UNIVERSITY OF WOLLONGONG

Detecting and Resolving Redundancies in EP3P Policies

A thesis submitted in fulfillment of the
requirements for the award of the degree

Master of Computer Science (Research)

from

UNIVERSITY OF WOLLONGONG

by

Farzad Salim

School of IT and CS.
September 2006

Dedicated to

My Family

Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Farzad Salim
September 7, 2006

Abstract

Current regulatory requirements on data privacy make it increasingly important for enterprises to be able to verify and audit their compliance with their privacy policies. Traditionally, a privacy policy is written in a natural language. Such policies inherit the potential ambiguity, inconsistency and mis-interpretation of natural text. Hence, formal languages are emerging to allow a precise specification of enforceable privacy policies that can be verified.

The EP3P language is one such formal language. An EP3P privacy policy of an enterprise consists of many rules. Given the semantics of the language, there may exist some rules in the ruleset which can never be used, these rules are referred to as *redundant* rules.

Redundancies adversely affect privacy policies in several ways. Firstly, redundant rules reduce the efficiency of operations on privacy policies. Secondly, they may misdirect the policy auditor when determining the outcome of a policy. Therefore, in order to address these deficiencies it is important to identify and resolve redundancies.

This thesis introduces the concept of *minimal privacy policy* - a policy that is free of redundancy. The essential component for maintaining the minimality of privacy policies is to determine the effects of the rules on each other. Hence, redundancy detection and resolution frameworks are proposed. Pair-wise redundancy detection is the central concept in these frameworks and it suggests a pair-wise comparison of the rules in order to detect redundancies. In addition, the thesis introduces a policy management tool that assists policy auditors in performing several operations on an EP3P privacy policy while maintaining its minimality. Formal results comparing alternative notions of redundancy, and how this would affect the tool, are also presented.

Acknowledgments

On completion of such a time-consuming journey, there are always many people to acknowledge. This thesis would have never been possible without them. It is a pleasant aspect that I have the opportunity to express my gratitude for all of them.

Firstly, I would like to express my sincere appreciation to both of my supervisors, Prof. Aditya Ghose and Prof. Rei Safavi-Naini for their guidance, encouragement and support through the course of this work. I would also like to thank Peter Harvey for his critical questions as well as supporting me at those times when my research encountered obstacles. Thanks also goes to Janos Tsakiris for proof reading my thesis.

I would like to acknowledge the friendship of Peter, Janos, Chee Fon, Victoria, Siamak, Sara, as well as that of Elinor, whose support during the thesis writing stage will not be forgotten. I'd especially like to thank my cousin Ehsan for bringing a smile to my face and providing motivation when times were tough.

I would like to express my deep and sincere gratitude to my father and mother who formed part of my vision and taught me to see the good in everything and be a constructive part of the whole. Their faith in me, advises, and encouragements provided a persistent inspiration for my journey in this life. I am grateful for my brother Farhad and my sisters Sara and Sohaila for always being there for me.

Contents

List of Figures

Chapter 1

Introduction

In this thesis we address the problem of *redundancy* in privacy policies written in EP3P language. We will show that detecting and resolving redundant rules is important for specification and maintenance of real world enterprises.

In this chapter, we will focus on the motivation behind the ideas presented in this thesis. We conclude by highlighting our contribution and presenting an outline of the structure of the thesis.

1.1 Motivation

Privacy is the right of individuals to determine for themselves how, when and to what extent information about them is used and shared with others. Protecting privacy is considered one of the fundamental requirements of Information Systems (IS). In order to balance the need for the free flow of information whilst respecting the human right to privacy, the Organization for Economic Cooperation and Development (OECD) defined a set of privacy principles back in the 1980's. The development of information technology in recent years has led to the use of information systems in various enterprises such as hospitals, banks, government agencies, etc. With the exponential growth in the amount of data flow within and between these enterprises, privacy violations has become commonplace. In response to these violations many countries have enacted legislation that is often based on OECD principles. For example, the United States has regulations for health care (HIPPA[1]) and the protection of childrens data (COPPA[2]).

With the growing need for protecting privacy, enterprises publish a *privacy policy* that is a representation of different legal regulations, promises made to data owners, as well as more restrictive internal practices of the enterprise. In addition, enterprises use a *privacy enforcement system* responsible for ensuring that all access to the private data occurs exclusively according to an enterprise privacy policy. The main components of a privacy enforcement system that is of interest to us is *Policy Decision Point* (PDP). A PDP uses the privacy

[1]http://www.hhs.gov/ocr/hipaa/

[2]http://www.coppa.org/

policy to make a decision on whether to authorize/deny an access request.

However, it is very difficult to develop a PDP that uses a privacy policy that is written in a natural language. Such policies inherit the potential ambiguity, inconsistency and misinterpretation of natural text [7, 10, 40]. To address these problems, formal *privacy policy languages* such as Platform for Enterprise Privacy Practices (EP3P) [3] [9], Enterprise Privacy Authorization Language (EPAL) [51], XACML [6] are emerging. These languages define a syntax and semantics that allows the expression of *authorization rules* that determine the enterprise's posture toward privacy. More precisely, a rule provides the constraints on who is allowed to perform what actions on which data for which purpose under what circumstances. In addition, it allows the expression of how long the data will be retained or who will be informed under what circumstances. In our study we have focused on EP3P language as the language for formalizing an enterprise privacy policy.

The dependability of enterprises on formal privacy policies has made them the subject of much recent research. As a result, several desirable attributes for privacy policies such as *correctness, consistency* and *minimality* have been identified and policy languages would like to provide support for them [16, 54, 6]. In addition, the broad scope of the principle of privacy has introduced many operations that need to be performed on privacy policies, hence policy management tools are also emerging to provide support for these operations [13, 46].

Sometimes the coexistence of some rules in a policy may result in the grant or the denial of an authorization request, despite the intention of the policy auditor, while writing the policy. A formal privacy policy is said to be *correct* if it only reflects the intentions of the policy auditor. Even though it is almost impossible to formally define the correctness attribute that was introduced, it is very important to provide methods that assist policy auditors in creating correct policies. By having an incorrect policy, a breach of privacy is highly likely.

Support for the consistency of a policy has been the subject of much research [52, 32, 42, 1, 2, 43, 14]. The general notion of consistency implies that a policy is *consistent* if it does not provide conflicting directives [43, 42]. For example, a policy that simultaneously authorizes and denies an access request. There are two types of techniques for detecting and resolving redundancies. The first, *specification-time*, tackles the inconsistent rules when they are added to the policy and the second, *run-time*, eliminates inconsistencies during the execution.

The third desirable attribute for policies is *minimality.* A policy is minimal if every rule within the policy can affect an authorization response. We refer to the rules that are ineffective as *redundant.* The desirability of having a minimal policy is universally accepted by policy researchers. However, the importance of detecting the rules that can not affect the outcome of the policy at specification time has not been addressed. At first glance, the minimality attribute can be considered an afterthought and not crucial, because the redundant rules

[3]In this thesis we write EP3P to solely refer to the formal language used in the Platform for Enterprise Privacy Practices. We write *E-P3P* to refer to the platform itself.

only increase the size of a policy. However, by taking into account the minimality along with the correctness attributes, it can be seen that the author of the policy must have had an intention for adding such rules to the policy. Hence, the fact that they are redundant does not suggest that they can be ignored.

Out of these three attributes, EP3P language supports runtime consistency by guaranteeing that an access request is either authorized or denied. However, there are several factors that add to the complexity of creating and maintaining real world enterprise privacy policies [7, 8, 58, 12, 16]:

- A privacy policy of a medium/large size enterprise constitutes tens to hundreds of rules.
- Privacy policies may change over time to address changing legislation. Hence, construction of a privacy policy is not a one-off process.
- There may be more than one policy auditor who is responsible for maintaining an enterprise privacy policy.
- In an enterprise, different departments (or branches) may have direct control over their own data. However, at the organizational level, interoperability of all these information subsystems is required. Hence, privacy policies may need to be combined to provide a unified view of the enterprise privacy practices.
- Frequently, enterprises share privacy sensitive data and exchange their privacy policies. Hence, these enterprises may frequently need to determine if privacy policies can accommodate the privacy of the incoming data.

The above factors have led to the introduction of mechanisms for automating the management operations for EP3P privacy policies. Some policy researchers in [13, 46, 12] have introduced *refinement* as a tool to verify if one policy is more restrictive than another. Others have introduced techniques that allow different EP3P privacy policies to be *combined* while they retain their independence. These authors have assumed that there already exists one or more formal privacy policy(s). They mutually emphasize the complexities involved in creating a formal privacy policy and report about the influence of redundant rules on the efficiency of their proposed mechanisms.

Currently, there exists no mechanism to assist policy auditors in constructing a correct EP3P privacy policy. Similarly, there is no way to ensure the minimality of an EP3P privacy policy. Since a privacy policy has a temporal attribute, the rules that constitute a policy may change and the changes to a rule may affect other rules in the policy. Sometimes, a new rule makes some of the existing rules redundant. By the same token, it is possible that a new rule is made redundant by some of the existing rules in the policy. Hence, the process of creating an EP3P privacy policy becomes error-prone and challenging. The same complexity can be observed when combining two or more privacy policies. As privacy policies get larger, these operations becomes even more complex. It is our belief that redundancies can be prevented if, at rule specification-time, the consequences of adding the rule to the policy can be determined. Such an early determination of the consequences of rules on each other can also assist in creating correct privacy policies. In addition, having minimal policies can improve the efficiency of the EP3P refinement and EP3P combination tools.

1.2 Contribution

In this thesis we address the problems that arise when redundant rules exist in an EP3P policy. To the best of our knowledge this problem has not been attempted yet. The thesis provides the necessary tools for maintaining the minimality of an EP3P privacy policy. The main contribution of this thesis is the definition of redundancy in the context of EP3P language, a succinct formal classification of redundant rules and a mechanism for detecting and resolving redundancies. As a proof of concept we also develop a tool that can assist policy auditors in creating minimal policies.

As a first step in dealing with the problem of redundancy, we formally define three different concepts of redundancy, 1) *Redundancy*, 2) *Absolute Redundancy*, 3) *Pair-wise Redundancy.* The first captures a common notion of redundancy: a rule is redundant if removing it from the policy does not change the authorization response of the policy. The second is that a redundant rule always remains redundant regardless of how many other rules are added to the policy. This ensures that non-redundant rules are not mistakenly labeled as redundant. In addition, it ensures that redundancy detection is not dependent on the order in which the rules are added to the policy. Hence, redundant rules can be detected incrementally as they are being added to the policy. With the first two notions of redundancy it is not clear whether a rule is made redundant by one rule or more. This lead us to the third notion of redundancy that only takes into account the rules that are absolute redundant with respect to one other rule in the policy.

Pairwise-redundant rules are further classified into three general categories, *include, contradict* and *shadow*. One rule includes another rule if it already provides the same outcome (authorization decision) and fires for all the access requests where the other rule fires. A rule shadows another rule if that rule has strictly higher priority and fires in the same situations. A "deny" rule contradicts an "authorize" rule if the "deny" rule has an equal or higher priority and fires in the same situations.

We prove that by detecting pairwise-redundant rules, we would be in a position to detect most of the redundancies in an EP3P privacy policy and the undetected redundancies are those whose occurrence is very rare. We develop an algorithm that allows us to detect all the pair-wise redundant rules.

