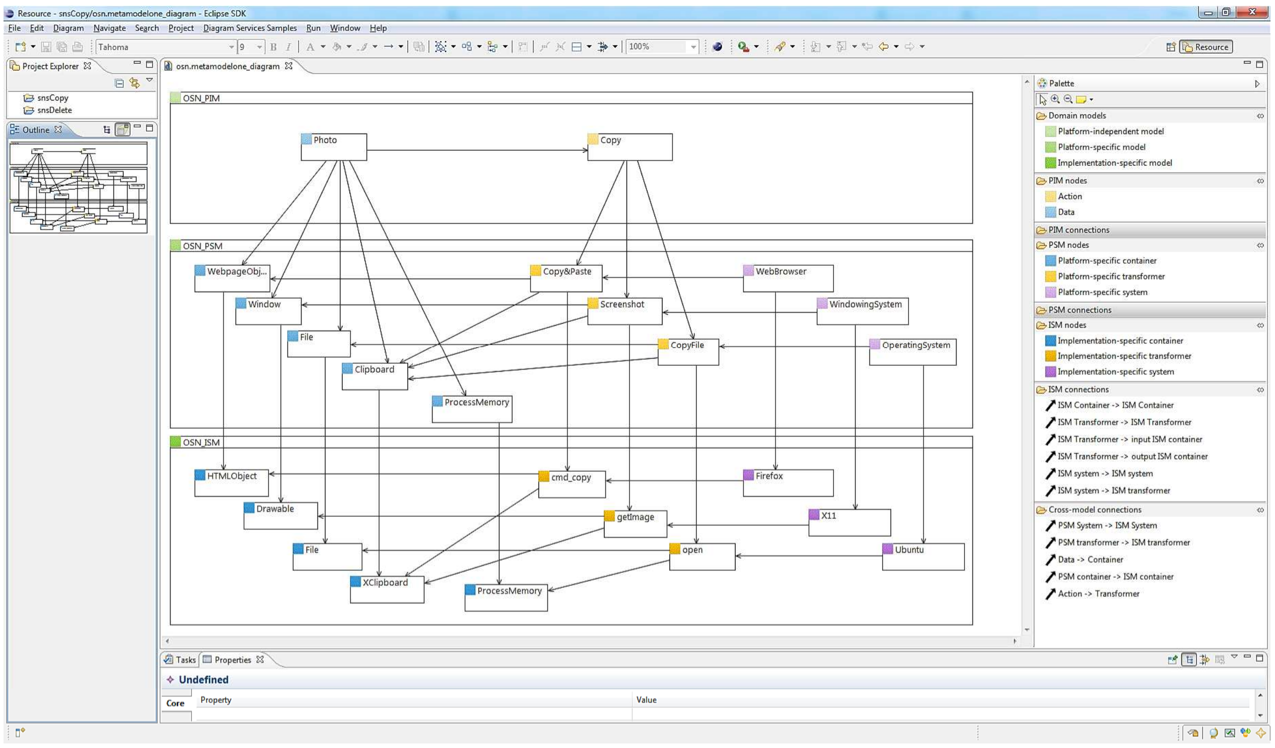


Figure 4.8.: The domain model editor with an example graphical model

Technical details of the implementation are in Appendix A. The corresponding Ecore metamodel diagram is given in appendix Figure A.1. The output of the domain model

[3]This scenario is explained in detail in Section 6.4.3

editor is a domain model in XML format. The XML model of the domain can also be generated without using this graphical editor, using the XML schema given in appendix Listing A.1. A snapshot of the domain model editor with a simple refinement of copy photo action is shown in Figure 4.8

The editor generates two files corresponding to the drawn domain model. A file with *_diagram* extension stores the model with all the information needed to load it in a graphical format. The second output is an XML file with the domain model without graphical details. As an example, Listing 4.1 shows the corresponding XML representation of the domain model that is shown in Figure 4.8, automatically generated by the editor.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<model:domain xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:model="http://model/1.0">
  <pims name="OSN_PIM">
    <pimactions executedas="//@psms/@psmtransformers.1 //@psms/@psmtransformers.0 //@psms/@psmtransformers.2" name="Copy"/>
    <pimdata validactions="//@pims/@pimactions.0" storedin="//@psms/@psmcontainers.0 //@psms/@psmcontainers.1 //@psms/@psmcontainers.2 //@psms/@psmcontainers.3 //@psms/@psmcontainers.4" name="Photo"/>
  </pims>
  <psms name="OSN_PSM">
    <psmcontainers contimplementedas="//@isms/@ismcontainers.2" name="WebpageObject"/>
    <psmcontainers contimplementedas="//@isms/@ismcontainers.0" name="Window"/>
    <psmcontainers contimplementedas="//@isms/@ismcontainers.1" name="Clipboard"/>
    <psmcontainers contimplementedas="//@isms/@ismcontainers.3" name="File"/>
    <psmcontainers contimplementedas="//@isms/@ismcontainers.4" name="ProcessMemory"/>
    <psmtransformers outputcontainer="//@psms/@psmcontainers.2" inputcontainer="//@psms/@psmcontainers.0" transimplementedas="//@isms/@ismtransformers.0" name="Copy&amp;Paste"/>
    <psmtransformers outputcontainer="//@psms/@psmcontainers.2" inputcontainer="//@psms/@psmcontainers.1" transimplementedas="//@isms/@ismtransformers.1" name="Screenshot"/>
    <psmtransformers outputcontainer="//@psms/@psmcontainers.2" inputcontainer="//@psms/@psmcontainers.3" transimplementedas="//@isms/@ismtransformers.2" name="CopyFile"/>
    <psmsystems systemtransformers="//@psms/@psmtransformers.1" sysimplementedas="//@isms/@ismsystems.0" name="WindowingSystem"/>
    <psmsystems systemtransformers="//@psms/@psmtransformers.0" sysimplementedas="//@isms/@ismsystems.1" name="WebBrowser"/>
    <psmsystems systemtransformers="//@psms/@psmtransformers.2" sysimplementedas="//@isms/@ismsystems.2" name="OperatingSystem"/>
  </psms>
  <isms name="OSN_ISM">
    <ismcontainers name="Drawable"/>
    <ismcontainers name="XClipboard"/>
    <ismcontainers name="HTMLObject"/>
    <ismcontainers name="File"/>
    <ismcontainers name="ProcessMemory"/>
    <ismtransformers inputimplecontainer="//@isms/@ismcontainers.2" outputimplecontainer="//@isms/@ismcontainers.1" name="cmd_copy"/>
    <ismtransformers inputimplecontainer="//@isms/@ismcontainers.0"
```