In addition, we introduce a framework that can assist a policy auditor in resolving the redundancies. Our main concern is to provide a resolution method that is intuitive and as close as possible to a decision that the author of the policy may make when confronted with the redundancy. We use an information theory approach to determine whether the authorization information that a rule provides can be derived for other rules. The result of this comparison allows us to make a decision on how to resolve the detected redundancies.

We tackle the problem of conflicting resolution suggestions. This occurs when a rule is made redundant by more than one rule and the resolution suggestions are different. We provide a procedure for dealing with these conflicts while resolving redundancies in a policy. We prove that the procedure is the most efficient way for resolving all the detected redundancies in an EP3P policy.

As a proof of concept, we present an integrated implementation of a policy management tool, the *EP3P Policy Management Console* (*EPMC*). It allows management operations such as adding new rules to a policy as well as the *evaluation* of the existing rules to ensure that the policy is minimal. *EPMC* can assist policy auditors in resolving the redundancies.

1.3 Thesis Structure

In Chapter 2 we discuss the emerging needs for privacy control systems within enterprises and the importance of formal privacy policy languages in addressing such needs. Further, we survey some of the related privacy policy languages and briefly describe their limitations. We then concentrate on the area of policy enforcement and describe some of the policy enforcement frameworks, more specifically we describe the E-P3P framework that can be used for enforcing the EP3P privacy policies.

Chapter 3 focuses on the EP3P privacy policy language. We formally describe the syntax and semantics of the language. Further, we introduce several desirable attributes that privacy policies would like to preserve. We then give a survey of several policy management operations that are performed on privacy policies. Finally, we give an account of some of the existing tools for managing policies.

Chapters 4 is devoted to introducing our redundancy detection framework. It introduces the fundamental concepts for detecting redundancies in an EP3P privacy policy. Further, we provide the necessary algorithms for detecting them.

In chapter 5 we provide our heuristics for resolving redundancies. We will also explain how to deal with conflicting ways to resolve a redundant rule and provide algorithms to automate the redundancy resolution procedure.

Chapter 6 gives some details of the tool *EP3P Policy Management Console (EPMC)* that we develop as a proof of concept. By using an example we will show how the tool uses the redundancy detection and redundancy resolution frameworks to assist policy auditors in performing operations such as update and revision of an EP3P privacy policy. We provide a brief demonstration of its design and implementation.

Finally, in Chapter 7 we conclude by summarizing our results, provide a brief critical analysis of these results and based on this analysis, suggest areas upon which this research can be expanded upon by future researchers.

Chapter 2

Background and Related Work

This chapter aims to provide a general overview of the literature relevant to this thesis. The first section gives the definition of privacy and briefly describes privacy issues within enterprises. The second section surveys the existing privacy policy languages. The third section discusses management of privacy polices and the operations that are usually performed on them. Finally, the last section describes some of the related privacy policy enforcement mechanisms.

2.1 Enterprises and Privacy Issues

The first real definition of privacy dates back to the 19th century, when Warren and Brandeis defined privacy as "the right to be let alone" [55]. Since this definition many new definitions have been given. One of these definitions that seems to cover most of the aspects of privacy that are generally agreed upon, was given by the British Committee on Privacy and Related Matters [5]:

> "The right of the individual to be protected against intrusion into his personal life or affairs, or those of his family, by direct physical means or by publication of information."

Whereas traditionally privacy was focused on the individual, the emergence of new technologies and social changes have broadened the areas for which privacy is of concern. Some of these include Media privacy, Territorial privacy, Communication privacy and Information privacy. Information privacy in particular is of concern because with the increase in electronically stored data, the ease and speed to which it can be accessed, the possibility of data mining and the decentralization of data storage, technically the information about people could be accessed without the individuals permission.

This aspect of privacy, which is also referred to as "data protection", is the subject of the discussions in many environments that handle personal information. Westin has given a more precise definition of this aspect [56]:

"Privacy is the claim of individuals, groups and institutions to determine for themselves, when, how and to what extent information about them is communicated to others."

The protection of privacy is becoming a great concern for enterprises. They collect and store large amounts of confidential data about their employees, customers and partners [30, 31]. In addition, managing and accessing this data is fundamental for their business: confidential information is retrieved, analyzed and exchanged between people (and applications) that have different roles within an organization (or across organizations) to enable providing services and transactions [40, 10, 31, 7]. For example, health care systems are developed based on common standards (e.g., standardized electronic patient case files) linking general practitioners, hospitals and social centers on national and international scales. While modern health care networks can improve costs and effectiveness of the health care system, the patient's privacy is affected yet many patients may wish for their privacy to be protected. With the development of such health care networks, an increasing amount of sensitive medical data will be collected, stored, shared among different health care professionals and transferred to different sites around the world. Hence, it has to be guaranteed that only authorized personnel can have access to the patient's medical information.

Critical guidelines for determining an approach to privacy center around two principles. The first is the privacy principle of necessity of data processing, which guarantees that users can only access/process personal data, if it is necessary for their authorized operations. The second is the privacy principle of purpose binding, which ensures that the purpose of data processing is compatible with the purposes for which the data is obtained [30].

The collection, retention and use of such data is of increasing concern to individuals, especially regarding what personal data they reveal to enterprises. Further, they are concerned about where this data might end up. Hence, the protection of privacy has become one of the foremost concerns of enterprises dealing with confidential data [50]. However, privacy protection in real world enterprises is very complex because of the following fundamental requirements [50, 9, 10, 46, 12, 11]:

- In complex organizations, different branches or departments may have direct control over their own data. However, at the organization level, interoperability of all these information subsystems is required.
- Frequently, different enterprises share part of their data and must agree on a unified data disclosure policy.
- Legislation provides another constraint on privacy protection.
- Even within a single, homogeneous system, different groups of users and different classes of objects may be treated according to different principles.

- A further degree of complexity is introduced by the temporal dimension.

This last point in particular introduces added complexity as the privacy rules may change over time, so there should be a way to express such a dynamic property of privacy in order for systems to enforce them.

To address this governing need for privacy management, enterprise privacy technologies are emerging. The fundamental goals for such technologies is to ensure the data practices satisfy the privacy rules [10, 13]. The successful implementation of such systems is based on the following components [45, 47]:

- *Flexible language(s):* For writing privacy policies that can adapt to changing legislation.
- *Enforcement mechanisms:* Strong and provable mechanisms to enforce privacy policies in a variety of contexts.

2.2 Privacy Policy Languages

An enterprise privacy policy often reflects different legal regulations, promises made to customers, as well as more restrictive internal practices of the enterprise [40, 38]. In general, a "privacy policy" defines what data is collected, for what purpose the data will be used, whether the enterprise provides access to the data, who are the data recipients, how long the data will be retained and who will be informed under what circumstances [40].

Privacy policies do not refer to specific applications or systems in the IT infrastructure, nor do they refer to specific technologies [50]. This permits the exchange of privacy policies between organizations that want to share private data. The data management system can then adapt to changing requirements by disabling policies or replacing old policies with new ones.

Privacy polices can be viewed from three different perspectives [53]:

1. Preferences: The data usage preferences of a particular data owner whose data may be collected by an organization. For example, a customer may have a preference such as:

   I do not want my medical record to be used for marketing purposes.

2. Promises: The privacy promises that an organization advertises to enable a data owner to determine whether their preferences match the data practices of the organization. For example, the following statements show the privacy promise of a hospital [18]:

   We will not use your medical record for any purpose other than the primary purpose for which the data was collected.

3. Privacy Practices: The fine-grained access or privacy-control policy that governs the actual usage of the data by users of one or more organizations. Privacy practices are more detailed and restrictive than promises. They are usually defined and/or implemented by non-legal players. For example,

   > A patient's medical-record can be read by their primary doctors, for treatment purposes.

Traditionally, privacy policies are written informally in a natural language. However, to use privacy policies as the core of privacy control technologies, they must be machine readable. There are several privacy policy languages that target different perspectives of privacy policies. Preferences may be formalized using APPEL [26]. Another example are the IBM Privacy Services that use data subject policies to protect the release of personal data that is stored in a personal wallet. Privacy promises can be formalized using P3P [23]. Privacy practices can be formalized using EP3P, EPAL or XACML [6]. These languages can be used to formalize the privacy policies associated with data that is handled inside an enterprise or exchange between two organizations in one or two enterprises.

Using formal policy languages reduce ambiguities and omissions that appear in natural language policy documents [7]. In the following sections we will describe each of the formal privacy policy languages in more details.

2.2.1 P3P

P3P is a semi-structured privacy policy language that allows an organization to specify its website privacy practices in a machine-readable format [18]. The Platform for Privacy Preferences Project (P3P) is a current project of the World Wide Web Consortium (W3C) [23]. The goal was to enable machine-readable privacy disclosures that could be retrieved automatically by web browsers [22, 25], ensuring that users are informed about privacy policies before they release personal information to enterprises.

The core of a P3P policy are "data schemas" and "statements". A data schema defines data elements that can be used in privacy statements. In other words schemas define a vocabulary of the language. Statements are the actual rules in a P3P policy. A P3P policy consists of a sequence of "statement" elements that include the following sub-elements:

- "Purpose": Describes purposes for which information is collected.
- "Recipient": Describes the intended users of the collected information.
- "Retention": Defines the duration for which the collected information will be kept.
- "Data-group": Provides a list of individual data items (specified using "data" tags) that are collected for stated purposes in the statement.

P3P has predefined values for "purpose" (12 choices), "recipient"(6), and "retention" (5). It also has predefined types of data items. It is also possible to assign *categories* to data items. A policy provides opt-in or opt-out values for the required attribute of "purpose" and "recipient" elements. The opt-in value says that the user must provide explicit consent to the stated purpose/recipient. The opt-out value gives the user flexibility to reject the specified purpose/recipient, but the user needs to take additional action for the opt-out to take effect.

Examples of "purpose" can be *current* or *contact.* The element *current* means completion and support of activity for which the data was provided. The element 'contact' element specifies contacting visitors for marketing of services or products through a communication channel other than voice telephone.

Examples of "recipient" can be *ours*, *same* or *unrelated.* The element 'ours' means the data will not be sent outside the enterprise which collects the data. The element *same* means legal entities following our practices. The *unrelated* element means legal entities whose practices are unknown to us.

Examples of "retention" can be *stated-purpose*, *business- practices*, or *indefinitely.* The *stated-purpose* means the data will be deleted when the purpose for which the data was collected is satisfied. The *business-practices* refers to a long term retention but with a destruction timetable.

Intuitively, a P3P statement enumerates the types of data collected, explains how the data will be used, identifies the data recipients and makes a variety of other disclosures including information about dispute resolution. The P3P language allows multiple elements to exist in a statement. The meaning of a statement with multiple elements of each type is that all listed data elements can be used by all listed recipients for all listed purposes (i.e., corresponding to a cross-product). If multiple statements contain the same data element, both can be applied (i.e., implementing a union). One important characteristics of P3P statements is that they are positive, meaning that they express what operations can be performed rather than what cannot [3].

```
<POLICY>
  ... ...
  <STATEMENT>
    <PURPOSE>
      <individual-decision required="opt-in"/>
      <contact required="opt-in"/>
    </PURPOSE>
    <RECIPIENT><ours/></RECIPIENT>
    <RETENTION>
      <business-practices/>
    </RETENTION>
    <DATA-GROUP>
      <DATA ref="#user.home-info.online.email"/>
      <DATA ref="#dynamic.miscdata">
         <CATEGORIES><purchase/></CATEGORIES>
      </DATA>
    </DATA-GROUP>
  </STATEMENT>
</POLICY>
```

Figure 2.1: P3P Policy

For example, assume there is a bookseller who needs to obtain some minimum personal information to complete a purchase transaction. This information includes full-name, shipping address, and credit card number. The bookseller also uses the purchase history of customers to offer personalized book recommendations, for which it needs customers email addresses. Figure 2.1 shows the bookseller's policy encoded in P3P policy language.

A segment of the policy shown in Figure 2.1 allows the bookseller to use miscellaneous purchase data to create personalized recommendations and email them to the customer. However, the opt-in value of the required attribute of the purposes individual-decision and contact implies that explicit customer consent is necessary. By default, the value of the required attribute is set to always, which precludes the possibility of customer opt-in or opt-out.

2.2.2 APPEL

The P3P language provides an standard language for encoding user's privacy preferences, called A P3P Preference Exchange Language (APPEL). APPEL is an XML language for specifying users privacy preferences. Privacy preferences are expressed in APPEL as a list of *rules* [3]. These rules can subsequently be matched against a P3P privacy policy to determine whether the website policy is acceptable to the user [21, 3, 24]. An APPEL rule consists of two parts:

- *Rule head*: Specifies the action (behavior) of the rule if it is activated (or "fires"). The behavior can be "authorize" the request, implying that the policy conforms to preferences specified in the rule body. Or it can be "block", implying that the policy does not respect users preferences.

- *Rule body*: Provides the pattern that is matched against a policy. The format of a pattern follows the XML structure used in specifying privacy policies described earlier.

An APPEL rule is satisfied by matching its constituent expressions and (recursively) their subexpressions. The elements used in the matching process are P3P elements: "statement", "data", "purpose", "recipient", and "retention". Every APPEL expression has a connective attribute that defines the logical operators between its sub-expressions. A connective can be: *or, and, non-or* (negated or), *non-and* (negated and), *or-exact*, and *and-exact*. The default connective is *and*. The two unusual connectives are *and-exact* and *or-exact*, whose semantics are as follows [3]:

- *and-exact*: A successful match is made if (a) all of the contained expressions can be found in the policy and (b) the policy contains only elements listed in the rule. For the *and* connective, only part (a) needs to be satisfied, not part (b).

- *or-exact*: A successful match is made if (a) one or more of the contained expressions can be found in the policy, and (b) the policy only contains elements listed in the rule. For the *or* connective, only part (a) needs to be satisfied, not part (b).

```
<appel:RULESET>
   <appel:RULE behavior="block">
    <POLICY>
      <STATEMENT>
         <PURPOSE appel:connective="or">
         <contact/>
         <telemarketing/>
         </PURPOSE>
      </STATEMENT>
    </POLICY>
   </appel:RULE>
   <appel:RULE behavior="request"/>
      <appel:OTHERWISE/>
   </appel:RULE>
 </appel:RULESET>
```

Figure 2.2: APPLE Preferences

Figure 2.2 shows an APPEL rule set that implements the following preference:

> Block sites whose policies indicate that the information collected can be used for contact or telemarketing.

The rule set consists of two rules. The first rule specifies conditions under which access to a site must be blocked. The second rule is guaranteed to fire if the first one does not, since the rule body contains the appel:OTHERWISE element and allows the site to be accessed.

2.2.3 EP3P

P3P was a major step toward formalizing privacy policies. However, P3P was mainly intended for privacy promises (i.e., readability purposes) [13] rather than expressing the fine-grained data practices nor the enforcement of these policies within an enterprise [9, 40, 22]. A language for supporting expression and enforcement of privacy policies must meet several requirements such as [6]:

- The language must support constraints on who is allowed to perform which actions on which resources, as well as constraints on the purposes for which data was collected or is to be used.
- The language must enable the expression of directly-enforceable policies: policies that describe subjects, resources, actions, and access conditions using identifiers that can be mapped directly and automatically to the actual objects and operations used by computer applications.
- To be the same as, or integrated with, the language used for access control policies, because both types of policies will usually control access to the same resources and must not be in conflict.

To fulfill these requirements, Karjoth et. al. [40] proposed a fine-grained *data model* which provides abstract elements that are represented in a privacy policy. Further, they extended the Flexible Authorization Framework (FAF) [35, 37, 36] with multiple granters (i.e., security officer and privacy officer). Finally, they have annotated an authorization rule with post conditions (i.e., obligations that must be carried out if the rule fires).

Following this, Ashley et. al. introduced EP3P (The Platform for Enterprise Privacy Practices) language based on the privacy policy model introduced by [40]. EP3P concentrates on the core privacy authorization while abstracting data models and user-authentication from all deployment details. EP3P policies, unlike P3P, are enforceable, as they are written and structured in similar to access control policies that one may find in the security domain [9].

The EP3P privacy policy data model consists of five components: “data users”, “data types”, “purposes”, “actions”, “conditions” and “obligation”. For conveniently specifying rules In EP3P, the data the users and the purposes are categorized. These categories are ordered in hierarchies of “data users”, “data types” and “purposes”. The hierarchies allow for writing general statements as well as exceptions. In addition EP3P specifies three sets of “actions”, “conditions” and “obligations” [9]. These categories are explained further below:

- *Data users* are individuals who are accessing or receiving data, and are seen in a privacy context, as privacy policies will depend on the relationship between the individual requesting data and the individual to whom the data refers to. For example, one type of data user might be "physician" while another might be a "finance-group".

  Although generally considered a separate class, the data owner, known as *data subject* becomes a data user when they access their own data. Granting rights to the data subject defines whether they can access and/or update their data stored at the enterprise.

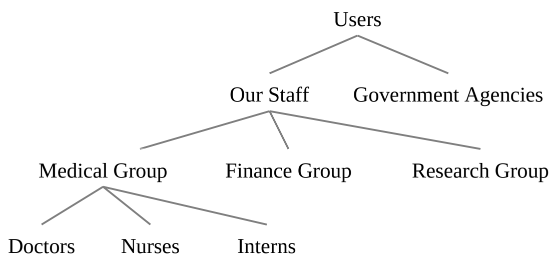


Figure 2.3: Data User Hierarchy

- *Data types* define the collected data that are handled differently from a privacy perspective. Typically, the data types used in privacy policies are high-level descriptions of data, such as medical-record or customer contact information. The concept of data type is very important for privacy enforcement because data are usually treated in groups in privacy policies. For example, a privacy policy may be:

  > Purchase history can be used for research analysis in an anonymous way and contact information cannot be used for marketing purpose.

  Contact information and purchase history are examples of data types. Name, address, zip code, and telephone number, etc. belong to contact information data type.

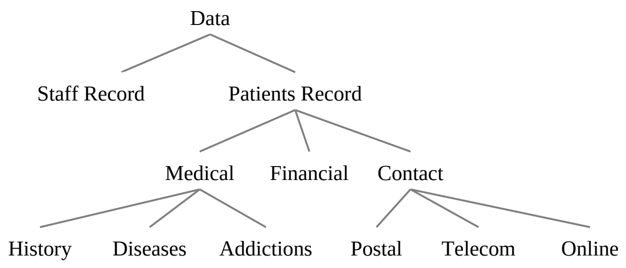


Figure 2.4: Data Type Hierarchy

- *Purposes* explain for what reason or business purpose the collected data will be used. Most privacy laws, codes of conduct, codes of ethics of different computer societies, as well as international privacy guidelines or directives, require the principle of purpose binding [31]:

  > The purpose for which personal data is collected and processed should be specified and legitimate. The subsequent use of personal data is limited to those specified purposes, unless there is an informed consent by the data subject.

  According to this principle, a subject may only access data for a purpose for which it has been collected. Thus, unlike in access control, a subject has to specify a purpose for accessing an object.

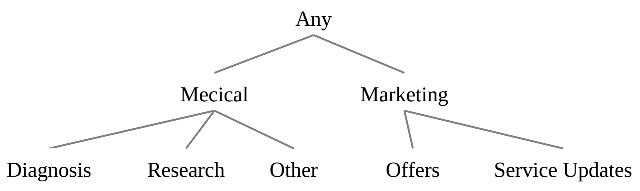


Figure 2.5: Purpose Hierarchy

- Privacy policies make distinctions about who can perform activities based on the *action* being performed. For example, a policy might say that anyone in the company can create a customer record, but that only certain data users are allowed to read that record.

- *Conditions* represent context under which a data handling practice can or cannot be performed. Often, legislation or privacy policies make statements based on the satisfaction of context conditions, so policy rules can be qualified based upon those conditions. For example, a policy might require that email addresses can only be used for marketing purposes if the owner of the address is older than 13 years old. In this case, “older than 13 years” is a condition that might have to be evaluated in order to determine if a particular data access request should or should not be allowed.

- *Obligations* are activities that must be fulfilled as a result of accessing or using personal information. In many privacy policies, the access control decisions are not simply a matter of making an “allow” or “deny” decision. But, granting or denying access incurs an obligation on the data user to take additional actions. A sample obligation can be ”delete patient’s record in 30 days”. An example of a real policy which specifies an obligation is the following [49]:

"Medical researchers may read protected health information for medical research projects if the patient explicitly authorizes the release and is either a legal adult or has obtained an opt-in from their legal guardian. When protected health information is released to the medical researcher, the patient must be notified within 90 days and the protected health information must be deleted by the researcher within one year."

```
<?xml version="1.0" encoding="UTF-8"?>
<vocabulary>
    <user-category id="Users" parent="" />
    <user-category id="Our Staff" parent="Users" />
    <user-category id="Medical Group" parent="Our Staff">
    </user-category>
    <user-category id="Government Agencies" parent="Our Staff">
      <short-description language="en">
         All the agencies such as low enforcement or ensurance
      </short-description>
    </user-category>
    <user-category id="Doctors" parent="Medical Group">
      <short-description language="en">
           Doctors in the hospital
      </short-description>
    </user-category>
    <user-category id="Nurses" parent="Medical Group">
      <short-description language="en"/>
    </user-category>
    <data-category id="Data" parent="">
      <short-description language="en"/>
      <long-description language="en"/>
    </data-category>
    <data-category id="Patient Record" parent="Date">
      <short-description language="en"/>
      <long-description language="en"/>
    </data-category>
    <data-category id="Staff Record" parent="Date">
      <short-description language="en"/>
      <long-description language="en"/>
    </data-category>
....
</vocabulary>
```

Figure 2.6: EP3P Vocabulary

These elements are used to formulate authorization rules that are the core of any privacy policy. A policy contains a collection of rules, where each rule specifies the actual conditions under which access is to be allowed or denied. A rule contains a special set of predicates used to determine whether the rule applies to a given decision request, a specification of the result to be returned if the rule is satisfied and an optional set of condition predicates that must be true with respect to a given decision request if the rule is to be satisfied. If either the applicability predicates or the condition predicates evaluate to false, the rule is treated as not applicable to the given request (we will elaborate more on this point in the next chapter). In EP3P, the predicates in the applicability section are grouped as subjects [users], resources [data], actions and purpose. At least one predicate in each group must be true for the rule to be applicable. The condition section may contain zero or more predicates. If the condition is missing, then the condition is treated as implicitly true. If present, it may contain either a single predicate or the boolean "and" combination of predicates. The following is an informal representation of an EP3P authorization rule:

> ALLOW/DENY [Data User] to perform [Action] on [Data Type] for [Purpose] provided [Condition]. Carry out [Obligation]

EP3P uses XML as the representation language - which allows easy processing, transfer and enforcement by control systems. A snippet of an EP3P rule is shown in Figure 2.7. These rules are the formalization of the following informal policies [49]:

Rule1: Any user is allowed to receive personal information from the hospital for the purpose of affecting the health and safety of an individual to whom the information relates if upon disclosure notification thereof is mailed to the last known address of the individual to whom the information relates.