```
        outputimplecontainer="//@isms/@ismcontainers.1" name="getImage"/>
    <ismtransformers inputimplecontainer="//@isms/@ismcontainers.3"
        outputimplecontainer="//@isms/@ismcontainers.4" name="open"/>
    <ismsystems implesystemtransformers="//@isms/@ismtransformers.1" name="X11"
        />
    <ismsystems implesystemtransformers="//@isms/@ismtransformers.0" name="
        Firefox"/>
    <ismsystems implesystemtransformers="//@isms/@ismtransformers.2" name="
        Ubuntu"/>
  </isms>
</model:domain>
```

Listing 4.1: XML Output of the domain model editor: refinement of *copy photo* in OSN

4.5. Summary

In this work, no proofs are given for the correctness of the domain model that is defined using the proposed metamodel. So, even though we hope for the domains to be modeled correctly, it is possible that the resulting models are incorrect and/or incomplete. This can be problematic for the correct enforcement of the policies because action refinement based on the domain model is central to the policy derivation in this thesis. This limitation is discussed in detail in sections 1.6 and 9.2.

In this chapter, classes of data and actions are refined to classes of transformers that target container classes in classes of system implementations. However, usage control policies are enforced in concrete systems with transformers that target specific containers. In order to use the domain models (defined using classes) for usage control policy translation, we must connect the classes of data, containers, transformers, etc. to their instances. Also, we must specify the semantics of the refinement associations within and across the PIM, PSM and ISM levels. The next chapter addresses these issues.

Chapter 5

Formal Semantics of Action Refinement

The results discussed in this chapter have already been published in [62], co-authored by the author of this thesis.

In the previous chapter, a domain metamodel that distinguishes between abstract actions and technical transformers and refines the former to the latter, was described and instantiated. This domain metamodel has, however, certain limitations. Firstly, the domain metamodel gives an intuition of the relationship between a high-level action and the more concrete transformers. But it does not explain the exact relationship between the two: classes of actions are mapped to classes of 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 (Figure 4.4 on page 46). 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.

Secondly, the domain metamodel is a UML class model. When we instantiate it for modeling a domain, the resulting domain model shows the static structure of the domain; it is not capable of capturing one specific snapshot of the runtime. This means that the domain model alone cannot be used to relate high-level actions to system states in order to understand how user actions at a higher level affect the runtime of the concrete systems.

Intuitively, modeling user actions in terms of system states is useful in those cases where the effect of the user action can be achieved by infinitely many sequences of systems events and it might either not be possible to list all of those sequences or it might be just much more convenient to instead mention the resultant system state for the action.

In order to give formal semantics to action refinement and to model user actions also in terms of system states, the domain metamodel is combined with the usage control model described in Chapter 3. The relationship among the elements of the two models is shown in Figure 5.1.

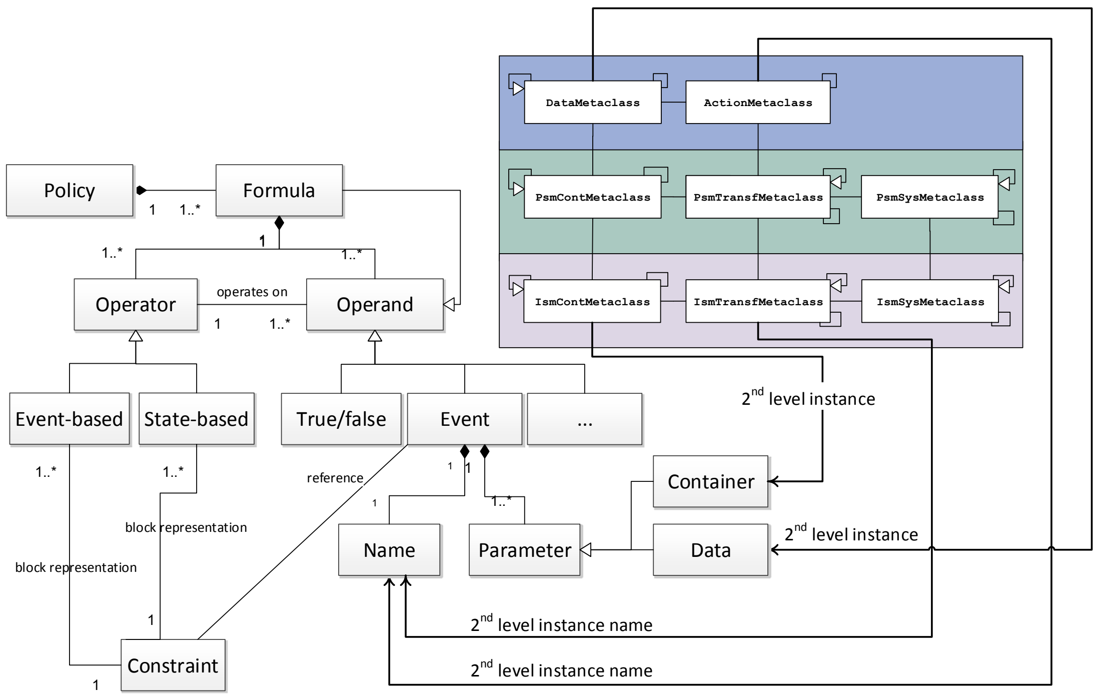


Figure 5.1.: The domain metamodel w.r.t. the OSL model of Chapter 3

Using the Meta Object Facility (MOF) notations, the domain metamodel is a M2-model which describes an M1-level domain model that consists of domain-specific classes of data, actions, containers etc. The data flow model, which has data and container elements (e.g., the abstract data d1, stored in container myphoto.jpg) is at the M0 level. The data and container elements of the data flow model are instances of the instances (let us call these 2^{nd} level instances from now on) of the members of the PIM and the ISM levels in the domain metamodel. This is shown in Figure 5.1 with labels "2^{nd} level instance" and "2^{nd} level instance name" on associations. The two models are combined formulating this relationship between the model elements. A specific example of corresponding elements at all the three levels is shown in Figure 5.2.

The formalization of action refinement is described in two steps: in the first step, one high-level action is refined into sets/sequences of transformers, both at the PSM and the ISM levels using the recursive function $\tau_{\mathcal{E}}$.

In the second step, the refinement of an action is given by modeling the state that the system reaches when the high-level action is performed by an end user. The resultant state is expressed as an OSL formula with state-based operators. Function τ_{σ} is used for this type of action refinement. Later both the refinements are combined to get the **complete** refinement of a high-level action using function $\tau_{\mathcal{A}}$.

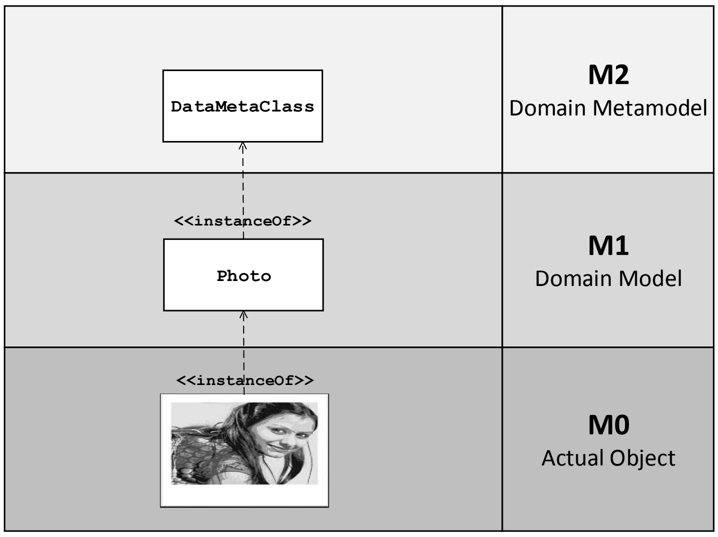


Figure 5.2.: An example depicting the relationships among the elements of the domain metamodel, the domain model and the usage control model. `DataMetaClass` is instantiated to the data class `Photo` which is instantiated to a specific photo (data d in the usage control model).

5.1. The Combined Model

Chapters 2 and 3, formally introduced the instances in a domain, e.g., data and containers. In this chapter, formal notations for the sets of *UML classes* of these instances are introduced. These UML classes of data, containers etc., are elements of the class diagram model that represents the domain. For instance, `Photo` in a model of an online social network (OSN), shown in Figure 4.2 on page 44, is a data class whose instances are all individual photos in different runs of the OSN.

In order to distinguish the classes from the instances, a different font style and a "breve" notation is used in the symbols representing the classes and their sets (as in $\breve{d} \in \breve{\mathfrak{D}}$ to denote data class $\breve{d}$ as member of set $\breve{\mathfrak{D}}$). A complete list of all symbols used in this chapter is in Appendix Table C.1.

$\breve{\mathfrak{D}}$ is the set of all classes of data at the PIM level, $\breve{\mathfrak{A}}$ is the set of all classes of user actions at the PIM level, $\breve{\mathfrak{C}}_{\mathfrak{psm}}$ is the set of all classes of containers at the PSM level, $\breve{\mathfrak{T}}_{\mathfrak{psm}}$ is the set of all classes of transformers at the PSM level, $\breve{\mathfrak{C}}_{\mathfrak{ism}}$ is the set of all classes of containers at the ISM level and $\breve{\mathfrak{T}}_{\mathfrak{ism}}$ is the set of all classes of transformers at the ISM level. In other words, these sets are collections of the domain model instances of `DataMetaclass`, `ActionMetaclass`, `PsmContMetaclass`, `PsmTransfMetaclass`, `IsmContMetaclass` and `IsmTransfMetaclass` respectively, e.g., referring to Figure 4.2, {`Profile, Post, Photo, Comment, ...`} is a set of classes of photos, posts, comments etc. in the OSN; each member of this set is an instance of `DataMetaclass`.

In Chapters 2 and 3, there has been no distinction between high-level user actions and system events: policies could be specified on both. But as we have seen in Chapter 4, the end user view of the domain only consists of high-level user actions. So let us split the set of all events $\mathcal{E}$ into two sets so that we can refer to actions and transformers separately, when needed. $\mathcal{E}_{\mathcal{A}}$ is the set of all user actions while $\mathcal{E}_{\mathcal{T}}$ is the set of all transformers in a system:

$$\mathcal{E}_{\mathcal{A}} \cup \mathcal{E}_{\mathcal{T}} \subseteq \mathcal{E} \wedge \mathcal{E}_{\mathcal{A}} \cap \mathcal{E}_{\mathcal{T}} = \varnothing$$

Each action is defined by a name (set $AName$) and a set of parameters. Ditto for the transformers, where the set $TName$ represents transformer names:

$$\begin{aligned}&\mathcal{E}_{\mathcal{A}} \subseteq AName \times \mathbb{P}(PName \times PValue)\\&\mathcal{E}_{\mathcal{T}} \subseteq TName \times \mathbb{P}(PName \times PValue)\\&AName \cup TName \subseteq EName\end{aligned}$$

Recall from Chapter 4 that each action class is *associated* with at least one data class that represents the data that is the target of the corresponding action. Therefore, an action class is defined by a name (set $\breve{A}Name$) and a set of target data classes.

$$\breve{\mathfrak{A}} \subseteq \breve{A}Name \times \mathbb{P}(\breve{\mathfrak{D}})$$

Similarly, classes of transformers at the PSM and the ISM levels are expressed using their names (sets $\breve{T}Name$ and set $\breve{T}Name'$ respectively) and the respective target container classes.

$$\begin{aligned}&\breve{\mathfrak{T}}_{\mathfrak{psm}} \subseteq \breve{T}Name \times \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{psm}})\\&\breve{\mathfrak{T}}_{\mathfrak{psm}} \subseteq \breve{T}Name' \times \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}})\end{aligned}$$

Before moving on, a recap of the oft-used terms:

- *Actions* refer to specific user actions at the PIM level. Each action is a member of $\mathcal{E}_{\mathcal{A}}$. Actions are instances of the members of set $\breve{\mathfrak{A}}$ that appear in the domain model, e.g., specific actions of copying data are instances of the action class `Copy` shown in the domain model of Figure 4.2 on page 44.

- *Transformers* are technical events. The term is generic in the sense that transformers are both the technical concepts (at the PSM level) and the implementation-specific concrete events in technical systems (at the ISM level), unless the model level is explicitly specified (as in 'PSM tranformer' and 'ISM transformer'). Each transformer is a member of $\mathcal{E}_{\mathcal{T}}$ and an instance of a member of ($\breve{\mathfrak{T}}_{\mathfrak{psm}} \cup \breve{\mathfrak{T}}_{\mathfrak{ism}}$) that is depicted in the domain model, e.g., *unlink myphoto.jpg* is an instance of `unlink` transformer class that appears in Figure 4.2.

- In order to distinguish them from the user actions of PIM level, the *Action part of ECA rules* is called "Authorization Action".

- The term *Event* and the set $\mathcal{E}$ hereafter refers to the union of all user actions and transformers, as mentioned above.

- In order to distinguish them from $\mathcal{E}$, the *Event part of ECA rules* is called "Trigger Event".

Following is an overview of the policy specification and derivation:

- *Step 0 – Domain model specification:* This must take place before the policies are derived using action refinement. The domain model contains UML classes that represent domain elements like data, container etc. for all runs of the systems.
- *Step 1 – Policy specification:* As already mentioned in Chapter 3, data does not exist in running systems, only corresponding containers exist. Therefore policies are specified only in terms of containers with an indication if the policy applies to that specific container or all copies of it, to be interpreted as data (via *dataUsage, containerUsage*). Although policies are specified in one specific run of a system, they address the protection of data in all the system runs.
- *Step 2 – Policy derivation:* After specification, policies are translated and deployed in either the same system run or in a different system run as the specification, but the policies are to be enforced for all the subsequent runs of the system (until they are revoked). Steps *2a* – *2c* take place in one arbitrary run of the system; the outputs of these steps are applicable to all system runs.
  - *Step 2a – Connecting instances to classes:* Policies are specified on instances, but the domain model used for policy refinement contains only corresponding UML classes. The instances in the policies therefore need to be connected to their classes (details in Section 5.1.1). Though this connection takes place in that one specific run of the system when the policies are translated, the class of a specific data, container etc. remains the same across all the system runs.
  - *Step 2b – Future to past derivation:* needed for monitoring of policies for preventive enforcement, details to come in Section 5.2.1.
  - *Step 2c – Action refinement:* takes place after the future to past translation; details are in Section 5.1.2. If data moves across machines, the containers (and possibly their names) change. This is reflected in that system run when the policies are translated, e.g., if a data is downloaded from the internet and stored in a specific file on a machine, this information can be known only after the data is downloaded (or at the earliest when the download starts). The complete path of the file, reflected in the corresponding policy remains valid for all system runs thereafter. Any changes in data storage mappings are taken care of on grounds of the data flow definitions. The output of this step is a policy that applies to all system runs, per machine.
  - *Step 2d – Information specific to system runs:* Policies might require some additional information that varies according to the different system runs, e.g., user login session. This type of information is added as a last step in policy derivation. Policies need not be re-translated from scratch, only the runtime-specific information part in policies is updated in each run (details in Section 6.2).

For action refinement in SLPs using the domain model, data, containers, actions and transformers must be connected to their classes. A generic function $getclass$ that returns the class of the given instance is used for this purpose. $getclass$ is overloaded with different signatures for data, containers, actions and transformers. The connection of data, containers etc. to their classes applies to all runs of the systems. For this reason, whenever a specific data, container etc. is addressed in the following sections, it refers to all the system runs.

5.1.1. Connecting Instances to Classes

Although one data, container or action can belong to multiple classes because of the inheritance hierarchies in the domain model,[1] we are here interested in computing the most specialized class which was instantiated to get the specific object (data, container etc.) for all systems runs. For this reason, the $getclass$ function maps an element (data, container, etc.) to exactly one class from the domain model.

Function $getclass : \mathcal{D} \rightarrow \breve{\mathfrak{D}}$ returns the class of a specific data. The exact definition of $getclass$ for data varies according to the domain and its implementation. The following simple example illustrates this: in the OSN context, a specific image (data 'd') can be an instance of `ProfilePhoto`, `Logo`, `Banner` or `CoverPhoto` (data classes). Different data classes might mean different storage format and different technical refinements of the instances, e.g., (small) logos stored as blobs vs. (large) photos stored as links in a table. Relevant data classes and assignment of data elements to one or many of them is intuitive and is driven by the domain design. This is because of two reasons: firstly, data classes describe how users understand different types of data in a particular context/domain and there is no rule for coming up with these classes. Secondly, even if the domain is fixed, there are several ways of implementing a domain and so a generalization of technically connecting specific data to their classes is not feasible. Therefore, it is not possible to give a generic definition of $getclass$ that formally connects data elements to their classes for all domains and their implementations.

Similarly, connecting specific containers to their ISM classes is implementation-dependent and the definition of the function $getclass : \mathcal{C} \rightarrow \breve{\mathfrak{C}}_{\mathfrak{ism}}$, that returns the class of a given container, cannot be generalized. Therefore, a formal definition of these functions is not provided. Instead, a methodological guidance in Section 6.4 illustrates how data and containers could be connected to their classes in one particular domain implementation.

Building upon the two $getclass$ functions that connect data and containers to their classes, the $getclass$ functions that connect actions and transformers to their classes are formally defined. Recall from Section 4.2.1 that actions have the same name as their classes, the only difference is that the former target specific data while the latter target classes of data at the PIM level. For each $\breve{a} \in \breve{\mathfrak{A}}$ let $\breve{a}.in$ denote the set of target data classes for the action class in the domain model. Then, $getclass : \mathcal{E}_{\mathcal{A}} \rightarrow \breve{\mathfrak{A}}$ is defined as:

[1] Because of multi-level inheritance in the domain model, a specific data, container or transformer can belong to several classes. E.g., *myphoto.jpg* is both a *File* and an *OS-Container* at the PSM level in Figure 4.2. Similarly, *rm myphoto.jpg* is an instance of ISM transformer class *rm File* which inherits from the class *removeFile File* at the PSM level.

$$\begin{aligned}
&\forall\, a \in \mathcal{E}_{\mathcal{A}}, \forall\, \breve{a} \in \breve{\mathfrak{A}} : getclass(a) = \breve{a} \iff \\
&\quad a.n = \breve{a}.n \wedge (\exists\, d \in \mathcal{D}, \exists\, \breve{d} \in \breve{\mathfrak{D}} : \\
&\qquad obj \mapsto d \in a.p \wedge \breve{d} \in \breve{a}.in \wedge getclass(d) = \breve{d})
\end{aligned}$$

where $a.n$ refers to the name and $a.p$ are the parameters of a; $\breve{a}.n$ is the name of $\breve{a}$.

Similarly, transformers and their classes at the ISM level have the same name with the instances targeting containers and the classes targeting container classes (Section 4.2.3). Additionally, in our language, a transformer may also target a specific data (dataUsage events, i.e., $\exists\, t \in \mathcal{E}_{\mathcal{T}}, \exists\, d \in \mathcal{D} : obj \mapsto d \in t.p$). A transformer class $\breve{t}$ is the class of "transformer t with data d as object" *iff* the classes of the containers where d is stored *in any data state*, are in the set of targets of $\breve{t}$ in the domain model. For each $\breve{t} \in \breve{\mathfrak{T}}_{\mathfrak{ism}}$ let $\breve{t}.in$ denote the set of target container classes for the transformer class. Then, function $getclass : \mathcal{E}_{\mathcal{T}} \to \breve{\mathfrak{T}}_{\mathfrak{ism}}$ is defined as:

$$\begin{aligned}
&\forall\, t \in \mathcal{E}_{\mathcal{T}}, \forall\, \breve{t} \in \breve{\mathfrak{T}}_{\mathfrak{ism}} : getclass(t) = \breve{t} \iff t.n = \breve{t}.n \wedge \\
&\quad ((\exists\, c \in \mathcal{C}, \exists\, \breve{c} \in \breve{\mathfrak{C}}_{\mathfrak{ism}} : obj \mapsto c \in t.p \wedge \breve{c} \in \breve{t}.in \wedge getclass(c) = \breve{c}) \\
&\qquad \vee (\exists\, d \in \mathcal{D} : obj \mapsto d \in t.p \wedge (\exists\, c' \in \mathcal{C}, \exists \sigma \in \Sigma : d \in \sigma.s(c') \\
&\qquad\quad \wedge\, getclass(c') \in \breve{t}.in)))
\end{aligned}$$

where $t.n$ refers to the name and $t.p$ refers to the parameters of t; $\breve{a}.n$ is the name of $\breve{a}$ and $\sigma.s$ denotes the storage function of state σ.

Because the computation of classes for actions and transformers uses their names and the values of their object parameters, we can only compute the classes of non-variable events (actions and transformers) and those variable events whose names and object parameters are known.

Function $getinstance$ is the dual of $getclass$ that computes all instances of a given class of data, container etc. for all system runs. For example, all instances of a data class can be computed as

$$getinstance : \breve{\mathfrak{D}} \to 2^{\mathcal{D}}$$

$$\forall\, \breve{d} \in \breve{\mathfrak{D}} : getinstance(\breve{d}) = \{\mathcal{D}' \in 2^{\mathcal{D}} \mid \forall\, d \in \mathcal{D} : d \in \mathcal{D}' \implies getclass(d) = \breve{d}\}$$

Function $getinstance$ is overloaded to compute the instances of classes of actions, containers and transformers in a similar way, leveraging the respective $getclass$ functions.

5.1.2. Action Refinement

By now we have a model that consists of both data, actions, containers, transformers and their respective classes. We have seen how the instances connect to their classes and vice versa. Now that we also have the distinction between abstract user actions and their concrete representations called the transformers, policies that talk of the former can be refined to policies that talk of the latter. And these refinements can be performed based on the relationships between the classes of actions and transformers.

Besides this, as we already have the notion of system states and a set of state-based OSL formulas that hold true under certain conditions, we can also refine actions in terms of states. That is, for each action, we can define which state-based formula should hold true after the action is performed by the user. The motivation for this type of action refinement is already introduced in the start of this chapter and is recapped in Section 5.1.2.2. In the next subsections, we will see both these types of action refinements.

Defining *Action Refinements* in OSL: Remember that formulas of the form $E(\cdot)$ and $I(\cdot)$ denote actual and intended events (see Section 2.1.3). Therefore, refinement of actions and transformers using the aforementioned class-level refinement relationships corresponds to the translation of OSL formulas of the form $E(\cdot)$ and $I(\cdot)$. Let us start with the refinement of actions to transformers.

5.1.2.1. Actions Refined to Transformers

Actions can be refined to transformers in two ways: an action refined to a *set* of transformers means that any of the resulting transformers correspond to the action. An action can also be refined to a *sequence* of transformers implying that all of the corresponding transformers, in the particular sequence, together refine the action. Further refinement of transformers to other transformers also works the same way.

As it is impossible to predict the length of executions between any two members of a sequence of transformers in running systems, we assume arbitrary executions between any two members of a sequence of transformers in *sequence refinements*. But this assumption introduces future-time liveness in the action refinement definitions if they are defined in future OSL.Therefore, the future OSL formulas of the SLPs are first translated to their past forms and then action refinement is performed on the resulting formula. Past-time liveness is not a problem for enforcement because indefinite past can be checked in a running system as opposed to the indefinite future. Indefinite past is expressed using $\underline{eventually}^{-}$ (represented as $\Diamonddot$ from now on), semantically equivalent to $\underline{not}^{-}(\boxdot(\underline{not}^{-}))$ in the language Φ_i^-. Intuitively, $\Diamonddot(\varphi)$ is true if the formula φ was true at least once in the past.

This way, executing action refinement on an SLP becomes a two-step process. In the first step, a translation function $\tau_{\mathcal{P}} : \Phi \rightarrow \Phi_i^-$ that works along the lines of the methodological guidance provided in Section 5.2, translates a formula in Φ to an equivalent formula in Φ_i^-. In the second step, actions are refined to transformers and also in terms of system states.

As mentioned earlier, both actions and transformers are refined to transformers via set and sequence relationships. Because action and transformer refinement works the same way, the distinction among them is not needed in the definition of action refinement. The refinement function is therefore defined on a set that is a collection of all the actions and transformers (set $\mathcal{E}$). For this, the objects of actions and transformers, i.e., data and containers, are put together in a set $\mathcal{O}$. Then, at the class level also, a set $\breve{\mathfrak{E}}$, that is a union of the sets of all the action and transformer classes is defined, to be used in the definition of action refinement.

$$\mathcal{O} \subseteq \mathcal{D} \cup \mathcal{C}$$
$$\breve{\mathfrak{E}} = \breve{\mathfrak{A}} \cup \breve{\mathfrak{T}}_{\mathfrak{psm}} \cup \breve{\mathfrak{T}}_{\mathfrak{ism}}$$

The translation function $\tau_{\mathcal{E}} : \Phi_i^- \to \Phi_i^-$ recursively refines an action or a transformer on the object addressed in the corresponding SLP. Recall that the way the *getinstance* function has been defined, it computes all the instances of a class. To get an instance of a *class of actions or transformers* that targets a specific object, function *getinstance* is overloaded to compute exactly one instance of a given class:

$$\begin{aligned}
&getinstance : \breve{\mathfrak{E}} \times \mathcal{O} \to \mathcal{E} \\
&\forall\, o \in \mathcal{O}, \forall\, \breve{e} \in \breve{\mathfrak{E}} : getinstance(\breve{e}, o) = \\
&\quad \{e \in \mathcal{E} \mid getclass(e) = \breve{e} \wedge obj \mapsto o \in e.p\}
\end{aligned}$$

At the class level, R_{set} maps a class of user actions to a *set* of PSM transformer classes with the intuitive semantics that instances of any one of the mapped transformers corresponds to the instance of the high-level action. R_{seq} maps an action class to a *sequence* of transformer classes at the PSM level: instances of all of the specified transformers in the particular sequence correspond to the high-level action instance.

Additionally, as PSM and ISM transformers can be further refined, both within and across their respective model levels, their refinement functions are overloaded to express both of these refinements. R_{set} and R_{seq} express refinements within the same level and, $\tilde{R}_{set}$ and $\tilde{R}_{seq}$ express cross-level refinements (from PIM to PSM and from PSM to ISM):

$$\begin{array}{rr}
R_{set} : \breve{\mathfrak{T}}_{\mathfrak{psm}} \to \mathbb{P}(\breve{\mathfrak{T}}_{\mathfrak{psm}}) & \tilde{R}_{set} : \breve{\mathfrak{A}} \to \mathbb{P}(\breve{\mathfrak{T}}_{\mathfrak{psm}}) \\
R_{set} : \breve{\mathfrak{T}}_{\mathfrak{ism}} \to \mathbb{P}(\breve{\mathfrak{T}}_{\mathfrak{ism}}) & \tilde{R}_{set} : \breve{\mathfrak{T}}_{\mathfrak{psm}} \to \mathbb{P}(\breve{\mathfrak{T}}_{\mathfrak{ism}}) \\
R_{seq} : \breve{\mathfrak{T}}_{\mathfrak{psm}} \to \mathrm{seq}\, \breve{\mathfrak{T}}_{\mathfrak{psm}} & \tilde{R}_{seq} : \breve{\mathfrak{A}} \to \mathrm{seq}\, \breve{\mathfrak{T}}_{\mathfrak{psm}} \\
R_{seq} : \breve{\mathfrak{T}}_{\mathfrak{ism}} \to \mathrm{seq}\, \breve{\mathfrak{T}}_{\mathfrak{ism}} & \tilde{R}_{seq} : \breve{\mathfrak{T}}_{\mathfrak{psm}} \to \mathrm{seq}\, \breve{\mathfrak{T}}_{\mathfrak{ism}}
\end{array}$$

With this intuition, $\tau_{\mathcal{E}}$ is formally defined as follows:

$$\begin{aligned}
&\forall\, o \in \mathcal{O};\ \forall\, e \in \mathcal{E}, \forall\{\breve{e}_1, .., \breve{e}_n\} \subseteq \breve{\mathfrak{E}} : \tau_{\mathcal{E}}(E(e \mapsto \{obj \mapsto o\})) = \\
&\begin{cases}
\tau_{\mathcal{E}}(E(getinstance(\breve{e}_n, o))\ \underline{and}^- & if(\exists\, \breve{e} \in \breve{\mathfrak{E}} : getclass(e) = \breve{e} \\
\diamondsuit(\tau_{\mathcal{E}}(E(getinstance(\breve{e}_{n-1}, o)))\ \underline{and}^- \diamondsuit(... & \wedge\ (R_{seq}(\breve{e}) = \langle \breve{e}_1, .., \breve{e}_n \rangle \vee \\
\quad ...\ \diamondsuit(\tau_{\mathcal{E}}(E(getinstance(\breve{e}_1, o))))...) & \tilde{R}_{seq}(\breve{e}) = \langle \breve{e}_1, .., \breve{e}_n \rangle)) \\
 & \\
\tau_{\mathcal{E}}(E(getinstance(\breve{e}_1, o)))\ \underline{or}^- & if(\exists\, \breve{e} \in \breve{\mathfrak{E}} : getclass(e) = \breve{e} \\
(\tau_{\mathcal{E}}(E(getinstance(\breve{e}_2, o)))\ \underline{or}^- ... & \wedge\ (R_{set}(\breve{e}) = \{\breve{e}_1, .., \breve{e}_n\} \vee \\
\quad ...\ \underline{or}^-\ \tau_{\mathcal{E}}(E(getinstance(\breve{e}_n, o))) & \tilde{R}_{set}(\breve{e}) = \{\breve{e}_1, .., \breve{e}_n\})) \\
 & \\
E(e \mapsto \{obj \mapsto o\}) & \text{otherwise}
\end{cases}
\end{aligned}$$

As $\diamondsuit(\psi)$ expresses that ψ must have been true at least once sometime in the past, the above definition covers the case of arbitrary number of events in a sequence (as in $\langle open$, $write+$, $close\rangle$).

The events are underspecified in the translation definitions. The existence of parameters other than the object parameter is implicit. As object ($obj \in PName$) is a part of the definition of the translation function, it is explicitly mentioned in the events above.

The above definition refines only actual events. Refinement of intended events works on similar lines: $\tau_{\mathcal{E}}$ on formulas of form $I(\cdot)$ gives the above output with each $E(\cdot)$ replaced by $I(\cdot)$.

5.1.2.2. Action Refinement using System State

Intuitively, expressing the semantics of high-level user actions only in terms of sets/sequences of transformers might not be sufficient in many cases because one high-level action can be refined to infinitely many sequences of transformers in different systems. It is therefore sometimes more useful and convenient to also refine user actions in terms of the effect that they cause in running systems with all the infinite sequences of transformers. To this end, another refinement of action, τ_{σ}, using state-based operators (Section 3.4) is defined. It is called the **state-based refinement** of actions. This translation captures the state a system reaches when a user performs a high-level action on some data. Therefore, for each action, the power user defines which state-based OSL formula must hold when the high-level action is performed. By default, when no such such formula is defined for an action, its state-based refinement is given by the proposition: *false*.

State-based operators specify storage relationships among specific data and containers in running systems. But the state-based refinement for an action must be defined for classes of actions because specific actions in policies (that target specific data items) are earliest known at the policy specification time. Also, the concrete containers that potentially store specific data are known only when a policy is deployed in a concrete system. For these reasons, state-based operators that address unknown data and container classes (ISM containers) are needed.

To specify unknown data in the state-based operators, variables are used and these variables are substituted by actual data when a policy is specified. To use classes of containers in operators, the language Φ_s is extended to also include state-based operators on set $\breve{\mathfrak{C}}_{\mathfrak{ism}}$. The semantics of the new operators are defined using the $getclass$ function discussed earlier.

ISM Containers in State-Based Operators: The language Φ_s is extended by overloading two existing operators: $isNotIn(\mathcal{D}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}}))$ and $isOnlyIn(\mathcal{D}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}}))$.

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 $not(isNotIn(d, \{C_{print}\}))$ where C_{print} represents the class of printer containers. When data d flows into any container that belongs to this class, a high-level print action is recognized. 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. No use case was found where the semantics of $isCombinedWith(d, d)$ needs to be specified using container classes.

The Special Case of the *Copy* Action: While modeling different high-level actions as transformers and system states, we recognized that the "copy" action is the trickiest. Not only is it hard to define a "one-fits-all" meaning of this action, it is also non-trivial to describe this action for well-known cases with the OSL operators we have as yet. Therefore, two new state-based operators were added to the language Φ_s to better model copy action: $\underline{\mathit{isNewIn}}(\mathcal{D}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}}))$ and $\underline{\mathit{isMaxIn}}(\mathcal{D}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}}), \mathbb{N})$.

$\underline{\mathit{isNewIn}}(d, \mathit{Cl})$ is true when in the current state, data is in at least one such container where it was not stored in the previous state and the class of that new storage container is in set Cl. This operator is useful in modeling copy action as the effect that moves data to new containers (of specific types). The possibility to specify a restriction on the type of container is given so that policies that inhibit copies of data do not render the data or the system useless. For example, modeling copy as "data moves to any new container" would render the data useless in case of policies that inhibit copy as nothing could be done with data. In that case, it would make more sense to say "copy data" means that data is in new files (file system level copies) or that data is in new memory locations (opened by new applications or written to new buffers) etc.

$\underline{\mathit{isMaxIn}}(d, \mathit{Cl}, n)$ is true when data d is in at most n containers of classes belonging to set Cl. Using this operator, we can express copy as the restriction on the maximum number of copies of data in a system. So if "copy d" is defined as $\underline{\mathit{not}}(\underline{\mathit{isMaxIn}}(d, \mathit{File}, 2))$, then a third copy of d in files will be counted as a new copy. If one of the two files is deleted, then data d can be stored in a new file.

Variable Data in State-Based Operators: We already have the notion of variables $\mathit{Var} = \mathit{VName} \to \mathit{VVal}$ for event names and event parameters in our language (see Section 2.1.1). As data exists as object parameter in events ($\mathit{obj} \in \mathit{PName}$), the set of possible variable values $\mathit{VVal} = \mathbb{P}(\mathit{PValue} \cup \mathit{EName})$ includes data. Variable data are therefore variables that map variable names to data:

$$\mathcal{VD} = \mathit{VName} \mapsto \mathcal{D}$$
$$\mathcal{VD} \subseteq \mathit{Var}$$

In order to specify state-based formulas without mentioning the specific data, state-based operators with variable data, in a language Φ_{sv}, are introduced:

$$\begin{aligned}\Phi_{sv} ::= {} & \underline{\mathit{isNotIn}}(\mathcal{VD}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}})) \mid \underline{\mathit{isOnlyIn}}(\mathcal{VD}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}})) \mid \\ & \underline{\mathit{isNewIn}}(\mathcal{VD}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}})) \mid \underline{\mathit{isMaxIn}}(\mathcal{VD}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}}), \mathbb{N})\end{aligned}$$

Elements from Φ_{sv} are instantiated into elements from Φ_s using the substitution of variables in formula as defined earlier (Section 2.2.5): a formula $\varphi \in \Phi_{sv}$ that contains a variable $v \in \mathit{Var}$ is instantiated by substituting all occurrences of vn in φ by value $vv \in \mathit{Var}(v)$. $\varphi[vn \mapsto vv]$ denotes the result of substitution. There may be multiple substitutions in the square brackets. Recall that $\mathit{VarsIn}(t)$ denotes the set of variables in a term t. Using it, function $\mathit{Inst}_{sv} : \Phi_{sv} \to \mathbb{P}(\Phi_s)$ generates all ground substitutions of a formula with variables:

$$\begin{aligned}&\forall\, \varphi \in \Phi_{sv} : \mathit{VarsIn}(\varphi) = \{vn_1 \mapsto \mathit{VS}_1, \ldots, vn_k \mapsto \mathit{VS}_k\} \\ &\quad \Rightarrow \mathit{Inst}_{sv}(\varphi) = \{\varphi[vn_1 \mapsto vv_1, \ldots, vn_k \mapsto vv_k] : \textstyle\bigwedge_{i=1}^{k} vv_i \in \mathit{VS}_i\}\end{aligned}$$

Finally, the language Φ_s of Section 3.4 is redefined as follows:

$$\begin{aligned}\Phi_s ::= {} & \Phi_{sv} \mid \underline{\mathit{isNotIn}}(\mathcal{D}, \mathbb{P}(\mathcal{C})) \mid \underline{\mathit{isNotIn}}(\mathcal{D}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}}) \mid \\ & \underline{\mathit{isNewIn}}(\mathcal{D}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}})) \mid \underline{\mathit{isMaxIn}}(\mathcal{D}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}}), \mathbb{N}) \mid \\ & \underline{\mathit{isCombinedWith}}(\mathcal{D}, \mathcal{D})\end{aligned}$$

plus the macros $\underline{isOnlyIn}(\mathcal{D}, \mathbb{P}(\mathcal{C}))$ and $\underline{isOnlyIn}(\mathcal{D}, \mathbb{P}(\breve{\mathfrak{C}}_{\mathfrak{ism}}))$ with $\underline{isOnlyIn}(d, Cl)$ $\Leftrightarrow \underline{isNotIn}(d, \breve{\mathfrak{C}}_{\mathfrak{ism}} \setminus Cl)$.

The semantics of Φ_s, given by $\models_s \subseteq (Trace \times \mathbb{N}) \times \Phi_s$, is redefined:

$$\begin{aligned}
&\forall\, t \in Trace;\ n \in \mathbb{N};\ \varphi \in \Phi_s;\ \sigma', \sigma \in \Sigma :\ (t, n) \models_s \varphi \iff \\
&\quad \sigma = states(t, n) \wedge \sigma' = states(t, n-1) \wedge (\\
&\qquad \exists\, i \in \mathbb{N}, vd \in \mathcal{VD}, Cl \subseteq \breve{\mathfrak{C}}_{\mathfrak{ism}} : (\varphi = \underline{isNotIn}(vd, Cl) \\
&\qquad\quad \vee\ \varphi = \underline{isNewIn}(vd, Cl) \vee \varphi = \underline{isMaxIn}(vd, Cl, i)) \wedge \\
&\qquad\quad \exists\, \chi \in \Phi_s : \chi \in Inst_{sv}(\varphi) \wedge (t, n) \models_s \chi \\
&\qquad \exists\, d \in \mathcal{D}, Cs \subseteq \mathcal{C} : \varphi = \underline{isNotIn}(d, Cs) \wedge \\
&\qquad\quad \forall\, c \in \mathcal{C} : d \in \sigma.s(c) \implies c \notin Cs \\
&\qquad \vee\ \exists\, d \in \mathcal{D}, Cl \subseteq \breve{\mathfrak{C}}_{\mathfrak{ism}} : \varphi = \underline{isNotIn}(d, Cl) \wedge \\
&\qquad\quad \forall\, c \in \mathcal{C} : d \in \sigma.s(c) \implies (getclass(c) \notin Cl) \\
&\qquad \vee\ \exists\, d_1, d_2 \in \mathcal{D} : \varphi = \underline{isCombinedWith}(d_1, d_2) \wedge \\
&\qquad\quad \exists\, c \in \mathcal{C} : d_1 \in \sigma.s(c) \wedge d_2 \in \sigma.s(c) \\
&\qquad \vee\ \exists\, d \in \mathcal{D}, Cl \subseteq \breve{\mathfrak{C}}_{\mathfrak{ism}} : \varphi = \underline{isNewIn}(d, Cl) \wedge \\
&\qquad\quad \{c \in \mathcal{C} \mid d \in \sigma.s(c) \wedge getclass(c) \in Cl\} \setminus \\
&\qquad\qquad \{c' \in \mathcal{C} \mid d \in \sigma'.s(c') \wedge getclass(c') \in Cl\} \neq \varnothing \\
&\qquad \vee\ \exists\, i \in \mathbb{N}, d \in \mathcal{D}, Cl \subseteq \breve{\mathfrak{C}}_{\mathfrak{ism}} : \varphi = \underline{isMaxIn}(d, Cl, i) \wedge \\
&\qquad\quad \#\{c \in \mathcal{C} \mid d \in \sigma.s(c) \wedge getclass(c) \in Cl\} \leq i)
\end{aligned}$$

where $\sigma.s$ the storage function of state σ.

The semantics of Φ_i, that is, $\models_{f,s} \subseteq (Trace \times \mathbb{N}) \times \Phi_i$, and the past variant of the language (Φ_i^-), $\models_{f,s}^- \subseteq (Trace \times \mathbb{N}) \times \Phi_i^-$ is as defined earlier in Section 3.4.

Event Declaration: To enable the power user to express the potential state-based refinement of a user action, and to define all possible events and their parameters that can be used in all policies in the domain, we introduce the concept of *Event Declaration*. An event declaration is given by the event name, a set of parameter names, and the state-based formula that must hold when the high-level action is performed.

$$\mathcal{ED}ecl \subseteq AName \times \mathbb{P}(PName) \times \Phi_i^-$$

The relationship between an event and its declaration is *bijective*. One example for an event declaration is $copy \mapsto \{obj, quality, subject\} \mapsto \underline{not}^-(\underline{isMaxIn}(x, \{UnixFile\}, 2))$. It says that the action copy can be used in policies with three parameters: obj, that denotes the primary object (data or container), quality and subject; and the state-based refinement of copy is modeled as $\underline{not}^-(\underline{isMaxIn}(x, \{UnixFile\}, 2))$ where x is a variable.

Action is a subset of the set of events that can be used in policies. For the state-based refinement of action, a function *getref* fetches the state-based formula from the declaration of the specific action:

$$\begin{aligned}
&getref : \mathcal{E}_{\mathcal{A}} \to \Phi_i^- \\
&\forall\, a \in \mathcal{E}_{\mathcal{A}};\ ed \in \mathcal{ED}ecl : getref(a) = ed.\varphi \iff a.n = ed.n
\end{aligned}$$

where $a.n$ and $ed.n$ refer to the names of a and ed respectively; $ed.\varphi$ refers to the state-based formula part of the event declaration ed.

The refinement of action in terms of states is therefore achieved using $\tau_\sigma : \Phi_i^- \rightarrow \Phi_i^-$ which essentially looks for the variable state formula in the declaration and each variable in the respective state formula is substituted by the actual data value from the parameter of the user action. The other parameters of the action are evaluated alongside.

$$\begin{array}{l}
\forall\, a \in \mathcal{E}_\mathcal{A}, d \in \mathcal{D}, vd \in \mathcal{VD}, \varphi \in \Phi_i^- : \\
\quad \tau_\sigma(E(a \mapsto \{obj \mapsto d, pn_1 \mapsto pv_1, \ldots, pn_k \mapsto pv_k\})) = \\
\qquad \begin{cases} \varphi[vd \mapsto d] \bigwedge_{i=1}^{k} eval(a.p.pn_i = pv_i) & \text{if } \varphi = getref(a) \\ \\ false & \text{otherwise} \end{cases}
\end{array}$$

where $a.p.pn$ is used to access the parameter value for parameter name pn in a.

By now we have the refinement of a high-level action in terms of sets/sequences of transformers (using function $\tau_\mathcal{E}$) and in terms of system states (using function τ_σ). Now both functions are combined to express the "complete" refinement of a high-level action, given by $\tau_\mathcal{A} : \Phi_i^- \rightarrow \Phi_i^-$. 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:[2]

$$\begin{array}{l}
\forall\, d \in \mathcal{D};\ e \in \mathcal{E};\ \psi \in \Phi_i^-;\ \varphi \in \Phi_i^- : \tau_\mathcal{A}(\psi) = \varphi \iff \\
\psi \in \Phi_s \wedge \varphi = \psi \\
\vee\ \psi \in \{true, false\} \wedge \varphi = \psi \\
\vee\ \exists\, \gamma \in \Gamma : \psi = eval(\gamma) \wedge \varphi = \psi \\
\vee\ \psi = E(e) \wedge \varphi = \underline{or}^-(\tau_\sigma(E(e)), \tau_\mathcal{E}(E(e))) \\
\vee\ \psi = T(e) \wedge \varphi = \underline{or}^-(\tau_\sigma(T(e)), \tau_\mathcal{E}(T(e))) \\
\vee\ \exists\, vn \in VName;\ vs \in PValue \setminus \mathcal{D};\ \chi \in \Phi_i^- : \\
\qquad \psi = (\underline{forall}\ vn\ \underline{in}\ vs : \chi) \wedge \varphi = (\underline{forall}\ vn\ \underline{in}\ vs : \tau_\mathcal{A}(\chi)) \\
\vee\ \exists\, \chi \in \Phi_i^- : \psi \in \{\underline{not}(\chi), \underline{not}^-(\chi)\} \wedge (\varphi = \underline{not}^-(\tau_\mathcal{A}(\chi))) \\
\vee\ \exists\, \chi, \xi \in \Phi_i^- : \psi \in \{\underline{or}(\chi, \xi), \underline{or}^-(\chi, \xi)\} \wedge (\varphi = \underline{or}^-(\tau_\mathcal{A}(\chi), \tau_\mathcal{A}(\xi))) \\
\vee\ \exists\, \chi, \xi \in \Phi_i^- : \psi \in \{\underline{and}(\chi, \xi), \underline{and}^-(\chi, \xi)\} \wedge (\varphi = \underline{and}^-(\tau_\mathcal{A}(\chi), \tau_\mathcal{A}(\xi))) \\
\vee\ \exists\, \chi, \xi \in \Phi_i^- : \psi \in \{\underline{implies}(\chi, \xi), \underline{implies}^-(\chi, \xi)\} \wedge (\varphi = \underline{implies}^-(\tau_\mathcal{A}(\chi), \tau_\mathcal{A}(\xi))) \\
\vee\ \exists\, \chi, \xi \in \Phi_i^- : \psi = \underline{since}^-(\chi, \xi) \wedge (\varphi = \underline{since}^-(\tau_\mathcal{A}(\chi), \tau_\mathcal{A}(\xi))) \\
\vee\ \exists\, i \in \mathbb{N};\ \chi \in \Phi_i^- : \psi = \underline{before}^-(i, \chi) \wedge (\varphi = \underline{before}^-(i, \tau_\mathcal{A}(\chi))) \\
\vee\ \exists\, l, x, y \in \mathbb{N};\ \chi \in \Phi_i^- : \psi = \underline{replim}^-(l, x, y, \chi) \wedge (\varphi = \underline{replim}^-(l, x, y, \tau_\mathcal{A}(\chi))) \\
\vee\ \exists\, i \in \mathbb{N};\ \chi, \xi \in \Phi_i^- : \psi = \underline{repsince}^-(i, \chi, \xi) \wedge (\varphi = \underline{repsince}^-(i, \tau_\mathcal{A}(\chi), \tau_\mathcal{A}(\xi)))
\end{array}$$

The goal of refining actions in SLPs is to derive ECA rules to be deployed for enforcement. The informal semantics of ECA rules is that when a trigger *event* is attempted and the corresponding *condition* holds true, some authorization *action* is taken. In other words, the trigger event and the condition parts together refer to a violation of the policy upon which, the authorization action takes place. For preventive enforcement, the propositions

[2]We can only refine those variable events where event name and the value of the object parameter is known. Hence $VVal = PValue \setminus \mathcal{D}$

to be checked in the condition part must refer to the past otherwise they cannot be checked at the enforcement time because running systems cannot look into the future. Therefore, an important requirement to be met for deriving ILPs from SLPs is to get those past-time equivalents that express the violations of the respective future-time policies. The next section describes a set of rules to derive the past-time formulas.

From these past-time formulas we get the conditions to be checked in corresponding ECA rules. It is not trivial to derive the trigger event from most of these formulas in an automated way. Also, the authorization action depends upon the enforcement strategy and needs to be specified additionally. The next chapter on policy derivation methodology discusses how past-time formulas are converted to ECA form. In this chapter we limit the discussion to the derivation of these formulas.

5.2. Future to Past Translation

For deriving past-time formulas that refer to the violation of a future-time OSL formula, a special proposition $\mathcal{S}tart$ is used. $\mathcal{S}tart$ denotes the moment in time when the policy is deployed and activated. This can be a policy activation event, the universally valid proposition *true*, or any other propositional formula described in OSL.

In the paragraph that follows, a methodological guidance for converting future-time formulas into past-form conditions to be checked in ECA rules is provided. The ECA rule is assumed to be applicable at the *first* violation of an obligation formula checked at any timestep $t \geq 1$; further violations are not considered. The first violation of a formula at timestep t means that the formula is false at t and after t, irrespective of what is added to the trace, the formula cannot be true. As policies with liveness properties cannot be enforced, we assume that liveness formulas are already filtered out from the set of formulas for whom the past violations are computed.

Note: The results in Section 5.2.1 were first published in [63] and were not written by the author of this thesis. Though these results have been modified by the author since the original version of [63], the core contribution of this author in this context is the proof of correctness in Section 5.2.2.

5.2.1. Deriving Past-Time Conditions

The translation function $\tau_{\mathcal{P}} : \Phi \to \Phi_i^-$, that gives the first violation of any arbitrary formula is defined as follows:

- The violation of propositions $\phi \in \Psi$ is given by simple negation: $\tau_{\mathcal{P}}(\phi)$ is transformed into $\neg\phi$. $\tau_{\mathcal{P}}(\phi \wedge \psi)$ for $\psi \in \Psi$ is transformed into $\neg\phi \vee \neg\psi$; $\tau_{\mathcal{P}}(\neg\phi)$ is transformed into ϕ; all other propositional operators can be expressed by virtue of conjunction and negation.

- $\tau_{\mathcal{P}}(\Box(\phi))$ for $\phi \in \Psi$ gives the condition $\underline{\mathit{before}}^-(1, \underline{\mathit{since}}^-(\phi, \mathcal{S}tart)) \wedge \neg\phi$; the ECA rule is applicable if ϕ has been true at every timestep since $\mathcal{S}tart$ except in the current timestep.

- $\tau_{\mathcal{P}}(\underline{until}(\phi, \psi))$ for $\phi, \psi \in \Psi$ gives the condition $\underline{before}^{-}(1, \underline{since}^{-}((\phi \wedge \neg\psi), \mathcal{S}tart)) \wedge \neg(\phi \vee \psi)$; the ECA rule is applicable if, ϕ has been true (and ψ false) at every timestep since $\mathcal{S}tart$ but in the current timestep, both ϕ and ψ are false.

- $\tau_{\mathcal{P}}(\underline{after}(n, \phi))$ for $\phi \in \Psi$ gives the condition $\underline{before}^{-}(n, \mathcal{S}tart) \wedge \neg\phi$; the ECA rule is applicable if $\mathcal{S}tart$ was true n timesteps ago and in the current timestep, ϕ is false.

- $\tau_{\mathcal{P}}(\underline{replim}(i, m, n, \phi))$ for $\phi \in \Psi$ gives the condition $\underline{before}^{-}(i, \mathcal{S}tart) \wedge (\underline{repsince}^{-}(m - 1, \phi, \mathcal{S}tart) \vee \neg(\underline{repsince}^{-}(n, \phi, \mathcal{S}tart)))$. Replim restricts the repetition of a formula in a range (from m to n) for *i* timesteps. So the ECA rule is triggered when $\mathcal{S}tart$ was *i* timesteps back and in the last *i* timesteps, there have been at most $m - 1$ occurrences of the propositional formula (via $\underline{repsince}^{-}(m - 1, \phi, \mathcal{S}tart)$) or, there have been at least $n + 1$ occurrences of the propositional formula (via $\neg(\underline{repsince}^{-}(n, \phi, \mathcal{S}tart))$). For $m = 0$, the past-form condition is $\underline{before}^{-}(i, \mathcal{S}tart) \wedge \neg(\underline{repsince}^{-}(n, \phi, \mathcal{S}tart))$.

- $\tau_{\mathcal{P}}(\underline{repuntil}(n, \psi, \phi))$ for $\phi, \psi \in \Psi$ gives the condition $\underline{since}^{-}(\neg\phi, \mathcal{S}tart) \wedge \neg\underline{repsince}^{-}(n - 1, \psi, \mathcal{S}tart) \wedge \psi$; the ECA rule is applicable if ϕ has always been false since $\mathcal{S}tart$ (via $\underline{since}^{-}(\neg\phi, \mathcal{S}tart)$); if furthermore, there have been at least n occurrences of the propositional formula ψ (via $\neg\underline{repsince}^{-}(n - 1, \psi, \mathcal{S}tart)$); 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 intended event in the transformation of ψ). Note that the semantics of $\underline{since}^{-}$ includes the current timestep.

- $\tau_{\mathcal{P}}(\underline{repmax}(n, \psi))$ for $\psi \in \Psi$ is transformed via the equivalence with the respective $\underline{repuntil}$ operator: $\underline{repmax}(n, \psi) = \underline{repuntil}(n, \psi, \underline{false})$

As there is no known general explicit translation procedure from past to future temporal logic, we restrict ourselves to propositional arguments for temporal operators. In practice, this is often sufficient because policies do not seem to contain many nested temporal operators. In those cases where there are nestings, these are either factored out by decomposition or they are so simple that they can be directly transformed on grounds of the transformation rules embodied in $\tau_{\mathcal{P}}$.

5.2.2. Proof of Correctness

For the purely propositional $\phi \in \Psi$, the violation conditions are intuitive. For the temporal operators with propositional operands, let

- $\mathit{prefix}(s, t)$ denote the prefix of a given trace s from the beginning of the trace till the given timestep t (including t)

- $\otimes$ denote trace concatenation

- $\not\models_f$ and $\not\models_{f^-}$ denote the negation of $\models_f$ and $\models_{f^-}$ respectively

  and

- $\mathit{min}\{.\}$ denote the minimum value in a set

To refer to the first violation of a formula ϕ, we build a set with those moments in time after which, ϕ cannot be true, irrespective of *any* concatenations to the trace. Such a set might be empty in case of formulas with unbounded eventuality, i.e., for formulas that express liveness properties. We argue that as liveness properties are not enforceable, they are filtered out (by an "oracle") from the set of formulas for whom the first violation is computed. So, the first violation time of ϕ is given by $min\{t \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace}, t' \in \mathbb{N}_1 : (\mathit{prefix}(s,t) \otimes s', t') \not\models_f \phi\}$. The correctness of the derivation of the past-time conditions is proven by showing that

- if the equivalent past-time condition is true, then the future-time formula does not hold,

  and

- if the equivalent past-time condition is false, then the future-time formula must be true.

The correctness argument for the temporal operators with propositional operands is as follows:

- To prove that the first violation of $\Box(\phi)$ for $\phi \in \Psi$ is correct according to the definition in Section 5.2.1, the following must be proven to be true:
  - if the equivalent past-time condition is true, then the future-time formula does not hold:

    $$\forall\, s \in \mathit{Trace};\ t \in \mathbb{N}_1;\ \phi \in \Psi : (s,t) \models_{f^-} \underline{\mathit{before}}^-(1, \underline{\mathit{since}}^-(\phi, \mathcal{S}\mathit{tart})) \wedge \neg\phi \to \big((s,t) \not\models_f \Box(\phi) \wedge t = min\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t') \otimes s', t') \not\models_f \Box(\phi)\}\big)$$

    Let us assume that $(s,t) \models_f \Box(\phi)$ holds. This means that $(s,t) \models_f \phi$ is true, which in turn means that $(s,t) \models_{f^-} \phi$ is true, because ϕ is propositional. But this contradicts the antecedent. Hence, $(s,t) \not\models_f \Box(\phi)$.

    If $t = min\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t') \otimes s', t') \not\models_f \Box(\phi)\}$ is false, then $\exists\, t'' \in \mathbb{N}_1 : t'' < t \wedge (\forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t'') \otimes s', t'') \not\models_f \Box(\phi))$. This means that $t'' < t \wedge (s, t'' \not\models_f \phi)$ and so, $t'' < t \wedge (s, t'' \not\models_{f^-} \phi)$ as ϕ is propositional. But this contradicts the antecedent $(s,t) \models_{f^-} \underline{\mathit{before}}^-(1, \underline{\mathit{since}}^-(\phi, \mathcal{S}\mathit{tart}))$.

    Hence, the consequent is proven true by contradiction.
  - if the equivalent past-time condition is false, then the future-time formula must be true:

    $$\begin{aligned}&\forall\, s \in \mathit{Trace};\ t \in \mathbb{N}_1;\ \phi \in \Psi : \big((s,t) \not\models_f \Box(\phi) \wedge\\ &\qquad t = min\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t') \otimes s', t') \not\models_f \Box(\phi)\}\big)\\ &\to (s,t) \models_{f^-} \underline{\mathit{before}}^-(1, \underline{\mathit{since}}^-(\phi, \mathcal{S}\mathit{tart})) \wedge \neg\phi\end{aligned}$$

    $(s,t) \models_{f^-} \neg\phi$ follows directly from $(s,t) \models_f \neg\phi$ because of $t = min\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t') \otimes s', t') \not\models_f \Box(\phi)\}$.

    Proving that $(s,t) \models_{f^-} \underline{\mathit{before}}^-(1, \underline{\mathit{since}}^-(\phi, \mathcal{S}\mathit{tart}))$ holds, means to show that $(s, t-1) \models_{f^-} \underline{\mathit{since}}^-(\phi, \mathcal{S}\mathit{tart})$ holds. We prove this by contradiction. Let us assume that $(s, t-1) \models_{f^-} \underline{since}^-(\phi, \mathcal{S}tart)$ is false. This means that $\exists\, t'' \in \mathbb{N}_1 : t'' < t \wedge (s, t'') \not\models_{f^-} \phi$, which in turn means that $\exists\, t'' \in \mathbb{N}_1 : t'' < t \wedge (s, t'') \not\models_f \phi$ as ϕ contains no temporal operator. But this negates the selection of t in the antecedent as the first violation time point of $\Box(\phi)$ because $\exists\, t'' < t : (\forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s, t'') \otimes s', t'') \not\models_f \Box(\phi))$. So $(s, t-1) \models_{f^-} \underline{since}^-(\phi, \mathcal{S}tart)$ must be true.

- To prove that the first violation of $\underline{until}(\phi, \psi)$ for $\phi, \psi \in \Psi$ is correct according to the definition in Section 5.2.1, the following must be proven to be true:
  - if the equivalent past-time condition is true, then the future-time formula does not hold:

$$\begin{aligned}
&\forall\, s \in \mathit{Trace};\; t \in \mathbb{N}_1;\; \phi, \psi \in \Psi :\\
&(s, t) \models_{f^-} \underline{before}^-(1, \underline{since}^-((\phi \wedge \neg\psi), \mathcal{S}tart)) \wedge \neg(\phi \vee \psi) \rightarrow\\
&\big((s, t) \not\models_f \underline{until}(\phi, \psi) \wedge\\
&\qquad t = \mathit{min}\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s, t') \otimes s', t') \not\models_f \underline{until}(\phi, \psi)\}\big)
\end{aligned}$$

    Let us assume that $(s, t) \models_f \underline{until}(\phi, \psi)$. This means that $(s, t) \models_f \phi \vee \psi$, which in turn means that $(s, t) \models_{f^-} \phi \vee \psi$, because ϕ is propositional. But this contradicts the antecedent. Hence, $(s, t) \not\models_f \underline{until}(\phi, \psi)$.

    If $t = \mathit{min}\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s, t') \otimes s', t') \not\models_f \underline{until}(\phi, \psi)\}$ is false, then $\exists\, t'' \in \mathbb{N}_1 : t'' < t \wedge (s, t'' \not\models_f (\phi \vee \psi))$. This means that there exists a timestep $t'' < t : (s, t'' \not\models_{f^-} (\phi \vee \psi))$ as ϕ is propositional. But this contradicts the antecedent $(s, t) \models_{f^-} \underline{before}^-(1, \underline{since}^-((\phi \wedge \neg\psi), \mathcal{S}tart))$. Hence, the consequent is proven true by contradiction.

  - if the equivalent past-time condition is false, then the future-time formula must be true:

$$\begin{aligned}
&\forall\, s \in \mathit{Trace};\; t \in \mathbb{N}_1;\; \phi, \psi \in \Psi : \big((s, t) \not\models_f \underline{until}(\phi, \psi) \wedge\\
&\quad t = \mathit{min}\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s, t') \otimes s', t') \not\models_f \underline{until}(\phi, \psi)\}\big)\\
&\rightarrow (s, t) \models_{f^-} \underline{before}^-(1, \underline{since}^-((\phi \wedge \neg\psi), \mathcal{S}tart)) \wedge \neg(\phi \vee \psi)
\end{aligned}$$

    Because $\phi, \psi \in \Psi$, $(s, t) \models_{f^-} \neg(\phi \vee \psi)$ follows directly from $(s, t) \models_f \neg(\phi \vee \psi)$ because $t = \mathit{min}\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s, t') \otimes s', t') \not\models_f \underline{until}(\phi, \psi)\}$.

    Proving that $(s, t) \models_{f^-} \underline{before}^-(1, \underline{since}^-((\phi \wedge \neg\psi), \mathcal{S}tart)) \wedge \neg(\phi \vee \psi)$ holds, means to show that $(s, t-1) \models_{f^-} \underline{since}^-((\phi \wedge \neg\psi), \mathcal{S}tart)$ holds. We can again prove this by contradiction. Let us assume that $(s, t-1) \models_{f^-} \underline{since}^-((\phi \wedge \neg\psi), \mathcal{S}tart)$ is false. This means that $\exists\, t'' \in \mathbb{N}_1 : t'' < t \wedge (s, t'') \not\models_{f^-} \phi \wedge \neg\psi$, which in turn means that $\exists\, t'' \in \mathbb{N}_1 : t'' < t \wedge (s, t'') \not\models_f \phi \wedge \neg\psi$ as $\phi, \psi \in \Psi$. This means that $\exists\, t'' \in \mathbb{N}_1 : t'' < t \wedge \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s, t'') \otimes s', t'') \not\models_f \underline{until}(\phi, \psi)$. But this negates the selection of t in the antecedent as the earliest point in time for the violation of $\underline{until}(\phi, \psi)$. So $(s, t-1) \models_{f^-} \underline{since}^-((\phi \wedge \neg\psi), \mathcal{S}tart)$ must be true. Hence, the consequent is proven true.

- To prove that the first violation of $\underline{after}(n, \phi)$ for $\phi \in \Psi$ is correct according to the definition in Section 5.2.1, the following must be proven to be true:

- if the equivalent past-time condition is true, then the future-time formula does not hold:

$$\begin{aligned}&\forall\, s \in \mathit{Trace};\ t \in \mathbb{N}_1;\ \phi \in \Psi;\ \exists\, n \in \mathbb{N} : (s,t) \models_{f^-} \underline{\mathit{before}}^-(n, \mathcal{S}tart) \wedge \neg\phi \rightarrow\\ &\big((s,t) \not\models_f \underline{\mathit{after}}(n,\phi) \wedge\\ &\quad t = \mathit{min}\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t') \otimes s', t') \not\models_f \underline{\mathit{after}}(n,\phi)\}\big)\end{aligned}$$

Assume, $\exists\, t'' \in \mathbb{N}_1 : t'' < t \wedge (\forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t') \otimes s', t') \not\models_f \underline{\mathit{after}}(n,\phi))$. Then the monitoring of the future formula $\underline{\mathit{after}}(n,\phi)$ started at least $n+1$ timesteps ago. This contradicts $(s,t) \models_{f^-} \underline{\mathit{before}}^-(n, \mathcal{S}tart)$. So, $t = \mathit{min}\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t') \otimes s', t') \not\models_f \underline{\mathit{after}}(n,\phi)\}$ holds.

If $(s,t) \models_f \underline{\mathit{after}}(n,\phi)$, then $(s,t) \models_f \phi$ because the monitoring of the future formula $\underline{\mathit{after}}(n,\phi)$ started n timesteps ago, as shown above. This means that $(s,t) \models_{f^-} \phi$ as ϕ is propositional. But this contradicts $(s,t) \models_{f^-} \neg\phi$ in the antecedent. Hence, the consequent is proven true by contradiction.

- if the equivalent past-time condition is false, then the future-time formula must be true:

$$\begin{aligned}&\forall\, s \in \mathit{Trace};\ t \in \mathbb{N}_1;\ \phi \in \Psi;\ \exists\, n \in \mathbb{N} : \big((s,t) \not\models_f \underline{\mathit{after}}(n,\phi) \wedge\\ &\quad t = \mathit{min}\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t') \otimes s', t') \not\models_f \underline{\mathit{after}}(n,\phi)\}\big) \rightarrow\\ &(s,t) \models_{f^-} \underline{\mathit{before}}^-(n, \mathcal{S}tart) \wedge \neg\phi\end{aligned}$$

Let us assume that $(s,t) \models_{f^-} \underline{\mathit{before}}^-(n, \mathcal{S}tart)$ is false. Then $\mathcal{S}tart$ was true m timesteps ago such that either $m > n$ or $m < n$. The first case, $m > n$ means that $\underline{\mathit{after}}(n,\phi)$ was violated at a timestep before t. But this contradicts the selection of t as the earliest violation of $\underline{\mathit{after}}(n,\phi)$. So m cannot be greater than n. For the case of $m < n$, $\underline{\mathit{after}}(n,\phi)$ is not violated at timestep t which again contradicts the antecedent. Therefore $m = n$ and $(s,t) \models_{f^-} \underline{\mathit{before}}^-(n, \mathcal{S}tart)$ must hold.

Now, if $(s,t) \models_{f^-} \neg\phi$ is false, i.e., $(s,t) \models_{f^-} \phi$, then $(s,t) \models_f \phi$. But this contradicts $(s,t) \not\models_f \underline{\mathit{after}}(n,\phi)$. So $(s,t) \models_{f^-} \neg\phi$ must be also true.

Hence, the consequent is proven true by contradiction.

- To prove that the first violation of $\underline{\mathit{replim}}(i,m,n,\phi)$ for $\phi \in \Psi$ is correct according to the definition in Section 5.2.1, the following must be shown to hold:
  - if the equivalent past-time condition is true, then the future-time formula does not hold:

$$\begin{aligned}&\forall\, s \in \mathit{Trace};\ t \in \mathbb{N}_1;\ \phi \in \Psi;\ \exists\, i,m,n \in \mathbb{N} : (s,t) \models_{f^-} \underline{\mathit{before}}^-(i, \mathcal{S}tart) \wedge\\ &\qquad (\underline{\mathit{repsince}}^-(m-1, \phi, \mathcal{S}tart) \vee \neg(\underline{\mathit{repsince}}^-(n, \phi, \mathcal{S}tart)))\\ &\rightarrow \big((s,t) \not\models_f \underline{\mathit{replim}}(i,m,n,\phi) \wedge\\ &\quad t = \mathit{min}\{t' \in \mathbb{N}_1 \mid \forall\, s' \in \mathit{Trace} : (\mathit{prefix}(s,t') \otimes s', t') \not\models_f \underline{\mathit{replim}}(i,m,n,\phi)\}\big)\end{aligned}$$

If $(s,t) \models_f \underline{replim}(i,m,n,\phi)$, then
(i) looking into the future from $\mathcal{S}tart$, in the next i timesteps, ϕ occurs minimum m times, and
(ii) from $\mathcal{S}tart$, in the next i timesteps, ϕ occurs maximum n times

(i) contradicts $(s,t) \models_{f^-} \underline{before}^-(i, \mathcal{S}tart) \wedge (\underline{repsince}^-(m-1,\phi,\mathcal{S}tart))$ and (ii) contradicts $(s,t) \models_{f^-} \underline{before}^-(i,\mathcal{S}tart) \wedge \neg(\underline{repsince}^-(n,\phi,\mathcal{S}tart))$. Therefore, (i)∧(ii) contradicts $(s,t) \models_{f^-} \underline{before}^-(i,\mathcal{S}tart) \wedge (\underline{repsince}^-(m-1,\phi,\mathcal{S}tart)$ $\vee \neg(\underline{repsince}^-(n,\phi,\mathcal{S}tart)))$.

If $t = min\{t' \in \mathbb{N}_1 \mid \forall s' \in Trace : (prefix(s,t') \otimes s', t') \not\models_f \underline{replim}(i,m,n,\phi)\}$ is false then $\exists t'' \in \mathbb{N}_1 : t'' < t \wedge \forall s' \in Trace : (prefix(s,t'') \otimes s', t'') \not\models_f \underline{replim}(i,m,n,\phi)\}$. This means that $\mathcal{S}tart$ was at least $n+1$ timesteps ago. But this contradicts $(s,t) \models_{f^-} \underline{before}^-(i,\mathcal{S}tart)$. Therefore the consequent is proven true by contradiction.

– if the equivalent past-time condition is false, then the future-time formula must be true:

$$\begin{aligned}&\forall s \in Trace;\ t \in \mathbb{N}_1;\ \phi \in \Psi;\ \exists i,m,n \in \mathbb{N} : \big((s,t) \not\models_f \underline{replim}(i,m,n,\phi) \wedge \\ &t = min\{t' \in \mathbb{N}_1 \mid \forall s' \in Trace : (prefix(s,t') \otimes s', t') \not\models_f \underline{replim}(i,m,n,\phi)\}\big) \rightarrow \\ &(s,t) \models_{f^-} \underline{before}^-(i,\mathcal{S}tart) \wedge (\underline{repsince}^-(m-1,\phi,\mathcal{S}tart) \\ &\quad \vee\neg(\underline{repsince}^-(n,\phi,\mathcal{S}tart)))\end{aligned}$$

If $(s,t) \models_{f^-} \underline{before}^-(i,\mathcal{S}tart)$ is false then $\mathcal{S}tart$ was true in a timestep j such that either $j < i$ or $j > i$. If $j < i$, then $\underline{replim}(i,m,n,\phi)$ is not violated at t. If $j > i$, then $\underline{replim}(i,m,n,\phi)$ was false at a timestep earlier than t. This contradicts the choice of t in the antecedent. Hence $(s,t) \models_{f^-} \underline{before}^-(i,\mathcal{S}tart)$ holds.

Now if, $(s,t) \models_{f^-} \underline{repsince}^-(m-1,\phi,\mathcal{S}tart) \vee \neg(\underline{repsince}^-(n,\phi,\mathcal{S}tart))$ is false, then $(s,t) \models_{f^-} \neg\underline{repsince}^-(m-1,\phi,\mathcal{S}tart) \wedge (\underline{repsince}^-(n,\phi,\mathcal{S}tart))$ must be true. This means that ϕ is true at least m times and at most n times since start. That is, looking into the future from $\mathcal{S}tart$, $(s,t) \models_f \underline{replim}(i,m,n,\phi)$ holds as ϕ is propositional. But this contradicts the antecedent. Thus proved that $(s,t) \models_{f^-} \underline{repsince}^-(m-1,\phi,\mathcal{S}tart) \vee \neg(\underline{repsince}^-(n,\phi,\mathcal{S}tart))$ also holds.

- To prove that the first violation of $\underline{repuntil}(n,\psi,\phi)$ for $\phi,\psi \in \Psi$ is correct according to the definition in Section 5.2.1, the following must be shown to hold:
  – if the equivalent past-time condition is true, then the future-time formula does not hold:

$$\begin{aligned}&\forall s \in Trace;\ t \in \mathbb{N}_1;\ \phi,\psi \in \Psi;\ \exists n \in \mathbb{N} : (s,t) \models_{f^-} \underline{since}^-(\neg\phi,\mathcal{S}tart) \\ &\wedge\neg\underline{repsince}^-(n-1,\psi,\mathcal{S}tart) \wedge \psi \rightarrow \\ &\big((s,t) \not\models_f \underline{repuntil}(n,\psi,\phi) \wedge \\ &\quad t = min\{t' \in \mathbb{N}_1 \mid \forall s' \in Trace : (prefix(s,t') \otimes s', t') \not\models_f \underline{repuntil}(n,\psi,\phi)\}\big)\end{aligned}$$

If $(s,t) \models_f \underline{repuntil}(n,\psi,\phi)$ holds, then, either ϕ is true in t or at a timestep $t'' < t$, or ψ must be true at most n times till (and including) t. But this violates the antecedent $(s,t) \models_{f^-} \underline{since}^-(\neg\phi, \mathcal{S}tart) \wedge \neg\underline{repsince}^-(n-1, \psi, \mathcal{S}tart) \wedge \psi$. Therefore, $(s,t) \not\models_f \underline{repuntil}(n, \psi, \phi)$ must be true.

If $t = min\{t' \in \mathbb{N}_1 \mid \forall\, s' \in Trace : (prefix(s,t') \otimes s', t') \not\models_f \underline{repuntil}(n, \psi, \phi)\}$ is false, then there is at least one timestep $t'' < t$ when ψ has already been true at least $n+1$ times starting from $\mathcal{S}tart$, as ϕ was never true after $\mathcal{S}tart$. But this contradicts the antecedent that says that ψ becomes true the $n + 1^{st}$ time in the timestep t. Therefore the consequent is proven true by contradiction.

– if the equivalent past-time condition is false, then the future-time formula must be true:

$$\forall\, s \in Trace;\ t \in \mathbb{N}_1;\ \phi, \psi \in \Psi;\ \exists\, n \in \mathbb{N} : \big((s,t) \not\models_f \underline{repuntil}(n, \psi, \phi) \wedge$$
$$t = min\{t' \in \mathbb{N}_1 \mid \forall\, s' \in Trace : (prefix(s,t') \otimes s', t') \not\models_f \underline{repuntil}(n, \psi, \phi)\}\big) \rightarrow$$
$$(s,t) \models_{f^-} \underline{since}^-(\neg\phi, \mathcal{S}tart) \wedge \neg\underline{repsince}^-(n-1, \psi, \mathcal{S}tart) \wedge \psi$$

Let $(s,t) \models_{f^-} \underline{since}^-(\neg\phi, \mathcal{S}tart) \wedge \neg\underline{repsince}^-(n-1, \psi, \mathcal{S}tart) \wedge \psi$ be false. Then ϕ has been true at least once since $\mathcal{S}tart$ or, ψ has been true at most n times including the current timestep, or both. Looking into the future from $\mathcal{S}tart$, this contradicts the selection of t as the earliest moment in time for $(s,t) \not\models_f \underline{repuntil}(n, \psi, \phi)$. Therefore $(s,t) \models_{f^-} \underline{since}^-(\neg\phi, \mathcal{S}tart) \wedge \neg\underline{repsince}^-$ $(n-1, \psi, \mathcal{S}tart) \wedge \psi$ holds. Hence, the consequent is proven true.

The next section demonstrates policy derivation using action refinement for different use cases. Before that, Algorithm 1 recaps action refinement in terms of the inputs and outputs of the different transformations that the OSL formula goes through.

ALGORITHM 1: Action refinement in policies