Rule2: A Canadian law enforcement agency is allowed to have personal information disclosed to it by an institution for any purpose if the institution that is disclosing the personal information is a law enforcement institution.

```
<rule id="rule1" ruling="allow" priority=3>
  <user-category refid="any-user" />
  <data-category refid="personal-information" />
  <purpose refid="affecting-health-and-safety" />
  <action refid="disclosure" />
  <obligation refid="institution-provides-proper-
     notice-to-thedata-subject-for-disclosure" />
</rule>

<rule id="rule2" ruling="allow" priority=2>
  <user-category refid="canadian-law-
                          enforcement-agency" />
  <data-category refid="personal-information" />
  <purpose refid="any-purpose" />
  <action refid="disclosure" />
  <condition refid="disclosure-by-law-
                      enforcement-institution"/>
</rule>
```

Figure 2.7: EP3P Rules

EP3P language has formal semantics to determine whether an access request may be authorized or denied. Given a query, the authorization service determines if the condition of a rule is satisfied. Then, the rules which satisfy the condition(s) are examined in the order of their priority. The evaluation process stops when it reaches an applicable rule that contains an "allow" or "deny" ruling. We will give further details about the EP3P language in the next chapter.

2.2.4 EPAL

IBM's Enterprise Privacy Authorization Language (EPAL) is a commercial version of EP3P and uses the same data model with a more comprehensive representation of conditions and obligations. EPAL rules are evaluated based on the order in which they are written. However, EP3P explicitly assign a priority to each rule. These priorities specify the order in which the rules must be evaluated. The main difference between EPAL and EP3P is is that the rules in EPAL are totally ordered whilst EP3P rules are patially ordered. Hence, there may exist two different rules in an EP3P policy with different priorities.

2.2.5 XACML

eXtensible Access Control Markup Language is a formal policy language that is not limited to expressing privacy policies. The language provides a policy model used to describe general access control requirements and has standard extension points for defining new functions, data types, etc. In addition, XACML has a request/response language that allows a user to form a query to ask whether or not a given action should be allowed. XACML and EPAL and therefore EP3P are very similar in concept as well as their implementations. Annie Anton [6] provides a comprehensive comparison of XACML and EPAL.

2.3 Limitations of Privacy Policy Languages

In this section we will describe the limitations that are reported in the literature for P3P and APPEL. We defer the limitations of EP3P to the next chapter where we formally define and analyze the language.

2.3.1 P3P Language

Many researchers have noted the limitations of P3P [58, 3, 53, 34, 33, 7]. The ambiguity of the language is clearly mentioned by Shunter et. al. [53], where they write:

> "P3P allows statements to be ambiguous and redundant without associating any meaning to such statements. Recipient, retention and purpose are not clearly separated."

Another study conducted by “citiGroup” in their position paper reveals:

> “The same P3P policy could be represented to users in ways that may be counter to each other as well as to the intent of the site...”

For example, P3P uses pre-defined labels to advertise how long data can be retained. The main label is *stated-purpose* that signals that the data will be retained for the stated purposes. Since P3P allows multiple purposes in a statement, this means that the data can be retained as long as any of the stated purposes are still active. The consequence is a variety of retention times for the same data element [53].

Another criticism refers to the expressiveness of the language and reveals that P3P is unable to express the primary question: “What can be derived from all that collected data” [53].

The other major problem with P3P is that it does not specify whether the same data element can be included in multiple statements. Because if each data element can only be used once, the resulting policy would be very coarse-grained. The following statement could no longer be expressed:

> “*ours* can use the data for *stated-purpose* and *contact* while *delivery* can use the data for *stated-purpose* only.”

If we assume that the same data element can be used in different statements, P3P would need a notion of conflict resolution that defines what statements are considered as conflicting and how conflicts are resolved. This holds in particular if the same data element is collected as ’always’ as well as ’opt-in’ or ’opt-out’ [7, 53, 58].

Yu et. al. compliment the work of Shunter et. al. and emphasize that the fundamental reasons for a variety of ambiguity problems in P3P is the lack of formal semantics.

> “There are often several ways to interpret a particular P3P policy and the P3P specification does not clearly state which way is correct. There are also many ways for a P3P policy to be semantically inconsistent as we now discuss.”

They introduce *data-centric* semantics in which every P3P policy’s data usage part is mapped into five relations: purpose, recipient, retention, data-optional and data-category. Contrast this with the original P3P that has only three relations: purpose, recipient and retention. In addition, there are situations in which P3P statements can be inconsistent. One instance would be when a statement has conflicting purposes and retention value. For example, a statement in a P3P policy that collects users postal information for the purpose *historical* with retention *no-retention*.

They handle the mentioned semantic problems by employing *integrity constraints* (i.e., if a data item is collected for the purpose *historical*, then its retention cannot be *no-retention*).

So, if a P3P policy is translated and violates these integrity constraints then this policy is invalid. Finally they provide an algorithm for converting a P3P policy to data-centric semantics.

Other perspectives for providing semantics to P3P exist, such as *purpose-centric* in which a data item, along with a purpose, determines other elements (i.e., recipients and retention) in a P3P statement. So it is not clear which perspective is preferred. Coarse-grained (data-centric) semantics enable users to act more conservatively when disclosing information without worrying about complex relationships between the major components of P3P. However, with fine-grained (purpose-centric) semantics, it is always possible to derive data-centric semantics. On the other hand, purpose-centric semantics reveal more details about the operation of enterprises, which may or may not be desirable.

In general, the above problems mainly flag the complexity involved in creating privacy policies and the care that must be taken while converting informal policies to P3P. It is commonly believed that there is a lack of guidelines for design and creation of P3P policies [7, 53, 58, 3, 54].

2.3.2 APPEL

At first glance, APPEL comes across as an attractive language. It is small, readable, and uses standard XML for syntax. Hence, it seems easy to write APPEL rules. The user has to provide the pattern of the substructure of interest in the rule body and associate the desired behavior in the rule head, like the one shown in Figure 2.2. Unfortunately, the APPEL constructs interact with the P3P policy language in unintended ways. Several researchers [34, 3, 53], who have tried to implement APPEL rules note that it is easy to write a preference that appears correct and find that it does not accomplish the intended goal. Quoting from the position paper by the Joint Research Center of the European Commission,

> "There are various problems with the preference expression language (APPEL). Constructing the logic of matching patterns is very complex and involves various inherent contradictions."

The limitations of APPEL has also been noticed by Agrawal et. al., where he reports that the shortcomings of APPEL arise from a fundamental design choice. Specifically, the fact that APPEL only allows logical operations at nodes corresponding to P3P elements. They believe that these shortcomings cannot be overcome without completely redesigning the language. Hence, they introduce a new preference language, XPref, which is a subset of XPath [3].

2.4 Policy Management Tools

In this section, we will review some of the tools developed to facilitate the formalization, detection and resolution of policy *anomalies*. Generally, an anomaly refers to an undesirable situation that may arise due to the coexistence of some rules in the policy. However, the exact definition of an anomaly is unique to each language.

One of the first attempts to address the complexity of creating security policies was the implementation of a tool called POWER (POlicy Wizard Engine for Refinement) [44]. The main design objective of POWER was to assist policy auditors in transforming an abstract policy to a collection of implementable rules. It provides a Graphical User Interface (GUI) to hide the syntax of the underlying policy language. Further, it checks the syntax. However, POWER did not consider the semantics of the language, hence it could not detect policy anomalies.

As more expressive policy languages were developed, tools emerged that check the semantics as well as the syntax of the language. Firstly we will discuss the anomaly detection tools for security policies written in Ponder [29, 32, 43].

Ponder is a declarative language for specifying security policies with role-based access control [29]. Ponder supports *authorizations* and *obligations*. It also has a number of supporting abstractions that are used to define rules: domains for hierarchically grouping objects and constraints for controlling the firing of rules at runtime. Ponder rules are defined over sets of objects formed by applying set operations, such as *union*, *intersection* and *difference* to the objects held within its domains. In the policy specification, two sets have special significance, the *subject* set and the *target* set. These sets are used to represent the managed objects that the policy operates on. Ponder supports both positive authorizations that permit an action, and negative authorizations that forbid an action. Authorization rules define what activities a subject can perform on a set of target objects and are essentially access control rules to protect resources from unauthorized access. On the other hand, obligation policies define what activities a subject must or must not perform on a set of target objects. In addition, authorizations can be constrained by boolean expression (conditions).

Lupu and Sloman [43] define two types of conflicts that can occur between rules written in Ponder language: *modality* conflicts and *application* specific conflicts. Modality conflicts arise when two rules with opposite modality refer to the same subject, actions, and targets. Application specific conflicts occur when two rules contradict each other due to the context of the application, for example conflict of duties or conflict of interest.

Lupu and Sloman [47, 43] identify three cases where modality conflicts may happen in Ponder policies:

- The subjects are both obligated-to and obligated-not-to perform actions on the targets.

- The subjects are both authorized and forbidden to perform actions on the targets.
- The subjects are obligated but forbidden to perform actions on the targets.

The following example gives the flavor of the language and a modality conflict. Consider a policy for regulating the medication of patients in a hospital. An initial informal policy may state “Nurses must maintain patients temperatures within normal limits”. To ensure that the above policy holds the policy auditor must write a set of rules specifying which drugs must be administered and which additional actions must be taken, plus an authorization policy to permit analgesics to be administered.

> */* Administer analgesics when temperature is too high */*
> Obligation rule: O+ on high-temperature nurses[administer (analgesics)]

Note that at this point there are no authorizations for patients to administer analgesics. If the administrator omits to specify such a policy, wrongly assuming it may have been specified in the general access control policies, the obligation rule may not be fulfilled. In addition, the following rule introduces conflict because of the obligation and negative authorization of the rules.

> */* Nurses are not authorized to administer analgesics */*
> Authorization rule: A- nurses[administer(analgesics)]

Sloman et. al., have implemented a tool for checking policies for conflicts, as they are specified. In addition, the authors also proposed possible ways to resolve the detected conflicts without human intervention. The main idea of conflict resolution is called domain nesting. It is based on the belief that policies defined for more specific domains are assigned higher priorities than the ones for more generic domains. Their method of conflict detection involves examining the entire set of rules, or at the very least all the rules found in one domain. In contrast Graham, et. al. [32] provided a method for incrementally checking a small number of rules when a new rule is added or modified, to determine if the new set of rules is consistent. They modify and use decision tables to store the policy rules, as decision tables have been proved applicable for detecting conflicts between the rules. In addition they introduce algorithms for detecting conflicts between the rules. The major drawback of their approach is the need for transforming policy rules to decision tables. Further, there is no treatment for obligation policies. Finally, for policies to be executable, they must be transformed from this internal representation into a form that can be executed.

Agrawal et. al., have taken a different view to the works mentioned above by defining four general tasks that are essential to policy management. They refer to these tasks as *policy ratification*:

- Dominance check: To determine rule x is dominated by a group of rules $y = \{y_1, \ldots, y_n\}(n \geq 1)$ when the addition of x does not effect the behavior of the policy.

- Conflict check: To determine if there are two rules in the policy that provide directives that can not be satisfied simultaneously.
- Coverage check: To know if rules have been defined for a certain range of input parameters. For example, the administrator may want to make sure, regardless of the value of the rule conditions, at least one rule has a true condition.
- Consistent priority assignment: In most policy systems, rules are prioritized by assigning an integer value to them. The goal of consistent priority assignment is to assign priorities to the rules, such that the minimum number of covered rules exist.