```
input          : formula $\psi \in \Phi$
output         : formula $\psi' \in \Phi_i^-$
local variables: temp1,temp2,temp3 $\in \Phi_i^-$
1 $temp1 \leftarrow \tau_{\mathcal{P}}(\psi)$; //get past condition
2 $temp2 \leftarrow \tau_{\mathcal{E}}(temp1)$; //action refined to transformers
3 $temp3 \leftarrow \tau_{\sigma}(temp1)$; //state-based action refinement
4 $\psi' \leftarrow \underline{or}^-(temp2, temp3)$; //combine both refinements
```

5.3. Evaluation

The refinement of actions with past form translations of specification-level policies is evaluated through several examples. For this, the example scenarios introduced in Chapter 4 are revisited. Actions are refined to transformers using the domain models described in Figures 4.4–4.7 (pages 46–48). The following policies are considered:

1. Friends must never copy this photo (OSN: running example, Section 5.3.1.1).

2. This video must be deleted after 30 days (OSN: running example, Section 5.3.1.2).

3. This message should not be sent to a destination more than 2 times in the next 24 hours (mobile application, Section 5.3.2.1).
4. Don't use data older than 3 days (Java third-party application, Section 5.3.3.1)

SLPs are translated at the data consumer's end with technical details in the ILPs reflecting the configurations of the data consumer's systems.

Before moving on to the respective examples, a rough overview of connecting data and containers: as mentioned earlier, data is an abstract concept that reflects how end users see the content of different containers. As such, data exists only in the minds of different persons who want to protect or misuse it, e.g., users and attackers. In running systems, only the corresponding containers exist. Therefore, policies (both for data and for containers) are specified on concrete containers in running systems. The data stored in a concrete container is identified by creating a data ID and mapping it to the container. All further copies of data are tracked based on this mapping. Hence data is identified by the *initial container* in which it appears in the concrete system. Data appears in different containers at different layers. The infrastructure is capable of tracking which containers at the data consumer's side store data at which layer of abstraction. So, for example, the infrastructure knows that Alice's photo is received by Bob at the web browser level in "http://fw-../9c73d9b7ff.jpg"; is stored in "/home/../7B835Bd01" in the cache folder and rendered in window "0x1a00005" in X11. This information is used to create initial mappings of data and container for every usage-controlled layer at Bob's end. The container is substituted by the data ID in the policy before the policy is deployed. Further details of how the mapping between data and container takes place are in Section 6.2.

Once the policy is deployed, with the usage control infrastructure, it is possible to track multiple representations of the same data at and across different abstract layers in a system. Data appears in different containers at different layers but all of these containers are mapped to the same data ID and that data ID also appears in the deployed policy.

A data flow model per layer is used to keep track of all the copies of that data in each layer. The relevant infrastructure components at respective layers communicate with each other in order to track all copies of data across all layers.

In the following section, the symbols $\rightarrow, \neg, \wedge, \vee, \forall, \exists, \in$ will be used instead of $\underline{implies}^{-}$, $\underline{not}^{-}$, $\underline{and}^{-}$, $\underline{or}^{-}$, $\underline{forall}$, $\underline{exists}$ and $\underline{in}$ for simplicity's sake.

5.3.1. Example One: Online Social Network

Revisiting the running example, actions are refined in two policies that Alice specifies, to be enforced in her friend Bob's machine. The first example refines the copy action.

5.3.1.1. Refining *Copy Photo*

Alice specifies a policy on the photo she accesses in her machine via URL "http://fw-../9c73d9b7ff.jpg". Because this photo enters Bob's machine at the web browser level via the same URL, this URL refers to the initial container at Bob's end.

The SLP "Friends must never copy this photo" on data in "http://fw-../9c73d9b7ff.jpg", expressed in future OSL is: $\Box(\neg(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\})))$. It is of the form $\Box(\varphi)$ where $\varphi = \neg(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\}))$.

Past form: $\tau_{\mathcal{P}}$ gives us the past-time condition to be checked in the respective ECA rules. $\tau_{\mathcal{P}}(\Box(\varphi)) = \underline{before}^{-}(1, \varphi\ \underline{since}^{-} \mathcal{S}tart) \wedge \neg\varphi$ where $\mathcal{S}tart$ denotes the policy activation event. This means that the respective ECA rule is triggered when φ has always been true since the policy was activated, except the current timestep. As $\varphi = \neg(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\}))$, the ECA rule is triggered when $E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\})$ is true.

Action Refinement: For refining actions to transformers, a domain model shown in Figure 4.4 on page 46 is used. For state-based action refinement, state formula is defined in the event declaration. In this example, copy photo means "data is in more than two files". The respective state-based formula in the declaration of copy action is $\neg(\underline{isMaxIn}(x, \{UnixFile\}, 2))$ where *x* is variable data and $UnixFile$ is the class of containers at the ISM level. Action refinement, $\tau_{\mathcal{A}}(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\}))$, works as follows when the data is received at Bob's end:

1. The state-based refinement of copy uses the following event declaration: $copy \mapsto \{obj, quality, subj, location, device\} \mapsto \neg(\underline{isMaxIn}(x, \{UnixFile\}, 2))$. State-based refinement is achieved by substituting variable *x* with *http://fw-../9c73d9b7ff.jpg* in the state formula:

   $$\tau_{\sigma}(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\})) =$$
   $$\neg(\underline{isMaxIn}(http://fw-../9c73d9b7ff.jpg, \{UnixFile\}, 2)) \wedge eval(subj = friend)$$

   The container name $http://fw-../9c73d9b7ff.jpg$ is substituted by a data ID after the initial mapping of data and container takes place as described in Section 6.2 on page 82.

2. For refinement to transformers, applying $\tau_{\mathcal{E}}$, copy is first refined to {Copy&Paste, Screenshot, CopyFile} and then recursively, each of these transformers is refined till the lowest ISM level. As mentioned earlier, the usage control infrastructure is capable of figuring out which containers at first store the data in Bob's machine at different layers.[3] These initial containers per layer are shown in the policy below. For dataUsage policies, these container names don't appear in the deployed ECA rules. Before deployment, each of these containers is substituted by the same data ID in the rule.

[3]How this is actually achieved, depends upon the technical systems in place. If, e.g., Bob runs a Firefox browser on a Windows machine, a Firefox extension which is a part of the usage control infrastructure can figure out for each picture, the corresponding file in the cache folder. If from each of these containers, data flows to other containers within that layer, e.g., a photo can be copied to another file from the filepath where it is initially cached, the data flow tracking components in place recognize this. See Chapter 7 for more details on the architecture of the usage control infrastructure.

$$
\begin{aligned}
&\tau_{\mathcal{E}}(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\})) = \\
&\tau_{\mathcal{E}}(E(cmd\_copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\})) \\
&\vee\ \tau_{\mathcal{E}}(E(getImage \mapsto \{obj \mapsto 0x1a00005, subj \mapsto friend\})) \\
&\vee\ (\tau_{\mathcal{E}}(E(close \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\})) \wedge \\
&\quad \diamondsuit(\tau_{\mathcal{E}}(E(write \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\})) \wedge \\
&\quad \diamondsuit(\tau_{\mathcal{E}}(E(read \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\})) \wedge \\
&\quad \diamondsuit(\tau_{\mathcal{E}}(E(open \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\}))))))
\end{aligned}
$$

3. Following the definition of $\tau_{\mathcal{A}}$

$$
\begin{aligned}
&\tau_{\mathcal{A}}(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\})) = \\
&\tau_{\sigma}(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\})) \\
&\vee\ \ \tau_{\mathcal{E}}(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\}))
\end{aligned}
$$

the translation results from (1) and (2) above are combined to get the action-refined rule to be checked in the *condition* part of the ECA rule corresponding to the SLP:

$$
\begin{aligned}
&\neg(\underline{isMaxIn}(http://fw-../9c73d9b7ff.jpg, \{UnixFile\}, 2)) \wedge eval(subj = friend) \\
&\vee\ E(cmd\_copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\}) \\
&\vee\ E(getImage \mapsto \{obj \mapsto 0x1a00005, subj \mapsto friend\}) \\
&\vee\ (E(close \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\}) \wedge \\
&\quad \diamondsuit(E(write \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\}) \wedge \\
&\quad \diamondsuit(E(read \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\}) \wedge \\
&\quad \diamondsuit(E(open \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\})))))
\end{aligned}
$$

Assuming data ID *d1* is generated and mapped to the initial container *http://fw-../9c-73d9b7ff.jpg* and then to all the others via data flow tracking, the final formula would look like this:

$$
\begin{aligned}
&\neg(\underline{isMaxIn}(d1, \{UnixFile\}, 2)) \wedge eval(subj = friend) \\
&\vee\ E(cmd\_copy \mapsto \{obj \mapsto d1, subj \mapsto friend\}) \\
&\vee\ E(getImage \mapsto \{obj \mapsto d1, subj \mapsto friend\}) \\
&\vee\ (E(close \mapsto \{obj \mapsto d1, subj \mapsto friend\}) \wedge \\
&\quad \diamondsuit(E(write \mapsto \{obj \mapsto d1, subj \mapsto friend\}) \wedge \\
&\quad \diamondsuit(E(read \mapsto \{obj \mapsto d1, subj \mapsto friend\}) \wedge \\
&\quad \diamondsuit(E(open \mapsto \{obj \mapsto d1, subj \mapsto friend\})))))
\end{aligned}
$$

The above formula demonstrates a condition with one data ID that substitutes different container names. As the data ID – container mapping does not take place at this stage of policy derivation, the usage of containers in dataUsage policies is continued in the following examples.

The nested OSL formula that we get from action refinement can be broken down to subformulas (Fischer-Ladner closure) and each subformula is then mapped to the condition part of one ECA rule. Then, several ECA rules are generated corresponding to one SLP.

5.3.1.2. Refining *Delete Video*

The SLP "This video must be deleted after 30 days" on data in "http://fw-../myvid.mp4" is expressed in future OSL as: $\underline{after}(30, E(delete \mapsto \{obj \mapsto http://fw-../myvid.mp4\}))$ where the time unit is 'day'. For simplicity's sake, deleting the video *before* 30 days is not captured in this formula. The video is accessed and downloaded by Bob and stored in his machine as "myvid.mp4". So the policy to be translated at Bob's end is $\underline{after}(30, E(delete \mapsto \{obj \mapsto myvid.mp4\}))$. It is of the form $\underline{after}(30, \varphi)$ where $\varphi = E(delete \mapsto \{obj \mapsto myvid.mp4\})$.

Past form: $\tau_{\mathcal{P}}$ gives us the past-time condition to be checked in the respective ECA rules. $\tau_{\mathcal{P}}(30, \underline{after}(\varphi)) = \underline{before}^{-}(30, \mathcal{S}tart) \wedge \neg\varphi$ where $\mathcal{S}tart$ denotes the policy activation event. For this case, the past-time condition to be checked in the ECA rule is:

$$\underline{before}^{-}(30, \mathcal{S}tart) \wedge \neg(E(delete \mapsto \{obj \mapsto myvid.mp4\}))$$

This means that the respective ECA rule is triggered when the rule was activated 30 timesteps ago and in the current timestep, myvid.mp4 is not deleted.

Action Refinement: For refining actions to transformers, the domain model shown in Figure 4.5 on page 47 is used. For state-based action refinement, the respective event declaration is $delete \mapsto \{obj, subj, location, device\} \mapsto \underline{isOnlyIn}(x, \varnothing)$ where x is a variable data. The respective state-based formula says that the deletion of data means that "data is in none of the containers".

The next step is action refinement as described in Section 5.1.2. Following steps take place for $\tau_{\mathcal{A}}(E(delete \mapsto \{obj \mapsto myvid.mp4\}))$ at Bob's end:

1. State-based refinement is achieved by substituting variable x with *myvid.mp4* in the state formula:
   $\tau_\sigma(E(delete \mapsto \{obj \mapsto myvid.mp4\})) = \underline{isOnlyIn}(myvid.mp4, \varnothing)$

2. Applying $\tau_{\mathcal{E}}$, delete is first refined to {Remove, Overwrite} transformers and then each of these transformers is refined to {unlink, shred}.

   $$\tau_{\mathcal{E}}(E(delete \mapsto \{obj \mapsto myvid.mp4\})) =$$
   $$\tau_{\mathcal{E}}(E(Remove \mapsto \{obj \mapsto myvid.mp4\})) \vee \tau_{\mathcal{E}}(E(Overwrite \mapsto \{obj \mapsto myvid.mp4\}))$$

   which is refined to

   $$E(unlink \mapsto \{obj \mapsto myvid.mp4\}) \vee E(shred \mapsto \{obj \mapsto myvid.mp4\})$$

3. Following the definition of $\tau_{\mathcal{A}}$

   $$\tau_{\mathcal{A}}(\underline{before}^{-}(30, \mathcal{S}tart) \wedge \neg(E(delete \mapsto \{obj \mapsto myvid.mp4\}))) = \underline{before}^{-}(30, \mathcal{S}tart)$$
   $$\wedge\neg(\tau_\sigma(E(delete \mapsto \{obj \mapsto myvid.mp4\})) \vee \tau_{\mathcal{E}}(E(delete \mapsto \{obj \mapsto myvid.mp4\})))$$

the translation results from (1) and (2) above are combined to get the action-refined rule to be checked in the *condition* part of the ECA rule corresponding to the SLP:

$$\underline{before}^{-}(30, \mathcal{S}tart) \wedge \neg(\underline{isOnlyIn}(myvid.mp4, \varnothing) \vee \\ E(unlink \mapsto \{obj \mapsto myvid.mp4\}) \vee E(shred \mapsto \{obj \mapsto myvid.mp4\}))$$

The next example is from the domain of Mobile Computing: actions in a mobile application for Android are refined.

5.3.2. Example Two: Mobile Applications Domain

Based on the example domain model in Figure 4.6 on page 48, user action in the SLP "This message should not be sent to a destination more than 2 times in the next 24 hours" is refined. The SLP in OSL at the data consumer's end is: $\forall\ d\ \in\ DEST : \underline{replim}(24, 0, 2, E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})))$[4] where $DEST$ is the set of all destinations and the time unit is 'hour'. The action "send SMS" is refined at the data consumer's end as follows:

5.3.2.1. Refining *Send SMS*

The SLP is of the form $\forall\ d\ \in\ DEST : \phi$ where $\phi = \underline{replim}(24, 0, 2, E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})))$.

Past form: Applying $\tau_{\mathcal{P}}$ to the SLP, we get $\forall\ d\ \in\ DEST : \tau_{\mathcal{P}}(\phi)$ in the first step. $\tau_{\mathcal{P}}$ essentially acts on $\underline{replim}(24, 0, 2, E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})))$ which is of the form $\underline{replim}(24, 0, 2, \varphi)$ where $\varphi = E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\}))$. This gives us the final past-time formula to be checked in condition

$$\forall\ d\ \in\ DEST : \underline{before}^{-}(24, \mathcal{S}tart) \wedge \\ \neg\underline{repsince}^{-}(2, E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})), \mathcal{S}tart)$$

that is, if the ECA rule was activated 24 hours ago and the message was already sent to a destination 2 times since the rule activation, then a 3rd attempt triggers the rule:

$$\begin{aligned} &\tau_{\mathcal{P}}(\forall\ d\ \in\ DEST : \underline{replim}(24, 0, 2, E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})))) \\ = \quad &\forall\ d\ \in\ DEST : \underline{before}^{-}(24, \mathcal{S}tart) \wedge \\ &\quad \neg\underline{repsince}^{-}(2, E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})), \mathcal{S}tart) \end{aligned}$$

Action Refinement: For refining actions to transformers, the domain model shown in Figure 4.6 on page 48 is used. $\tau_{\mathcal{A}}(E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})))$ works as follows:

[4]SMS_10263 is used an as SMS identifier in this example for simplicity's sake. In reality, an SMS is uniquely identified within a phone either by a UID or by a hash of some unique parts of the SMS like the sender's phone number and the receiving time stamp; across phones, other unique content and added identifiers could be combined to generate hash and create a unique identifier for an SMS.

1. The respective event declaration is $send \mapsto \{obj, subj, dest\} \mapsto false$. The state-based refinement gives us $\tau_\sigma(E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\}))) = false$

2. Applying $\tau_\mathcal{E}$, send is first refined to SendMessage and then to the SendTextMessage transformer in the Android application.

$$\tau_\mathcal{E}(E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\}))) = \\ \tau_\mathcal{E}(E(SendMessage \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})))$$

   which is refined to

$$E(sendTextMessage \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\}))$$

3. Following the definition of $\tau_\mathcal{A}$

$$\tau_\mathcal{A}(\forall\ d\ \in\ DEST : \underline{before}^-(24, \mathcal{S}tart) \wedge \\ \neg\underline{repsince}^-(2, E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})), \mathcal{S}tart)) \\ = \forall\ d\ \in\ DEST : \underline{before}^-(24, \mathcal{S}tart) \wedge \\ \neg\underline{repsince}^-(2, \tau_\sigma(E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\}))) \\ \vee\ \tau_\mathcal{E}(E(send \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\}))), \mathcal{S}tart)$$

   the translation results from (1) and (2) above are combined to get the action-refined rule to be checked in the *condition* part of the ECA rule corresponding to the SLP:

$$\forall\ d\ \in\ DEST : \underline{before}^-(24, \mathcal{S}tart) \wedge \neg\underline{repsince}^-(2, (false\ \vee \\ E(sendTextMessage \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\}))), \mathcal{S}tart)$$

   which is equivalent to

$$\forall\ d\ \in\ DEST : \underline{before}^-(24, \mathcal{S}tart) \wedge \neg\underline{repsince}^-(2, \\ E(sendTextMessage \mapsto (\{obj \mapsto SMS\_10263, dest \mapsto d\})), \mathcal{S}tart)$$

5.3.3. Example Three: A Java Application

The original policy for this use case is "don't use outdated data" for Google+. For demonstration purpose, outdated data is interpreted as data older than 3 days. This means that if a data d was neither created nor updated in the past 3 days, it's considered outdated.

Revisiting the Birthday Reminder third-party OSN application, written in Java and modeled in Figure 4.7 on page 48, actions are refined in the following SLP: "Don't use data older than 3 days" for data stored in String object *fid* (friend ID).

This time we refer to Section 2.6 and specify this policy in past: $\boxdot(E(use \mapsto \{obj \mapsto fid\}) \rightarrow \underline{within}^-(3, E(create \mapsto \{obj \mapsto fid\})\ \vee\ E(update \mapsto \{obj \mapsto fid\})))$ where the timestep is on the scale of 'day'.[5] The actions to be defined are "create text data", "update text data" and "use text data".

[5]As already mentioned in Section 2.6, the choice of future time OSL for policy specification is driven by intuition; policies could as well be specified using past time operators. This example is used to demonstrate this fact.

5.3.3.1. Refining *Create, Update* and *Use Data*

Past form: As the policy is specified in past-time OSL, and the formula is of the form $\boxdot(\varphi)$ where $\varphi = E(use \mapsto \{obj \mapsto fid\}) \rightarrow \underline{within}^{-}(3, E(create \mapsto \{obj \mapsto fid\}) \lor E(update \mapsto \{obj \mapsto fid\}))$, the violation of the formula is given by $\neg\varphi$. The condition to be checked in the ECA rule is $E(use \mapsto \{obj \mapsto fid\}) \land \neg\underline{within}^{-}(3, E(create \mapsto \{obj \mapsto fid\}) \lor E(update \mapsto \{obj \mapsto fid\}))$.

Action Refinement: Actions are refined to transformers according to the domain model shown in Figure 4.7 on page 48. The state-based action refinement of "create data" and "update data" is given by the proposition *false* in their respective event declarations. The state-based refinement of use data is given as $\neg(\underline{isNotIn}(x, ResultSet))$ where x is variable data and $ResultSet$ is the Java's table of data representing a database result set, which is generated by executing a statement that queries the database. This means that the data is considered to be used whenever it is read from the database. The respective event declarations are as follows:
- Create: $create \mapsto \{obj, subj, location, device\} \mapsto false$
- Update: $update \mapsto \{obj, subj, location, device\} \mapsto false$
- Use: $use \mapsto \{obj, subj, location, device\} \mapsto \neg(\underline{isNotIn}(x, ResultSet))$

Action refinement, as described in Section 5.1.2 and given by, $\tau_{\mathcal{A}}(E(use \mapsto \{obj \mapsto fid\}) \land \neg\underline{within}^{-}(3, E(create \mapsto \{obj \mapsto fid\}) \lor E(update \mapsto \{obj \mapsto fid\})))$ is straightforward:

1. $\tau_{\sigma}(E(create \mapsto \{obj \mapsto fid\})) = false$
   $\tau_{\sigma}(E(update \mapsto \{obj \mapsto fid\})) = false$
   $\tau_{\sigma}(E(use \mapsto \{obj \mapsto fid\})) = \neg(\underline{isNotIn}(fid, ResultSet))$

2. Applying $\tau_{\mathcal{E}}$, create, update and use actions on text data are refined to {createData}, {updateData} and {useData} respectively. Then each of these transformers is refined to {eu.demirbas.app.Main.createData}, {eu.demirbas.app.Main.updateData} and {eu.demirbas.app.Main.useData} respectively. As the prefixes of the method names are too long, let us use a placeholder term "prefix" instead of "eu.demirbas.app.Main" here onwards. Finally,

$$\tau_{\mathcal{E}}(E(create \mapsto \{obj \mapsto fid\})) = E(prefix.createData \mapsto \{obj \mapsto fid\})$$
$$\tau_{\mathcal{E}}(E(update \mapsto \{obj \mapsto fid\})) = E(prefix.updateData \mapsto \{obj \mapsto fid\})$$
$$\tau_{\mathcal{E}}(E(use \mapsto \{obj \mapsto fid\})) = E(prefix.useData \mapsto \{obj \mapsto fid\})$$

3. Following the definition of $\tau_{\mathcal{A}}$

$$\begin{aligned}&\tau_{\mathcal{A}}(E(use \mapsto \{obj \mapsto fid\}) \land \neg\underline{within}^{-}(3, E(create \mapsto \{obj \mapsto fid\})\\&\lor E(update \mapsto \{obj \mapsto fid\}))) =\\&\tau_{\sigma}(E(use \mapsto \{obj \mapsto fid\})) \lor \tau_{\mathcal{E}}(E(use \mapsto \{obj \mapsto fid\})) \land\\&\neg\underline{within}^{-}(3, (\tau_{\sigma}(E(create \mapsto \{obj \mapsto fid\})) \lor \tau_{\mathcal{E}}(E(create \mapsto \{obj \mapsto fid\})) \lor\\&\qquad \tau_{\sigma}(E(update \mapsto \{obj \mapsto fid\})) \lor \tau_{\mathcal{E}}(E(update \mapsto \{obj \mapsto fid\}))))\end{aligned}$$

the translation results from (1) and (2) above are combined to get the action-refined rule to be checked in the *condition* part of the ECA rule corresponding to the SLP as:

$$\begin{aligned}&\neg\underline{\mathit{within}}^{-}(3, (\mathit{false} \;\vee E(\mathit{prefix.createData} \mapsto \{\mathit{obj} \mapsto \mathit{fid}\})\\ &\qquad\qquad \vee \mathit{false} \;\vee E(\mathit{prefix.updateData} \mapsto \{\mathit{obj} \mapsto \mathit{fid}\}))) \wedge\\ &(\neg\underline{\mathit{isNotIn}}(\mathit{fid}, \mathit{ResultSet}) \;\vee\; (E(\mathit{prefix.useData} \mapsto \{\mathit{obj} \mapsto \mathit{fid}\})))\end{aligned}$$

which is equivalent to

$$\begin{aligned}&\neg\underline{\mathit{within}}^{-}(3, (E(\mathit{prefix.createData} \mapsto \{\mathit{obj} \mapsto \mathit{fid}\})\\ &\qquad\qquad \vee E(\mathit{prefix.updateData} \mapsto \{\mathit{obj} \mapsto \mathit{fid}\}))) \wedge\\ &(\neg\underline{\mathit{isNotIn}}(\mathit{fid}, \mathit{ResultSet}) \;\vee\; E(\mathit{prefix.useData} \mapsto \{\mathit{obj} \mapsto \mathit{fid}\}))\end{aligned}$$

What's next? We will revisit the above examples for further derivation of policies in the next chapter.

5.4. Summary

This chapter describes a model-based policy refinement for usage control enforcement. Through this, the fundamental problem of the lack of clear semantics of actions like copy and delete is addressed. In addition, a methodological guidance to derive the past-time first violation of future OSL formulas and the corresponding proof is also provided.

Policy derivation in this chapter does not discuss the case of enforcing policies about interdependent events at multiple layers of abstraction as the the solution does not depend upon policy derivation. This problem is addressed in Section 7.3.

It is hard to establish a notion of correctness between the semantics of low-level and high-level policies. This is because the semantics of high-level propositions is not precisely defined but rather exists in the (end) user's mind. In fact, the domain metamodel and its usage in policy derivation should be seen as a way to *define* the semantics of high-level policies by assigning machine-level events and state changes to high-level actions.

The next chapter addresses the methodological part of the problem and describes the further translation of these partially derived policies that we have at the end of this chapter.

Chapter 6

The Policy Derivation Methodology

The results discussed in this chapter have already been published in [64], co-authored by the author of this thesis.

Recapping Section 4.1, there are several requirements that must be met in order to automatically derive ILPs from the SLPs. **Firstly**, SLP data and action must be refined. **Secondly**, we need to convert the future-time SLP constraints into their past forms so that enforcement mechanisms can come to decisions on grounds of the past executions in the system. **Thirdly**, we must convert the SLPs into rules of event-condition-action form. This step is non-trivial because of the fact that one SLP can be enforced in several ways (e.g., Alice's "don't copy" can be enforced either by *inhibiting* all copy attempts, or by *executing* the action with logs, or by *modifying* the data being copied). **Fourthly**, as described earlier, we must achieve the initial binding of data and containers in an automated fashion. **Additionally**, enforcement mechanisms might require some *context information* in order to decide upon the action when they are notified of an event, e.g., browsing session identifier in case of a web application. This context information must be added to the ECA rules before deployment. **Also**, a well-defined methodology for automating the policy derivation is required. The rest of this chapter describes how all of these requirements are met from the methodology point of view.

6.1. The Generic Approach to Policy Derivation

The core contribution of this chapter is a methodology for automating usage control policy derivation in order to include non-technical end users in policy specification. As a fundamental concern is about the capability of end users in terms of understanding and specifying usage control policies [59], power users help them by specifying policy templates which apply to classes and not specific instances of data. End users merely create instances of these templates while specifying policies.

In the overall policy specification and translation, the initial steps are about defining an instance of the domain metamodel, with the three domain models providing the vocabulary for specifying SLPs and their translations till the ISM layer. The definition of the models is done by the sophisticated, power user.

In the next step, the power user creates specification-level policy templates Then 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. So in the next steps, the enforcement strategies corresponding to every SLP template are chosen and event-condition-action (ECA) rules are defined.

When the end user specifies policies using the available templates, as their translation is already defined by the power user, the policies are automatically translated containing implementation-specific representations of the abstract data and action according to their refinements specified in the models. In theory, 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.

The following sections provide a methodological guidance with details for automating the policy derivation. Some phases in the generic methodology are described only at a high-level. This is because certain context-dependent information in policies can be figured out in different domains in different ways. There is no universal set of precise instructions to figure out specific context information in all the domain applications.

Regardless of this, as the policy derivation in this thesis relies on the domain expertise of the power users, it is reasonable to assume that a high-level methodological guidance at places is sufficient for a power user to fill the details according to the domain context.

6.2. The Policy Derivation Methodology

This section shows how the complete policy derivation process can be automated: the power user only configures the policy derivation in the beginning, no further human intervention is needed. Policy derivation is completed in five steps as described in the following subsections.

6.2.1. Policy Templates

Although end users would like to be able to specify security requirements, in particular usage control policies [60], they are in general not capable of reasoning holistically about such policies [59]. Hence the need for a policy specification and derivation framework that requires *simple and limited user input* for the specification of usage control policies. Our generic policy derivation approach is based on policy templates that are essentially classes of policies. Two types of policy templates are used. Their specification is part of the infrastructure setup tasks that the power user performs before the domain application is ready to be used in a usage-controlled way.

6.2.1.1. Step 1: Setting up the Policy Derivation

The power user specifies two types of templates: one set of templates for the SLPs, to be instantiated manually by an end user for each data element that he wants to protect. The second set of templates specify enforcement strategies for automatically generating ECA rules for different classes of SLPs. Automating the generation of ECA rules is motivated by the fact that the enforcement strategy (viz. execute, inhibit, modify) for each SLP is usually according to the technical infrastructure and/or organizational goals. E.g., a company enforces a policy "don't copy document" by inhibition because the company wants to prevent data leakage and its infrastructure supports it; or, a film producer enforces "don't play unpaid videos" by allowing corresponding events with a lower quality video because either it's technically not feasible to inhibit the events or he wants to give a limited preview to its prospective customers. As the reasons for deciding upon an enforcement strategy don't change very often, it is reasonable to specify the enforcement strategy for ECA rules generation in a template in order to automate the process.

6.2.1.2. Step 2: Policy Specification

We looked into usage control requirements in several domains like online social networks, ambient assisted living systems and video surveillance systems. We recognized that most of the relevant usage control policies could be specified using limited combinations of OSL operators. Based on this experience, we came up with a list of *policy patterns*, where each pattern could be used to specify *templates of policies*, explained shortly.

Table 6.1 shows a sample list of patterns of policies. This list could be extended to cover special data protection needs in specific domains. For most of the policies, we already have OSL operators and macros that cover the majority of the patterns in temporal logic. There are however some more type of policies that need to be expressed by nesting some of the OSL operators. In the following, $\{\varphi_1, \varphi_2, \varphi_3\} \in \Phi$:

Pattern text	**OSL Formula Structure**
φ_1 or φ_2 must be true	$\varphi_1 \vee \varphi_2$
φ_1 and φ_2 must be true	$\varphi_1 \wedge \varphi_2$
φ_1 must be true n timesteps after φ_2 became true	$\Box(\varphi_2 \rightarrow \underline{\mathit{after}}(n, \varphi_1))$
φ_1 must immediately succeed φ_2	$\Box(\varphi_2 \rightarrow \underline{\mathit{after}}(1, \varphi_1))$
φ_1 must always be true, e.g., "all events must be logged"	$\Box\varphi_1$
φ_1 must never be true, e.g., "never send data outside the network"	$\Box\neg\varphi_1$
φ_1 must hold until φ_2 becomes true, e.g., "no data should be copied without permission"	$\varphi_1 \ \underline{\mathit{until}} \ \varphi_2$
If φ_1 is true, then φ_2 must be true within n timesteps or φ_2 must never be true, e.g., "a gift coupon can be used only within an year from the date of purchase"	$\Box(\varphi_1 \rightarrow (\underline{\mathit{within}}(n, \varphi_2) \vee \neg\varphi_2))$

If φ_1 is true, then φ_2 must be true in every moment in n timesteps after φ_1, e.g., "if a user is fined, he is not allowed to download sensitive documents for the next 3 months"	$\Box(\varphi_1 \rightarrow \underline{during}(n, \varphi_2))$
φ_1 must be repeated minimum m times and maximum n times in the next i timesteps, e.g., "at most 50 SMS may be sent in the next 24 hours"	$\underline{replim}(i, m, n, \varphi_1)$
φ_1 must be true at most n number of times, until φ_2 becomes true, e.g., "accounts are suspended after 3 consecutive failed login attempts"	$\underline{repuntil}(n, \varphi_1, \varphi_2)$
φ_1 must be true maximum n number of times, e.g, "a user account may be restored from suspension at most 2 times"	$\underline{repmax}(n, \varphi_1)$
φ_2 must be true immediately after φ_1, until φ_3 becomes true, e.g., "discovery of malicious user activity must trigger account suspension in the next moment; the account must remain suspended until the charges are proven wrong"	$\Box(\varphi_1 \rightarrow \underline{after}(1, \varphi_2\ \underline{until}\ \varphi_3))$

Table 6.1.: Sample policy patterns

Using policy patterns, we came up with *classes of SLPs that specify constraints upon classes of data and actions* for the domains where we tested our policy specification and derivation. SLP templates are user-friendly representations of policy classes as end users need not know policy specification languages. They instantiate the policy classes using templates to specify data elements, actions and other constraints like time and cardinality (max. number of log in attempts before the account is blocked, amount, currency, etc.).

6.2.2. Policy Derivation

6.2.2.1. Step 3: Past Translation & Action Refinement

When the policy is specified at the data provider's end, first, the class of data addressed in the policy is computed using the $getclass$ function (introduced in Section 5.1.1) and this information is added to the policy. When the data and the policy are propagated to all the consumers, the binding of data and its class is also propagated, which helps in translating the policy at the consumers' ends. Then the derivation of implementation-level ECA rules from specification-level policies uses the formal policy derivation described in Chapter 5: specification-level OSL formulas are translated to their past forms and actions are refined according to the domain model.

Technically, if the SLP in Obligation Specification Language (OSL) is expressed as φ, the first output of this step is $\tau_{\mathcal{A}}(\varphi)$. After action refinement, we get a complex, nested formula that could be broken down to subformulas.[1] This way, we would get a set of ECA rules corresponding to one specification-level policy. The generic format of ECA rules at this stage is as follows (where φ_i is one subformula)

$$\begin{aligned}&Event : any\\&Condition : \varphi_i\\&Action : MODIFY/INHIBIT/EXECUTE\end{aligned}$$

To limit the set of trigger events for each ECA rule (for performance reasons), whenever φ_i is of the form $E(e) \wedge x$ or $I(e) \wedge x)$ where *x* is an OSL formula, *e* is moved to the trigger event part and only *x* is checked in the condition part of the rule. In case of multiple events, e.g., $\varphi_i = E(e_1) \wedge E(e_2) \wedge ... \wedge E(e_n) \ \wedge x$, any of the events can be the trigger event.

$$\begin{aligned}&Event : I(e)/E(e)\\&Condition : x\\&Action : MODIFY/INHIBIT/EXECUTE\end{aligned}$$

We know that one high-level policy can be enforced in many ways. Hence theoretically, the action part of ECA rules ($MODIFY/INHIBIT/EXECUTE$) cannot be specified in an automated way. In order to avoid human intervention at this stage, predefined templates that express the action part for classes of policies are used, as mentioned earlier.

6.2.2.2. Step 4: Connecting Data and Container

As such, data does not exist in the real world, except in the mind of the humans who want to protect or abuse it. In real systems, only corresponding containers exist. Therefore, an end user writes policies only on concrete containers and specifies if he means to address the data in that container (i.e., all copies of the container) or the container itself. This is done by specifying the *policy type* in the SLP. "Policy type" corresponds to the type of event in the policy: e.g., if a policy talks of *dataUsage event*, then it is called a *dataUsage policy*). Conceptually, data enters a system in an *initial container* and gets a data ID assigned. Practically, this would mean that corresponding to that container, a data ID is created and mapped to the container and the mapping is somehow stored in the usage control infrastructure. This process of initially mapping containers to data is called the *initial binding of data and container*. How this initial binding takes place and gets reflected in policies, varies according to the type of the policy:

- ContainerUsage policy: applies only to the specific container in the policy; useful in specifying integrity policies like "don't modify */config.cfg* file". There is no need of data-container binding in this case.
- DataUsage policy: applies to all copies of the container mentioned in the policy. The data-container mapping is created and reflected in the policy before deployment.

[1] We used a very simplistic approach to compute the subformulas: if $\chi = \phi \vee \psi$, then $subformula(\chi) = \{\phi, \psi\}$. Each subformula is then mapped to the condition part of one ECA rule. We recognize the fact that creating multiple ECA rules, each corresponding to one subformula might be problematic for performance during enforcement. We implemented this solution for improving the readability of the generated ECA rules.

- DataContainerUsage policy: the end user can specify a data ID for the particular container in the policy, or this data ID can be retrieved from the infrastructure if the data-containers mapping already exists, as in the case of data distribution [65]. This type of policy is useful in distributed setup when the data ID for a particular container must be maintained across machines.

- DataOnly policy: this type of policy is useful when the mapping between data and containers already exists in one machine and both data and the policy are propagated to other machines via different channels. The policy has a data ID without any container; data enters the other machines in some initial containers and the policy with the data ID applies to all of them.

In Listing B.1 in the Appendix, a concrete syntax of SLPs is described as XML schema. In this syntax, the *type* of a policy is expressed via *ParamMatchDataTypes* (lines 193 and 199).

Algorithm 2 shows all the four cases of initial binding of data and container reflected in a policy. Post initial binding, policies that are finally deployed for enforcement are of two types: *dataOnly* policies apply to all representations of data at and across layers of abstraction in a machine while *containerOnly* policies apply to the specific container mentioned in the policy.

6.2.2.3. Step 5: Adding Context-Specific Details

According to the domain and system implementations in place, there are a number of context details that might be added to ECA rules before deployment. E.g., policies in an OSN might address specific users and their relationships to the profile owner. This information is critical to the correct enforcement of the usage control policy and it can be known only at runtime by identifying the browsing session.

Together, step 4 and 5 are called *policy instantiation* and they might be switched in order, depending upon the context detail to be added and the corresponding system implementation.

6.3. The Workflows

Figure 6.1 shows the sequences of human activities for the systematic specification and translation of 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 models because this provides the vocabulary for specifying policies at the PIM level (that is, the SLPs) and their translations to the ISM layer. The platform-independent domain model is defined as the ***first task*** of the power user. As the ***second*** and the ***third tasks***, the platform and implementation -specific models of the application are defined. The ***fourth task*** is to map the model elements: high level components from the PIM level are mapped to PSM level technical

ALGORITHM 2: Initial Binding of Data and Container in a Policy