In their work, Agrawal et. al. use techniques from constraint satisfaction programming and linear algebra to provide algorithms to find a feasible solution for insertion and deletion of rules and correct priority assignment. Although some of the introduced tasks are essential for policy management, their proposed solutions are very general and not directly applicable to a specific policy language, such as EP3P.

In a similar fashion to the previous policy management tools, Al-Shaer et. al. [4] have studied the anomalies that can arise within firewall policies. A firewall is a network element that controls the traversal of packets across the boundaries of a secured network based on a specific firewall policy. A firewall policy is a list of ordered filtering rules that define the actions performed on matching packets. A rule is composed of fields such as protocol type, source IP address, destination IP address, source port and destination port, and an action field. Each network field could be a single value or range of values. Filtering actions are either to accept, which passes the packet into or from the secure network, or to deny, which causes the packet to be discarded. The packet is accepted or denied by a specific rule if the packet header information matches all the network fields of this rule. Otherwise, the following rule is examined and the process is repeated until a matching rule is found or the default policy action is performed [4]. Note that the main similarity of a firewall policy and an EP3P policy is in the mid-policy termination of the rule evaluation.

Al-shaer et. al, emphasizes that when firewall rules are defined, attention has to be given to rule relations and interactions between rules in order to determine the proper rule ordering and guarantee correct firewall policy semantics. They believe that:

> ...the effectiveness of firewall security is dependent on providing policy management techniques and tools that enable network administrators to analyze, purify and verify the correctness of written firewall rules.

Al-Shaer et. al., identifies five relationships (i.e., completely disjoint, exactly matched, inclusively matched, partially disjoint, correlated) that may relate two or more rules. For instance, two rules are said to be *completely disjoint* if every field in the first rule is not a subset, not a superset, and not equal to the corresponding fields in the second rule. Further, given these relationships they present various types of anomalies that may arise in a firewall policy. They have identified the following anomalies:

- Correlation anomaly: Two rules are correlated if the first rule matches some packets that match the second rule and the second rule matches some packets that match the first. The correlated rules introduce anomalies when they have different actions (i.e., if one swaps the order of the rules, the outcome of the policy changes).
- Shadow anomaly: A rule is shadowed when a previous rule matches all the packets that match this rule and both rules have different actions.
- Generalization anomaly: A rule is a generalization of another if the first rule can match all the packets that match the second rule that is more specific.
- Redundancy anomaly: A rule is redundant if it performs the same action on the same packets as another rule, such that if the redundant rule is removed, the security policy will not be affected.

The authors then developed a tool, "Firewall Policy Adviser" to detect anomalies and provide feedback to administrators.

There is only one tool, an editor, for authoring privacy policies: EPAL Policy Editor. The editor was developed by a team of students at North Carolina State University[1]. It provides an interface which facilitates the creation of EPAL privacy policies (vocabulary and rules). Hence, a policy auditor does not need to have prior knowledge of XML language for creating a privacy policy. The created EPAL policies can then be used in Tivoli software for enforcement. However, similar to POWER, the EPAL editor can only assist in creating a syntactically correct policy - it does not provide any method for ensuring the minimality of the policy nor does it provide feedback about the possible interference of the rules in the policy.

2.5 Privacy Policy Enforcement

The enforcement of a privacy policy is the ultimate goal of any privacy management system. There have also been some approaches addressing the policy enforcement using different techniques. Here we will briefly introduce some of these approaches.

Myersb and Liskov [48] proposed an approach for control of information flow in operating systems with mutual distrust. Their aim was to ensure that access to information be given to the applications that have the required level of credentials. Langheinrich [41] introduced a privacy awareness system for ubiquitous computing environments. This system uses a

[1]Privacy Authoring Editor is an open source project and available for download at http://sourceforge.net/projects/ epaleditor

privacy-aware database to combine the collected data and their privacy policies to ensure the data usage corresponds to the data usage policy. Although these two approaches are used to protect privacy, they suggest the protection mechanism to be built within the agents (devises). These approaches are different to access control type approaches where there is a central entity that makes sure policies are enforced, hence, the above mentioned approaches can not be used for enforcing privacy policies within enterprises.

Fischer-Hubner [30, 31] proposed a task-based privacy policy model for enforcing privacy policies. The model addresses two privacy principles: "purpose binding" and the principle of "necessity" by annotating the tasks with purposes for which they are being performed. Further, the model can express for each given task, what sub-tasks must be perfomred as well. However, this model does not consider the context of the tasks. The context-based access control takes into account the environmental conditions such as consent of the data owners or their age while processing authorization requests.

The other approache that is most relevant to our work is the use of tags or metadata to govern how the data associated with each group of data is used. The metadata is checked to decide whether an operation is allowed or not when an access request arrives. This approach is very flexible and well suited for handling privacy within and between enterprises because it enhances the traditional access control[2] by incorporating two additional variables into access requests: the "preferences of the data owner" and the "specific business purpose" for which the data will be used.

2.5.1 E-P3P Framework

Karjoth et. al. adapted the tag-based approach and introduced an abstract framework for privacy-enabled management and exchange of private date, called the Platform for Enterprise Privacy Practices (E-P3P) [10, 50]. The E-P3P framework uses the EP3P language for expressing access restrictions [39].

Figure 2.8 shows main components of the E-P3P framework [3], in the E-P3P framework all requests for access to a protected resource go through an abstract component called a Policy Enforcement Point (PEP). The PEP formulates a request for an authorization decision that includes a description of the request: who is making the request, what resource is being requested, what operation or action is to be performed on the resource and what is the purpose(s) for the access request.

[2]Traditionally, access control has dealt only with authorization decisions on user's access to target resources.

[3]E-P3P is based on the abstract policy enforcement model defined by Internet Engineering Task Force (IETF) and International Organization for Standardization.

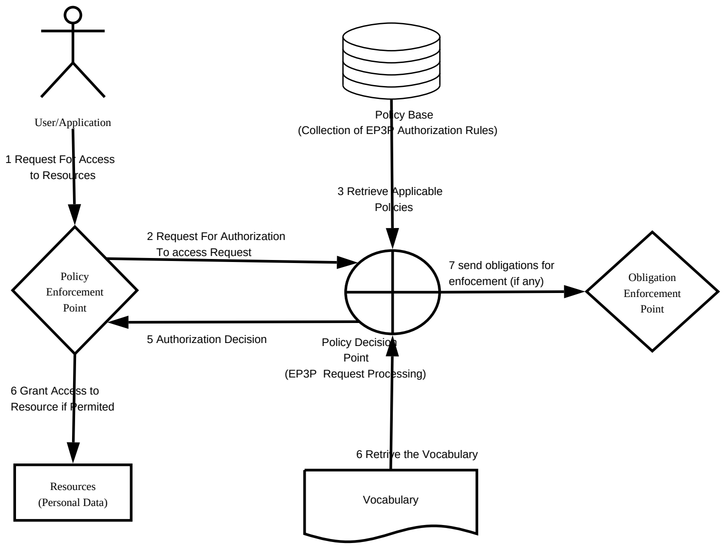

User/Application
1 Request For Access to Resources
Policy Base (Collection of EP3P Authorization Rules)
3 Retrieve Applicable Policies
2 Request For Authorization To access Request
7 send obligations for enfocement (if any)
Policy Enforcement Point
Obligation Enforcement Point
5 Authorization Decision
Policy Decision Point (EP3P Request Processing)
6 Grant Access to Resource if Permited
6 Retrive the Vocabulary
Resources (Personal Data)
Vocabulary

Figure 2.8: EP3P Enforcement Model

The authorization request is sent to another abstract component called a Policy Decision Point (PDP). The PDP retrieves the policies applicable to the request, along with any additional information required to evaluate those policies, it evaluates the policies with the information available and returns an authorization decision for PEP. In E-P3P, the authorization can be *permit, deny*, or *don't care*. The later implies that the PDP is unable to provide an authorization decision because it has no rule is applicable to the request.

Given the authorization decision, the PEP either grants the requested access to the resource or denies access. An authorization decision may be accompanied by obligations, which are sent to the Obligation Enforcement Point (OEP) that is responsible for enforcing obligations.

Note that the PDP is the evaluation engine that processes a set of policy rules . It has no control over the enforcement of the policy decision. The framework depends on having every request for protected resources go through a PEP, which is responsible for policy enforcement. The implementation of a PEP is usually application-specific, as it is usually built into the platform on which the application is built.

Any rule in the PDP must be stated based on a specific vocabulary. Vocabularies are defined by the particular applications or domains that use policies - they are domain-specific. The PDP does not need to understand the domain-specific meaning of each term used in a policy, it only needs to understand how to determine whether the values supplied for each term in an authorization decision request satisfy the conditions for access specified in the policy. It is the responsibility of each application or domain that will be using policies to specify a vocabulary that can be mapped to the entities in the domain such as data users or data types. The meaning of the values is not used by the PDP however and is significant only the application or domain. Thus, the format (syntax) of the vocabulary is domain independent, even though the meaning (semantics) of the terms in the vocabulary are domain specific.

Privacy policy languages specify the format of policy rules and how the rules must be evaluated.

2.5.2 Tivoli Privacy Manager

Based on the E-P3P model, IBM has introduced the application "Tivoli Privacy Manager for e-business". This software is a commercial version of E-P3P that uses EPAL as the privacy policy language. It is designed to help large organizations build privacy policies and privacy practices directly into their e-business applications and infrastructure for enforcement. Figure 2.9 illustrates the essential components of Tivoli software.

- Policy Editor: An interface for creating machine-readable privacy policies (P3P) from written privacy policies.
- Policy Deployment: Links privacy policies to personal information by creating data types and then connecting data types to users, groups and storage locations.
- Report Generator: Generates enterprise wide reports, showing policies deployed, enforcement locations and audit trails detailing personal information management according to privacy policies.

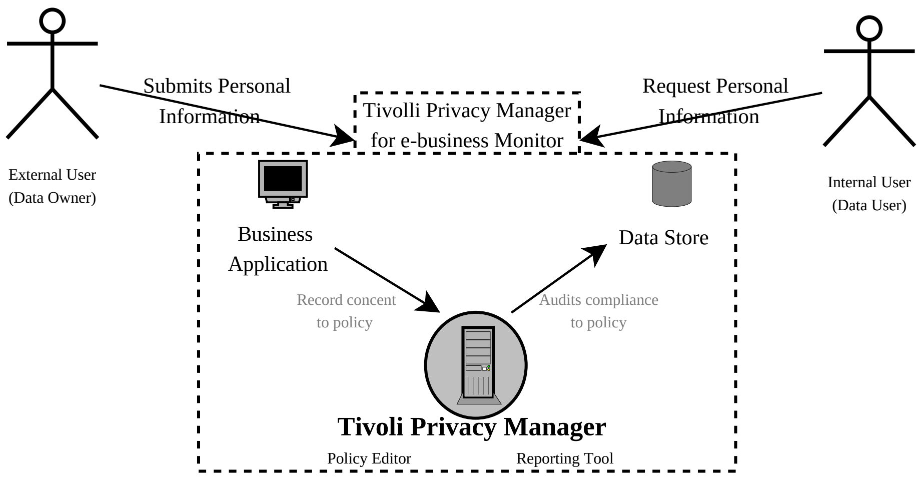

Submits Personal Information
Tivolli Privacy Manager for e-business Monitor
Request Personal Information
External User (Data Owner)
Internal User (Data User)
Business Application
Data Store
Record concent to policy
Audits compliance to policy
Tivoli Privacy Manager
Policy Editor
Reporting Tool

Figure 2.9: Tivoli: Privacy Enforcement

Thus far we have defined the concept of privacy, the significance of preserving privacy in enterprises, the languages used for formally representing privacy policies to ensure enforcement, the policy management tools and the mechanisms for enforcing policies. Our focus from this point on is on the EP3P language itself and the operations performed on it. In the next chapter we will formally define the EP3P language, analyze it and determine its shortcomings.

Chapter 3

EP3P Language & Policy Management Operations

In the previous chapter we informally described the EP3P language data model and the structure of an authorization rule. In this chapter, first we will formally define the syntax and semantics of the EP3P language. Second, we will specify desirable attributes that privacy policies languages would like to preserve. Finally, we will introduce some of the operations on privacy polices.

3.1 EP3P Language

In this section we formally define the syntax and semantics of the EP3P privacy policy language [9].

3.1.1 EP3P Syntax

The EP3P language is a triple consisting of a *vocabulary*, a set of *authorization rules*, and a *default ruling*. The vocabulary defines actions, obligations and conditions as well as hierarchies of users, data and purposes.

Formally, a hierarchy is a pair $(H, \geq_H)$ consisting of a finite set H and a transitive, non-reflexive relation $\geq_H \subseteq H \times H$, where every $h \in H$ has at most one predecessor (parent). We write $a \geq_H b$ (i.e., read a is more general than b) iff a is a parent of b.

Definition 1 The EP3P vocabulary is a six tuple $\mathcal{V} = (UH, DH, PH, A, C, O)$ where $UH = (U, \geq_U)$, $DH = (D, \geq_D)$, $PH = (P, \geq_P)$ are hierarchies of user categories (U), data categories (D), purpose categories (P). A is a set of actions. Conditions C and obligations O are a set of propositions.

Definition 2 A *ruleset* $\mathcal{R}$ for a vocabulary $\mathcal{V}$ is a subset of $U \times D \times A \times P \times \mathbb{Z} \times \{+, -\} \times C \times O$.

Definition 3 An *authorization rule* is denoted as $\rho = (\langle u, a, d, p\rangle\langle c, i, r, o\rangle) \in \mathcal{R}$. It consists of a *Data user* $u \in U$, an *Action* $a \in A$, a *Data Category* $d \in D$, *Purpose* $p \in P$, a set of *Conditions* $c \subseteq C$, a *Priority* $i \in \mathbb{Z}$, a *Ruling* $r \in \{+, -\}$ and a set of *Obligations* $o \subseteq O$. The rules with $'+'$ ruling are referred to as an *authorize-rule* and those with $'-'$ ruling referred to as a *deny-rule*.

The intuition for assigning a priority to an authorization rule is to quantify the importance of one rule in the face of others. As we will show in the following chapters, sometimes a policy author would like to express that in a specific circumstance one authorization rule is more appropriate than another.

Definition 4 The condition in the EP3P language is a predicate of the form $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ where c_i is a proposition belonging to C or the negation of such a proposition. The condition of an authorization rule is said to be *satisfied* if $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ evaluates to true.

The intuitive reading of an EP3P authorization rule $(\langle u, a, d, p\rangle\langle c, \mathit{1}, +, o\rangle)$ is: "if the condition c is *satisfied* the user u can execute action a on data d for purpose p; obligation o must be *fulfilled*". Similarly, the tuple $(\langle u, a, d, p\rangle\langle c, \mathit{1}, -, o\rangle)$ states: "if the condition c is satisfied, the user u can not execute action a on data d for purpose p; obligation o must be fulfilled".

The following example shows how a *policy auditor*, the one who is responsible for managing privacy policies within an enterprise, may transform an informal privacy policy based on the syntax of the EP3P language.

***Example* 1** Let us assume the informal privacy policy of an arbitrary hospital consists of the following policy statements:

- We do not allow anyone other than your primary doctor to access your medical record for any purpose.
- Under some emergency conditions we will allow others in the medical group to read your medical information, unless you specify otherwise. However, you will be informed about its access.

Further, assume we have a vocabulary consisting of the user (Figure 2.3), data (Figure 2.4) and purpose (Figure 2.5) hierarchies and sets of actions = $\{\mathit{read}, \mathit{write}, \mathit{execute}\}$, conditions = $\{\mathit{optin}, \mathit{optout}, \mathit{emergency}\}$ and obligations= $\{notify, delete\}$.

Given the informal policy and the above vocabulary, the policy auditor may write the following formal rules to represent the above policies[1]:

$(\langle \mathit{users}, \mathit{read}, \mathit{medical}, \mathit{any}\rangle\langle\emptyset, \mathit{1}, -, \emptyset\rangle)$**:** This authorization rule states that *users* must not *read* the data type *medical* for *any* purposes and no conditions need to be considered (i.e., this rule applies to all situations without any restrictions). Similarly, there are no obligations to be fulfilled in case the rule fires for a request.

[1]Note that the relation between informal policy and formal policy is not one to one as the policy auditor may write multiple formal rules that all represent an informal privacy policy.

($\langle$*doctors*, *read*, *medical*, *diagnosis*$\rangle\langle$*optin*, $3, +, \emptyset\rangle$): This authorization rule states that *doctors* can execute action *read* on the data type *medical* for *treatment* purposes, if the condition *optin* (i.e., the doctor is a primary doctor and has been given a consent to read the data) holds; There is no obligation to be fulfilled.

($\langle$*nurses*, *read*, *medical*, *diagnosis*$\rangle\langle\neg$*optout*, *emergency*, $2, +,$ *notify*$\rangle$): The authorization rule states that *nurses* can *read* the data type *medical* for *diagnosis* purposes, if the condition $\neg$*optout*[2] (i.e., The data owner has not explicitly denied access to the data); In the case of performing such an action the hospital must inform the user about the data usage.

3.1.2 EP3P Operational Semantics

The central point for interpretation of rules in the EP3P language is the operational semantics [9]. It allows a unique authorization decision to be made for each access request.

Definition 5 An access request in the EP3P language is a four tuple: $(u, a, d, p) \in U \times A \times D \times P$ with one element of each type.

Access requests in EP3P language are not restricted to "ground terms" as in some other languages, (i.e., minimal elements in the hierarchies). This is useful if one starts with a coarse policy and refines it, so the elements that are initially leaf nodes in the hierarchy may later bare children. For instance, the individual users in a "department" of an "enterprise" may not be mentioned in the enterprise's global privacy policy, but in the department's privacy policy.

Processing a request involves two stages: *pre-processing* and *request-processing*. Pre-processing takes place once for a policy unless the vocabulary or rules change, while request-processing is repeated for every request. The result of the pre-processing stage is a set of authorizations rules that we denote as *policy-base* ($\mathcal{PB}$).

A policy-base consists of explicit rules (i.e., written by the policy auditor in $\mathcal{R}$) and the rules that are derived from them. The rules are propagated through the three hierarchies of user, data and purposes, by taking into account the propagation strategy defined bellow.

Definition 6 A *policy-base* $\mathcal{PB}$ is a set of rules in an EP3P language, where the rules are constructed using the following three steps:

- For every $\rho = (\langle u, a, d, p\rangle\langle c, i, r, o\rangle) \in \mathcal{R}$, a tuple $(\langle u, a, d, p\rangle\langle c, i, r, o\rangle)$ is added to $\mathcal{PB}$.
- For every $(\langle u, a, d, p\rangle\langle c, i, r, o\rangle) \in \mathcal{PB}$, if there exists (u', d', p') s.t. $(u \geq_U u') \wedge (d \geq_D d') \wedge (p \geq_P p')$, a tuple $(\langle u', a, d', p'\rangle\langle c, i, r, o\rangle)$ is added to $\mathcal{PB}$.

[2]$\neg$ is a classical negation.

- For every $(\langle u, a, d, p\rangle\langle c, i, r, o\rangle) \in \mathcal{PB}$ where $r = -$, if there exists a (u', d', p') s.t. $(u' \geq_U u) \wedge (d' \geq_D d) \wedge (p' \geq_P p)$, a tuple $(\langle u', a, d', p'\rangle\langle c, i, -, o\rangle)$ is added to $\mathcal{PB}$.

A crucial point in the pre-processing stage is the fact that "deny" rules are inherited both downwards and upwards along the four hierarchies, while "allow" rules are inherited downwards only. The reason is that the hierarchies are considered groupings: if access is forbidden for some element of a group, it is also forbidden for the group as a whole.

Given the rules in the first example, after the pre-processing stage the following rules will exist in the policy-base. For clarity purposes we use letters to represent the explicit rules; letters and numbers to represent the derived rules.

A	$(\langle \mathit{users}, \mathit{read}, \mathit{medical}, \mathit{any}\rangle\langle\emptyset, 1, -, \emptyset\rangle)$
A1	$(\langle \mathit{users}, \mathit{read}, \mathit{history}, \mathit{any}\rangle\langle\emptyset, 1, -, \emptyset\rangle)$
A2	$(\langle \mathit{users}, \mathit{read}, \mathit{disease}, \mathit{any}\rangle\langle\emptyset, 1, -, \emptyset\rangle)$
A4	$(\langle \mathit{users}, \mathit{read}, \mathit{addiction}, \mathit{any}\rangle\langle\emptyset, 1, -, \emptyset\rangle)$
A5	$(\langle \mathit{users}, \mathit{read}, \mathit{patient\ Record}, \mathit{any}\rangle\langle\emptyset, 1, -, \emptyset\rangle)$
A6	$(\langle \mathit{users}, \mathit{read}, \mathit{data}, \mathit{any}\rangle\langle\emptyset, 1, -, \emptyset\rangle)$
A7	$(\langle \mathit{our\ staff}, \mathit{read}, \mathit{medical}, \mathit{any}\rangle\langle\emptyset, 1, -, \emptyset\rangle)$
.	.
A432	$(\langle \mathit{government\ agencies}, \mathit{read}, \mathit{data}, \mathit{service\ update}\rangle\langle\emptyset, 1, -, \emptyset\rangle)$
B	$(\langle \mathit{doctors}, \mathit{read}, \mathit{medical}, \mathit{diagnosis}\rangle\langle \mathit{optin}, 3, +, \emptyset\rangle)$
B1	$(\langle \mathit{doctors}, \mathit{read}, \mathit{history}, \mathit{diagnosis}\rangle\langle \mathit{optin}, 3, +, \emptyset\rangle)$
B2	$(\langle \mathit{doctors}, \mathit{read}, \mathit{diseases}, \mathit{diagnosis}\rangle\langle \mathit{optin}, 3, +, \emptyset\rangle)$
B3	$(\langle \mathit{doctors}, \mathit{read}, \mathit{addictions}, \mathit{diagnosis}\rangle\langle \mathit{optin}, 3, +, \emptyset\rangle)$
C	$(\langle \mathit{nurses}, \mathit{read}, \mathit{medical}, \mathit{diagnosis}\rangle\langle \neg \mathit{optout}, \mathit{emergency}, 2, +, \mathit{notify}\rangle)$
C1	$(\langle \mathit{nurses}, \mathit{read}, \mathit{history}, \mathit{diagnosis}\rangle\langle \neg \mathit{optout}, \mathit{emergency}, 2, +, \mathit{notify}\rangle)$
C2	$(\langle \mathit{nurses}, \mathit{read}, \mathit{diagnosis}, \mathit{diagnosis}\rangle\langle \neg \mathit{optout}, \mathit{emergency}, 2, +, \mathit{notify}\rangle)$
C3	$(\langle \mathit{nurses}, \mathit{read}, \mathit{addiction}, \mathit{diagnosis}\rangle\langle \neg \mathit{optout}, \mathit{emergency}, 2, +, \mathit{notify}\rangle)$

The pre-processing phase is an approach to computing the inheritance effects at compile-time rather than run-time. In effect, the pre-processing phase computes a deductive closure of $\mathbb{R}$, under the upward and downward inheritance rules.