```
input           : Policy p
output          : Policy p′
local variables: Data d, Container c
switch getPolicyType(p) do
    case containerUsage
        p′ ← p;
        //the returned policy is of type containerOnly
        setPolicyType(p′, containerOnly);
    case dataUsage
        //get the value of object parameter
        c ← getObjectIn(p);
        //get the ID of the data stored in the container object
        //this method creates a data ID, if it already does not exist
        d ← getDataForContainer(c);
        //substitute existing value of object parameter with data d
        substituteObjectIn(p, c, d);
        p′ ← p;
        setPolicyType(p′, dataOnly);
    case dataContainerUsage
        d ← getDataIdIn(p);
        c ← getObjectIn(p);
        //create data ID for the given container
        createDataFor(d, c);
        substituteObjectIn(p, c, d);
        p′ ← p;
        setPolicyType(p′, dataOnly);
    case dataOnlyUsage
        p′ ← p;
        setPolicyType(p′, dataOnly);
endsw
```

representations. As the implementation-specific model is also a refinement of the platform-specific model, any power user who maps the PIM components to the PSM components also completes the mapping of the PSM-level counterparts at the ISM level. Therefore, this step is not shown as a separate task in the process.

In the ***fifth task***, the power user creates specification-level policy templates. The ***sixth task*** of the power user is to decide upon the enforcement strategy corresponding to each SLP template. These enforcement-related decisions are used to define the templates of executable event-condition-action rules as the ***seventh task***.

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

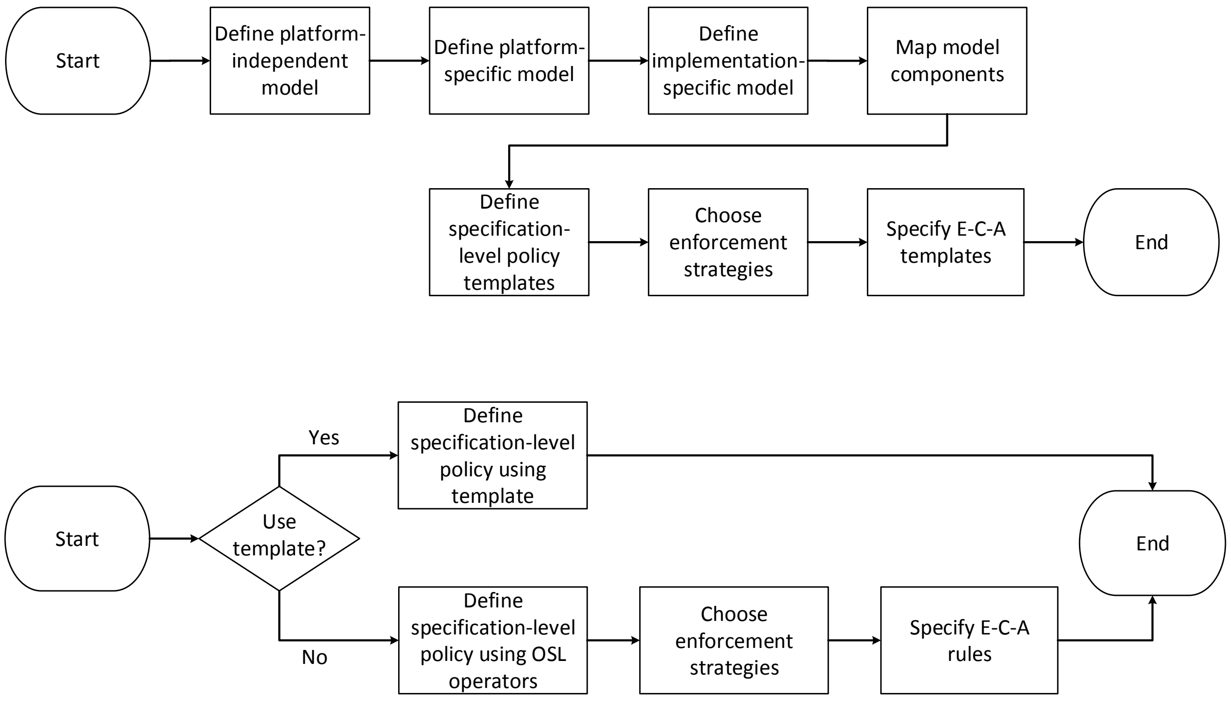

Start
Define platform-independent model
Define platform-specific model
Define implementation-specific model
Map model components
Define specification-level policy templates
Choose enforcement strategies
Specify E-C-A templates
End
Start
Use template?
Yes
No
Define specification-level policy using template
Define specification-level policy using OSL operators
Choose enforcement strategies
Specify E-C-A rules
End

Figure 6.1.: Policies specification and translation

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 as shown in the second flow-chart.

A third possibility of variation in the end user's work flow is for those end users who are technically proficient and are capable of specifying policies directly at the implementation level (ECA rules with concrete technical details). Such cases also fit perfectly in the context of this work and their possibility is not ruled out.[2] However, as this use case bypasses the typical policy specification and derivation process, it is not shown explicitly in Figure 6.1.

6.4. Implementation & Evaluation

In the following subsections, the partially derived policies and their scenarios from Section 5.3 will be revisited in the context of the respective concrete implementations.

6.4.1. Example One: Online Social Network

The generic architecture, to be described in Chapter 7 was instantiated to derive policies for the Online Social Network SCUTA[3], which already had basic usage control capabilities [51]. This is an implementation of the scenario introduced in the running example in

[2]Technically-proficient end users like system administrators might also specify policies by hand and directly deploy them via shell commands or hand-written tools.

[3]http://tiny.cc/scuta

Section 1.4.2. SCUTA was extended to provide two types of policy specification for two classes of end users. As shown in Figure 6.2, *basic users* only specify how sensitive they consider a particular data. Based on this sensitivity rating and a predefined trust model that assigns trust ratings on a scale of 0 to 1 to different types of friends and maps these trust ratings to a list of allowed actions, SLPs are generated by the system on behalf of the users. The intuition here is that basic users are incapable of understanding and specifying complex usage control policies. Further details of this type of policy specification are discussed in [51].

Advanced Users can specify their policies by instantiating admin-defined SLP templates that are loaded every time the page is requested. A template can be brought to editable mode by double-clicking on it. In Figure 6.2, we also see an SLP template in edit mode. One example policy that could be specified using this template is "acquaintances must not copy this data without permission". All these policies protect end users' data from misuse by other social network users.

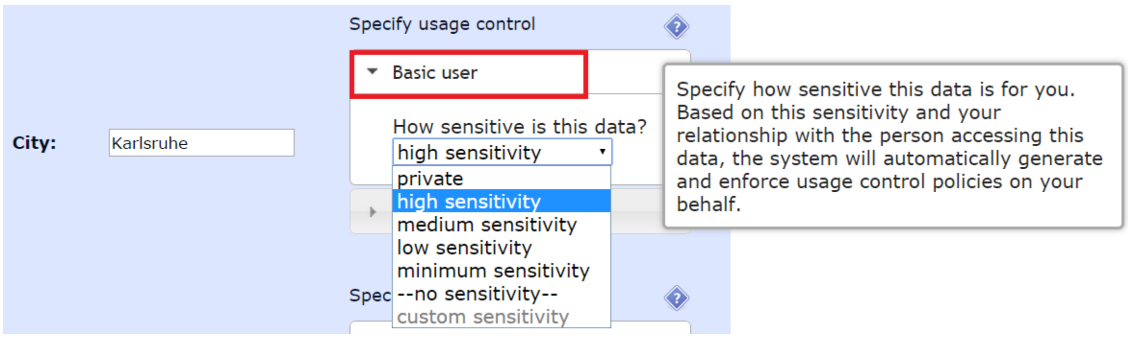
Policies were translated and their enforcement was evaluated on two system configurations. One configuration consists of Mozilla Firefox web browser [51] running on a Windows 7 operating system (for protecting cache files) [12].[4] In the second configuration, the Firefox web browser runs on an Ubuntu Linux distribution. In the following subsection, the second implementation that has been part of the running example till now will be discussed: refinement of copy and delete in Linux.

6.4.1.1. Connecting instances to classes.

As mentioned in Section 5.1.1, there is no universal way to compute the class of a given data or container. The definition of the *getclass* function varies according to the domain and its implementation. That's why the connection of a data to its class has not been implemented as part of the *policy derivation core* (see Section 7.1). Instead, the policy derivation infrastructure expects this connection information to come from the domain where policies are specified.

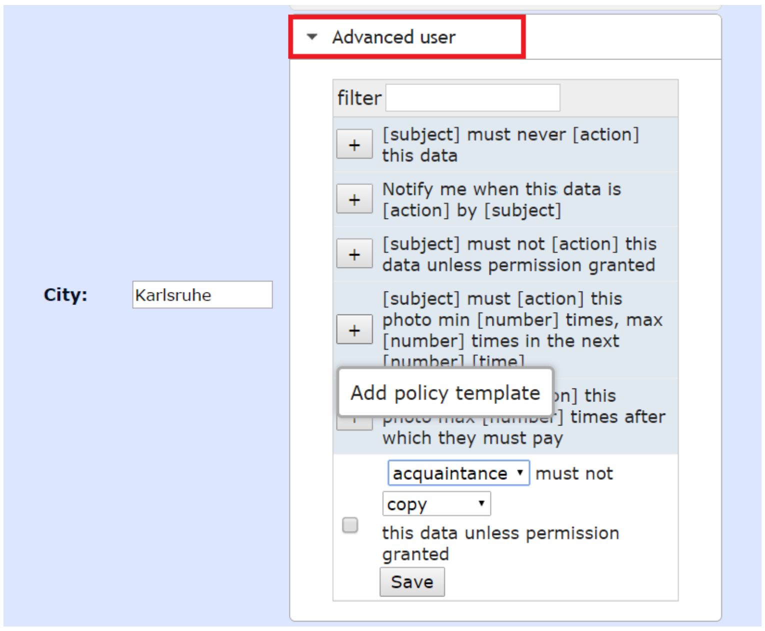
In order to connect data elements to their classes for this scenario, the database schema of SCUTA was used: the class of a data element is represented by the name of the table where that element is stored. E.g., an image stored in table "profile_photo" is a profile photo. Recall from Chapter 3 that a data flow model is used to keep track of data in multiple containers at different layers of abstraction. The data flow model of a particular layer describes all sets of containers in that layer. E.g., C_{File} is the set of all files in the data flow model of Windows 7. These names of sets of containers have a one-to-one correspondence with the container class names. So it's possible to reuse the set names as class names and connect containers to their classes, e.g., all members of C_{File} are instances of `File`.

[4]A video demonstrating policy specification and translation for this use case is available at https://www.youtube.com/watch?v=6i9Mfmbj2Xw. For space reasons, the discussion in this section is limited to selected SLP templates, instantiated policies and the different translations. Other templates are available at the SCUTA website.



Specify how sensitive this data is for you. Based on this sensitivity and your relationship with the person accessing this data, the system will automatically generate and enforce usage control policies on your behalf.

(a) 'Basic User' policies are generated by the system on behalf of the user. Details of how data sensitivities result in usage control policies are already published in related work.



[subject] must never [action] this data

Notify me when this data is [action] by [subject]

[subject] must not [action] this data unless permission granted

[subject] must [action] this photo min [number] times, max [number] times in the next [number] [time]

…n] this photo max [number] times after which they must pay

acquaintance must not copy this data unless permission granted

(b) 'Advanced Users' can specify their policies using admin-defined SLP templates

Figure 6.2.: Screenshot of SCUTA showing basic and advanced policy specification

6.4.1.2. Enforcing *Don't Copy* by *Inhibition*

Now let us go through the derivation of the example policy introduced in Section 1.4.2: "Friends must never copy this photo" on data in "http://fw-../9c73d9b7ff.jpg" and expressed in future OSL as: $\Box(\underline{not}(E(copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\})))$. The concrete syntax of this SLP that conforms to the schema of appendix Section B.1 is given in Listing 6.1.

As mentioned in Section 1.4.2, in the attacker model, the SCUTA provider is trusted. So the SLP templates (specified by the power user) and their instances (specified by end users) are stored in the SCUTA database. Table 6.2 shows an example SLP template as stored in the database.

Template ID	102
Template Text	#{[subject]}# must never _[[action]]_ this data
Data Class	city; email; photo
Policy Class	`<policy subject=\"#{subject}#\"> <obligation> <always> <not> <event name=\"_[action]_\"  tryEvent=\"true\"> <params> <param class=\"@{class}@\" name=\"obj\" value= \"@{obj}@\" policyType=\"dataUsage\"/> </params> </event> </not> </always> </obligation> </policy>`

Table 6.2.: An example SLP template, specified by the power user

- Each template is identified by a *Template ID*, e.g., 102 in Table 6.2.
- The *template text* appears in the web interface without the special character delimiters; the delimiters are interpreted by the web application logic to recognize certain fields and to fill the list of options for that field that a user can choose from: a list of actions, a list of OSN users, etc.
- *Data class* refers to the type of data for which the policy could be instantiated. The web application logic shows the template in the context of only those type of data. E.g., if the template class is photo, then that template could be instantiated for specifying policies for all photos in the OSN.
- *Policy class* is the concrete XML representation of the template text with delimiters for the aforementioned purpose.

In the GUI, the end user sees the 'template text' with subject and action parts configurable via dropdown selections. The values of data class and container (to be interpreted as data) on which the policy is specified, is filled in by the web application logic. When a template is instantiated, data and other parameter values substitute the placeholders delimited by special symbols in the policy class. Generated policies are shown to the user in form of structured, meaningful sentences in natural language.

At the backend, XML representations of the SLPs are generated that conform to the concrete syntax shown in appendix Listing B.1. SLPs in XML representations are used for further translation and enforcement steps. The SLP generated by instantiating the template of Table 6.2 is shown in Listing 6.1.

```
<policy name="nocopy" subject="friend">
  <obligation>
    <always>
      <not>
        <event name="copy" tryEvent="true">
          <params>
            <param name="obj" value="http://fw-../9c73d9b7ff.jpg"
                policytype="dataUsage"/>
          </params>
        </event>
      </not>
    </always>
  </obligation>
</policy>
```

Listing 6.1: "Don't copy" policy in concrete syntax

Policy derivation and deployment in our implementation is a one-click process. In order to generate ECA rules from the action-refined policy, ECA generation templates, per SLP template, are used. These templates are stored as part of configuration files in the usage control infrastructure.

Listing 6.2 shows the ECA template (in the format of a JSON object) for the SLP template shown in Table 6.2. Note how template ID from Table 6.2 is used to connect the two templates. The ECA template states that in the generated ECA rule, the trigger *event* is a *-event, a concrete representation of "any" event mentioned in Section 6.2.2.1 and which matches all events in the system; the *condition* comes from the action refinement which in turn takes as input the past form of the specified OSL policy; the *action* is to inhibit the event.

```
{
 "id" : "102",
 "templates" : [
  {
    "event" : "<*/>",
    "condition" : "<actionRef(pastForm)/>",
    "action" : ["<inhibit/>"],
    "type" : "preventive"
  }
 ]
},
```

Listing 6.2: ECA template for "inhibit" enforcement strategy

Using this ECA template, ECA rules corresponding to instances of SLP template 102 are generated. We will pick the example policy from where we left it in Section 5.3.1.1:

We have the action-refined rule to be checked in the *condition* part of the ECA rule from the previous chapter. Let's call it φ.

$$\begin{aligned}\varphi = \neg(&\underline{isMaxIn}(http://fw-../9c73d9b7ff.jpg, \{UnixFile\}, 2)) \wedge eval(subj = friend)\\ &\vee E(cmd\_copy \mapsto \{obj \mapsto http://fw-../9c73d9b7ff.jpg, subj \mapsto friend\})\\ &\vee E(getImage \mapsto \{obj \mapsto 0x1a00005, subj \mapsto friend\})\\ &\vee (E(write \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\}) \wedge\\ &\quad \diamondsuit(E(read \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\}) \wedge\\ &\quad \diamondsuit(E(open \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\})))))\end{aligned}$$

In the implementation, two minor modifications were introduced to the policy in derivation: one, copy is modeled as a sequence of $\langle open, read, write\rangle$ system calls without *close*. The reason is that from the usage control point of view, it is more useful to recognize a copy action when the *write* system call is triggered instead of waiting for it to complete in which case the violation of the policy would have already taken place if the policy was to inhibit the copy action.

The second modification is that the condition part is decomposed to several subformulas in order to make the generated ECA rules readable for demonstration purposes, as already mentioned in Section 6.2. This decomposition is automated. The above condition φ is decomposed into

$$\begin{aligned}
\varphi_1 &= \neg(\underline{isMaxIn}(http://fw - ../9c73d9b7ff.jpg, \{UnixFile\}, 2)) \wedge eval(subj = friend)\\
\varphi_2 &= E(cmd\_copy \mapsto \{obj \mapsto http://fw - ../9c73d9b7ff.jpg, subj \mapsto friend\})\\
\varphi_3 &= E(getImage \mapsto \{obj \mapsto 0x1a00005, subj \mapsto friend\})\\
\varphi_4 &= E(write \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\}) \wedge\\
&\quad \diamondminus(E(read \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\}) \wedge\\
&\quad \diamondminus(E(open \mapsto \{obj \mapsto /home/../7B835Bd01, subj \mapsto friend\})))
\end{aligned}$$

where $\varphi_1 \vee \varphi_2 \vee \varphi_3 \vee \varphi_4 = \varphi$. Here, φ_2, φ_3 and φ_4 are of the form $E(e) \;\wedge\; true$. So the respective *e* becomes the trigger event as described above, and the condition part of the respective ECA rules is *true*.

With these different conditions, **several** ECA rules are generated automatically according to the template of Listing 6.2. The corresponding ECA rules are shown in Listing 6.3.

```
<!-- for state-based refinement -->
<preventiveMechanism name="Mechanism_102_1_preventive">
  <trigger action="*" tryEvent="true"/>
  <condition>
   <not>
    <stateBasedFormula operator="isMaxIn"
      param1="http://fw-../9c73d9b7ff.jpg" param2="UnixFile" param3="2" />
   </not>
  </condition>
  <authorizationAction name="Authorization␣1">
    <inhibit/>
  </authorizationAction>
  <description>#{friend}# must never _[copy]_ this data</description>
</preventiveMechanism>

<!-- for FF Browser -->
<preventiveMechanism name="Mechanism_102_2_preventive">
  <trigger action="cmd_copy" tryEvent="true">
    <paramMatch name="obj" value="http://fw-../9c73d9b7ff.jpg" type="dataUsage"/>
    <paramMatch name="scope" value="536a624a87644"/>
  </trigger>
  <condition>
    <true/>
  </condition>
  <authorizationAction name="Authorization␣2">
    <inhibit/>
  </authorizationAction>
  <description>#{friend}# must never _[copy]_ this data</description>
</preventiveMechanism>

```

```
31 <!-- for X11 -->
32 <preventiveMechanism name="Mechanism_102_3_preventive">
33   <trigger action="getImage" tryEvent="true">
34     <paramMatch name="obj" value="0x1a00005" type="dataUsage"/>
35   </trigger>
36   <condition>
37     <true/>
38   </condition>
39   <authorizationAction name="Authorization␣3">
40     <inhibit/>
41   </authorizationAction>
42   <description>#{friend}# must never _[copy]_ this data</description>
43 </preventiveMechanism>
44
45 <!-- for Linux -->
46 <preventiveMechanism name="Mechanism_102_4_preventive">
47   <trigger action="write" tryEvent="true">
48     <paramMatch name="obj" value="/home/uc/.mozilla/firefox/143mebnh.dev/Cache/7B835Bd01"
         type="dataUsage"/>
49   </trigger>
50   <condition>
51    <eventually>
52     <and>
53      <eventMatch action="read" tryEvent="true">
54       <paramMatch name="obj" value="/home/uc/.mozilla/firefox/143mebnh.dev/Cache/7B835Bd01
           " type="dataUsage"/>
55      </eventMatch>
56      <eventually>
57       <eventMatch action="open" tryEvent="true">
58        <paramMatch name="obj" value="/home/uc/.mozilla/firefox/143mebnh.dev/Cache/7
            B835Bd01" type="dataUsage"/>
59       </eventMatch>
60      </eventually>
61     </and>
62    </eventually>
63   </condition>
64   <authorizationAction name="Authorization␣4">
65     <inhibit/>
66   </authorizationAction>
67   <description>#{friend}# must never _[copy]_ this data</description>
68 </preventiveMechanism>
```

Listing 6.3: ECA rules for "don't copy" by inhibition

The *scope* in line 4 and 22 of Listing 6.3 is added to the ECA rules as part of the context information that identifies each logged-in SCUTA user and his browsing session uniquely. In this case, it identifies Bob's login session as the policy is to be enforced in his machine.

Apart from the identification of a user session, scope's computation also includes the relationship of Alice with Bob (i.e., Friend). The inclusion of friendship relationship in scope means that we have both the context information and the subject of the policy merged into one for compactness. It's obvious that if Alice changes her relationship with Bob in the OSN after Bob logged in and the while his login session was still valid, the change in the relationship would not reflect at Bob's end until he logs out and logs in again (i.e., starts a new user session). Of course, the web application can enforce the update at Bob's end by forcibly logging him out [51].

The *type="dataUsage"* in lines 21, 36, 50, 56 and 60 means that the respective containers are to be interpreted as data and substituted by a data ID.

6.4.1.3. Enforcing *Don't Copy* by *Modification*

As templates are used for ECA rules generation, changing enforcement strategies (inhibit, modify, execute) is easy. The power user modifies the ECA templates and notifies the responsible component of the infrastructure for re-translating, revoking and deploying policies. In the current implementation, all deployed policies are overwritten. Modifying the implementation to selectively revoke and redeploy only those policies whose enforcement strategy has changed is trivial and can be achieved by maintaining a list of deployed ILPs with respect to each SLP.

Listing 6.4 shows another ECA template that generates ECA rules with "modify" enforcement strategy for the class of SLP in Table 6.2. In this template, we see that the authorization action type (modify vs inhibit) and the values of the modified parameters vary according to the trigger event. The line 7 in this listing specifies how the enforcement mechanism must replace the different parameters of the respective trigger events in question. The format of specifying authorization action is shown by example in Figure 6.3: *enforcement strategy, followed by event and parameter name, value pairs*. Any number of parameters can be specified. A list of keywords with fixed semantics were also used in ECA templates. Some example keywords are *inhibit*, *modify*, *allow*, *event*, *method* and *destination*.

When there is only one enforcement strategy for all ECA rules and that strategy is "inhibit" then it's sufficient to mention "inhibit" in the action part (as done in Listing 6.2). However, when there is more than one enforcement strategy for different ECA rules, then it's required to specify the authorization action part in full format as described above and shown by example in Listing 6.4:

- The ECA rule with *-event as trigger event is to enforced by inhibition.
- The ECA rule with cmd_copy as trigger event is to enforced by modification where the object parameter's new value is string "Action not allowed!".
- The ECA rule with getImage as trigger event is to enforced by modification where the planeMask parameter's new value is "0x0".
- The ECA rule with write as trigger event is to enforced by modification where the object parameter's new value is "/icon/error.jpg".

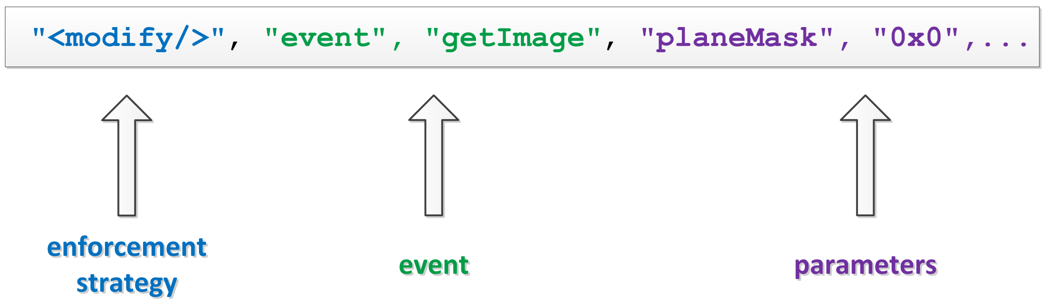


Figure 6.3.: Specification of authorization actions in ECA templates

```
{
  "id" : "102",
  "templates" : [
    {
      "event" : "<*/>",
      "condition" : "<actionRef(pastForm)/>",
      "action" : ["<inhibit/>", "event", "*",
                  "<modify/>", "event", "cmd_copy", "obj", "Action not allowed!",
                  "<modify/>", "event", "getImage", "planeMask", "0x0",
                  "<modify/>", "event", "write", "obj", "/icon/error.jpg"],
      "type" : "preventive"
    }
  ]
},
```

Listing 6.4: ECA template for "modify" enforcement strategy

The generated ECA rules are shown in Listing 6.5: line 30 says that the object of the trigger event should be substituted by a fixed value "Action not allowed!", hence modifying the event attempted in the trigger (*cmd_copy* in this case). Similarly, lines 48 and 77 specify the modification of values of planeMask and object parameters of *getImage* and *write* trigger events.

```
1   <!-- for state-based refinement -->
2   <preventiveMechanism name="Mechanism_102_1_preventive">
3     <trigger action="*" tryEvent="true">
4       <paramMatch name="scope" value="536a624a87644"/>
5     </trigger>
6     <condition>
7      <not>
8       <stateBasedFormula operator="isMaxIn"
9         param1="http://fw-../9c73d9b7ff.jpg" param2="UnixFile" param3="2" />
10     </not>
11    </condition>
12    <authorizationAction name="Authorization 1">
13      <inhibit/>
14    </authorizationAction>
15    <description>#{friend}# must never _[copy]_ this data</description>
16  </preventiveMechanism>
17
18  <!-- for FF Browser -->
19  <preventiveMechanism name="Mechanism_102_2_preventive">
20    <trigger action="cmd_copy" tryEvent="true">
21      <paramMatch name="obj" value="http://fw-../9c73d9b7ff.jpg" type="dataUsage"/>
22      <paramMatch name="scope" value="536a624a87644"/>
23    </trigger>
24    <condition>
25      <true/>
26    </condition>
27    <authorizationAction name="Authorization 2">
28      <allow>
29        <modify>
30          <parameter name="obj" value="Action not allowed!"/>
31        </modify>
32      </allow>
33    </authorizationAction>
34    <description>#{friend}# must never _[copy]_ this data</description>
35  </preventiveMechanism>
36
37  <!-- for X11 -->
38  <preventiveMechanism name="Mechanism_102_3_preventive">
39    <trigger action="getImage" tryEvent="true">
40      <paramMatch name="obj" value="0x1a00005" type="dataUsage"/>
```

```
  </trigger>
  <condition>
    <true/>
  </condition>
  <authorizationAction name="Authorization 3">
    <allow>
      <modify>
        <parameter name="planeMask" value="0x0"/>
      </modify>
    </allow>
  </authorizationAction>
  <description>#{friend}# must never _[copy]_ this data</description>
</preventiveMechanism>

<!-- for Linux -->
<preventiveMechanism name="Mechanism_102_4_preventive">
  <trigger action="write" tryEvent="true">
    <paramMatch name="obj" value="/home/uc/.mozilla/firefox/143mebnh.dev/Cache/7
          B835Bd01" type="dataUsage"/>
  </trigger>
  <condition>
   <eventually>
    <and>
     <eventMatch action="read" tryEvent="true">
      <paramMatch name="obj" value="/home/uc/.mozilla/firefox/143mebnh.dev/Cache
           /7B835Bd01" type="dataUsage"/>
     </eventMatch>
     <eventually>
      <eventMatch action="open" tryEvent="true">
       <paramMatch name="obj" value="/home/uc/.mozilla/firefox/143mebnh.dev/Cache
            /7B835Bd01" type="dataUsage"/>
      </eventMatch>
     </eventually>
    </and>
   </eventually>
  </condition>
  <authorizationAction name="Authorization 4">
    <allow>
      <modify>
        <parameter name="obj" value="/icon/error.jpg"/>
      </modify>
    </allow>
  </authorizationAction>
  <description>#{friend}# must never _[copy]_ this data</description>
</preventiveMechanism>
```

Listing 6.5: ECA rules for "don't copy" by modification

6.4.1.4. Enforcing *Delete Video* by *Execution*

The policy to be specified is "this video must be deleted after 30 days". The corresponding SLP template is shown in Table 6.3. Again, though the template text in this figure is shown with special characters, the end user only sees corresponding meaningful words in the UI. E.g., *subject* instead of #*[subject]*#. Similarly, the user is prompted to specify a number via #*NU0[[number]]NU*# and the time unit via #*TI[[time]]TI*# (e.g., line 3 in Listing 6.6).

Template ID	103
Template Text	#{[subject]}# must _[[action]]_ this data after #NU0[[number]]NU# #TI[[time]]TI#
Data Class	document; photo; video; song

Policy Class	`<policy subject=\"#{subject}#\">`
	` <obligation>`
	`  <after>`
	`   <timesteps>`
	`    <num value=\"#NU0[number]NU#\"/>`
	`    <#TI[time]TI#/>`
	`   </timesteps>`
	`   <event name=\"_[action]_\" tryEvent=\"true\">`
	`    <params>`
	`     <param class=\"@{class}@\" name=\"obj\" value=`
	`         \"@{obj}@\"policyType= \"dataUsage\" />`
	`    </params>`
	`   </event>`
	`  </after>`
	` </obligation>`
	`</policy>`

Table 6.3.: SLP template used for “delete video” policy

The SLP generated by instantiating the template of Table 6.3 is shown in Listing 6.6.

```xml
<policy name="dodelete">
  <obligation>
    <after amount="30" unit="DAYS">
      <event name="delete" tryEvent="true">
        <params>
          <param name="obj" value="http://fw-../myvid.mp4" policytype="dataUsage"/>
        </params>
      </event>
    </after>
  </obligation>
</policy>
```

Listing 6.6: “Delete video” policy in concrete syntax

As it is already mentioned, Alice specifies the policy on “http://fw-../myvid.mp4” but at Bob’s end, this data is stored in “myvid.mp4”.

This policy is enforced by *execution*: whatever event is attempted, we check if the data should be deleted at that moment and if the data is not deleted, then the responsible person is notified of the policy violation. The corresponding ECA template is shown in Listing 6.7.

```json
{
      "id" : "103",
      "templates" : [
        {
          "event" : "<*/>",
          "condition" : "<actionRef(pastForm)/>",
          "action" : ["<execute/>", "notify", "pxpID", "pxp",
                      "obj", "<@objectInstance@>",
                      "method", "email",
                      "destination", "kumari@in.tum.de",
                      "subject", "###policy violation###",
                      "message", "data not deleted!"],
          "type" : "preventive"
        }
      ]
    },
```

Listing 6.7: ECA template for "execute" enforcement strategy

Line 7 of Listing 6.7 says that the policy is to be enforced by execution of the authorization action "notify" whose object parameter's value is the same as the object of the trigger event. The method of notification is "email", destination (or recipient) of the email is "kumari@in.tum.de", the subject is "###policy violation###" and the message in the body is "data not deleted!". The term *pxp* in line 7 refers to a component, the Policy eXecution Point, which is used in the implementation of usage control enforcement for executing synchronous and asynchronous events both inside and outside the usage controlled infrastructure. Some examples of such events are send email, send SMS and delete files. *Asynchronous events* are those events for whom the policy enforcement mechanism does not wait to know if they have been successful or not; it just sends the event request to PXP and continues. A detailed discussion of the usage control policy derivation and enforcement architecture is given in Chapter 7.

For further derivation, let us revisit the example where the condition part of the ECA rule is:

$$\underline{before}^{-}(30, Start) \wedge \neg(\underline{isOnlyIn}(myvid.mp4, \varnothing) \vee E(unlink \mapsto \{obj \mapsto myvid.mp4\}) \vee E(shred \mapsto \{obj \mapsto myvid.mp4\}))$$

In our implementation, $Start$ proposition is an event *activateMechanism* with object value as the "ID of the mechanism". This event is used to activate the corresponding ECA rule after deployment. The ECA rule automatically generated according to the ECA template of Listing 6.7 is shown in Listing 6.8.

```
<preventiveMechanism name="Mechanism_103_1_preventive">
  <trigger action="*" tryEvent="true"/>
  <condition>
    <and>
     <before amount="30" unit="DAYS">
      <eventMatch action="activateMechanism" tryEvent="false">
        <paramMatch name="obj" value="Mechanism_103_1_preventive"/>
      </eventMatch>
     </before>
     <or>
      <not>
       <stateBasedFormula operator="isOnlyIn"
        param1="myvid.mp4" param2="null" />
      </not>
      <or>
       <eventMatch action="unlink" tryEvent="false">
        <paramMatch name="obj" value="myvid.mp4"/>
       </eventMatch>
       <eventMatch action="shred" tryEvent="false">
        <paramMatch name="obj" value="myvid.mp4"/>
       </eventMatch>
      </or>
     </or>
    </and>
  </condition>
  <authorizationAction name="Authorization 1">
    <allow>
```

```
      <executeSyncAction id="pxp" name="notify">
        <parameter name="pxpID" value="pxp"/>
        <parameter name="obj" value="myvid.mp4"/>
        <parameter name="method" value="email"/>
        <parameter name="destination" value="kumari@in.tum.de"/>
        <parameter name="subject" value="###policy violation###"/>
        <parameter name="message" value="data not deleted!"/>
      </executeSyncAction>
    </allow>
  </authorizationAction>
  <description>delete video after 30 days</description>
</preventiveMechanism>
```

Listing 6.8: ECA rules for "delete video" by execution

By now, we have seen how ILPs are derived from the respective SLPs that are specified using templates in one concrete implementation of our running example of an OSN. It has also been demonstrated how one SLP can be enforced with different strategies (viz. inhibition and modification) with very little effort needed for changing the infrastructure configurations. Now let us revisit the other two use cases where the automated derivation of policies has also been successfully evaluated.

6.4.2. Example Two: Mobile Applications Domain

The policy specification tool of SCUTA in the previous example, is integrated with the web application. This means that if we want to specify policies for another web application, we cannot reuse that *policy editor*. To specify policies in a generic way for several applications, an independent policy editor was developed, which runs as a daemon on a machine (e.g., a physical or virtual machine, a phone) and can be called by any layer on that machine to specify policies. The user interface and the functioning of the editor is the same as shown in the case of advanced user in Figure 6.2, the only difference being in the storage of templates and other configurations: they are stored as *config* files instead of in a database as in SCUTA. Also, the connection of instances to their classes has not been implemented as part of the policy editor. Instead, the policy editor relies on this information to come from the component that called it for policy specification. The SLP template is as shown below in Table 6.4:

In this example a specific destination for the SMS is provided (via *#TXT0[[address]]TXT#*) as there was no implementation-level support for variables in the Mobile Application use case. The concrete SLP generated by instantiating the template of Table 6.4 is shown in Listing 6.9.

```
<policy name="sendsms">
  <obligation>
    <repLim upperLimit="2" lowerLimit="0" amount="24" unit="HOURS">
      <event name="send" tryEvent="true">
        <params>
          <param name="obj" value="SMS_10263" policytype="dataUsage" />
          <param name="dest" value="+01-234-5678" />
        </params>
      </event>
    </repLim>
  </obligation>
</policy>
```

Listing 6.9: "Send SMS" policy in concrete syntax

Template ID	101
Template Text	Send SMS max #NU0[[number]]NU# times to destination #TXT0[[address]]TXT# in #NU1[[number]]NU# #TI[[time]]TI#
Data Class	sms

Policy Class	`<policy>`
	` <obligation>`
	`  <replim>`
	`   <timesteps>`
	`    <num value=\"#NU1[number]NU#\"/>`
	`    <#TI[time]TI#/>`
	`   </timesteps>`
	`   <num value=\"0\"/>`
	`   <num value=\"#NU0[[number]]NU#\"/>`
	`   <event name=\"send\" tryEvent=\"true\">`
	`    <params>`
	`     <param class=\"@{class}@\" name=\"obj\" value=`
	`         \"@{obj}@\" policyType= \"dataUsage\"/>`
	`     <param name=\"dest\" value= \"#TXT0[address]TXT#\"/>`
	`    </params>`
	`   </event>`
	`  </replim>`
	` </obligation>`
	`</policy>`

Table 6.4.: SLP template used for “send SMS” policy

This policy is enforced by *inhibition*. The corresponding ECA template is shown in Listing 6.10.

```
{
      "id" : "101",
      "templates" : [
        {
          "event" : "sendTextMessage", "obj",  "<@objectInstance@>",
                    "dest",  "<@destInstance@>",
          "condition" : "<actionRef(pastForm)/>",
          "action" : ["<inhibit/>"],
          "type" : "preventive"
        }
      ]
    },
```

Listing 6.10: ECA template for enforcing “send SMS” policy by inhibition

Based on this ECA template, the generated ECA rule in shown in Listing 6.11.

```
<!-- for state-based refinement -->
<preventiveMechanism name="Mechanism_101_preventive">
  <trigger action="sendTextMessage" tryEvent="true">
    <paramMatch name="obj" value="SMS_10263" type="dataUsage"/>
    <paramMatch name="destination" value="+01-234-5678" />
  </trigger>
  <condition>
   <and>
    <before amount="24" unit="HOURS">
     <eventMatch action="activateMechanism" tryEvent="false">
      <paramMatch name="obj" value="Mechanism_101_preventive"/>
     </eventMatch>
    </before>
    <not>
     <repsince limit="2">
      <eventMatch action="sendTextMessage" tryEvent="false">
       <paramMatch name="obj" value="SMS_10263" type="dataUsage"/>
       <paramMatch name="destination" value="+01-234-5678" />
      </eventMatch>
      <eventMatch action="activateMechanism" tryEvent="false">
       <paramMatch name="obj" value="Mechanism_101_preventive"/>
      </eventMatch>
     </repsince>
    </not>
   </and>
  </condition>
  <authorizationAction name="Authorization 1">
    <inhibit/>
  </authorizationAction>
  <description>not more than 2 SMS per day</description>
</preventiveMechanism>
```

Listing 6.11: ECA rules for "not more than 2 SMS per day"

6.4.3. Example Three: A Java Application

Another use case where this policy derivation approach has been successfully evaluated is described in [1]. The scenario is as follows: though the OSN providers regularly check third-party applications for policy violations, these checks are neither transparent nor fool-proof. No documentations were found to show the existence of any policy enforcement frameworks that performs these checks in automated ways without the side effects of manual checks.

The work described in [1] addresses this issue and enables a technical enforcement of platform policies for violations: once the third-party application providers agree to the platform policies, their infrastructures are continuously monitored for the violation of these policies. The implementation took into account the enforcement of third-party platform policies in three OSNs: Facebook, Google+ and Tumblr. An overview of the implementation is shown in Figure 6.4.

When third-party application providers access OSN users' data, they also get the respective policies from the OSN. A *proxy server*, acting as the OSN's policy manager, mediates between the OSN and the third party application and modifies responses from the OSN servers in order to achieve this. The implementation only enforces usage control policies for third party applications written in Java. This is not a fundamental limitation as the infrastructure could be easily extended by adding enforcement points (called PEP in Figure 6.4, for further details on PEP, see Section 7.2), capable of intercepting, inhibiting and possibly modifying system events at different other layers of abstraction.

Conceptually, the *modified interface (MIF)* and the *policy database (PDB)* are components of the proxy. MIF intercepts and modifies messages between the OSN and the third party application and the PDB stores all the policies in XML format.

Policy derivation requests are processed by the same usage control infrastructure servers as used by SCUTA. In this work, these components (viz. PMP, PDP and PIP) are treated as black box. They could as well be together called a component Θ that provides policy-related services. Hence they are not described here (see Chapter 7 for details on them). The

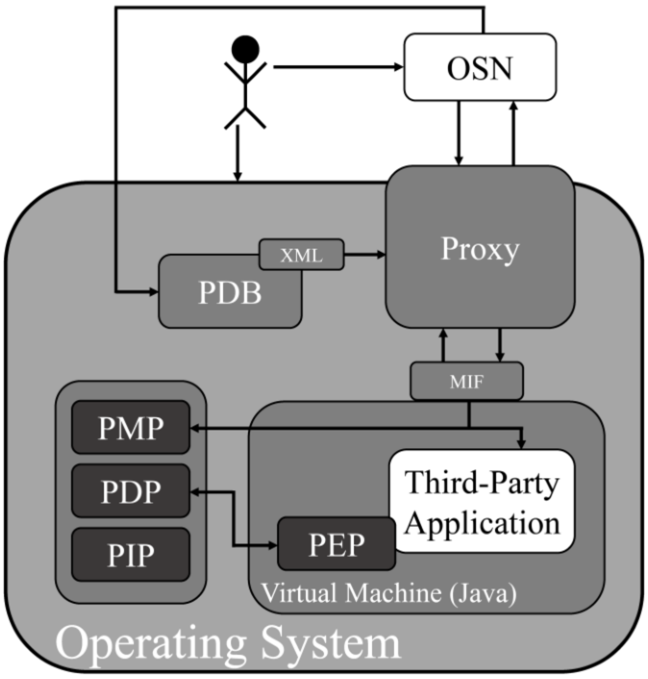


Figure 6.4.: Infrastructure for enforcing third-party platform policies [1]

power users specify a domain model and policy templates without any idea of the other details. A partial domain model for a third-party java application is shown in Figure 4.7 on page 48. It refines create, update and use actions for text data (name, ID, friendlist etc.).

One of the platform policies extracted from free text is “do not use outdated data” from Google+ [66]. We already saw a partial translation of this policy in Section 5.3.3. The policy was specified in past form. No SLP templates were used in the specification. The corresponding ECA template is shown in Listing 6.12.

```
{
      "id" : "110",
      "templates" : [
        {
          "event" : "<*/>",
          "condition" : "<actionRef(pastForm)/>",
          "action" : ["<inhibit/>"],
          "type" : "preventive"
        }
      ]
    },
```

Listing 6.12: ECA template for “inhibit outdated data usage” policy

The past condition to be checked is $\neg\underline{within}^{-}(3, (E(eu.demirbas.app.Main.createData \mapsto \{obj \mapsto fid\}) \vee E(eu.demirbas.app.Main.updateData \mapsto \{obj \mapsto fid\}))) \wedge (\neg\underline{isNotIn}(fid, ResultSet) \vee E(eu.demirbas.app.Main.useData \mapsto \{obj \mapsto fid\}))$. The ECA rule generated from this template is shown in Listing 6.13.

```
<preventiveMechanism name="Mechanism_110_preventive">
  <trigger action="*" tryEvent="true"/>
  <condition>
   <and>
    <or>
     <not>
      <stateBasedFormula operator="isNotIn" param1="fid" param2="ResultSet" />
     </not>
     <eventMatch action="eu.demirbas.app.Main.useData()" tryEvent="true">
      <paramMatch name="obj" value="fid"/>
     </eventMatch>
    </or>
    <not>
     <within amount="3" unit="DAYS">
      <or>
       <eventMatch action="eu.demirbas.app.Main.createData()" tryEvent="false">
        <paramMatch name="obj" value="fid"/>
       </eventMatch>
       <eventMatch action="eu.demirbas.app.Main.updateData()" tryEvent="false">
        <paramMatch name="obj" value="fid"/>
       </eventMatch>
      </or>
     </within>
    </not>
   </and>
  </condition>
  <authorizationAction name="Authorization 1">
    <inhibit/>
  </authorizationAction>
  <description> Prevent using outdated data</description>
</preventiveMechanism>
```

Listing 6.13: ECA rule for “inhibit outdated data usage” in 3rd party Java application

6.5. Summary

This chapter has provided a methodological guidance for transforming specification-level policies into implementation-level policies that configure enforcement mechanisms at different layers of abstraction. This methodology helps translate policies in an automated manner. The methodological guidance is at two levels: At one level, there is a description of a work-flow with all the technical (system components) and non-technical users (power users and end users). The second type of methodological guidance sets the order of computation in policy derivation (derivation of past-time rules followed by action refinement followed by ECA generation and so on).

The usage of policy templates is also proposed in order to enable end users specify complicated usage control policies with minimum effort and zero understanding of the underlying policy language. Templates are user-friendly representations of policy classes. However, no concrete syntax or design for the templates is suggested. Instead, one concrete implementation has been discussed for demonstration.

In this chapter, policy derivation and deployment components are treated as black box. Power users only specify a domain model and policy templates without any idea of the other details. The next chapter provides the details of the relevant usage control infrastructure components.

Chapter 7

Architecture

Some parts of this chapter are based on a similar chapter from our usage control book under publication [2].

In order to operationalize the policy derivation concepts, we build upon an existing generic usage control infrastructure with three main components:

1. a *Policy Enforcement Point* (PEP), able to observe, intercept, possibly modify and generate events in the system;
2. a *Policy Decision Point* (PDP), responsible for deciding on admission and necessary modifications of system events according to deployed ILPs and representing the core of the usage control monitoring logic;
3. and a *Policy Information Point* (PIP), which implements the data-flow model (Chapter 3) and maintains the data state $\sigma \in \Sigma$ and therefore provides the data-container mapping to the PDP.

   These three components are called the *core of usage control enforcement*.

To this setup, a *Policy Management Point* (PMP) was added in order to specify, translate and deploy policies. The following sections elaborate on the internal structure and the behaviour of the PMP and the interactions among all of the above usage control components.

7.1. Policies: Specification to Deployment

This section describes the so-called *policy derivation core*, which is the Policy Management Point (PMP) component with its two main subparts: the Policy Translation Point (PTP) and the Policy Editor (PE). A component diagram of the PMP with respect to other usage control components is shown in Figure 7.1.

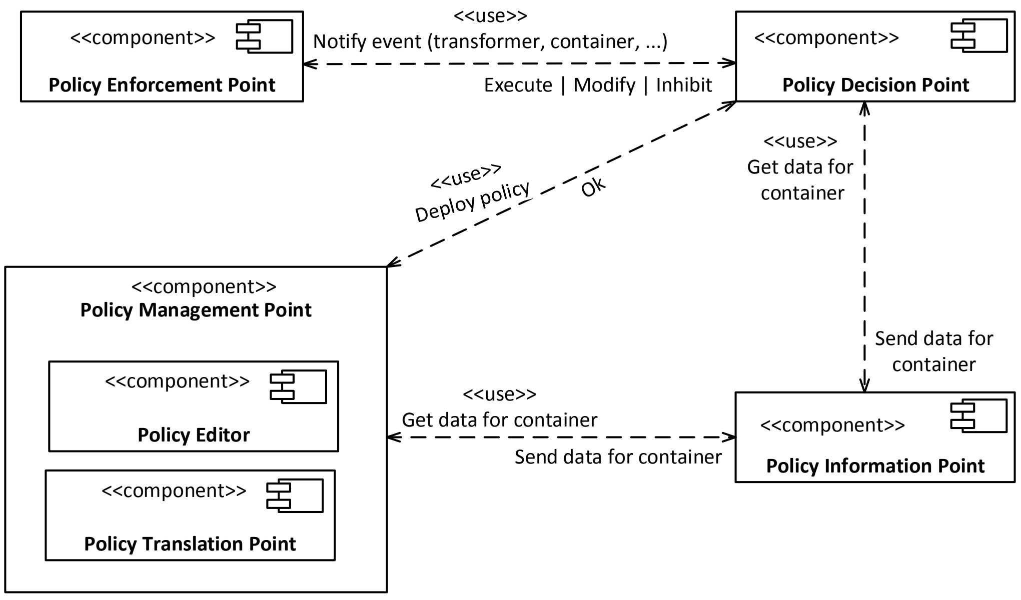


Figure 7.1.: Core components of the usage control infrastructure

The *PTP* represents the core logic of (i) deriving past-time conditions from future-time SLPs, (ii) action refinement according to the domain model and (iii) the derivation of ECA rules based on the ECA templates. Because of the sequential nature of policy derivation, the implementation instantiates the "pipe and filter" design pattern [67, 68] where the result of each step is an input to the next step. In order to generate ECA rules from the action-refined policy, ECA generation templates, per SLP template, are used. These templates are stored as part of configuration files in the PTP. All calls to the PTP go via the PMP.

Unlike the PTP, the *Policy Editor* is conceptually a part of the PMP but it can reside anywhere in the environment: it can be integrated with the domain application logic (as in the case of SCUTA) or it can run independently as a background application that is called by PEPs at different layers of abstraction for policy specification. SLP templates along with event declaration are part of the policy editor configuration.

From the methodology point of view, the PE is the entry point of policy derivation and this is where the connection of instances to their classes is generally done. But this connection of instances to their classes has not been implemented as part of the policy editor. Instead, the policy editor expects this connection information to come from the component/ layer of abstraction which calls it for policy specification. This is because of the fact that connecting instances to their classes is domain and implementation -specific and cannot be generalized to be encoded inside the policy derivation logic. So PE is called by different layers via a method where instance and class mapping is passed as parameter.

After future to past translation and action refinement, ECA rules are generated and forwarded to the PMP for instantiation and deployment, as described in Section 6.2, Algorithm 2. The corresponding sequence of component interactions is depicted in Figure 7.2. If the policy is of type "containerUsage" or "dataOnly", the PMP instantiation component skips the initial binding of the data and the container. If the policy is of type "data usage",

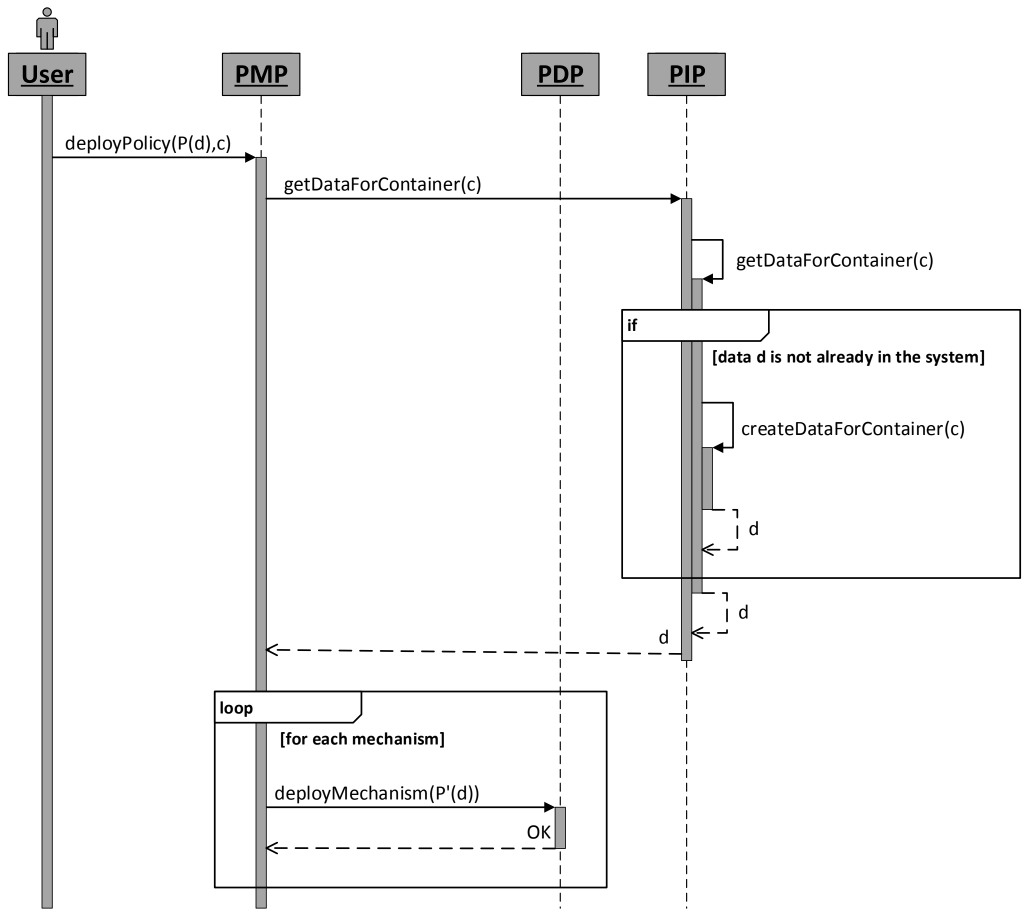


Figure 7.2.: Policy instantiation and deployment (figure adapted from [2])

the PMP queries the PIP for the data IDs mapped to the container addressed in the policy. If no data ID already exists, the PIP creates a new data ID and maps it to the container in question. The PIP then forwards the data ID which replaces the container in the policy at the PMP. If the policy type is "dataContainerUsage", the PMP notifies the PIP to create a new data with the given ID and map it to the corresponding container. After this initial binding, the policies are marked as either *containerOnly* in cases of container usage type, or as *dataOnly* in cases of the rest of the three policy types. After this, the policies are deployed at the PDP. In Figure 7.2, P and P' are the policies before and after instantiation respectively.

Policy instantiation with context details is dependent on the application technicalities; a concrete case of how this is done has been shown in Section 6.4.1.

If the domain model or the ECA templates need to be updated, the power user modifies them and notifies the PMP for re-translating, revoking and deploying policies. In the current implementation, all deployed policies are overwritten. Modifying the implementation to selectively revoke and redeploy only those policies whose enforcement strategy has changed is trivial and can be achieved by maintaining a list of deployed ILPs with respect to each SLP.

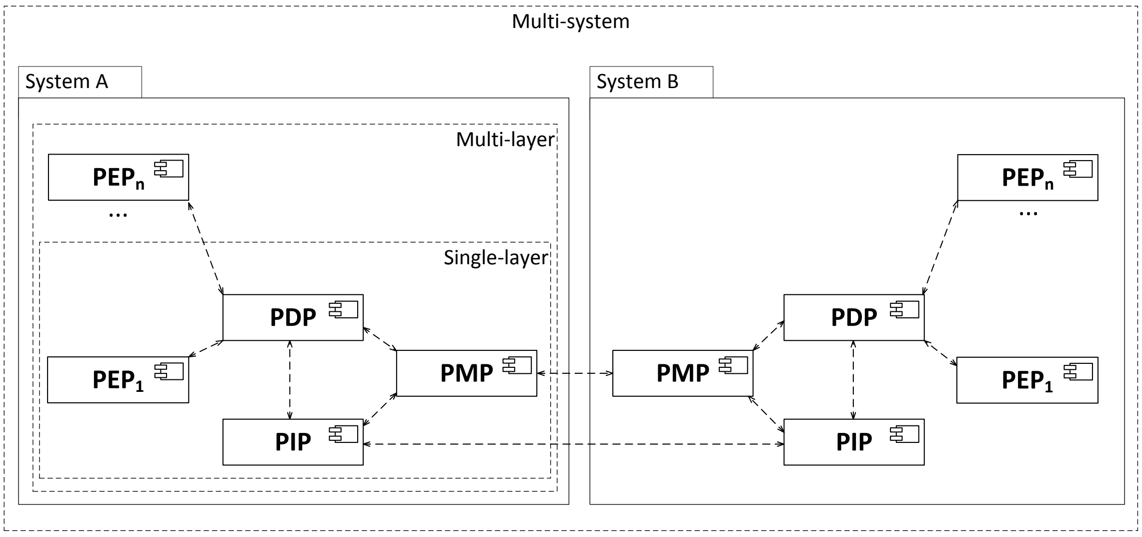


Figure 7.3.: Component structure: single-layer, multi-layer and multi-system [2]

During policy enforcement, when an event is intercepted and notified by the PEP to the PDP, the object parameter is always a container as only concrete containers exist in running systems. For a *dataOnly* policy, the PDP queries the PIP for the data-container mapping before deciding upon the enforcement mechanism. In case of *containerOnly* policies, the PDP needs no communication with the PIP as the policy and the event notification, both address containers.

Although enforcement of policies is out of the scope of this work, for the sake of completeness, an overview of the policy enforcement core is presented in the next section.

7.2. Overview of Enforcement

To ensure data protection via usage control, enforcement must be done at and across all layers and in all the machines of the distributed system. In the existing usage control scheme, the enforcement relies on one PEP per layer of abstraction with PDP, PMP and PIP shared across all layers in one machine. Figure 7.3 shows a layer-agnostic generic architecture. This generic architecture can be instantiated for arbitrary system layers.

Figure 7.3 depicts the static component structure of the overall infrastructure including i) *single-layer*, ii) *multi-layer*, and iii) *multi-system* or *multi-machine* enforcement of usage control.

In the *single-layer* case, the PEP intercepts desired and actual events of a specific system layer and forwards them to the PDP for evaluation. Based on the PDP's decision the PEP then either allows, inhibits, or modifies a signaled event, and potentially generates additional events if a matching policy demands so. To conduct this evaluation, the PDP tries to match the events, coming from the PEP, against the deployed ILPs. To evaluate the corresponding policies, in particular the condition part of the ECA rules, the PDP uses a runtime verification algorithm [69].

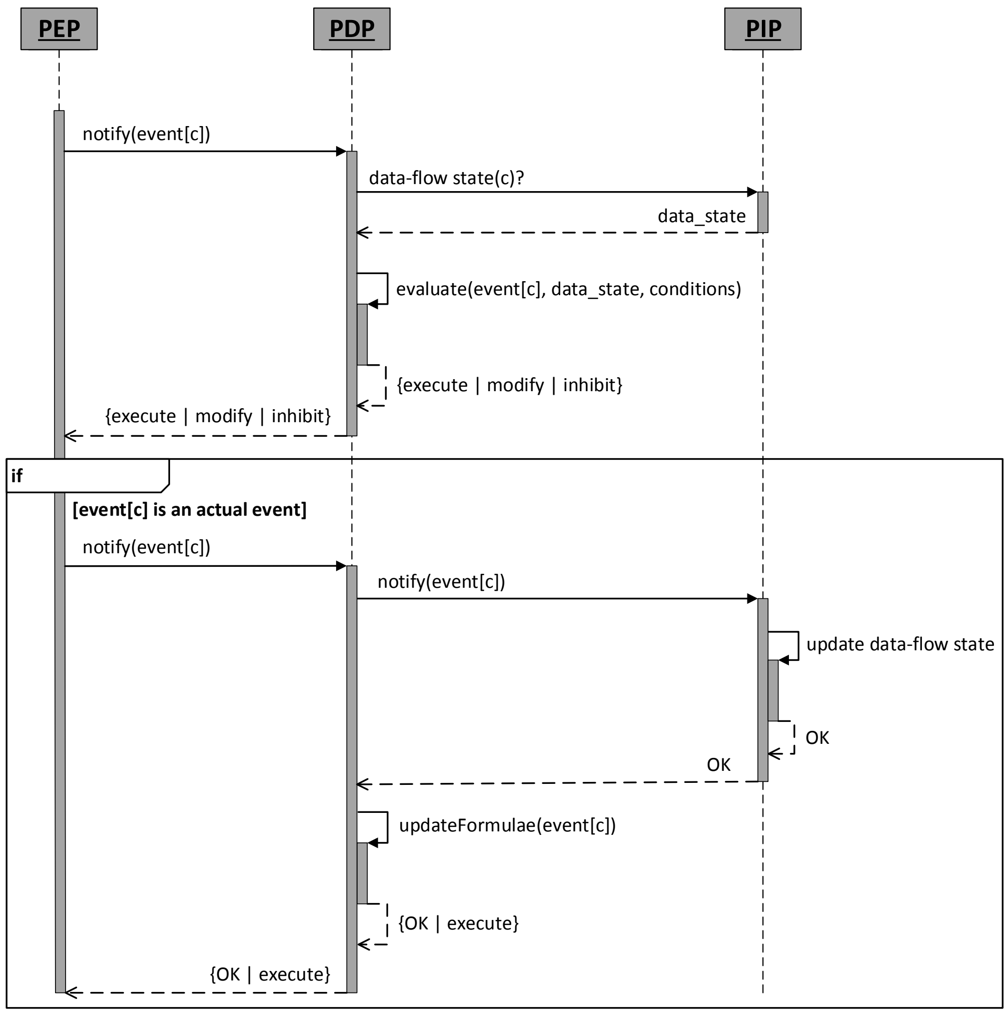


Figure 7.4.: Interaction among components during enforcement [2]

In some cases, e.g., if the trigger part of an ECA rule concerns a data-usage event, or the condition part contains a state-based formula, the PDP needs additional information for the policy evaluation. In these cases the PDP queries the PIP for data state related information to e.g., get to resolve the relationship between a data item and its representations. Finally, if the desired event was transformed into an actual event and executed by the PEP, the PEP notifies the PDP about this execution, which then in turn updates corresponding formulas. The PDP then finally notifies the PIP about the executed actual event to allow it to update its data-flow model accordingly. The evolution of the data-flow model is then internally conducted by updating the data state $\sigma \in \Sigma$ according to the defined transition relation $\mathcal{R}$ for the respective actual event. Figure 7.4 depicts the corresponding sequence of interactions at a high level.

The architecture for a *multi-layered* system corresponds to the architecture for the single layer, with the only difference that there exists a PEP for each layer of abstraction. All the different PEPs signal events to the same PDP. The PIP in this scenario is supposed to maintain a *common* state of the system *across all layers*. How this is achieved theoretically is out of the scope of this work as it does not impact the policy derivation nor does it require any further change in the architecture. The sequence diagram in this case would be identical to the one depicted in Figure 7.4 for the single layer case, where the PEP agent represents one of the multiple PEPs in the system. One exception to this rule is the case of cross-layer events in a multi-layered system, discussed in Section 7.3.

In a *multi-machine* setup, each PIP holds the data flow state of the machine (i.e., all the monitored layers of abstraction together) on which it is deployed. The product of the *storage*, *alias* and *naming* functions of all PIPs thus corresponds to the *global* data flow state *across machines*. At the same time, each PMP manages the deployed policies (ILPs) for data within the machine on which it is deployed.

In addition to what has been described in the cases of single-layer and multi-layer systems, both the PIP and the PMP are able to communicate with their respective counterparts on other machines. Thus, they allow for the exchange of cross-machine data flow information and policies whenever attempts to transfer usage controlled data across systems are observed at the operating system layer [30, 70].

7.3. Cross-layer Events

Till this point in this dissertation, only those cases of policy enforcement have been considered where events at one layer of abstraction in a system are unrelated to events at the other layers with respect to a high-level action. E.g., copying a photo via taking a screenshot is achieved independently of saving the photo in a file and the two events, when recognized on a data d in the system, are be counted as two copies of the same data.

However, many other events across different layers of abstraction in the system may together achieve one higher-level goal in the system. Such interdependent events are called *cross-layer events* in the literature [3]. E.g., *context-savepage* command at the Firefox layer and a series of *write* system calls in a Linux distribution together achieve one copy of data d. During enforcement, the usage control infrastructure must also see these events at different layers to be somehow connected, in order to identify one (and not two) copy of d. In the absence of such a connection, policies that specify cardinality constraints (e.g., "pay after three copies") or express *stateful* constraints on events (e.g., "lower buyer ranking upon copy until payment is done") will be wrongly enforced. In these examples, the buyer may be wrongly issued two invoices upon one copy or his ranking may be wrongly lowered because he did not pay for the extra copy wrongly recognized by the enforcement.

In related work [3], cross-layer events are modeled by considering different layers pairwise, e.g, Firefox and Linux. For the sake of simplicity, in this discussion, just these two layers are assumed to exist in a machine. In a generic case where the system is composed of $n \geq 2$ layers, the same approach could be applied recursively considering the n^{th} layer as layer X and the composition of the other $n-1$ layers as layer Y.

Each data-flow in the cross-layer model is supposed to start at one layer (e.g., in the Firefox), go through the other layer (e.g., in Linux) and end again at the first layer where it started (Firefox). So the layer where the event started is seen as a *long* event with duration, which from the other layer's perspective, may span over several timesteps.

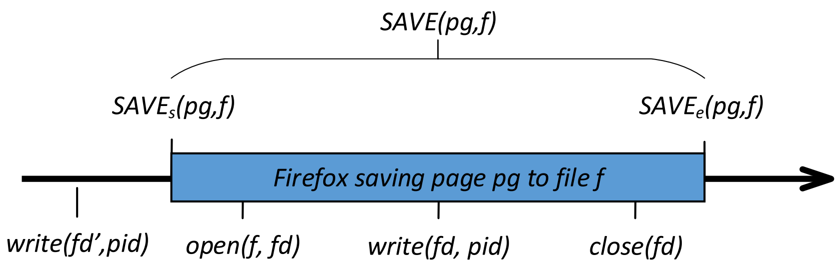


Figure 7.5.: Cross-layer events (figure adapted from [3])

An example of cross-layer events is shown in Figure 7.5. For simplicity, the context-savepage event at the Firefox layer is called $SAVE$ event. The $SAVE$ event spans over multiple timesteps from the OS layer's point of view where several events (3 system calls shown in Figure 7.5) take place in this duration. In [3], the duration of such *long* events that correspond to several events at the other layer is modeled using two atomic events that respectively represent the moment in which the long event starts (e.g., $SAVE_s$) and the moment in which the long event ends (e.g., $SAVE_e$). All events between, and including, the $SAVE_s$ and $SAVE_e$ events shown on the timeline in Figure 7.5 perform one single data flow across the two layers.

A concept of *Scope* is used to connect all events that are part of a single data flow across layers. *Scope* is essentially a label attached to each system event at all layers at the runtime. A scope is *activated* when the cross-layer flow starts and it is *deactivated* when the cross-layer flow is completed. In Figure 7.5, the scope is *activated* by the start of the $SAVE$ event (given by event $SAVE_s$) at the Firefox level and it is *deactivated* with the end of the $SAVE$ event (given by event $SAVE_e$). All events that are together a part of a single data flow (at both the layers), have the same scope label attached to them. At any moment in time, there may be multiple cross-layer events in progress (started, but not ended) and hence several scopes may be active at a moment in time in a machine. Events belonging to different cross-layers flows are distinguished from each other by their scope labels. Therefore, in this example, one execution of context-savepage and all the related write system calls have the same scope.

For policy enforcement, the scope is needed in the runtime after the policy is deployed. Without going into the theoretical details, we refer to an "*Oracle*" that, given a system event, returns the scope of the event in that system run. How exactly the oracle computes the scope of each event is described in [3] and is not in the scope of this work.

The oracle has been implemented as part of the PIP. When a PEP notifies the PDP of an event, the PDP requests the corresponding scope from the PIP where the oracle component computes and returns the scope for the event. The oracle computes the same scope for all the cross-layer events of one single data flow. In the example of context-savepage command at the Firefox layer and a series of write system calls at the Linux level, both the Firefox command (intercepted by the Firefox PEP) and the set of Linux system calls (intercepted by the Linux PEP) have the same scope when they create one copy of the webpage.

Essentially, the following should take place technically, when any PEP notifies the PDP of an event: When the PDP receives the notification of a *start-scope* event ($SAVE_s$ in the above example), it flags the start of a new scope. It compares events for this scope and collectively evaluates the ECA rules with *stateful* conditions (e.g., with cardinality constraints) and updates the corresponding conditions. Algorithm 3 gives an abstract view of the evaluation of the cross-layer events at the PDP.

ALGORITHM 3: Evaluating cross-layer events at the PDP