We shall describe below a procedure that EP3P language [9] follows for handling access requests which involves a sequence of steps to obtain a final ruling r and obligation o for the request.

Definition 7 Given an authorization rule, the pair $\langle r, o\rangle$ denots the *authorization decision.*

Definition 8 Given a $\mathcal{PB}$, an arbitrary request $(\hat{u}, \hat{a}, \hat{d}, \hat{p})$ where $\hat{u}$ = '*doctor*', $\hat{a}$ = '*read*', $\hat{d}$ = '*medical history*' and $\hat{p}$ = '*treatment*' is processed as follows:

- Find all the rules where $(\hat{u}, \hat{a}, \hat{d}, \hat{p}) \equiv (u, a, d, p)$

- Remove all the rules $(\langle u, a, d, p\rangle\langle c, i, r, o\rangle) \in \mathcal{PB}$ where c contains an unsatisfied condition. The rules with satisfied conditions are referred to as *applicable rules*[3].
- If there exist a $(\langle u, a, d, p\rangle\langle c, i, -, o\rangle) \in \mathcal{PB}$ all the rules with $(\langle u, a, d, p\rangle\langle c, i, +, o\rangle)$ $\in\mathcal{PB}$ are removed from $\mathcal{PB}$ regardless of their obligation element.
- For priority i, select all the rules with $(\langle u, a, d, p\rangle\langle c, i, r, o\rangle) \in \mathcal{PB}$. If such a rule exists, the pair $\langle r, RO\rangle$ is returned where $RO = \{o|\ \langle(u, a, d, p),\ (i, r, c, o)\rangle \in \mathcal{PB}\}$. Otherwise, the rule(s) with the next highest priority are processed. Note that RO is a set of obligation sets for the rule(s) with the priority i.
- If there are no rules with the priority i, the default ruling with the empty obligation $\{(dr, \emptyset)\}$ is returned.

Note that for our purposes we say that a rule is *fired* when the authorization decision of the rule is returned in the request-processing step. In addition, the authorization decision of a firing rule is referred to an *authorization response*.

The previous table illustrates the policy-base of the arbitrary hospital that was introduced in example 1. Now let us assume that a patient's condition is classed as *emergency* and a nurse needs to read their medical history to apply the correct treatment. Further, we assume that the patient has not explicitly denied medical record access to the nurses. Hence, when a nurse sends the following access request: $\langle nurse, read, history, diagnosis\rangle$ to a policy enforcement system, there will be two rules that are applicable for this request:

1) $(\langle \mathit{nurses}, \mathit{read}, \mathit{history}, \mathit{diagnosis}\rangle\langle\emptyset, \mathit{1}, -, \emptyset\rangle)$
2) $(\langle \mathit{nurses}, \mathit{read}, \mathit{history}, \mathit{diagnosis}\rangle\langle\neg \mathit{opt\ out}, \mathit{emergency}, \mathit{2}, +, \mathit{notify}\rangle)$

Since the conditions of both rules are satisfied and the rules have different priorities, none of them needs to be removed from the policy-base. However, considering that rule 2 has a higher priority than rule 1, the authorization decision, $(+, notify)$ will be returned.

Note that we used the above examples to simplify understanding of the syntax and semantics of the EP3P language. Consequently, we have deliberately skipped many of the details regarding the creation of formal privacy rules, such as:

- How to construct a formal vocabulary for an EP3P policy.
- How to ensure the collection of EP3P rules actually mirrors the informal privacy policy.
- How to decide on the priority of the rules.

[3]We assume there exists machinery that evaluates the condition of the rules in policy-base and returns true (satisfied) or false (unsatisfied).

- How to decide whether to write an abstract rule (i.e. the rules written for the elements at the top of the hierarchies rather than leaf nodes) or more specific rules.

There are many complex issues involved in answering such questions, specifically when dealing with a policy for which the number of rules is large.

3.2 Desirable Attributes

In this section we will describe three desirable attributes, *consistency, safety* and *minimality* of a policy [8, 12, 16, 42]. Now that we have described the EP3P language, we are in a position to evaluate whether EP3P preserves these attributes.

3.2.1 Consistency

Support for the consistency of a policy is by far the most studied of their attributes [52, 32, 42, 1, 2, 43, 14]. The definition of consistency appropriate for a specific policy language is directly dependent on the language itself. However, the general notion of consistency implies that a policy is *consistent* if it does not provide conflicting directives [43, 42]. With respect to EP3P language, an EP3P policy must not consist of the following [16]:

- Authorization responses that contradict one another. For example, a policy that allows doctors to read medical records, yet simultaneously restricts hospital employees from accessing these records.
- Incompatible obligations. For example, a policy that returns two obligations, *delete patient-record within one month* and *retain patient-record for one year.*

Maintaining Consistency

Maintaining the consistency of a policy is one of the core issues in policy management. Here we will describe the approaches taken in the literature for dealing with inconsistencies in a policy. Also, we will explain how the EP3P language provides support for the two notions of inconsistency described above.

There are two types of techniques for detecting and resolving inconsistencies as described below:

- *Specification-time*: Some inconsistencies may be due to specification errors made by the policy auditor while constructing the rules. Hence, it is critical to deal with these inconsistencies before they are introduced into the system.

  Bai et. al. [14], have examined several situations in which inconsistent rules could be detected at specification time (while updating a security policy) and suggested techniques for the automatic resolution of these inconsistencies.

Cuppens and Saurel [27] have proposed the use of a deontic logic[4] to model the concept of permission, obligation and prohibition with a modal logic of action for formalizing and detecting inconsistent rules in security policies.

Angos et. al. [32] have proposed the use of decision tables to detect inconsistencies between the rules as well as proposing algorithms for incremental validation of a policy. However, to date, there is no mechanism to detect inconsistent rules in privacy policies.

- *Run-time*: The run-time inconsistency resolution mechanism for policies can be represented as a mapping function that takes a set of authorization rules, removes inconsistent rules and returns as output a set of authorization rules that is a subset of the original set [35, 37]. There are several approaches for resolving inconsistencies at run-time. Some of these are as follows:
  - **No Conflict:** It assumes no inconsistency can occur. The presence of inconsistent rules is considered an error.
  - **Denial Takes Precedence:** In this case, negative authorizations are always adopted when an inconsistency occurs. In other words, the principle says that if we have one reason to authorize an access, and another to deny it, then we deny it.
  - **Permissions Takes Precedence:** Positive authorizations are always adopted when an inconsistency occurs.
  - **Nothing Takes Precedence:** This principle says that we neither authorize nor deny an access when there is an inconsistency. Instead, the decision is made to deferring to the default ruling.
  - **Most Specific Takes Precedence:** The explicit rules from which derived rules originated are determined and the authorization decision of the closest explicit rule is considered the authorization response.

The decision about which approach is suitable for a language is dependent on the domain in which the language is being used. For example, Bertino et. al. in [17] used the Most Specific Takes Precedence approach for dealing with inconsistency in their languages.

The EP3P language does not treat the inconsistencies that are due to the incompatible obligations. However, it provides a run-time inconsistency resolution that enforces consistency by three means:

1. Deny rules always take precedence. Hence, for any authorize rule in the policy, if there exists an overlapping deny rule, the authorize rule is removed.

[4]Deontic logic is the field of logic that is concerned with obligation, permission, and related concepts. Alternatively, a deontic logic is a formal system that attempts to capture the essential logical features of these concepts.

2. The rules are processed according to their priorities.
3. The rule evaluation is terminated whenever the first applicable rule is found. It disallows further processing of the rules, thus avoiding ambiguity.

The drawback of addressing inconsistencies at run-time but neglecting them at specification-time is that inconsistent rules may continue to persist within the policy until run-time which adds to the complexity of run-time inconsistency resolution. Furthermore, there are several specification-time operations that are usually performed on policies, such as merge and update (to be elaborated on later), which if applied to such policies may result in unpredictable outcomes.

3.2.2 Safety

The second desirable attribute of a policy is *safety* [16]. An EP3P policy is *safe* if the authorizations granted to a data type are always more restrictive than those given to subclasses of that data type.

For example, giving an authorization to access a medical record is tantamount to granting access to a patient's allergy history because the medical record contains this history.

Imposing fewer obligations on accessing medical records than on allergy history is regarded as unsafe because a hospital employee can undermine the intent of the policy by accessing the entire medical record instead of only the allergy history. Hence, the obligations of the policy are compromised. The following two rules represent a safe EP3P policy.

1) $(\langle \textit{nurses}, \textit{read}, \textit{medical record}, \textit{diagnosis}\rangle\langle\emptyset, \textit{1}, +, \textit{notify}, \textit{delete record}\rangle)$
2) $(\langle \textit{medical team}, \textit{read}, \textit{medical record}, \textit{diagnosis}\rangle\langle\emptyset, \textit{2}, +, \emptyset\rangle)$

However, interchanging the priority of these rules results in an unsafe policy, because in doing so, nurses could avoid being required to delete the record by querying the authorization service as a generic medical team.

Unsafe policies are problematic because they allow obligations to be avoided, undermining the policy. However, EP3P does not address safety issues and currently there is no mechanism to check the safety of an EP3P policy [16].

3.2.3 Minimality

The third desirable attribute for policies, *minimality* is the focus of our work. A policy is minimal if every rule within the policy can affect an authorization response. For example, the hospital policy shown below consists of two rules, both of which specify that "doctors can read patient's medical history for the purpose of treatment". However, the first rule has a higher priority than the second. Hence, the second rule does not affect the authorization response of the policy.

1) $(\langle \textit{doctors}, \textit{read}, \textit{medical history}, \textit{treatment}\rangle\langle\emptyset, \mathit{2}, +, \emptyset\rangle)$
2) $(\langle \textit{doctors}, \textit{read}, \textit{medical history}, \textit{treatment}\rangle\langle\emptyset, \mathit{1}, +, \emptyset\rangle)$

The EP3P language does not preserve the minimality attribute. Hence, the policy auditor is allowed to write rules which will never be used in the authorization process. Since the ineffective rules do not affect the authorization response, one could argue against the importance of detecting and removing these rules. There are two main reasons why their detection remains important:

- The rules that do not contribute to the authorization response, still add to the size of the policy and increase the time required to process the rules.
- These rules may flag unexpected runtime behaviors that could have been prevented had they been detected and dealt with at the early policy specification stage.

A valid question would be; why would ineffective rules exist in the policy in the first place? As will be explained in the following sections, such rules may exist in the policy due to errors made by the policy auditor while creating the EP3P privacy policy. These errors are a consequence of the large number of rules and the associated complexity in predicting the interaction of these co-existing rules.

3.3 Operations on Privacy Policies

So far, internal attributes of policies such as consistency, safety and minimality have been examined. In this section we will explain some operations that must be performed on a policy, such as *policy formalization.* In addition, we will explain three operations that can be performed *between* policies in a multiple policy enterprise environment. These include *refinement, composition and alignment of privacy promises with privacy practices.* We will describe these four operations, focusing firstly on inter-policy operations.