```
input            : Event e
local variables  : Scope s
global variables: Time t

s ← getscope(e);
if (e.type == start-scope) then
    flag_new_scope(s);
    create_list(s);
end
add_to_list(s,e);
forall the r in ECARule do
    if (getcondition(r) == stateful) then
        evaluate_collectively(e,r,s);
    end
    else
        evaluate(e,r);
    end
end
if (e.type == end-scope) then
    wait(t);
    flag_end_scope(s);
    delete_list(s);
end
```

In evaluation functions, $evaluate()$ and $evaluate\_collectively()$, the data state is implicit, unlike shown explicitly in Figure 7.4.

Implementation The OSN scenario of the running example was extended to also include the demonstration of the case of cross-layer events:

> User Alice wants to share some Excel files with her OSN friends. At the same time, she wants to protect certain parts of these files (viz. cells and worksheets) from being used in an unwanted way by her friends. She wants that her friends should be able to view everything but they should not be able to e.g., print, modify, delete the sheets. Again, Alice would like to specify and enforce relevant policies without any technical knowledge. Her policies must be automatically translated, delivered, deployed and enforced at and across different layers of abstraction in her friends' machines.

A usage control policy specification, translation and deployment infrastructure has been set up which helps Alice. It comprises of three layers of abstraction: Ms Excel, Windows and a Java applet using which the Excel files and policies are exchanged across machines (from Alice to Bob). Both the Excel PEP (an AddIn written in C#) and the Windows PEP (an API call interceptor) together enforce the policies.

In the implementation, data and policies are assumed to be generated at the Excel layer: Alice writes a worksheet (or already has one) and wants to specify a policy. She can select a cell (or a set of cells or a worksheet) and launch the Policy Editor described earlier via a context-menu option. As the smallest container at the Excel layer is a cell in the worksheet, Alice can specify policies at the granularity of multiple policies per cell. After policy specification, when Alice saves and deploys the policies, several data-container mappings are created by the PIP as described in Algorithm 2 on page 87. The worksheet is saved in a file to be uploaded at the web server and shared with friends in the OSN.

When Alice uploads the Excel file, technically, the following takes place: There are two sets of cross-layer events: a $LOAD$ event in the applet and a set of $readFile$ API calls at the Windows layer. These are connected together by two atomic events $LOAD_s$ and $LOAD_e$ that mark the beginning and the end of the file upload respectively (similar to the $SAVE$ event and the set of system calls that represent one save page event in Figure 7.5). After the file is completely read from the disk, the applet queries the PIP to get a list of all associated data IDs. Thereafter, the list of data IDs and the file is sent to the web server. This complete sequence is shown in Figure 7.6.

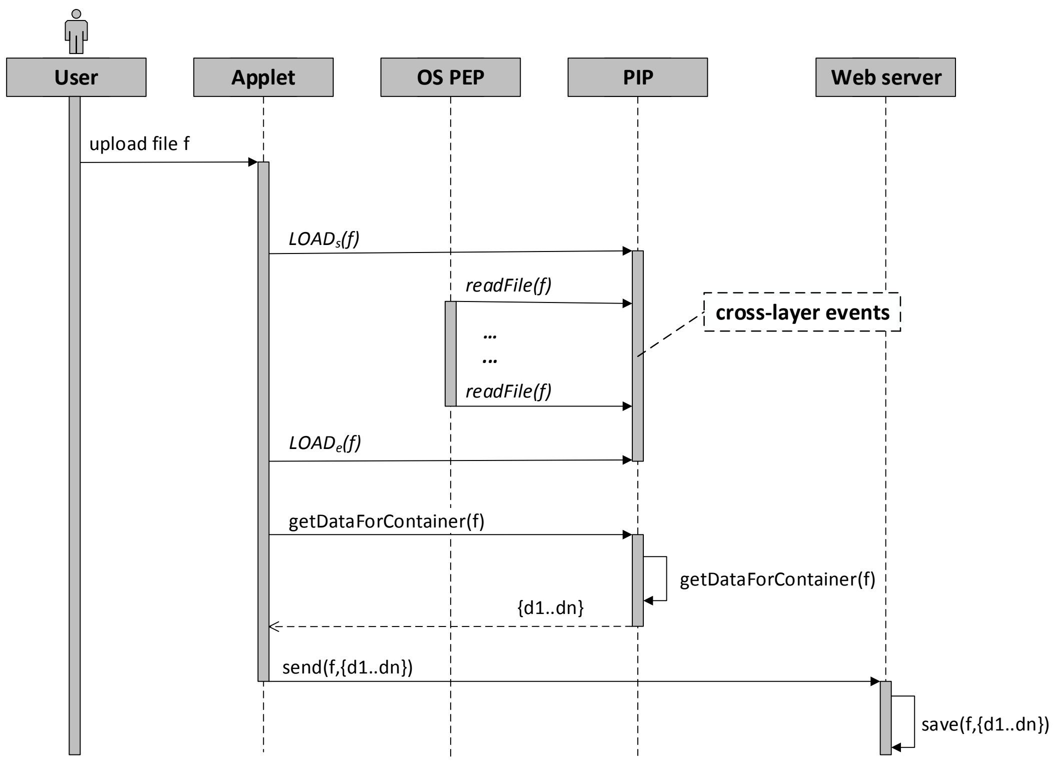


Figure 7.6.: Cross-layer events in case of *file upload*

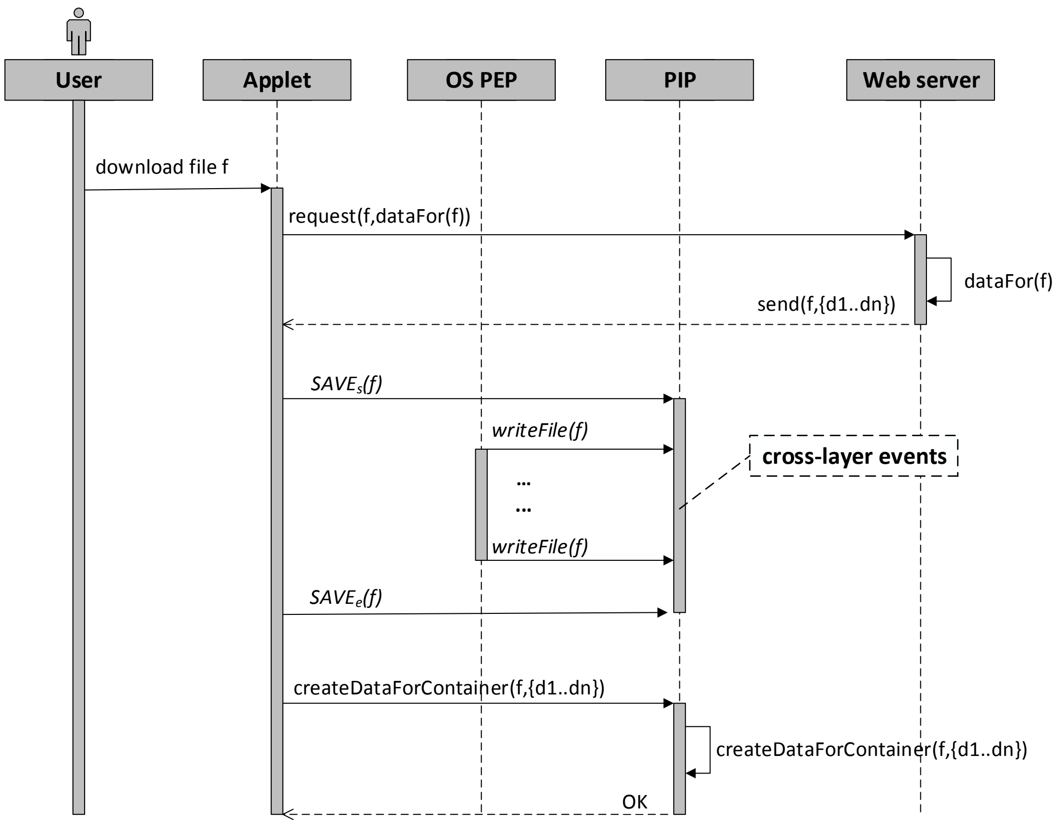


Figure 7.7.: Cross-layer events in case of *file download*

The web application server stores all the files and for each file, it stores a list of data IDs. At Bob's end, when a file download starts, the dual of events that took place while uploading, happen. The applet receives a file and a list of data IDs. The file is written to the file system at the OS layer and in the local PIP, new data-container mappings are created. A new $SAVE$ event in the applet and a set of corresponding $writeFile$ API calls are executed (cross-layer). When the corresponding PEPs (the applet and the Windows PEP) notify the PDP of the cross-layer events, the events could be connected via the scope label at the PDP. The usage control monitoring infrastructure would therefore recognize that *only one copy of the data* is created. Figure 7.7 depicts the complete download process in a sequence diagram. Owing to time reasons, the PDP was not modified for this case.

With the related work on *structured data* [29], where different *parts* of a single container store different data (e.g., one Excel file has several cells and worksheets, each storing a different data item; an email has header, body and attachment fields that store data of different sensitivity etc.), the infrastructure is capable of retrieving the correct mapping of policies for each cell (or set of cells or worksheet) in the Excel file.

7.4. Summary

This chapter described the core components of the usage control infrastructure with interactions among them for policy deployment and enforcement. A solution for enforcing policies on cross-layer events was also discussed with the help of a concrete implementation. The next chapter argues about the relevance of this work in the context of the existing literature.

Part III.

Conclusion

Chapter 8

Related Work

There are two major challenges in the area of Requirements Engineering (RE) that must be met in order to get rid of ambiguities in system implementations. The first challenge is about bridging the gap between requirements elicitation approaches based on contextual inquiry and more formal, specification and analysis techniques [71, 72, 73]. The second challenge is to bridge the gap between user and system requirements by automated, unambiguous derivation of the latter from the former.

The gap between user and system requirements [74, 75] has been a major problem in the domain of Requirements Engineering in general [76, 77] and though a lot of work has been done on understanding, eliciting and analyzing user requirements [78, 79, 74, 80, 81, 82, 83] and translating them into system requirements for different domains [84, 85, 86, 87, 88], it remains one of the prominent topics in software engineering research [89, 90]. One major reason is that often, user requirements are not adequately translated into system requirements which results in systems with functionality that are not desired while missing out on the desired ones [90].

This work targets this fundamental problem of Requirements Engineering in the context of usage control. The focus of this work is the derivation of implementation-level policies from specification-level policies using a domain metamodel that is instantiated to define the technical semantics of domain-specific abstractions in high-level usage control policies.

Though the problem we tackle here can be generalized to filling the gap between user and system requirements for software systems in general, in this discussion we keep ourselves limited to security requirements and policies.

Before we argue for our contribution with respect to related work, we recap the relevant literature in the following section. We start with a rough overview of different techniques to express and analyze security requirements followed by different approaches to refine security requirements from a higher level to the lower levels. We then discuss these in the specific context of access and usage control requirements.

As one major contribution of this work is the definition of generic semantics of high-level actions, we also highlight our contribution in the context of the established understanding of high-level actions like copy and delete.

8.1. Security Requirements Engineering in General

In the context of Security Requirements Engineering, related work has addressed modeling assets, threats and vulnerabilities, failures and countermeasures [91, 81, 92, 93], security requirements [94, 95, 96] and their potential conflict with other functional and non-functional requirements [97], application of standards [98], access control policies [99, 100], policy description languages [50, 101, 21, 102] and policy refinements [103, 41, 37, 35, 36].

To start with, several techniques have been employed to understand, analyze and elicit security requirements. The following paragraphs discuss a few of the techniques that are used to elicit security issues with capabilities to refine them incrementally and therefore, they act as a motivation for the hierarchical refinement approach in this work.

Compared to text-based elicitation and analysis of requirements, graphical expressions of requirements have been found to be more useful because they could be easily understood by all stakeholders, both technically proficient and non-proficient ones.

Two of the most well-known techniques that focus on threats and attack scenarios are *abuse cases* [91] and *misuse cases* [104, 81] which are extensions of the Unified Modeling Language (UML) use cases. Compared to the UML use cases that elicit the desired functionality of the system, abuse cases describe the interactions of users with the system that might cause harm to the system. A misuse case elicits potential threats from mis-actors. Countermeasures against threats are modeled as *security use cases* and a *prevent* relationship is used to relate misuse cases to the countermeasures. The major drawback of these techniques is that there is no hierarchy in the expressions (hence limited ways of refining cases) and the case diagrams very quickly become too large and complicated to cover all details of the relevant scenarios.

A better way to express harmful events/cases in systems is to organize them hierarchically. Such diagrammatic representations of the problem usually have a tree or graph structure where threats/harmful events are nodes connected to each other via AND/OR gates. The root node of the tree is top threat in the system (or subsystem) that materializes upon the success of the lower events that are either composite (intermediate nodes) or atomic events (leaves). Analysis of such structures aims at finding a minimal cut set, which is a collection of events/failures that can lead to the occurrence of the top-event. Attack trees [92], fault trees [93], attack graphs [105] and attack nets [106] are some of the widely used techniques in this respect.

The above techniques focus on the *problem side*. The *solution side* comprises of approaches like UMLsec [95] and SecureUML [94] that address modeling of security requirements. UMLsec is an extension of UML that uses stereotypes, tags and constraints in different type of UML diagrams like state charts and sequence diagrams to express security-relevant information. SecureUML is a modeling language that supports model-driven development of secure systems by combining Role-Based Access Control (RBAC) with UML. Though SecureUML is quite useful in expressing both *declarative* and *programmatic* access control requirements, it has two major limitations: one, it is limited to the generation of access control infrastructures and two, requirements about what a user must not do cannot be expressed in SecureUML notations.

In addition to SecureUML and UMLSec, several policy specification languages and policy frameworks have also been developed to express security requirements and manage them. Most of these frameworks target access control (e.g., KAoS [107] and Cassandra

[108]). Some well-known security policy languages include eXtensible Access Control Markup Language (XACML) [109], Security Assertion Markup Language (SAML) [110], WS-Trust [111], Open Digital Rights Language (ODRL) [50], eXtensible Rights Markup Language (XRML) [101], Enterprise Privacy Authorization Language (EPAL) [112], Open Mobile Alliance - DRM Rights Expression Language (OMA-DRM) [113], Obligation Specification Language (OSL) [21] and Ponder [102] that focus on various flavors of access and usage control.

There are two levels at which policies are specified: in some cases, policies are specified directly at the implementation level as executable rules; in others, policies are specified at high level either using a Graphical User Interface (GUI) or in terms closer to natural languages. In the latter cases, high-level policies are somehow translated from the specification to the implementation level in order to be enforced in technical systems. This is called *Policy Derivation*. Terms like *policy refinement*, *policy translation* and *policy decomposition* have also been synonymously used in related work.

There are several approaches to policy refinement. Based on this, several policy refinement frameworks exist in literature [103, 41, 37, 35, 36].

8.1.1. Different Approaches to Policy Refinement

In the spirit of knowledge refinement [114], all type of policy refinements distinguish between two levels of information: a high-level and a set of low-level information.

Most of the prominent work on policy refinement is based on the concept of *goal* which is a desired objective that should be met in a system or domain. In goal-based policy refinements, a policy is a typical means of achieving the goal [115]. Hence a goal is abstract while policy is concrete.

Similar view is expressed in [116] and [117] where goals are decomposed into successive levels of detail to form a tree of arbitrary depth. These are some of the earliest, prominent work that uses goals for policy refinement. Lower level goals support achievement of higher level policy. Policy is a type of goal. Hence goal is more general than policy.

In a nutshell, a goal is either similar to a policy or at least has the same purpose in the system, that is, to control the behavior of the system in some way [47]. Goal decomposition and refinement is therefore often used to refine policies in a hierarchical fashion. We mention some of the prominent goal-based refinement techniques below as they serve as the motivation for the hierarchical action and policy refinement in our work.

Darimont, Lamsweerde et al. have described a goal-based policy refinement in the context of the KAOS method [35, 36]. The high-level goal is formally refined to subgoals in an AND/OR tree-structure until implementable constraints are reached.

In later work by Lamsweerde, goals are refined to software requirements from which, software specifications and abstract architecture of the system are derived [118, 119].

Another hierarchical policy refinement is described in [120]. Policies are refined by mapping abstract elements to concrete policy elements. A database called the Information and System Model (ISM) is used to store all policy-refinement information which is the set of all objects and their relationships with each other (which can be hierarchical, inheritance or associations).

Bandara et. al in [37] describe a goal-based policy refinement which uses abductive reasoning techniques to derive the sequence of operations that allow a given system to achieve a desired high-level goal. Policies are refined by mapping abstract entities defined as part of a high-level policy to concrete objects/devices that make up the underlying system. High-level goals in policies are refined into operations supported by the concrete objects / devices. Goal refinement is achieved using the refinement patterns defined in [36]. Unlike our work, the policy refinement is semi-automated: towards the end of refinement, subject refinement requires user intervention. Several other goal-based policy refinement approaches are completely automated.

Rubio-Loyola [38] build upon the work by Bandara et al. [37]. Using model checking techniques to obtain system trace executions aimed at fulfilling lower-level goals that logically entail high-level goal that was refined according to [37]. Output policies are in Ponder [102]. The policy derivation is automated.

Apart from goal-based policy refinement for access control, some other well-known approaches to policy refinement are as follows:

Su et al. have proposed a policy refinement methodology based on resource hierarchies [40]. Klei et al. [41], Guerrero et al. [42] and Basile et al. [43] have proposed different ontology-based have proposed different ontology-based approaches to policy translation. Udupi et al. have presented an automated, domain independent approach based on data classification [46]. Lodderstedt et al. specify RBAC policies in UML and translate them to code or deployment files [94] and do not consider the semantics of atomic actions or resources. Aziz et al. [39] have proposed a resource hierarchy metamodel for translating domain-specific elements in XACML policies at the level of virtual organizations to generate corresponding XACML resource-level policies.

In terms of derivation of privacy policies, Yee and Korba [121] have presented two approaches for semi-automated policy derivation. One approach relies on third-party surveys of user perceptions of data privacy and derives rules based on community consensus. The other approach is to make use of existing policies in a peer-to-peer community. Policy content comprises of rules based on privacy principles that have been enacted into legislation.

Young and Anton [45] have proposed a commitment (obligations) analysis methodology to derive software requirements from privacy policies.

In terms of automated policy refinement, [41] have proposed an architecture called SEMPR that uses Web services and automatic Web services composition as a complementary technique to policy refinement in order to automate policy-based management. As a proof of concept they have implemented automated policy refinement for a Home Area Network. The limitation of this work is that it relies entirely on a Web service infrastructure in the form of Web service based management interfaces of the devices.

[122, 123, 124] describe a UML-based high-level, graphical, modeling language called UML-based Web Engineering (UWE) for graphically modeling system's security aspects in a simple and intuitive way. These graphical specifications are transformed to access control policies in XACML in a model-driven way. These XACML policies are then translated to Formal Access Control Policy Language (FACPL) policies which are evaluated by means of a Java-based software tool. The policy transformation is automated.

8.2. Differences with this Work

Now we identify the distinction between the state of the art in policy derivation and our work in this thesis based on several factors.

8.2.1. The Policy Refinement Context

We argue in the context of those policy refinement work that come close to our approach. Only fully-automated policy derivations are considered.

Aziz et al. [39] have proposed a resource hierarchy metamodel 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.

Ontology-based policy derivations that hierarchically refine policy constructs according to their *meanings* are also similar to our approach in spirit. Klei et al. [41], Guerrero et al. [42] and Basile et al. [43] have proposed different semi-automated approaches to the translation of access control policies. *In our work, such ontologies could be used at each level of the domain model.* For example, an ontology with all the different interpretations of secure deletion could be used to model the "delete" action in the domain model.

Another work which is quite similar to ours in terms of approach is described by Craven et al. in [44]. 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.

Two types of policies are refined this way: authorizations and obligations. The term "obligations" is used in a limited sense: obligation policies require an action to be performed within a certain period. *Compared to this, obligations in our work refer to all constraints upon future usages of protected data.*

Rochaeli in his PhD thesis [47] (SicAri project)[1] uses a similar technique like [37, 38] to refine policies into sub-policies. A hierarchical policy refinement tree is constructed where the set of previously refined policies are the nodes of the tree connected with AND/OR gates. The policy refinement tree is similar to a Fault Tree which consists of three types of policies as nodes: specified policies, abstract policies and concrete policies. Inputs of gates are connected to policies and outputs are connected to the relative sub-policies. The formal semantics of policy refinement is given in terms of nodes and connecting gates. The policy refinement is automated.

Differences with our work: *firstly, the policy refinement in [47] is goal based; enforceable rules are generated from high-level security goals. Secondly, the scope is the refinement of workflow policies, there is no attempt to address the generic refinement of user actions like copy, delete and*

[1]http://sicari.sourceforge.net/introduction.html

distribute. Thirdly, for policy representation, XACML is used and thus policies in SicAri are limited to the access-control capabilities provided by XACML [125]. Fourthly, policy refinement relies on refinement templates that are created by collecting best practices. An example best-practice solution is to refine a policy "prevent fictive loans" into "apply separation of duty with XACML policy" because banks usually prevent fictive loans by not allowing the same entity to apply for and approve a loan. Additionally, dynamic domain structure where devices join and leave and policies have to be changed at runtime are not considered by SicAri [125]. Compared to this, *it is possible in our work to capture and reflect the real-world domain structure changes into the policies* (see Appendix D).

All of the above techniques have certain points of differences with our work: Usually, policy refinements result in a set (or sequence) of operations that achieve the desired high-level goal in the system or are in some way, a direct technical representation of the high-level policy constructs. These approaches do not specify the semantics of the high-level terms in policies. Also, except [37], they also do not formally or informally describe the semantics of refinements.

We, on the other hand, tackle a different problem. We want to give a way to define formal semantics of high-level user actions in concrete systems taking into consideration the underlying platform and, use this semantics to refine policies that contain these terms.

Besides this, we target policies for usage control enforcement while most of the policy refinement literature concerns access control and privacy policies only. To the best of our knowledge, the translation of usage control policies has not been investigated except in [49] (see below).

8.2.2. Domain Modeling for Policy Refinement

We refine actions by modeling the concerned domain in a hierarchical way with distinction between the abstract entities at high level and the corresponding technical entities at the lower levels. This approach of capturing details of a system with several levels of abstraction has also been adopted in many architecture frameworks [126, 127, 128] and is also common in the embedded systems domain [129, 130, 131].

Our approach uses the domain modeling/model-based development paradigm. Attacking the refinement of actions and related policies at the domain level is inspired by the early work in the area of Requirements Engineering by Jackson et al. [132, 133, 134, 73]. In this approach, a significant proportion of the RE process is about developing domain descriptions [71]. Some of the prominent work by Jackson et al. explicitly distinguish between the domain and the system. The system in turn consists of the machine and its connection to the domain. Domain analysis is therefore limited to understanding the problem from a high-level perspective. Connecting the domain to the technical systems is a separate step. In comparison to this approach, our definition of domain and its description consists of both the domain and the system. The domain described in these work is represented in our metamodel at the PIM level. The similarity of our approach lies in the fact that we also propose to analyse the problem at a high-level and this level sets the context of our policy refinement. Policies are refined not according to the systems but according to the domain context.

We have adopted a model-based approach which is analogous to the MDA viewpoints [135] 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.

8.2.3. Semantics of Actions

In the context of *semantics of actions*, although various ways of secure deletion at different levels of abstraction in the system are described at length in [48], there is no further discussion on the different interpretations of deletion (weak deletion, quarantine items, irreversible encryption, etc.). In comparison to this paper, our approach is to formally define policy refinement along with a generic way of modeling semantics of actions like "copy" and "delete". We classify the abstract and the concrete elements according to the technical details into different models and perform transformations on the elements of these models.

We are not aware of any other related work that either discusses the meanings of some other high-level actions like *copy* or provides a generic way of modeling the meanings of such actions.

8.2.4. Policy Templates and Roles of Users

Intuitively, policy templates are graphical or textual representations of different types (or classes) of policies. We use two types of policy templates in our work-flow: one type of templates are used to help non-proficient end users specify usage control policies which could be otherwise quite complicated for them to understand and very tedious to specify. The second type of policy templates are used to automate the derivation of policies as ECA rules.

Use of templates and patterns is not new in Software Engineering (SE). Our template-based specification is inspired by the patterns approach in SE [136] especially, in security [137, 138, 139]. In general, the goal is to reuse knowledge and collect best practices in form of patterns. In this work, we have two additional motives for using templates: facilitating policy specification by end users and automation of the refinement process.

Rochaeli [47] also uses patterns, inspired by the work of Christopher Alexander in [140]. But the patterns are used to refine policies. These patterns contain best-practice solutions for problems that have occurred in the past. From these refinement patterns, XACML policies are generated using XML templates. This approach of [47] is similar to the usage of ECA templates described in this thesis, though the purpose of ECA templates is different in our case: they are used to specify enforcement strategies in ILPs and, for syntax transformation from obligation formulas to rules of ECA format.

Our policy specification and derivation requires users in two roles. Sophisticated users called power users configure the policy translation infrastructure and other templates and, the lesser users called *end users* use the templates to specify policies and automatically translate and deploy them.

Similar to our approach of classifying users according to their expertise, [120] have proposed two classes of users. The *expert* is the person with deep domain knowledge in the field of security or network QoS related mechanisms. A *consultant* is the person who has deep knowledge of the business for which policies are to be specified. This approach also uses *policy templates*, created by experts for specification of policies by consultants. Again, the context of this work is access control.

8.2.4.1. Usability of Policy Specification

As early as [134], researchers have recognized the problem with the understanding of requirements. From this point of view, we have found that specifications, both formal and informal, are often hard to understand. Although end users are interested in specifying security requirements, in particular usage control policies [60], they are in general not capable of reasoning holistically about such policies [59].

There are two ways suggested in literature to address this issue. One is to help users directly specify security policies in a simple and user-friendly way [141]. The other approach is to specify policies on behalf of the user (limiting user input to selecting from options via dropdowns or checkboxes) and help users understand already specified policies in terms of how their data is affected by these choices [142].

This was the reason behind introducing templates for policy specification, as described earlier. ***An in-depth look into the usability of policies and policy specification is out of the scope of this work.***

8.2.5. The Usage Control Context

In the area of usage control, several enforcements exist for a variety of cases: for various policy languages [21, 50, 101, 102, 112, 113, 143, 144], at [51, 145, 146, 147, 148, 149, 150, 151, 152] and across [27] different layers of abstraction in various types of systems [7, 52]. All of these work focus on the implementation of event monitors. Policies are supposed to exist (or are directly specified at the implementation level) and their derivation from end user requirements to enforcement mechanisms has not been addressed.

In the context of *policy derivation/refinement for usage control enforcement*, [49] describes an approach where high-level policies are specified for an abstract system model and automatically refined with the help of policy refinement rules to implementation-level policies. A major difference with our work here is the absence of a generic way to refine policy constructs: the refinement rules are specific to the domain in consideration and they do not reflect the technicalities of the underlying platform.

We are not aware of any work that provides an automated translation of specification-level usage control policies into implementation-level policies in a generic, domain- and system-independent way.[2] *Policy enforcement, evolution of policies and policy life-cycle management is out of the scope of this work.*

This chapter established the contribution of this thesis with respect to related work. The next chapter presents a critical analysis of the work, limitations and other left over issues.

[2]The domain metamodel defines a way to model domain-specific semantics, but the metamodel itself is generic and can be used for any domain.

Chapter 9

Conclusion and Future Work

This chapter concludes by a critical discussion on the achievements of this thesis and pointing to interesting directions of future work.

This work describes 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 power user. The policy derivation methodology describes with several examples, how the process can be automated end to end.

Specification-level policies consist of constraints on abstract user actions and data. There are two major challenges in the enforcement of such policies: firstly, because specification-level policies talk of abstract data but in real systems only concrete containers exist, it should be possible to keep track of all representations of the same data at different layers of abstraction. For instance, if a picture downloaded from an OSN is to be protected, then the corresponding window content, the cache file, and the browser-internal representation, as well as their copies, need to be protected. A summary of addressing different representations of the same data in the system without explicitly listing them has been presented in Chapter 3 and forms an important part of the foundation on which this work is built.

The second challenge is to translate the policies specified in abstract terms into implementation-specific policies with concrete technical details at different layers of abstraction. User actions in SLPs must be refined according to their technical semantics in order to derive policies for enforcement.

In Section 4.1, a set of requirements were laid down in order to achieve policy derivation in an automated way. The following section revisits these requirements and describes how each of them were met, addressing the second aforementioned challenge.

9.1. Revisiting the Requirements

At first, three high-level user requirements were identified in order to achieve the three sub-goals of policy derivation. Each of these goals and requirements correspond to one of the three steps in which the policy derivation problem was to be approached:

R1⟩ *Refinement of constructs:* There must be a generic way to refine the basic policy constructs viz. data and action into technical equivalents.

R2⟩ *Policy derivation:* SLPs must be translated to ILPs using the refinement of data and action.

R3⟩ *Automation of policy derivation:* Policy derivation from the specification to the implementation levels must be automated.

Following is a recap of the work presented in this thesis in the context of the refinements of these requirements:

The first requirement was to come up with a language to model data and actions in a generic way. A three-layer domain metamodel described in Chapter 4 fulfills this requirement. Data and action are modeled as UML classes at a high-level in the platform-independent model (PIM) and are refined recursively to their technical counterparts at the platform-specific and implementation-specific levels. Associations among classes represent the data and action refinements (*Requirements R1.1 – R1.3*).

The domain metamodel and the usage control model are combined in Chapter 5 by connecting data and containers (of the usage control model) to their classes (that are instances of elements of the domain metamodel: *Requirement R2.1*). The semantics of refining data and actions is formally defined in the combined model (*Requirement R2.2*) and past-equivalents of future-time OSL formulas are also derived (*Requirement R2.3*). The purpose of past-time translation of OSL formulas is to know the condition in which a future-time formula of an SLP will be violated so that relevant actions as mentioned in the respective policy could be taken.

A detailed methodology from setting up the infrastructure to the translation, deployment and update of policies is described in Chapter 6. Two roles of users are suggested: power users define a domain model instantiating the aforementioned metamodel. Finally transforming SLPs to ECA format is essentially a problem of syntax transformation with the tricky part being the specification of enforcement strategy for policies (*Requirements R3.1 – R3.3*). Classes of SLPs and ILPs, used as patterns and called *policy templates* have been used with two motives:

1. End users can specify policies for their data using the SLP templates which are user-friendly representations of OSL formulas with object parameters that talk of classes of data and containers instead of the instances (*R3.4 – R3.5*).

2. For each SLP template, the power user can specify an enforcement strategy. All policies that are specified using one SLP template are enforced in one particular way.

Different events at various layers of abstraction in a machine that belong to the same higher level event (e.g., the save page event in the above paragraphs) are grouped together by tagging them with a label that is unique for each such group of events (*Requirement R2.5*). Adding context-specific information like user session identifier to policies is case-dependent. Examples demonstrate how context information can be added in a generic way for certain domain implementations (*Requirement R2.4*).

In Chapter 7, an infrastructure design that supports the policy derivation concepts and the methodology is described via component diagrams. The interactions among different components are also outlined. Instantiating this generic architecture, several example scenarios were implemented to demonstrate the feasibility of achieving policy derivation in an automated way.

In the following section, we critically analyze the achievements of this thesis.

9.2. Discussion

Domain Modeling: This thesis discusses an action refinement approach based on domain models. In general, model-based approaches have been advantageous in automating concepts and processes because they allow flexibility of requirements. It is much easier to incorporate changes at one place (i.e., in the model) and to reflect the change everywhere in the environment.

Besides this, building explicit domain models provides two key advantages: they allow detailed reasoning about what is assumed about the domain and they provide opportunities for requirements reuse within a domain [71]. It's also possible to treat the domain as a black box and reason about the environment as we do in our implementation of the policy derivation core.

There are also some downsides to the model-based approach in the context of this work: Domain models are built by power users. As the domain metamodel proposed in this work could not be validated for correctness, this renders the usefulness of the domain models highly reliant on the expert knowledge of power users. The reason is that the meanings of data and actions are not precisely defined but they are intuitively understood both by end users and power users. Because of this, it is hard to establish a notion of correctness between the semantics of low-level and high-level policies based on these semantics.

Because action refinement is central to policy derivation in this work, and there is no inductive way to prove or disprove the completeness and correctness of a given domain model, we may end up with policy derivations that do not reflect the high-level requirements. From a security point of view, we do not consider this a misuse case where a power user might deliberately model the domain in a wrong way because the power users are assumed to be trusted. But if the power user wrongly models the domain, the errors would accumulate and propagate throughout the enforcement. Also, if data is stored at a layer of abstraction which is not included in the domain model or which could not be modeled for any reasons (business secret, lack of understanding of the functionality, closed source with no documentation, etc.), then no implementation-level policies would be generated for monitoring data usage in that layer of abstraction. Hence corresponding usages might be wrongly allowed during the enforcement of policies. Therefore, the domain metamodel should be seen as a *way* to define the semantics of high-level actions by assigning machine-level events and state-based formulas to high-level actions.

Automation: In the general scheme of things, the power used is supposed to configure the policy translation before the derivation components are deployed and running. As the action refinement is model-based and the syntax transformation of OSL formulas to ECA format is template-based, no further human intervention is required after the end user specifies and saves policies. Although the definition of enforcement mechanisms needs human intervention if each SLP within a class of SLPs needs to be enforced in a different way, we did not find this to happen very often. Such cases therefore can be considered as *exceptions* and handled differently via human intervention.

Another issue is in the derivation of past equivalents of OSL formulas. As we restrict ourselves to propositional arguments for temporal operators, complex obligations with nested future-time temporal formulas, whose past translation cannot be automatically achieved, may need to be decomposed. A fully-automatic solution of this problem would be to translate each subformula into the past form and map it to the condition part of a separate ECA rule. But in that case, there would be the need to show that the compositionality property holds.

Generalizability: Though we have discussed the application of action refinement in the context of usage control policy derivation, the domain metamodel used to define the meanings of actions is fundamentally independent of the usage control context. We see it as a way to to define the generic semantics of actions. So the domain metamodel could be used in any area where the meanings of actions are to be explicitly modeled at the domain level. E.g., any security policy model could be combined with the domain metamodel to use the general semantics of actions in policies. In fact the domain metamodel is not related to security per se: it's just a language to model domains and hence it could also be used as a standard to refine both security and non-security, functional and non-functional requirements.

Besides this, the formal semantics of policy derivation using action refinement could also be used in existing access control frameworks to translate policies because (i) usage control is an extension of access control with shared concepts between the two and (ii) it is possible to translate OSL to access control policy languages: this possibility has already been explored in the concrete case of translation from OSL to ODRLC and vice versa in [21]. As long as the generated policies can be translated to OSL, they can be transformed into ECA rules and successfully deployed and enforced in the usage control infrastructure described in Chapter 7. Because the policy management components, the Policy Editor(PE), the Policy Translation Point (PTP) and the Policy Management Point (PMP) are implemented to generate policies in a specific syntax, they cannot be used as-is to deploy policies that do not conform to the the concrete syntax given in the Appendix B. But this is not a fundamental limitation.

Usability Concerns: 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.

In the implementation, the policy templates have been *deliberately* kept simple with user inputs limited to multiple-choice format. This should not be seen as a limitation though, as the policy derivation is fundamentally independent of the concrete syntax of the templates.

In principle, policy templates could also be written in XML, in plain structured text, in simple graphical format with dropdowns and checkboxes or, in advanced graphical format where end users could drag and connect different blocks to write policies (on lines of Scratch, MIT Open Blocks). Policies could also be specified using free text in any natural language or standardized solutions like ODRLC (as mentioned above, a translation from OSL to ODRLC and vice versa is already provided in [21]).

Of course, technically sophisticated users can still write policies directly at the implementation level or they can write policies that address specific containers and concrete events on them. This work does not exclude such cases.

Ubiquity of infrastructure: Finally, a general problem with usage control enforcement is the requirement of a ubiquitous infrastructure in order to ensure that the translated policies are indeed enforced at and across all layers of abstraction in different machines in a distributed setup. Also, a basic assumption in this work is that the policy enforcement components are implemented correctly, that they are up and running and that they are not tampered with (at least not without the tamper being recognized). In a distributed setup with remote data consumers and closed sub-environments, it might be hard to verify if the assumptions indeed hold.

9.3. Future Work

The results in this thesis could be extended in multiple dimensions. The following paragraphs point to some of the areas and provide an intuition of the probable solutions.

Using Machine Learning Techniques: As mentioned earlier, the specification of enforcement strategies per policy is a stumbling block in the automation of policy derivation in this work. In Chapter 6, this has been handled by the suggestion of templates. The feasibility of using templates of ECA rules for this purpose has also been demonstrated by several examples.

A possibility to improve this solution could be the application of machine learning techniques where end user preferences for enforcing their specification-level policies for specific data and actions could be used as part of the input pattern to (i) enforce each policy, instead of class of policies, in a different way, and (ii) to include end users in specification of policy enforcement also. For instance, case-based reasoning (CBR) [153] has been shown to help end users specify privacy policies in pervasive computing [154, 155]. In the context of this work, it could be used to train a system to learn users' preferences for enforcement of specific policies. Based on the results, some of the human intervention from the power user in setting up the infrastructure could be eliminated for future cases.

Dynamic domains: There is a deliberately introduced limitation in this work: the dynamic nature of domains has not been considered in detail. Appendix D provides an intuition of handling the case where the domain structure is not static as defined initially in the domain model, but it changes with new types of and implementations being added, removed or updated from time to time. Along with the fundamental concepts and a basic algorithm to compare and merge any two classes from two domain models (one that exists in the infrastructure, represents an earlier version of the domain structure and is used for policy derivation, and the other model that contains the modifications to the domain), a proof of concept has also been implemented, tested and validated. In a nutshell, the infrastructure exists to support the dynamic domain case. However, the conceptual solution needs further attention for coming up with better ways of bringing together elements from different domain models.

Apart from this, one limitation of the work in Appendix D is the reliance on existing ontologies to establish relationships among terms at the abstract level. This limitation can be got rid of by defining detailed ontology of the specific domain where usage control has to be enforced. For example, an ontology with terms from computing industry, or legal departments or educational institutions can be employed to more easily establish relationships between two domain vocabulary. Therefore a next step in improving the solution would be to come up with such ontologies that provide semantic support to fill gaps in an incomplete definition of a domain.

Policy Evolution: Another limitation on the solution is posed by the evolution of specification-level policies in a distributed setup where consumers of data transform themselves into providers of data over time (as explained in Chapter 1), and so the policies that they attach with data might evolve with this change [31]. In the present solution, this aspect of the problem has not been considered. As Chapter 2 has explained, in the current scheme of things, policies may only evolve by strengthening. How the semantics of the actions in policies would change in that case could be worth exploring in future.

Policy Conflicts: This work also describes a policy editor for specifying SLPs where no mechanism are implemented for checking specification-time conflicts. Conflicts in usage control policies can arise from various sources. E.g., one policy might define a duty that includes performing certain actions that are forbidden by another policy, as in "no non-anonymised data should leave the system" vs. "report complete logs (including user profile) whenever this data is accessed". Conflicts can also arise when policies put constraints upon an action that targets an object which is a collection of other objects or forms part of another object in the domain model. For example, an album consists of several photos where each photo might be associated with a different policy ("copy max 5 times", "copy upon payment", "copy but notify", "copy with lower quality", "never copy" etc.). In this case, a policy for the album that constraints copy action might be in conflict with one or more of the policies for individual photos. Apart from object hierarchy, policy conflicts can also arise from subject hierarchy: for example, rights and duties of subjects at one level can restrict rights and duties of subjects at other levels; and because of several other reasons.

Intuitively, detecting these conflicts at the time of policy specification in isolated cases is not very difficult. However, in a large enterprise, it might be challenging for an administrator to manually compare a new policy with all the existing ones and detect and resolve any conflicts within realistic time limits. Besides this, deployment in a distributed setting is difficult because conflicts need to be identified (which is simple) and resolved at the deployment time (which might be very difficult). For this reason, a tool support for the process is required.

An interesting part of future work in this direction would be to see in the context of usage control, (i) how to form patterns of conflicts; (ii) if for all such cases, a hierarchical order could be assigned to the policies (or to classes of policies) to create conflict taxonomy and (iii) if these patterns and hierarchies (and other conflict resolution strategies) could be used to successfully resolve policy conflicts automatically.

Application to Other Areas: This work provides an application of the domain metamodel in the usage control context. For interoperability among domains with different other security models, the metamodel could be connected to other well-known usage and access control models for respective policy derivations.

Usability of Policies: This work does recognize the issue of usability vs. security in general and particularly in the case of specification of security policies (see Chapter 8). Usability concerns led us to introduce templates for policy specification, as described earlier. However, an in-depth look into the usability of templates and policies and a related discussion on the usability of policy specification has been out of the scope of this work. This is a vast area of research and there are innumerable possibilities to explore how usage control could be made more usable in terms of the representation of policies.

Bibliography

[1] Erol Demirbas. Monitoring compliance of third-party applications in online social networks. Bachelor thesis, TU München Germany, October 2014.

[2] Alexander Pretschner, Florian Kelbert, Prachi Kumari, Enrico Lovat, and Tobias Wüchner, editors. *Distributed Data Usage Control*. Yet to be published.

[3] Enrico Lovat. *Cross-layer Data-centric Usage Control*. Dissertation in informatik, Technische Universität München, München, 2015.

[4] Exponential bandwidth growth and cost declines. `http://www.networkworld.com/news/tech/2012/041012-ethernet-alliance-258118.html`. Accessed: 2014-04-17.

[5] K.G. Coffman and Andrew Odlyzko. Internet growth: Is there a 'moore's law' for data traffic? *Handbook of massive data sets*, pages 47–93, 2002.

[6] Kryder's law. `http://www.scientificamerican.com/article/kryders-law`. Accessed: 2014-04-17.

[7] 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.

[8] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[9] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Revised Versions of Lectures Given During the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures*, FOSAD '00, pages 137–196, London, UK, 2001. Springer-Verlag.

[10] Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, November 1993.

[11] Xin Jin, Ram Krishnan, and Ravi Sandhu. A Unified Attribute-based Access Control Model Covering DAC, MAC and RBAC. In *Proceedings of the 26th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy*, DBSec'12, pages 41–55, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] Tobias Wüchner and Alexander Pretschner. Data loss prevention based on data-driven usage control. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*, pages 151–160, 2012.

[13] 2013 an epic year for data breaches with over 800 million records lost. `http://nakedsecurity.sophos.com/2014/02/19/2013-an-epic-year-for-data-breaches-with-over-800-million-records-lost/`. Accessed: 2014-04-17.

[14] Jaehong Park and Ravi Sandhu. Towards usage control models: Beyond traditional access control. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, SACMAT '02, pages 57–64, New York, NY, USA, 2002. ACM.

[15] Jaehong Park and Ravi Sandhu. The UCONABC Usage Control Model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, February 2004.

[16] Alexander Pretschner, Manuel Hilty, and David A. Basin. Distributed Usage Control. *Communications of the ACM*, 49(9), 2006.

[17] Christoph Bier. How usage control and provenance tracking get together - a data protection perspective. In *Proceedings of the 2013 IEEE Security and Privacy Workshops*, SPW '13, pages 13–17, Washington, DC, USA, 2013. IEEE Computer Society.

[18] How Dropbox knows you're a dirty pirate, and why you shouldn't use cloud storage to share copyrighted files. `http://www.extremetech.com/computing/179495-how-dropbox-knows-youre-a-dirty-pirate-and-why-you-shouldnt-use-cloud-storage-to-share-copyrighted-files`. Accessed: 2014-04-17.

[19] Dropbox says it isn't poking around in our stuff. `http://nakedsecurity.sophos.com/2014/04/01/dropbox-says-it-isnt-poking-around-in-our-stuff`. Accessed: 2014-04-17.

[20] RSA survey: Target breach has larger impact than Snowden leakes on cybersecurity budgets, executive awareness. `http://www.tripwire.com/company/news/press-release/rsa-survey-target-breach-has-larger-impact-than-snowden-leaks-on-cybersecurity-budgets-executive-awareness`. Accessed: 2014-04-17.

[21] Manuel Hilty, Alexander Pretschner, David A. Basin, Christian Schaefer, and Thomas Walter. A policy language for distributed usage control. In *Proceedings of 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007*, pages 531–546. Springer-Verlag, 2007.

[22] Alexander Pretschner. An overview of distributed usage control. In *2nd Conference on Knowledge Engineering: Principles and Techniques*, pages 17–25, 2009.

[23] Alexander Pretschner, Manuel Hilty, David A. Basin, Christian Schaefer, and Thomas Walter. Mechanisms for usage control. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security, ASIACCS, Tokyo, Japan, March 18-20, 2008*, pages 240–244. ACM, 2008.

[24] Dean Povey. Optimistic security: A new access control paradigm. In *Proceedings of the Workshop on New Security Paradigms*, NSPW '99, pages 40–45, New York, NY, USA, 2000. ACM.

[25] Alexander Pretschner, Manuel Hilty, Florian Schütz, Christian Schaefer, and Thomas Walter. Usage control enforcement: Present and future. *IEEE Security & Privacy*, 6(4):44–53, 2008.

[26] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[27] Alexander Pretschner, Enrico Lovat, and Matthias Büchler. Representation-independent data usage control. In *Proceedings of 6th International Workshop on Data Privacy Management*, volume 7122 of *Lecture Notes in Computer Science*. Springer, 2011.

[28] Enrico Lovat and Alexander Pretschner. Data-centric multi-layer usage control enforcement: A social network example. In *Proceedings of ACM Symposium on Access Control Models and Technologies*. ACM, 2011.

[29] Enrico Lovat and Florian Kelbert. Structure matters - A new approach for data flow tracking. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA, May 17-18, 2014*, pages 39–43, 2014.

[30] Florian Kelbert and Alexander Pretschner. Data Usage Control Enforcement in Distributed Systems. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 71–82, New York, NY, USA, 2013. ACM.

[31] Alexander Pretschner, Florian Schütz, Christian Schaefer, and Thomas Walter. Policy evolution in distributed usage control. *Electronic Notes in Theoretical Computer Science*, 244:109–123, 2009.

[32] Halvard Skogsrud, Boualem Benatallah, and Fabio Casati. Trust-serv: Model-driven lifecycle management of trust negotiation policies for web services. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 53–62, New York, NY, USA, 2004. ACM.

[33] Naranker Dulay, Emil Lupu, Morris Sloman, and Nicodemos Damianou. A policy deployment model for the ponder language. In *2001 IEEE/IFIP International Symposium on Integrated Network Management, IM, Seattle, USA, May 14-18, 2001. Proceedings*, pages 529–543, 2001.

[34] Nicodemos Damianou, Naranker Dulay, Emil Lupu, Morris Sloman, and Toshio Tonouchi. Tools for domain-based policy management of distributed systems. In *Management Solutions for the New Communications World, 8th IEEE/IFIP Network Operations and Management Symposium, NOMS, Florence, Italy, April 15-19, 2002. Proceedings*, pages 203–217, 2002.

[35] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[36] Robert Darimont and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '96, pages 179–190, New York, NY, USA, 1996. ACM.

[37] Arosha K. Bandara, Emil Lupu, Jonathan D. Moffett, and Alessandra Russo. A goal-based approach to policy refinement. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), 7-9 June 2004, Yorktown Heights, NY, USA*, pages 229–239, 2004.

[38] Javier Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, G. Pavlou, and A.L. Lafuente. Using linear temporal model checking for goal-oriented policy refinement frameworks. In *Policies for Distributed Systems and Networks. Sixth IEEE International Workshop on*, pages 181–190, June 2005.

[39] Benjamin Aziz, Alvaro E. Arenas, and Michael D. Wilson. Model-based refinement of security policies in collaborative virtual organisations. In *Engineering Secure Software and Systems - Third International Symposium, ESSoS, Madrid, Spain, February 9-10, 2011. Proceedings*, pages 1–14. Springer Verlag, 2011.

[40] Linying Su, David W. Chadwick, Andrew Basden, and James A. Cunningham. Automated decomposition of access control policies. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), 6-8 June 2005, Stockholm, Sweden*, pages 3–13, 2005.

[41] Torsten Klie, Benjamin Ernst, and Lars Wolf. Automatic policy refinement using OWL-S and semantic infrastructure information, 2007.

[42] Antonio Guerrero, Víctor A. Villagrá, Jorge E. López de Vergara, Alfonso Sánchez-Macián, and Julio Berrocal. Ontology-based policy refinement using SWRL rules for management information definitions in OWL. In *Large Scale Management of Distributed Systems, 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM, Dublin, Ireland, October 23-25, 2006, Proceedings*, pages 227–232, 2006.

[43] Cataldo Basile, Antonio Lioy, Salvatore Scozzi, and Marco Vallini. Ontology-based policy translation. In Álvaro Herrero, Paolo Gastaldo, Rodolfo Zunino, and Emilio Corchado, editors, *Computational Intelligence in Security for Information Systems*, volume 63 of *Advances in Intelligent and Soft Computing*, pages 117–126. Springer Berlin Heidelberg, 2009.

[44] Robert Craven, Jorge Lobo, Emil C. Lupu, Alessandra Russo, and Morris Sloman. Decomposition techniques for policy refinement. In *Proceedings of the 6th International Conference on Network and Service Management, CNSM, Niagara Falls, Canada, October 25-29, 2010*, pages 72–79, 2010.

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

[46] Yathiraj B. Udupi, Akhil Sahai, and Sharad Singhal. A classification-based approach to policy refinement. In *Integrated Network Management, IM. 10th IFIP/IEEE International Symposium on Integrated Network Management, Munich, Germany, 21-25 May 2007*, pages 785–788, 2007.

[47] Taufiq Rochaeli. *An Automated Policy Refinement Process Supported by Expert Knowledge*. PhD thesis, TU Darmstadt, April 2009.

[48] Joel Reardon, David A. Basin, and Srdjan Capkun. Sok: Secure data deletion. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 301–315, Washington, DC, USA, 2013. IEEE Computer Society.

[49] R. Neisse and J. Doerr. Model-based specification and refinement of usage control policies. In *Eleventh Annual International Conference on Privacy, Security and Trust, PST, Tarragona, Catalonia, Spain, July 10-12, 2013*, pages 169–176. IEEE, 2013.

[50] Renato Iannella. Open Digital Rights Language (ODRL) Version 1.1. `http://www.w3.org/TR/odrl/`, 2008. [Online; accessed 26-August-2015].

[51] Prachi Kumari, Alexander Pretschner, Jonas Peschla, and Jens Kuhn. Distributed data usage control for web applications: a social network implementation. In *Proceedings of 1st ACM Conference on data and application security and privacy, CODASPY*, pages 85–96. ACM, 2011.

[52] Denis Feth and Alexander Pretschner. Flexible data-driven security for android. In *Sixth International Conference on Software Security and Reliability, SERE, Gaithersburg, Maryland, USA, 20-22 June 2012*, pages 41–50, 2012.

[53] Manuel Hilty, Alexander Pretschner, Christian Schaefer, and Thomas Walter. Enforcement for usage control — a system model and an obligation language for distributed usage control. Technical Report I-ST-020, DOCOMO Euro-Labs, December 2006.

[54] Prachi Kumari and Alexander Pretschner. Deriving implementation-level policies for usage control enforcement. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy, CODASPY*, pages 83–94. ACM, 2012.

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

[56] Manuel Hilty, David A. Basin, and Alexander Pretschner. On obligations. In *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*, pages 98–117, 2005.

[57] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 163–173, New York, NY, USA, 1980. ACM.

[58] Dov M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*, pages 409–448, London, UK, UK, 1987. Springer-Verlag.

[59] Lujun Fang and Kristen LeFevre. Privacy wizards for social networking sites. In *Proceedings of WWW '10*, pages 351–360, New York, NY, USA, 2010. ACM.

[60] Manuel Rudolph, Reinhard Schwarz, Christian Jung, Andreas Mauthe, and Noor ul Hassan Shirazi. Deliverable 3.2 - policy specification methodology. Technical report, SEcure Cloud computing for CRitical infrastructure IT, June 2014.

[61] Prachi Kumari. Requirements Analysis for Privacy in Social Networks. In *8th International Workshop for Technical, Economic and Legal Aspects of Business Models for Virtual Goods, Namur*, pages 28–40, 2010.

[62] Prachi Kumari and Alexander Pretschner. Model-based usage control policy derivation. In *Engineering Secure Software and Systems - 5th International Symposium, ESSoS, Paris, France, February 27 - March 1, 2013. Proceedings*, pages 58–74, 2013.

[63] Alexander Pretschner, Ricardo Neisse, Oliver Maschino, and Prachi Kumari. Security enforcement language. Technical Report D5.1.1, MASTER – Managing Assurance, Security and Trust for Services, January 2010.

[64] Prachi Kumari and Alexander Pretschner. Automated translation of end user policies for usage control enforcement. In *Data and Applications Security and Privacy XXIX - 29th Annual IFIP WG 11.3 Working Conference, DBSec 2015, Fairfax, VA, USA, July 13-15, 2015, Proceedings*, pages 250–258, 2015.

[65] Florian Kelbert and Alexander Pretschner. Towards a policy enforcement infrastructure for distributed usage control. In *17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, Newark, NJ, USA - June 20 - 22, 2012*, pages 119–122. ACM, 2012.

[66] Google Plus. `https://plus.google.com`, 2015. [Online; accessed 01-March-2015].

[67] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[68] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[69] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT)*, 6, Aug 2004.

[70] Florian Kelbert and Alexander Pretschner. Decentralized Distributed Data Usage Control. In Dimitris Gritzalis, Aggelos Kiayias, and Ioannis Askoxylakis, editors, *Cryptology and Network Security*, volume 8813 of *Lecture Notes in Computer Science*, pages 353–369. Springer International Publishing, 2014.

[71] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 35–46, New York, NY, USA, 2000. ACM.

[72] Michael Jackson and Pamela Zave. Deriving specifications from requirements: An example. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 15–24, New York, NY, USA, 1995. ACM.

[73] Michael Jackson. The meaning of requirements. *Annals of Software Engineering*, 3:5–21, January 1997.

[74] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering, 2nd Ed.* Springer, 2005.

[75] Itzhak Shemer. Systems analysis: A systemic analysis of a conceptual model. *Commun. ACM*, 30(6):506–512, 1987.

[76] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. Worldwide series in computer science. 1998.

[77] Ian K. Bray. *An Introduction to Requirements Engineering*. Addison Wesley, aug 2002.

[78] Gerald Kotonya, Gerald Kotonya, Ian Sommerville, and Ian Sommerville. Requirements engineering with viewpoints. *Software Engineering Journal*, 11:5–18, 1996.

[79] Gordon B. Davis. Strategies for information requirements determination. *IBM Systems Journal*, 21(1):4–30, 1982.

[80] James E. Rumbaugh. Getting started: Using use cases to capture requirements. *JOOP*, pages 8–12, 23, 1994.

[81] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements by misuse cases. In *TOOLS Pacific 2000: 37th International Conference on Technology of Object-Oriented Languages and Systems, Sydney, Australia*, pages 120–131, 2000.

[82] Pankaj Jalote. *An integrated approach to software engineering (2. ed.)*. Undergraduate texts in computer science. Springer, 1997.

[83] Sari Kujala. Bridging the gap between user needs and user requirements. In *Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE'02)*, pages 45–50, 2001.

[84] Anisha Vemulapalli and Nary Subramanian. Transforming functional requirements from UML into BPEL to efficiently develop soa-based systems. In *On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Confederated International Workshops and Posters, ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009, Vilamoura, Portugal, November 1-6, 2009. Proceedings*, pages 337–349, 2009.

[85] Derek Babb. User requirements for security in wireless mobile systems. *Information Security Technical Report*, 9:51–59, December 2004.

[86] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications-extended abstract. In *Proceedings of the 4th International Conference on Requirements Engineering, ICRE, Schaumburg, Illinois, USA, June 19-23, 2000*, page 189, 2000.

[87] Jonathan D. Moffett and Bashar A. Nuseibeh. A framework for security requirements engineering, 2003.

[88] Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Software Eng.*, 6(1):2–13, 1980.

[89] Betty H. C. Cheng and Joanne M. Atlee. Research directions in requirements engineering. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE, May 23-25, 2007, Minneapolis, MN, USA*, pages 285–303, 2007.

[90] Joy Beatty and James Hulgan. Experiences with a requirements object model. In Martin Glinz and Patrick Heymans, editors, *Requirements Engineering: Foundation for Software Quality*, volume 5512 of *Lecture Notes in Computer Science*, pages 104–117. Springer Berlin Heidelberg, 2009.

[91] John P. McDermott and Chris Fox. Using abuse case models for security requirements analysis. In *15th Annual Computer Security Applications Conference (ACSAC), 6-10 December 1999, Scottsdale, AZ, USA*, pages 55–64, 1999.

[92] Bruce Schneier. Attack trees, 1999.

[93] W.E. Vesely, U.S. Nuclear Regulatory Commission. Division of Systems, and Reliability Research. *Fault Tree Handbook*. Systems and Reliability Research, Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Commission, 1981.

[94] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, pages 426–441, 2002.

[95] Jan Jürjens. Umlsec: Extending uml for secure systems development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 412–425, London, UK, UK, 2002. Springer-Verlag.

[96] Haralambos Mouratidis, Paolo Giorgini, Gordon Manson, and Ian Philp. A natural extension of tropos methodology for modelling security. In *in the proceedings of the agent oriented methodologies workshop (OOPSLA 2002), SEATTLE-USA*, 2002.

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

[98] John Wilander and Jens Gustavsson. Security requirements - a field study of current practice. In *E-Proceedings of Symposium on Requirements Engineering for Information Security*, 2005.

[99] Qingfeng He and Annie I. Antón. Requirements-based access control analysis and policy specification (recaps). *Information & Software Technology*, 51(6):993–1009, 2009.

[100] Gustaf Neumann and Mark Strembeck. A scenario-driven role engineering process for functional rbac roles. In *Proceedings of 7th ACM symposium on Access control models and technologies, SACMAT*, pages 33–42. ACM, 2002.

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

[102] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. In *Proceedings of Workshop on Policies for Distributed Systems and Networks*, pages 18–39, 1995.

[103] Marco Casassa Mont, Adrian Baldwin, and Cheh Goh. POWER prototype: towards integrated policy-based management. In *The Networked Planet: Management Beyond 2000, 7th IEEE/IFIP Network Operations and Management Symposium, NOMS 2000, Honolulu, HI, USA, April 10-14, 2000. Proceedings*, pages 789–802, 2000.

[104] Guttorm Sindre and Andreas L. Opdahl. Templates for misuse case description. In *Proceedings of the 7th International Workshop on Requirements Engineering, Foundation for Software Quality (REFSQ'2001), Switzerland*, pages 4–5, 2001.

[105] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, pages 71–79, New York, NY, USA, 1998. ACM.

[106] James P. McDermott. Attack net penetration testing. In *Proceedings of the 2000 Workshop on New Security Paradigms*, NSPW '00, pages 15–21, New York, NY, USA, 2000. ACM.

[107] Andrzej Uszok and Jeff Bradshaw. Kaos policies for web services. W3C Workshop on Constraints and Capabilities for Web Services, 2004.

[108] Cassandra: Distributed access control policies with tunable expressiveness. In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '04, pages 159–168, Washington, DC, USA, 2004. IEEE Computer Society.

[109] XACML 1.0 Specification Set Approved as an OASIS Standard. `http://xml.coverpages.org/ni2003-02-11-a.html`, February 2003. [Online; accessed 26-August-2015].

[110] Assertions andProtocols for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`, March 2005. [Online; accessed 26-August-2015].

[111] Web Services Trust Language (WS-Trust). `http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf`, February 2005. [Accessed 26-August-2015].

[112] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. Enterprise Privacy Authorization Language (EPAL 1.2). IBM Technical Report, `http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification/`, 2003. Online; accessed 26-August-2015.

[113] Open Mobile Alliance. DRM Rights Expression Language V2.1. `http://www.openmobilealliance.org/Technical/release_program/drm_v2_1.aspx`, 2008. [Online; accessed 26-August-2015].

[114] Eric K. Jones. Knowledge refinement using a high level, non-technical vocabulary. In *Proceedings of the Eighth International Workshop (ML91), Northwestern University, Evanston, Illinois, USA*, pages 18–22, 1991.

[115] Martin S. Feather. Requirements reconnoitring at the juncture of domain and instance. In *Proceedings of IEEE International Symposium on Requirements Engineering, RE 1993, San Diego, California, USA, January 4-6, 1993*, pages 73–76, 1993.

[116] Jonathan D. Moffett and Morris Sloman. Policy hierarchies for distributed systems management. *Selected Areas in Communications, IEEE Journal on*, 11(9):1404 –1414, December 1993.

[117] Alistair G. Sutcliffe and Neil A. M. Maiden. Bridging the requirements gap: Policies, goals and domains. In *Proceedings of the 7th International Workshop on Software Specification and Design*, IWSSD '93, pages 52–55, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[118] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 249–, Washington, DC, USA, 2001. IEEE Computer Society.

[119] Axel van Lamsweerde. From system goals to software architecture. In Marco Bernardo and Paola Inverardi, editors, *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 25–43. Springer Berlin Heidelberg, 2003.

[120] Marco Casassa Mont, Adrian Baldwin, and Cheh Goh. POWER prototype: towards integrated policy-based management. In *The Networked Planet: Management Beyond 2000, 7th IEEE/IFIP Network Operations and Management Symposium, NOMS, Honolulu, HI, USA, April 10-14, 2000. Proceedings*, pages 789–802, 2000.

[121] George Yee and Larry Korba. Semiautomatic derivation and use of personal privacy policies in e-business. *International Journal of E-Business Research*, 1(1):54–69, 2005.

[122] Marianne Busch, Alexander Knapp, and Nora Koch. Modeling secure navigation in web information systems. In *Perspectives in Business Informatics Research - 10th International Conference, BIR 2011, Riga, Latvia, October 6-8, 2011. Proceedings*, pages 239–253, 2011.

[123] Marianne Busch and Nora Koch. Magicuwe - A CASE tool plugin for modeling web applications. In *Web Engineering, 9th International Conference, ICWE 2009, San Sebastián, Spain, June 24-26, 2009, Proceedings*, pages 505–508, 2009.

[124] Marianne Busch, Nora Koch, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Towards model-driven development of access control policies for web applications. In *Proceedings of the Workshop on Model-Driven Security*, MDsec '12, pages 4:1–4:6, New York, NY, USA, 2012. ACM.

[125] Julian Hendrik Schütte. *Security Policies in Pervasive Systems*. Dissertation in informatik, Technische Universität München, München, 2013.

[126] Carol O'Rourke, Neal Fishman, and Warren Selkow. *Enterprise architecture using the Zachman Framework*. Course Technology, 2003.

[127] John A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 26(3):276–292, 1987.

[128] The Open Group. TOGAF Version 9. The Open Group Architecture Framework. 2009. [Online; accessed 26-August-2015].

[129] Alexander Gruler, Alexander Harhurin, and Judith Hartmann. Modeling the functionality of multi-functional software systems. In *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS), 26-29 March 2007, Tucson, Arizona, USA*, pages 349–358, 2007.

[130] Dirk Ziegenbein, Peter Braun, Ulrich Freund, Andreas Bauer, Jan Romberg, and Bernhard Schätz. Automode - model-based development of automotive software. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE), 7-11 March 2005, Munich, Germany*, pages 171–177, 2005.

[131] Birgit Penzenstadler. Tackling Automotive Challenges with an Integrated RE & Design Artifact Model. In *Proceedings of OTM Workshops*, pages 426–431. Springer-Verlag, 2008.

[132] Guillermo Francisco Arango. Domain analysis: From art form to engineering discipline. *SIGSOFT Softw. Eng. Notes*, 14(3):152–159, April 1989.

[133] Michael Jackson, editor. *System Development*. Prentice-Hall International, 1983.

[134] Michael Jackson and Pamela Zave. Domain descriptions. In *International Symposium on Requirements Engineering*, pages 56–64, 1993.

[135] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.0.1. Technical Report omg/03-06-01, Object Management Group (OMG), June 2003.

[136] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[137] Markus Schumacher. *Security Engineering with Patterns: Origins, Theoretical Models, and New Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[138] Andrew Moore, Robert Ellison, and Richard Linger. Attack modeling for information security and survivability. Technical Report CMU/SEI-2001-TN-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2001.

[139] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Pearson Higher Education, 2004.

[140] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[141] Robert W. Reeder, Clare-Marie Karat, John Karat, and Carolyn Brodie. Usability challenges in security and privacy policy-authoring interfaces. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-computer Interaction - Volume Part II*, INTERACT'07, pages 141–155, Berlin, Heidelberg, 2007. Springer-Verlag.

[142] R. Iannella and A. Finden. Privacy awareness: Icons and expression for social networks. In *Proceedings of 8th International Workshop for Technical, Economic and Legal Aspects of Business Models for Virtual Goods Incorporating the 6th International ODRL Workshop, Namur, Belgium*, 2010.

[143] Xinwen Zhang, Jaehong Park, Francesco Parisi-Presicce, and Ravi S. Sandhu. A logical specification for usage control. In *SACMAT, 9th ACM Symposium on Access Control Models and Technologies, Yorktown Heights, New York, USA, June 2-4, 2004, Proceedings*, pages 1–10. ACM, 2004.

[144] W3C. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification. `http://www.w3.org/TR/2005/WD-P3P11-20050104/`, 2005. [Accessed 26-August-2015].

[145] Alexander Pretschner, Matthias Büchler, Matus Harvan, Christian Schaefer, and Thomas Walter. Usage control enforcement with data flow tracking for x11. In *Proceedings of 5th Intl. Workshop on Security and Trust Management*, pages 124–137, 2009.

[146] Matus Harvan and Alexander Pretschner. State-based Usage Control Enforcement with Data Flow Tracking using System Call Interposition. In *Proceedings of 3rd International Conference on Network and System Security*, pages 373–380, 2009.

[147] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreaded java. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 546–569, 2009.

[148] Iulia Ion, Boris Dragovic, and Bruno Crispo. Extending the java virtual machine to enforce fine-grained security policies in mobile devices. In *23rd Annual Computer Security Applications Conference (ACSAC), December 10-14, 2007, Miami Beach, Florida, USA*, pages 233–242, 2007.

[149] Lieven Desmet, Wouter Joosen, Fabio Massacci, Katsiaryna Naliuka, Pieter Philippaerts, Frank Piessens, and Dries Vanoverberghe. The S3MS.NET run time monitor: Tool demonstration. *Electronic Notes in Theoretical Computer Science*, 253(5):153–159, December 2009.

[150] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In *Proceedings of the Workshop on New Security Paradigms, Caledon Hills, ON, Canada, September 22-24, 1999*, pages 87–95, 1999.

[151] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy (S&P), 17-20 May 2009, Oakland, California, USA*, pages 79–93, 2009.

[152] Gabriela Gheorghe, Stephan Neuhaus, and Bruno Crispo. xesb: An enterprise service bus for access and usage control policy enforcement. In *Trust Management IV - 4th IFIP WG 11.11 International Conference, IFIPTM, Morioka, Japan, June 16-18, 2010. Proceedings*, pages 63–78, 2010.

[153] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, March 1994.

[154] Jason Cornwell, Ian Fette, Gary Hsieh, Madhu K. Prabaker, Jinghai Rao, Karen P. Tang, Kami Vaniea, Lujo Bauer, Lorrie Faith Cranor, Jason I. Hong, Bruce McLaren, Mike Reiter, and Norman M. Sadeh. User-controllable security and privacy for pervasive computing. In *Eighth IEEE Workshop on Mobile Computing Systems and Applications, HotMobile, Tucson, Arizona, USA, March 8-9, 2007*, pages 14–19, 2007.

[155] Norman M. Sadeh, Jason I. Hong, Lorrie Faith Cranor, Ian Fette, Patrick Gage Kelley, Madhu K. Prabaker, and Jinghai Rao. Understanding and capturing people's privacy policies in a mobile social networking application. *Personal and Ubiquitous Computing*, 13(6):401–412, 2009.

[156] George A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, November 1995.

[157] Christiane Fellbaum, editor. *WordNet: an electronic lexical database*. MIT Press, 1998.

[158] Rita.wordnet: Java library to access to the wordnet ontology, cited September 2014.

Appendix

Appendix A

Implementation Details of the Domain Model Editor

A.1. XML Schema of the Domain Model

The UML syntax of the domain model is given by the metamodel described in Chapter 4. Following is the concrete syntax of the domain model in XML. XML models generated from this schema can be fed to our policy translation infrastructure without any syntactic modifications for refining actions in implementation-level policies.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" attributeFormDefault="unqualified"
    elementFormDefault="qualified">
  <xs:element name="pims">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="pimactions" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="name" use="optional"/>
                <xs:attribute type="xs:string" name="synonym" use="optional"/>
                <xs:attribute type="xs:byte" name="seq" use="optional"/>
                <xs:attribute type="xs:string" name="actionRefmnt" use="optional"/>
                <xs:attribute type="xs:string" name="paramData" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="pimdata" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="storedin" use="optional"/>
                <xs:attribute type="xs:string" name="name" use="optional"/>
                <xs:attribute type="xs:string" name="synonym" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="name"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="psms">
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element name="psmcontainers" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="contimplementedas" use="optional"/>
                <xs:attribute type="xs:string" name="name" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="psmtransformers" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="name" use="optional"/>
                <xs:attribute type="xs:byte" name="seq" use="optional"/>
                <xs:attribute type="xs:string" name="outputcontainer" use="optional"/>
                <xs:attribute type="xs:string" name="inputcontainer" use="optional"/>
                <xs:attribute type="xs:string" name="crossPsmRefmnt" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="psmsystems" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="systemtransformers" use="optional"/>
                <xs:attribute type="xs:string" name="sysimplementedas" use="optional"/>
                <xs:attribute type="xs:string" name="name" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="name"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="isms">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ismcontainers" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="name" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="ismtransformers" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="name" use="optional"/>
                <xs:attribute type="xs:byte" name="seq" use="optional"/>
                <xs:attribute type="xs:string" name="inputimplecontainer" use="optional"/>
                <xs:attribute type="xs:string" name="outputimplecontainer" use="optional"
                    />
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="ismsystems" maxOccurs="unbounded" minOccurs="0">
```

```
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="implesystemtransformers" use="
                    optional"/>
                <xs:attribute type="xs:string" name="name" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="name"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing A.1: Domain model schema

A.2. Ecore Metamodel of the Domain Model Editor

Figure A.1 shows the graphical representation of the Ecore metamodel of the domain editor.

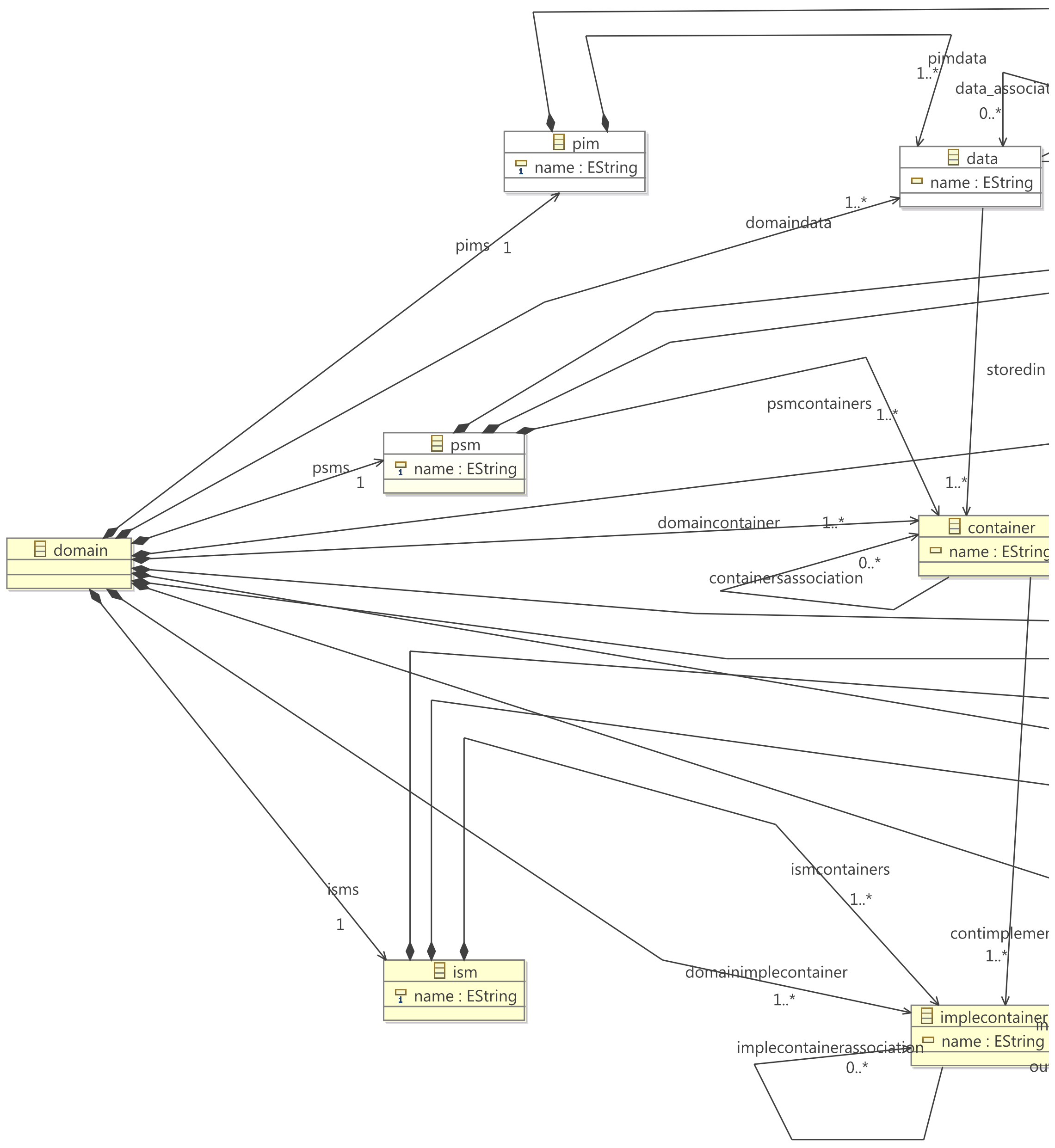
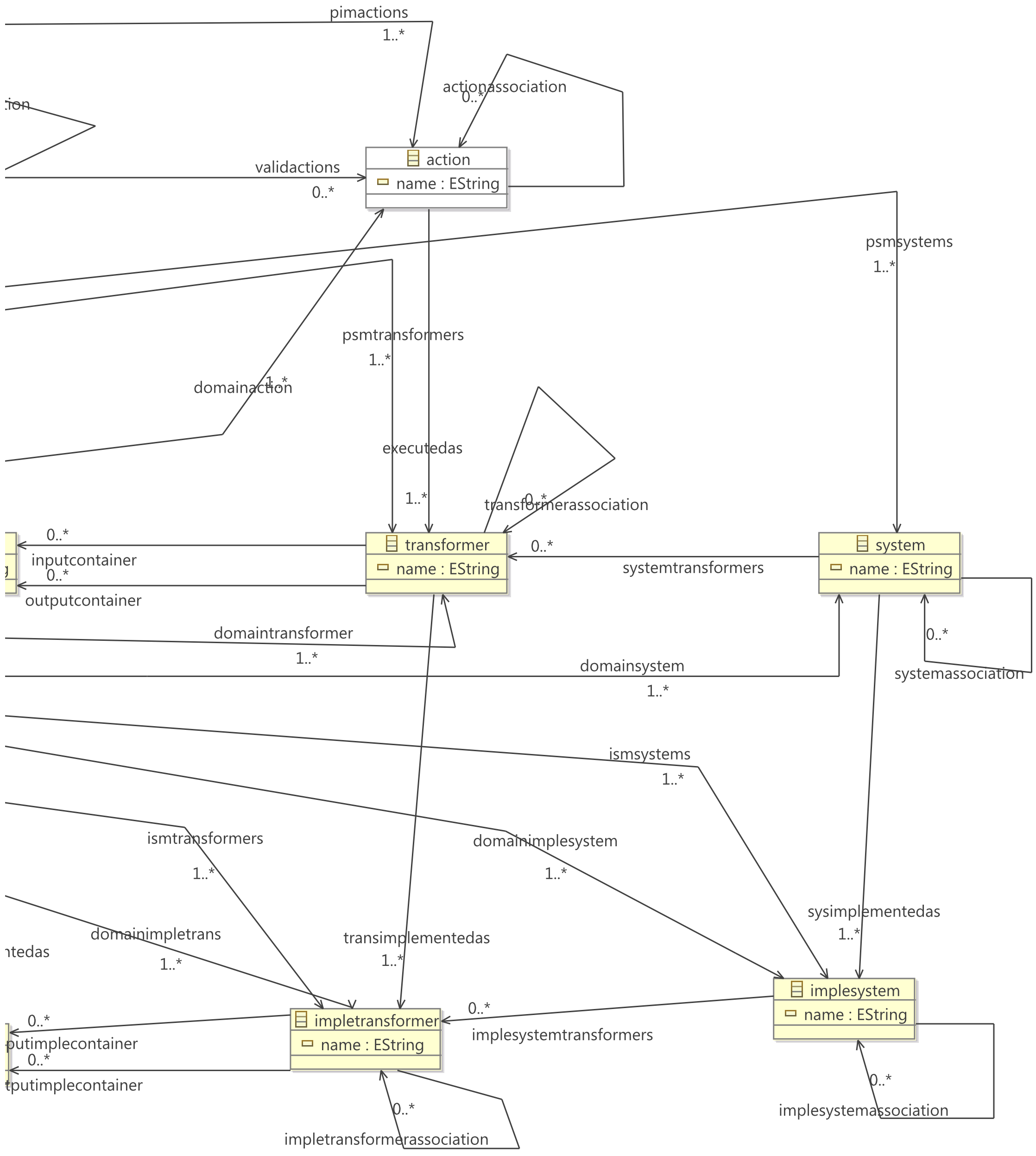


Figure A.1.: Ecore metamodel of the domain model editor


pimactions
1..*
actionassociation
0..*
action
name : EString
validactions
0..*
psmsystems
1..*
psmtransformers
1..*
domainaction
1..*
executedas
1..*
transformerassociation
0..*
0..*
inputcontainer
0..*
outputcontainer
transformer
name : EString
0..*
systemtransformers
system
name : EString
0..*
systemassociation
domaintransformer
1..*
domainsystem
1..*
ismsystems
1..*
ismtransformers
1..*
domainimplesystem
1..*
sysimplementedas
1..*
domainimpletrans
1..*
transimplementedas
1..*
0..*
putimplecontainer
0..*
tputimplecontainer
impletransformer
name : EString
0..*
implesystemtransformers
implesystem
name : EString
0..*
impletransformerassociation
0..*
implesystemassociation

Appendix B

Concrete Syntax of Policies

B.1. Specification-Level Policies

In the concrete syntax that is discussed in this chapter and is used in the implementation, SLPs are expressed with an identifier, a subject (optional, as subjects can also be expressed as event parameters) and a set of obligation formula expressed in future-time OSL with state-based operators.

The complete concrete XML syntax is described in three XML schema below. The schema in Listing B.1 describes the Specification Language with all the OSL operators (lines 42–66). It additionally uses Time and Event schemas in Listings B.2 and B.3 to describe events and time in SLP.

In this syntax, the *type* of a policy as described in Section 6.2 is expressed via different *ParamDataTypes* (lines 193 and 199). The boolean-valued attribute *tryEvent* (line 178) is used to express if an event is an intended event event (in case of tryEvent="true") or an actual event (in case of tryEvent="false").

B.1.1. Specification Language

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www22.in.tum.de/
        specificationLanguage"
 3    xmlns:tns="http://www22.in.tum.de/specificationLanguage" xmlns:time="http://www22.in.tum
        .de/time"
 4    xmlns:event="http://www22.in.tum.de/event" elementFormDefault="qualified">
 5
 6    <import namespace="http://www22.in.tum.de/time" schemaLocation="time.xsd" />
 7    <import namespace="http://www22.in.tum.de/event" schemaLocation="event.xsd" />
 8
 9
10    <element name="policy" type="tns:PolicyType" />
11
12    <complexType name="PolicyType">
13      <sequence>
14        <element name="initialRepresentations" type="tns:InitialRepresentationType"
15          minOccurs="0" />
16        <element name="obligation" type="tns:ObligationType" minOccurs="1"/>
17      </sequence>
18      <attribute name="name" type="string" use="required" />
19      <attribute name="subject" type="string" use="optional" />
20    </complexType>
```

```

  <complexType name="InitialRepresentationType">
    <sequence>
      <element name="container" type="tns:ContainerType" minOccurs="1"
        maxOccurs="unbounded" />
    </sequence>
  </complexType>

  <complexType name="ContainerType">
    <sequence>
      <element name="dataId" type="string" minOccurs="1" maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="string" use="required" />
  </complexType>

  <complexType name="ObligationType">
    <sequence>
      <group ref="tns:Operators" minOccurs="1" maxOccurs="1" />
    </sequence>
  </complexType>

  <group name="Operators">
    <choice>
      <element name="true" type="tns:TrueType" />
      <element name="false" type="tns:FalseType" />
      <element name="not" type="tns:NotType" />
      <element name="or" type="tns:OrType" />
      <element name="and" type="tns:AndType" />
      <element name="implies" type="tns:ImpliesType" />

      <element name="event" type="tns:EventOperatorType" />

      <element name="until" type="tns:UntilType" />
      <element name="always" type="tns:AlwaysType" />
      <element name="after" type="tns:AfterType" />
      <element name="during" type="tns:DuringType" />
      <element name="within" type="tns:WithinType" />

      <element name="repLim" type="tns:RepLimType" />
      <element name="repUntil" type="tns:RepUntilType" />
      <element name="repMax" type="tns:RepMaxType" />

      <element name="stateBasedFormula" type="tns:StateBasedOperatorType" />
      <element name="eval" type="tns:EvalOperatorType" />
    </choice>
  </group>


  <complexType name="TrueType">
    <sequence />
  </complexType>

  <complexType name="FalseType">
    <sequence />
  </complexType>

  <complexType name="NotType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
  </complexType>

  <complexType name="OrType">
    <sequence>
      <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
    </sequence>
  </complexType>

  <complexType name="AndType">
    <sequence>
      <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
    </sequence>
  </complexType>

  <complexType name="ImpliesType">
    <sequence>
      <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
    </sequence>
  </complexType>

  <complexType name="UntilType">
    <sequence>
      <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
    </sequence>
  </complexType>

  <complexType name="AlwaysType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
  </complexType>

  <complexType name="AfterType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attributeGroup ref="time:TimeAmountAttributeGroup" />
  </complexType>

  <complexType name="DuringType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attributeGroup ref="time:TimeAmountAttributeGroup" />
  </complexType>

  <complexType name="WithinType">

    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attributeGroup ref="time:TimeAmountAttributeGroup" />
  </complexType>

  <complexType name="RepLimType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attributeGroup ref="time:TimeAmountAttributeGroup" />
    <attribute name="lowerLimit" type="long" use="required" />
    <attribute name="upperLimit" type="long" use="required" />
  </complexType>

  <complexType name="RepUntilType">
    <sequence>
      <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
    </sequence>
    <attribute name="limit" type="long" use="required" />
  </complexType>

  <complexType name="RepMaxType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attribute name="limit" type="long" use="required" />
  </complexType>

  <complexType name="StateBasedOperatorType">
    <attribute name="operator" type="string" use="required" />
    <attribute name="param1" type="string" use="required" />
    <attribute name="param2" type="string" use="optional" />
    <attribute name="param3" type="string" use="optional" />
  </complexType>

  <complexType name="EvalOperatorType">
    <sequence>
      <element name="content" type="string" />
    </sequence>
    <attribute name="type" type="string" use="required" />
  </complexType>

  <complexType name="EventOperatorType">
    <sequence>
      <element name="params" type="tns:ParamsType" minOccurs="0"
        maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="string" use="required" />
    <attribute name="tryEvent" type="boolean" use="required" />
  </complexType>

  <complexType name="ParamsType">
    <sequence>
      <element name="param" type="tns:ParamType" minOccurs="0"
        maxOccurs="unbounded" />
    </sequence>

  </complexType>

  <complexType name="ParamType">
    <attribute name="name" type="string" use="required" />
    <attribute name="value" type="string" use="required" />
    <attribute name="class" type="string" use="optional" />
    <attribute name="policytype" type="tns:ParamDataTypes" use="optional"
      default="string" />
    <attribute name="dataID" type="string" use="optional"
      default="" />
  </complexType>

  <simpleType name="ParamDataTypes">
    <restriction base="string">
      <pattern value="string|dataUsage|container|data" />
    </restriction>
  </simpleType>

</schema>
```

Listing B.1: Specification language schema

B.1.2. Specifying Time in Policies

The XML schema shown in Listing B.2 describes the concrete syntax of time in OSL operators. At the abstract level, $Time$ is represented by a $(TAmount, TUnit)$ pair. In the concrete

XML syntax below, these two elements of the pair are denoted by *TimeAmountType* and *TimeUnitType*.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www22.in.tum.de/
    time"
  xmlns:tns="http://www22.in.tum.de/time" elementFormDefault="qualified">

  <attributeGroup name="TimeAmountAttributeGroup">
    <attribute name="amount" type="long" use="required" />
    <attribute name="unit" type="tns:TimeUnitType" use="optional"
      default="TIMESTEPS" />
  </attributeGroup>

  <complexType name="TimeAmountType">
    <annotation>
      <documentation>
        A time amount is a sum of elapsed time, which need not be of any
        specific intervals.
      </documentation>
    </annotation>
    <attributeGroup ref="tns:TimeAmountAttributeGroup" />
  </complexType>

  <simpleType name="TimeUnitType">
    <annotation>
      <documentation>
        Possible time units to quantify a time amount. One month is 30 days and a
        year is 12 months or 360 days.
      </documentation>
    </annotation>
    <restriction base="string">
      <enumeration value="TIMESTEPS" />
      <enumeration value="NANOSECONDS" />
      <enumeration value="MICROSECONDS" />
      <enumeration value="MILLISECONDS" />
      <enumeration value="SECONDS" />
      <enumeration value="MINUTES" />
      <enumeration value="HOURS" />
      <enumeration value="DAYS" />
      <enumeration value="WEEKS" />
      <enumeration value="MONTHS" />
      <enumeration value="YEARS" />
    </restriction>
  </simpleType>

</schema>
```

Listing B.2: Time schema

B.1.3. Specifying Events in Policies

The XML schema shown in Listing B.3 describes the concrete syntax of events in policies: it comprises of event name, event type to express if it is an actual or attempted event and a set of parameters (name, value pairs).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www22.in.tum.de/event"
  xmlns:tns="http://www22.in.tum.de/event" xmlns:time="http://www22.in.tum.de/time"

  elementFormDefault="qualified">

  <import namespace="http://www22.in.tum.de/time" schemaLocation="time.xsd" />

  <simpleType name="EventParameterDataTypes">
    <restriction base="string">
      <enumeration value="string" />
      <enumeration value="data" />
    </restriction>
  </simpleType>

  <complexType name="EventParameterType">
    <attribute name="name" type="string" use="required" />
    <attribute name="value" type="string" use="required" />
    <attribute name="type" type="tns:EventParameterDataTypes"
      use="optional" default="string" />
  </complexType>

  <complexType name="EventType">
    <annotation>
      <documentation>
        Events have a name and parameters.
        A try event represents an attempted usage event by a user.
      </documentation>
    </annotation>
    <sequence>
      <element name="parameter" type="tns:EventParameterType"
        minOccurs="0" maxOccurs="unbounded" />
    </sequence>
    <attribute name="event" type="string" use="required" />
    <attribute name="tryEvent" type="boolean" use="required" />
  </complexType>

  <element name="event" type="tns:EventType" />
</schema>
```

Listing B.3: Events schema

B.2. Implementation-Level Policies

The following is the concrete XML syntax of the ILPs used in the implementations.

B.2.1. ECA Rules

The XML schema of Listing B.4 shows the concrete syntax of the ECA rules. It describes both the preventive and the detective mechanisms (lines 280 – 297).

The *action* part of ECA rules is called Authorization Action. In the authorization action, apart from modification (lines 217 – 222) and inhibition (lines 224 – 229), we can also express both synchronous and asynchronous events to be executed as part of the execution enforcement strategy (lines 195 – 211). Contrary to the case of asynchronous events, when a synchronous event is executed, the enforcement waits for the status of the executed event (success or fail) from the executor processors. *Executor processors* (lines 188 – 193) are those components in the overall architecture that execute events for policy enforcement. There are two type of executor processors: one is a Policy Enforcement Point (PEP) described in Chapter 7 and the other is a Policy eXecution Point (PXP) which is an umbrella term for all components that are capable of executing some events both in the usage controlled infrastructure (e.g., delete a file from a usage controlled file system) and outside of it (send an email using webmail). An example use of PXP is shown in Section 6.4.1.4.

The Condition and Authorization Action part of the ECA rules are described below while the concrete syntax of Event and Time (time is also a part of the Condition) is the same as shown in Listings B.3 and B.2. These schema are referenced below.

Recall from Section 2.2.3 that conditions outside temporal and first-order logic are expressed using *eval*. One choice for expressing *eval* is XPath as shown in line 177 below. The rest of the past-time OSL and state-based operators are also a part of the condition.

Apart from event-condition-action, the ILPs also include a timestep declaration in order to express the granularity of the evaluation of the ECA rule (line 266). Timestep is expressed in numbers without the time unit: the default time unit is implementation-specific.

Additionally, the *InitialRepresentationType* (lines 306 – 311) allows us to declare a set of initial containers of data for specific type of policies as described in Section 6.2.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www22.in.tum.de/
       enforcementLanguage"
3    xmlns:tns="http://www22.in.tum.de/enforcementLanguage" xmlns:time="http://www22.in.tum.
       de/time"
4    xmlns:event="http://www22.in.tum.de/event" xmlns:cnd="http://www.iese.fhg.de/pef/1.0/
       condition"
5    elementFormDefault="qualified">
6
7    <import namespace="http://www22.in.tum.de/time" schemaLocation="time.xsd" />
8    <import namespace="http://www22.in.tum.de/event" schemaLocation="event.xsd" />
9
10   <complexType name="ConditionType">
11     <sequence>
12       <group ref="tns:Operators" minOccurs="1" maxOccurs="1" />
13     </sequence>
14   </complexType>
15
16   <complexType name="ConditionParamMatchType">
17     <attribute name="name" type="string" use="required" />
18     <attribute name="value" type="string" use="required" />
19   </complexType>
20
21   <complexType name="NotType">
22     <sequence>
23       <group ref="tns:Operators" />
24     </sequence>
25   </complexType>
26
27   <complexType name="TrueType">
28     <sequence />
29   </complexType>
30
31   <complexType name="FalseType">
32     <sequence />
33   </complexType>
34
35   <complexType name="OrType">
36     <sequence>
37       <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
38     </sequence>
39   </complexType>
40
41   <complexType name="AndType">
42     <sequence>
43       <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
```

```
    </sequence>
  </complexType>

  <complexType name="ImpliesType">
    <sequence>
      <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
    </sequence>
  </complexType>

  <complexType name="SinceType">
    <sequence>
      <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
    </sequence>
  </complexType>

  <complexType name="AlwaysType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
  </complexType>

  <complexType name="BeforeType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attributeGroup ref="time:TimeAmountAttributeGroup" />
  </complexType>

  <complexType name="DuringType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attributeGroup ref="time:TimeAmountAttributeGroup" />
  </complexType>

  <complexType name="WithinType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attributeGroup ref="time:TimeAmountAttributeGroup" />
  </complexType>

  <complexType name="RepLimType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attributeGroup ref="time:TimeAmountAttributeGroup" />
    <attribute name="lowerLimit" type="long" use="required" />
    <attribute name="upperLimit" type="long" use="required" />
  </complexType>

  <complexType name="RepSinceType">
    <sequence>
      <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
    </sequence>
    <attribute name="limit" type="long" use="required" />
  </complexType>

  <complexType name="RepMaxType">
    <sequence>
      <group ref="tns:Operators" />
    </sequence>
    <attribute name="limit" type="long" use="required" />
  </complexType>

  <complexType name="StateBasedOperatorType">
    <attribute name="operator" type="string" use="required" />
    <attribute name="param1" type="string" use="required" />
    <attribute name="param2" type="string" use="optional" />
    <attribute name="param3" type="string" use="optional" />
  </complexType>

  <complexType name="EvalOperatorType">
    <sequence>
      <element name="content" type="string" />
    </sequence>
    <attribute name="type" type="string" use="required" />
  </complexType>

  <simpleType name="ParamMatchDataTypes">
    <restriction base="string">
      <pattern value="string|dataUsage|container|data" />
    </restriction>
  </simpleType>

  <complexType name="ParamMatchType">
    <attribute name="name" type="string" use="required" />
    <attribute name="value" type="string" use="required" />
    <attribute name="type" type="tns:ParamMatchDataTypes" use="optional"
      default="string" />
    <attribute name="dataID" type="string" use="optional"
      default="" />
  </complexType>

  <complexType name="EventMatchingOperatorType">
    <sequence>
      <element name="paramMatch" type="tns:ParamMatchType"
        minOccurs="0" maxOccurs="unbounded" />
    </sequence>
    <attribute name="action" type="string" use="required" />
    <attribute name="tryEvent" type="boolean" use="required" />
  </complexType>

  <group name="Operators">
    <choice>
      <element name="true" type="tns:TrueType" />
      <element name="false" type="tns:FalseType" />
      <element name="not" type="tns:NotType" />
      <element name="or" type="tns:OrType" />
      <element name="and" type="tns:AndType" />
      <element name="implies" type="tns:ImpliesType" />

      <element name="eventMatch" type="tns:EventMatchingOperatorType" />
      <element name="conditionParamMatch" type="tns:ConditionParamMatchType" />

      <element name="since" type="tns:SinceType" />
      <element name="always" type="tns:AlwaysType" />
      <element name="before" type="tns:BeforeType" />
      <element name="during" type="tns:DuringType" />
      <element name="within" type="tns:WithinType" />

      <element name="repLim" type="tns:RepLimType" />
      <element name="repSince" type="tns:RepSinceType" />
      <element name="repMax" type="tns:RepMaxType" />

      <element name="stateBasedFormula" type="tns:StateBasedOperatorType" />
      <element name="eval" type="tns:EvalOperatorType" />
    </choice>
  </group>

```

```
<simpleType name="ParamInstanceTypes">
  <restriction base="string">
    <enumeration value="string" />
    <enumeration value="xpath" />
  </restriction>
</simpleType>

<complexType name="ParameterType">
  <attribute name="name" type="string" use="required" />
  <attribute name="value" type="string" use="required" />
  <attribute name="type" type="tns:ParamInstanceTypes" use="optional"
    default="string" />
</complexType>

<simpleType name="ExecutorProcessors">
  <restriction base="string">
    <enumeration value="pep" />
    <enumeration value="pxp" />
  </restriction>
</simpleType>

<complexType name="ExecuteActionType">
  <sequence>
    <element name="parameter" type="tns:ParameterType" minOccurs="0"
      maxOccurs="unbounded" />
  </sequence>
  <attribute name="name" type="string" use="required" />
  <attribute name="id" type="string" />
</complexType>

<complexType name="ExecuteAsyncActionType">
  <complexContent>
    <extension base="tns:ExecuteActionType">
      <attribute name="processor" type="tns:ExecutorProcessors"
        use="optional" default="pxp" />
    </extension>
  </complexContent>
</complexType>

<complexType name="DelayActionType">
  <attributeGroup ref="time:TimeAmountAttributeGroup" />
</complexType>

<complexType name="ModifyActionType">
  <sequence>
    <element name="parameter" type="tns:ParameterType" minOccurs="0"
      maxOccurs="unbounded" />
  </sequence>
</complexType>

<complexType name="AuthorizationInhibitType">
  <sequence>
    <element name="delay" type="tns:DelayActionType" minOccurs="0"
      maxOccurs="1" />
  </sequence>
</complexType>

<complexType name="AuthorizationAllowType">
  <sequence>
    <element name="delay" type="tns:DelayActionType" minOccurs="0"
      maxOccurs="1" />
    <element name="modify" type="tns:ModifyActionType" minOccurs="0"
      maxOccurs="1" />
    <element name="executeSyncAction" type="tns:ExecuteActionType"
      minOccurs="0" maxOccurs="unbounded" />
  </sequence>
</complexType>

<complexType name="AuthorizationActionType">
  <choice>
    <element name="allow" type="tns:AuthorizationAllowType" />
    <element name="inhibit" type="tns:AuthorizationInhibitType" />
  </choice>
  <attribute name="name" type="string" use="required" />
  <!-- indicates starting point in authorizationAction hierarchy -->
  <attribute name="start" type="boolean" use="optional"
    default="false" />
  <!-- reference to fallback authorizationAction (name), if executeActions/modification
    could not be performed successfully -->
  <attribute name="fallback" type="string" use="optional"
    default="inhibit" />
</complexType>

<!-- Preventive mechanisms can only come to decisions on the grounds of
  their current knowledge, so they use past-time formulas. The mechanism consists
  of an Event, a Condition, and an Action part (ECA). The Event is called trigger
  Event. When the condition evaluates to true the action part is executed. -->
<complexType name="MechanismBaseType">
  <sequence>
    <element name="description" type="string" minOccurs="0"
      maxOccurs="1" />
    <!-- Timestep size must not use timestep time unit! -->
    <element name="timestep" type="time:TimeAmountType"
      minOccurs="0" maxOccurs="1" />
    <element name="trigger" type="tns:EventMatchingOperatorType"
      minOccurs="0" maxOccurs="1" />
    <element name="condition" type="tns:ConditionType" minOccurs="0"
      maxOccurs="1" />
    <element name="authorizationAction" type="tns:AuthorizationActionType"
      minOccurs="0" maxOccurs="0" />
    <element name="executeAsyncAction" type="tns:ExecuteAsyncActionType"
      minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="name" type="string" use="required" />
</complexType>

<complexType name="DetectiveMechanismType">
  <complexContent>
    <extension base="tns:MechanismBaseType">
      <sequence />
    </extension>
  </complexContent>
</complexType>

<complexType name="PreventiveMechanismType">
  <complexContent>
    <extension base="tns:MechanismBaseType">
      <sequence>
        <element name="authorizationAction" type="tns:AuthorizationActionType"
          minOccurs="1" maxOccurs="unbounded" />
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="ContainerType">
  <sequence>
    <element name="dataId" type="string" minOccurs="1" maxOccurs="unbounded" />
  </sequence>
  <attribute name="name" type="string" use="required" />
</complexType>

<complexType name="InitialRepresentationType">
  <sequence>
    <element name="container" type="tns:ContainerType" minOccurs="1"
      maxOccurs="unbounded" />
  </sequence>
</complexType>

<complexType name="PolicyType">
  <sequence>
    <element name="initialRepresentations" type="tns:InitialRepresentationType"
      minOccurs="0" />
    <choice minOccurs="0" maxOccurs="unbounded">
      <element name="detectiveMechanism" type="tns:DetectiveMechanismType" />
      <element name="preventiveMechanism" type="tns:PreventiveMechanismType" />
    </choice>
  </sequence>
  <attribute name="name" type="string" use="required" />
</complexType>

<element name="policy" type="tns:PolicyType" />
</schema>
```

Listing B.4: Enforcement language schema

B.2.2. Specifying Time in ILPs

The XML schema shown in Listing B.2 describes the concrete syntax of time in ILPs also.

B.2.3. Specifying Events in ILPs

The XML schema shown in Listing B.3 also describes the concrete syntax of Event in the ECA rules.

Appendix C

A Summary of Symbols in Policy Derivation

Symbol	**Meaning**
$\mathcal{E}_{\mathcal{A}}$	Set of actions
$\breve{\mathfrak{A}}$	Set of classes of actions
$\mathcal{C}$	Set of containers
$\breve{\mathfrak{C}}_{\mathfrak{psm}}$	Set of classes of containers at the PSM level
$\breve{\mathfrak{C}}_{\mathfrak{ism}}$	Set of classes of containers at the ISM level
$\mathcal{D}$	Set of data
$\breve{\mathfrak{D}}$	Set of classes of data
$\mathcal{E}$	Set of events
$\breve{\mathfrak{E}}$	Set of classes of events
$\mathcal{E}\mathcal{D}ecl$	Event Declaration
$\mathcal{O}$	Set of objects ($\mathcal{O} \subseteq \mathcal{D} \cup \mathcal{C}$)
Φ_{sv}	The language containing state-based operators with variable data
$\Diamonddot$	Past eventually; semantically equivalent to $\underline{not}^{-}(\boxdot(\underline{not}^{-}))$
R_{seq}	*Sequence* refinement function that maps classes of transformers within the PSM and the ISM levels
R_{set}	*Set* refinement function that maps classes of transformers within the PSM and the ISM levels
$\tilde{R}_{seq}$	*Sequence* refinement function that maps classes of actions to classes of transformers at the PSM level; it also maps classes of transformers at the PSM level to those at the ISM level

$\tilde{R}_{set}$	*Set* refinement function that maps classes of actions to classes of transformers at the PSM level; it also maps classes of transformers at the PSM level to those at the ISM level
$\mathcal{S}tart$	Meaning
$\mathcal{E}_{\mathcal{T}}$	Set of transformers
$\breve{\mathfrak{T}}_{\mathfrak{psm}}$	Set of classes of transformers at the PSM level
$\breve{\mathfrak{T}}_{\mathfrak{ism}}$	Set of classes of transformers at the ISM level
$\tau_{\mathcal{A}}$	A translation function that is computed by taking a disjunction over the transformer-based and state-based refinements of action
$\tau_{\mathcal{E}}$	Transformer-based action refinement function
τ_{σ}	State-based action refinement function
$\tau_{\mathcal{P}}$	Future to past translation function that computes the first violation of a future-time formula in past time
$\mathcal{VD}$	Set of variable data

Table C.1.: List of symbols used in policy derivation

Appendix D

The Case of Dynamic Domains

In real world, domains evolve over time with services, logical systems and physical hosts being added, removed or modified. E.g., in an online social network (OSN), modifications might include the OSN provider offering encrypted file sharing service, the ad-service might be outsourced and various third-party applications can hook into the OSN. The client machines accessing the social network at various users' end might also change in their technical configurations, e.g., a new web browser or a new mail client can be installed. Any realistic policy translation must reflect these changes. In order to do so, **firstly**, we must be able to *recognize changes* in the domain structure. **Secondly**, recognized changes must be reflected in the domain model and we must *decide if and where* the changes must be incorporated in the policy derivation. The **third** requirement is to *apply the changes*. The new refinements should be reflected in the policy derivation by re-translation of the SLPs followed by revocation and deployment of the ILPs.

This chapter gives an initial intuition of a semi-automated approach to handle changes in a dynamic domain. Changes in domain are recognized manually by the power user and the delta is specified in a new (partial) domain model. E.g., if there is a new implementation class at the ISM level, it is assumed that some power user knows this and he specifies a 3-level domain model where existing (and potentially new) PSM classes are refined in terms of the new classes. In order to include these new features in policy derivation and enforcement, the new model is merged with the existing domain model, based on certain rules.

There is no universal way to merge any two domain models as different domains models might represent the same concept or artifact differently. Intuitively, existing ontologies could be leveraged to compare and bring together different classes based on their meanings. At the PIM level, another approach could be to compare classes based on their natural language meanings as the PIM classes describe the end user view of the domain and are therefore expressed in natural language terms. This chapter adopts the second approach for automatically merging two platform-independent models. The platform-specific and implementation-specific models could only partially be merged automatically, based on the closeness of the well-known technical terms as no known ontology for technical vocabulary was found to exist.

D.1. Merging Domain Models

Two type of relationships between elements[1] could be established after comparison: *containment*, between sequences in action refinements and *equivalence*. In order to establish one of the relationships, these information about elements were taken into consideration:

1. *syntactic information*, given by the name and the type of the element, e.g., a data can be compared only with a data, an action only with an action;
2. *structural information*, given by the element refinement, e.g., a sequence of transformer can be compared only with another sequence (and not a set) of transformers, and a set can be compared only with another set; and
3. *semantic information*, provided by the lexical equivalence/synonymity of terms or, in case of actions and transformers, by the formal semantics of their respective refinements.

For lexical equivalence among model elements, the concept of *Synonyms* is used. Apart from labels and associations with each other, each class in the model also has a set of synonyms which are intuitively, alternative names. E.g., a data class labeled 'photo' might have {picture, diagram, image} as synonyms. Synonyms are by default *equivalent* to each other. Hence policies specified using synonyms can be easily translated and deployed without requiring any negotiation between parties for agreement on terms. The set of synonyms for each class can be defined in two ways: firstly, when the domain model is defined by a power user and secondly, after the definition of the domain model during the merging process. In the second case, *lexical similarities* could be discovered using existing ontologies or using a dictionary update. The discovered lexically similar terms are added to the set of synonyms for the class in the existing domain model.

Basic Rules for comparing and merging two domain models are as follows:

Rule 1 *Comparison and merging is possible only between elements at the same levels in the domain, i.e., PIM to PIM, PSM to PSM, ISM to ISM.*

Rule 2 *At the PSM and the ISM levels, classes associated to the same system class are compared; classes associated to different systems exist independent of each other.*

Rule 3 *Elements are named using standard vocabulary at the PSM and the ISM levels across models. E.g., file, records, system calls are universal terms. Synonyms, if any, are added only manually by the power users.*

Rule 4 *The new domain model is always an addition to the existing model, it's not a replacement.*

When applying these rules to compare model elements and merge different classes in domain models, the following order based on the dependencies among elements is followed: data and container classes are compared and included first, followed by the transformer classes because transformer classes definitions need container classes as their targets. System classes are included at last because *systems are collections of transformers* and complete system definitions require all the relevant transformers.

[1]Recall from Chapter 4 that domain model elements are *classes* of data, actions, containers, etc.

The next section discusses the changes needed in the existing architecture in order to operationalize these concepts.

D.2. Modifications to the Existing Architecture

Chapter 7 describes in detail the component architecture of the usage control infrastructure. Policy derivation is performed by the Policy Translation Point (PTP) which is a part of the Policy Management Point (PMP). For incorporating domain changes, The PTP was extended and split into two main components: a *Translation Engine* (TE), which is an implementation of the translation logic and the policy derivation process and, an *Adaptation Engine* (AE), which provides the adaptation and merging part for the model-based policy translation. Figure D.1 gives an overview of the architecture.

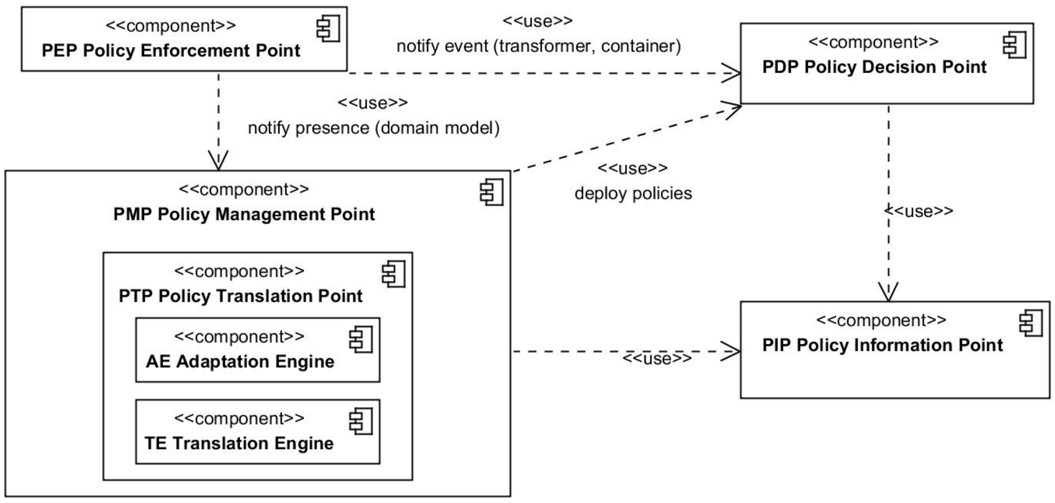

<<component>>
PEP Policy Enforcement Point
<<use>>
notify event (transformer, container)
<<component>>
PDP Policy Decision Point
<<use>>
notify presence (domain model)
<<use>>
deploy policies
<<use>>
<<component>>
PMP Policy Management Point
<<component>>
PTP Policy Translation Point
<<component>>
AE Adaptation Engine
<<component>>
TE Translation Engine
<<use>>
<<component>>
PIP Policy Information Point

Figure D.1.: PTP split into adaptation engine and translation engine

D.2.1. Recognizing Changes

In dynamic domains, one of the initial challenges is how to recognize change in the environment: in this case, how to recognize that there is a new *type* of system (e.g., database replaced by file system storage) or a new system implementation (e.g., a new database implementation) and the existing domain model needs to be updated. It is assumed that the respective power users specify the model of the new systems, but somehow this new model must be injected in the usage control infrastructure and the infrastructure must know that a new model is there. From the perspective of the usage control infrastructure, there are two ways to do it: *active* and *passive*.

In the *active* approach, the usage control infrastructure would detect a new component, query for a corresponding domain model and incorporate the changes. It was recognized that this approach is not feasible for several reasons. E.g., there may be parts of applications that are installed but are not subject to the usage control monitoring, such as extensions or Windows updates. Then, malicious applications might not be detected upon installation. Also the usage control infrastructure would have to broadcast messages and listen for responses on every start of the infrastructure and thereafter at periodic intervals. This could be a performance overhead.

In the *passive* approach, the infrastructure would wait and listen for a new system to initiate a handshake and notify its presence or update by sending its domain model as a parameter of the notification. Upon notification, the infrastructure would incorporate the changes in the domain model, e.g., re-translate the existing policies from the specification to the implementation level, revoke deployed policies and deploy new policies.

In this implementation, *the passive approach is adopted* that relies on a notification from the PEP of the new system that sends a three-layer domain model. The core task of the Adaptation Engine is to then compare the two models element by element and update the existing domain model.

The usage control infrastructure can start in one of two initial conditions: either, there is no initial model and the domain model provided by the first installed PEP will become the *base* domain model for future adaptations; or, the infrastructure has an existing model that has been specified for standard layers of abstraction, e.g., an operating system.

D.2.2. Including Changes in Policies

Once the domain model has been updated, the updates need to be reflected in deployed policies. This can be done in two ways: *delayed adaptation* and *real-time adaptation*. In delayed approach, already deployed policies are maintained until the infrastructure components restart. Any new specification-level policies are translated using the updated base domain model.

In the real-time adaptation, after the domain model is merged, the already deployed policies are immediately revoked, retranslated and deployed. This would require that a) the infrastructure keep track of all the deployed implementation-level policies corresponding to one specification-level policy along with their instantiation parameters and data ids generated at deployment and, b) the enforcement state of each deployed policy be exported before revocation and be imported for the newer version of that policy.

System As A New Instance Sometimes, the domain model need not change because a new instance of an existing system is installed. For example, a new backup server is added to the network. The backup server has the same implementation details as the main server. The PEP of this instance will notify its presence but the domain model has no new elements to be included. The only difference is in the storage of the protected data. In order to keep track of the data, the backup server must be populated with data via a monitored channel. This means that the protected data is tracked by the usage control infrastructure and the data-container mapping on the new server is included in the data state of the monitored system at the PIP.

The next section describes one implementation of merging domain model for the SCUTA instantiation of the running example.

D.3. Implementation & Evaluation

The online social network SCUTA with usage control enforcement capabilities has already been introduced in Chapter 6. Enforcement of policies in two different system configurations was also mentioned and one of them with a Linux distribution OS was discussed there. The second usage control enforcement with Mozilla Firefox web browser and Windows 7 32-bit operating system was augmented with capabilities to handle the dynamic domain case.

Lexical similarity In terms of merging the domain models at the PIM level, a natural language lexical library, namely RitaWordnet [156, 157, 158] was used to establish semantic relations between terms. For two words, e.g., "picture" and "photo", a normalized distance value was chosen which is closer to 0 if the concepts are related and closer to 1 if they are not semantically related. The PSM and the ISM -level classes were specified using standard terms and synonyms were only specified by the power user. No automated comparison for new synonyms was performed for these two levels in the domain model.

The synonyms provide a vocabulary negotiation mechanism between two end-users who want to use the same domain model for their policy translation but still want to use the terms with which they are most accustomed to. Synonyms are either provided in a static manner as a property of an element of the domain model, or synonyms are discovered in a dynamic manner while merging by the use of a lexical similarity comparison of terms.

D.3.1. Performance Analysis

The machine on which the tests were run was configured as follows: Intel i7-2640M 2.8ghz, 8Gb Ram, Windows7 64bit, SSD Intel 160Gb. In order to replicate a real case scenario, 160 different policies were used as fix set of input policies for evaluating the performance. The time required for the translation, revocation, deployment and instantiation of a single policy depends on the complexity of the domain model.

At first, the performance was measured per component. The first evaluated component was the translation engine (TE), part of the PTP. The base domain model contained 54 unique elements to be evaluated and had as defined systems the browser and the operating system both at PSM and ISM. The time required by the TE to translate 160 policies was an average of 11.4 seconds. Next component, PMP, required in average 24.3 seconds to revoke, re-translate, and redeploy all 160 affected policies. The third step was to evaluate the adaptation engine (AE). The base domain model consisted of 42 elements, the new domain model 43 elements, and the merged domain model 54 elements. In total 54 elements were affected by the update and 12 elements were added to the base domain model. The average time required for this to take place was 0.4 seconds.

The final step, was to measure the end-to-end performance. End-to-end performance refers to the time perceived by an user from the moment the request to adapt the domain model is sent to the moment when the updated policies are deployed and ready to be used. The average time required by the UC infrastructure was of 24.38 seconds to revoke affected policies, translate and redeploy them. This time also includes the communication

Operation on 160 policies	average time
1. Translate	11.4 seconds
2. Translate + Revoke + Deploy	24.3 seconds
3. Adaptation	0.04 seconds
4. Adapt + Revoke + Translate + Deploy	24.38 seconds

Table D.1.: Performance overhead

overhead induced by the communication between the PEP and the PMP. The results are shown in Table D.1.

During the performance test, the following observations were made. First of all, optimizing the translation process affects the performance of the entire infrastructure and improves the user experience. Secondly, the time required by the adaptation depends directly on the number of the elements composing the domain model because every element must be compared to its equivalent. On the other hand, adaptation takes place only when a new PEP is installed and new PEPs are installed less frequently compared to adding or revoking policies. This means that this time is similar to the time required when a newly configured system starts up for the first time. It is a common case that such a time is greater. Thirdly, in the context of a social network, the end-to-end time appears only at the first access of the data, not every consequent access. A good analogy would be that the end-to-end time is the time required for a ticket machine to issue a ticket, once the ticket is issued the performance overhead is only that introduced by checking the ticket at each access point, in our case the PDP verifying the policies.

In conclusion, an adaptive model-based policy translation introduces an insignificant performance overhead, whose cost is relatively minor compared to having static model-based policy translation.

D.3.2. Validation

Validation was done on the lines of *Metamorphic Testing*. The adaptation engine acts as a transformation function which takes as input domain models M_i and M_j, and merges them to M_k, i.e., $M_i \oplus M_j \rightsquigarrow M_k$. We have observed that the properties that this function verifies are commutativity and associativity. Commutativity means that $M_i \oplus M_j = M_j \oplus M_i$ and associativity means that $(M_i \oplus M_j) \oplus M_j = M_i \oplus (M_j \oplus M_k)$. Another observed property is that for a series of transformations $M_i \oplus Mj \oplus ... \oplus M_k \rightsquigarrow M_n$, there is new domain model M_y such that $M_i \oplus M_y \rightsquigarrow M_n$. Because elements which define sequences can have different names, the merged domain model and the expected domain model need not be identical. Therefore, the verification of these properties and the comparison of equivalent domain models is done by comparing the translated policies.

D.3.3. Limitations

This chapter described and implemented one way of including real world domain changes into the policies derived for usage control enforcement. The approach is semi-automated because the initial changes to a domain structure are always recognized by the power user who includes them in the domain model or constructs a new domain model. The further steps of recognizing and comparing the new or updated domain model with an existing one in the infrastructure are automated. Policies are also re-translated, revoked and deployed seamlessly.

With the example implementation, two major limitations were recognized. The data from a domain model can be specified with business domain terms which are not recognized by a lexical ontology. Therefore, a specific use case ontology should be put in place. This is a limitation introduced by the use of an external lexical library. We have not taken into consideration domain model elements which are labeled with composed terms, e.g., UserProfie, Project-portofolio, UpdateUserTable. A composed term cannot be recognized "as is" by the lexical library.

A second limitation is introduced by the fact that the domain model is always updated and backward compatibility is maintained. This means that the base domain model always expands. In case a domain model must be simplified, this must be done either manually or by using a special notification signed by the administrator to mark that the new domain model must be accepted as a replacement for the elements it contains.

Appendix E

Acronyms

DME	Domain Model Editor
ECA	Event-Condition-Action
EMF	Eclipse Modeling Framework
EPAL	Enterprise Privacy Authorization Language
FACPL	Formal Access Control Policy Language
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
ILP	Implementation-level Policy
ILPs	Implementation-level Policies
ISM	Implementation-Specific Model
KAoS	Knowledge Acquisition in automated specification
LTL	Linear Temporal Logic
ODRL	Open Digital Rights Language
OMA-DRM	Open Mobile Alliance - DRM Rights Expression Language
OSL	Obligation Specification Language
OSN	Online Social Network
PDP	Policy Decision Point
PE	Policy Editor
PEP	Policy Enforcement Point
PIM	Platform-Independent Model

PMP	Policy Management Point
PSM	Platform-Specific Model
PTP	Policy Translation Point
RBAC	Role-Based Access Control
RE	Requirements Engineering
SAML	Security Assertion Markup Language
SE	Software Engineering
SLP	Specification-level Policy
SLPs	Specification-level Policies
UML	Unified Modeling Language
UWE	UML-based Web Engineering
XACML	eXtensible Access Control Markup Language
XRML	eXtensible Rights Markup Language