3.3.1 Policy Refinement

Given the multiple business and legal requirements placed on enterprise privacy policies, they should be maintained, authored, and audited in a decentralized way. Backes et. al. in [13, 46, 12] introduced *refinement* as a solution for managing enterprise privacy policies. Intuitively, for each access request, we say a policy P_2 *refines* another policy P_1 if the authorization responses of both policies are equal or the authorization response of P_1 is "don't care", which implies that there is no rule that can explicitly deny or authorize the access. However, this is a gross simplification of the definition of refinement and for further information it would be best to consult.

The comparison between policies, as to whether one refines another, is used in many situations when the interaction of policies is of importance. For example, it allows us to verify whether a policy within a subgroup of an enterprise refines the privacy policy of the whole enterprise. In addition, it enables us to determine when the data was transfered from the realm of one policy to another and whether the policy of the receiving realm matches the policy of the transmitter's realm.

Although the concept of refinement is simple, implementing an algorithm that actually evaluates the refinement of EP3P policies is non-trivial for two reasons:

1. Priorities and obligations, as well as conditions on context information have to be taken into account.

2. The rule by rule comparison between policies is computationally expensive.

A simplistic solution is to devise a brute force search. But Backes et. al. [12] observed that EP3P rules usually overlap for a large number of elements. As a result several rules are typically useless for a particular request and assignment because they are always hidden by matching rules that have higher priority. Hence, they propose a "scope-based comparison" in which they restructure the rules before comparisons takes place. An important prerequisite for a scope-based search is the notion of "extended rules". Extended rules, instead of having one qualifier (i.e., given a rule $(\langle u, a, d, p\rangle\langle c, r, o\rangle)$, the qualifier is (c, r, o)) they can have multiple qualifiers in each rule. For example, a rule $\langle(u, d, p, a), (r_1, c_1, o_1)\rangle$ followed by another rule $\langle(u, d, p, a), (r_2, c_2, o_2)\rangle$ can be described by the extended rule $\langle(u, d, p, a), \langle(r_1, c_1, o_1); (r_2, c_2, o_2)\rangle\rangle$, where the evaluation of qualifiers is from left to right and thus respects the precedence of the original rules. By using this method the number of rules that need to be evaluated are significantly reduced.

Backes et. al., mentioned that there may exist many rules in the policies that do not need to be evaluated because they do not have any effect on the authorization response of their policy. However, they did not provide any method for detecting or eliminating these rules from policies in order to increase the efficiency. Hence, their algorithm does not take into account the minimality property. Moreover, they do not verify policies for the safety property.

3.3.2 Policy Composition

Current approaches to policy management suffer from an unrealistic view that privacy policies are considered standalone. In reality, enterprises may have several policies that need to be enforced at once. For example, several departments in an enterprise that exchange private information need to combine their privacy policies to have a common view of their privacy practices [10, 12]. This situation calls for techniques that allow different privacy policies to be integrated while they retain their independence. One of the main challenges in providing techniques for integrating policies is to ensure that the result of combining two policies is a syntactically correct privacy policy [46].

Backes et. al. [46] introduced algebra for combining EP3P policies. They define the composition of EP3P policies as “constructively combining two or more policies such that the resulting policy refines them all”. The difference between their work and previous works [19, 57] on policy composition is the existence of obligations, default rules and sophisticated treatment of deny-rules in EP3P language. In addition, an analysis of EP3P [46, 16] shows that the EP3P language is not closed under operations such as *conjunction* and *disjunction*, hence it is non-trivial to represent the resulting policy as syntactically correct. Backes et. al. [46] restrict the scope of their work to correct policies. They introduce two operators *conjunction* and *disjunction*, which serve as building blocks for constructing a new (more/less restrictive) policy. For instance, an enterprise might take all applicable regulations and combine them into a minimum policy by the means of conjunction. Another important use of this is to ensure that promises made to customers (in P3P), are always respected. This can be achieved by having the P3P policy as one of the inputs.

Similar to Backes et. al., Barth et. al. [16] have also investigated the expressiveness of privacy policy languages and mentioned that EPAL is not closed under combination. They believe that the reason for this is the mid-code termination. Hence, they introduced a new language, *Declarative Privacy Authorization Language* (*DPAL*). Barth et. al. claim that their language enforces each of their rules in the policy, enabling combination. However, *DPAL* can not avoid inconsistencies because the language evaluates all the rules instead of returning the first matching rule. It is assumed that there is an algorithm (pre-processor) that can detect and flag inconsistencies in DPAL policies. However, the details of the algorithm were not given, nor is there any information about the effectiveness or efficiency of the algorithm. In addition, there is no implementation yet reported for DPAL. Furthermore, DPAL policies do not adhere to the minimality property. So, there may be some rules in a *DPAL* policy which could be removed with the outcome of the policy not changing. In addition, although the authors appreciate the importance of the safety property for privacy policies, they did not provide any mechanism for preserving the safety of *DPAL* policies.

3.3.3 Alignment of Privacy Promises with Privacy Practices

One of the challenges in managing privacy is to ensure that the promises made to customers are actually enforced within the enterprise. Hence, it is important to be able to examine whether P3P policies correspond to EP3P policies used in an enterprise [38, 15, 58]. However, this is a challenging issue because EP3P policies aim to control data access within the enterprise. Also, they are very fine-grained and can define access rights down to individual employees. So, frequent changes are commonplace [38, 15, 58]. As a consequence, it is a challenging task to keep the promises current with the actual practices.

Since currently there is no standard algorithm for synchronizing both privacy promises and privacy practices, they are synchronized manually in an ad hoc fashion, however, overtime as changes are made to privacy practices, they begin to desynchronize [38, 15].

To enable up to date privacy promises, Karjoth et. al. provide a mechanism for translating EP3P policies to P3P, by providing a procedure to map the vocabulary of EP3P to P3P. However, to derive P3P statements from EP3P rules they assume that the EP3P policy is *fine-grained*, by which they mean "it has no deny rule and the default rule is to deny access". The reason for such a restriction is to ensure that the rules that are in the policy are not overwritten by deny rules while request-processing. Unfortunately, they do not explain how to deal with the priorities of the rules, because even if we assume that the policy only consists of the authorize rule, rules with higher priority may prevent lower priority rules from firing.

Similar to Karjoth et. al., Barth et. al, have commented on the importance of the transformation of privacy practices to privacy promises. They use modal logic to reason about permission inheritance across data hierarchies to connect P3P with DPAL. The approach Barth et. al. take is significantly different from Karjoth et. al., as they provide a formal uniform model for translating P3P and DPAL. With such a model, they can then check if one policy refines another. In addition, they provide a method to extract P3P privacy promises from DPAL policies.

In contrast to the views mentioned above, Yu et. al., [58] believe that because privacy practices have numerous *minor* details which impact little upon privacy premises, frequent changes in these details, which are commonplace, do not need to be accounted for. They state:

> "It is undesirable to change an enterprises promises to customers every time an internal access control rule changes and yet it is important to propagate these changes."

However, later they follow on from this with:

> "Because P3P policies and EPAL policies may be authored and changed independently, these changes need to be compared to ensure consistency."

Despite their criticism, Yu et. al. did not propose an alternate method for performing synchronization nor have they explained under what conditions changes to privacy promises need to be made.

The common point of agreement for all works mentioned above is that for deriving privacy promises from practices one must not consider rules only individually but as a collective.

Further, all the authors implicitly state that there exist several rules in privacy policies that do not have any effect on the authorization response of the policy.

3.3.4 Formalizing Privacy Policy in EP3P

Formalizing a privacy policy arguably is the most important and one of the most complicated steps in constructing a privacy control system. In all the above mentioned operations (e.g., refinement, composition), the authors have bypassed this step by assuming that a formal privacy policy already exists.

In general, the process of constructing a formal privacy policy is defined by the following steps that can be repeated through the life-cycle of a privacy policy [28]:

1. The enterprise formalized its internal terminology as "EP3P vocabulary", which fixes the scope of the enterprise privacy practices.
2. The enterprise develops "EP3P Rules" that formalize the legal regulations and the business practices of the enterprise.
3. These rules are then used as the default policy for using data and enforcing privacy throughout the enterprise. This can be done using traditional access control, EP3P-aware business processes, or privacy-enabled access control systems [39].

Vocabulary is defined through a rigorous requirements analysis that identifies the data flow within the organization [20, 54]. Such an analysis determines the potential data, users and the purposes for which the data may be used. In addition, they specify the operations that may be performed on the data. Several techniques have been proposed in the literature to determine such access control requirements and ultimately extract the required vocabulary.

The process of writing authorization rules can further be refined into two steps performed by the policy auditor(s). An policy auditor:

1. Reads and interprets the current informal privacy policies.
2. Writes EP3P rules such that their collection reflects the existing informal policy.

Studies [7, 18, 40, 9, 25] show that most of the informal privacy policies lack the clarity necessary to have a unique interpretation. One of the causes of those clarity problems to the monolithic characteristics of privacy policies. For example when there is one large privacy policy document, or several large privacy documents, rather than many smaller, topic-specific policies [8, 18]. In addition, it has been observed by researchers like Bolchini et. al. [18] and Anton et. al, [8, 7] that natural language privacy policies are typically laden with inconsistencies, ambiguities and redundancy. These characteristics increase the likelihood that the policy statements they contain will be misinterpreted. As an example of one such ambiguity, consider a statement from CIGNA's health care Notice of Privacy Practices [8]:

> “CIGNA health care also discloses confidential information to accreditation organizations such as the National Committee for Quality Assurance (NCQA) when the NCQA auditors collect Health Plan Employer Data and Information Set (HEDIS) data for quality measurement purposes. When we enter into these types of arrangements, we obtain a written agreement to protect your confidential information.”

The above policy can be interpreted in two ways. One may interpret the policy as being written agreements between CIGNA and individual customers, while another may interpret this as a written agreement between CIGNA and NCQA auditors.

Researchers like Lamswardy and Milopolus in the field of “requirements engineering” have been using different techniques for extracting correct information from informal (requirements, security, privacy) specifications. Anton et. al. [8] employed a goal-driven technique that allows one to examine policy documents at a fine level of granularity, at the individual statement level.

After constructing a formal vocabulary for the privacy policy of an enterprise, the policy auditor shall write formal rules based on the extracted vocabulary. In the process of writing these rules, the policy auditor must ensure that these rules actually represent the informal policy. In this process, the major obstacle is the potential misinterpretation of the formal policy behavior in the face of access requests. This misinterpretation is due to the complex reasoning involved when determining the effect of the rules on each other, when they coexist in the policy. We have identified the following as potential causes for such complexity with regards to an EP3P privacy policy:

1. For any medium sized enterprise it is common to have tens or even hundreds of rules in a privacy policy.

2. A privacy policy may be constructed by more than one policy auditor, in other words each policy auditor may construct a part of the policy which addresses one section of the enterprise policy.

3. For every rule written by the policy auditor, there may be many other derived rules based on the three hierarchies . Therefore, potentially there are many rules in addition to those that are visible to the policy auditor. However, they need to be taken into account while determining the effect of the rules on each other.

4. Authorization rules have a complex structure, containing priorities and conditions and obligations that must be considered.

5. It is not sufficient to merely check the syntax of the rule in a policy.

To tackle the above obstacles, tools and techniques are emerging to assist policy auditors. In the following section we will review some of these policy editing tools for several policy languages. Our goal is two-fold, first to show the importance of these tools, second, to illustrate the lack of such tools for the EP3P privacy policy language.

In the next chapter we provide the theoretical framework that allow us to determine the minimality of EP3P privacy policies. The framework is further used to build a policy management tool that assists policy auditors to reason about the co-existence of the rules in the policy. Such a reasoning technique is then employed in performing operations such as update and composition of EP3P privacy policies.

Chapter 4

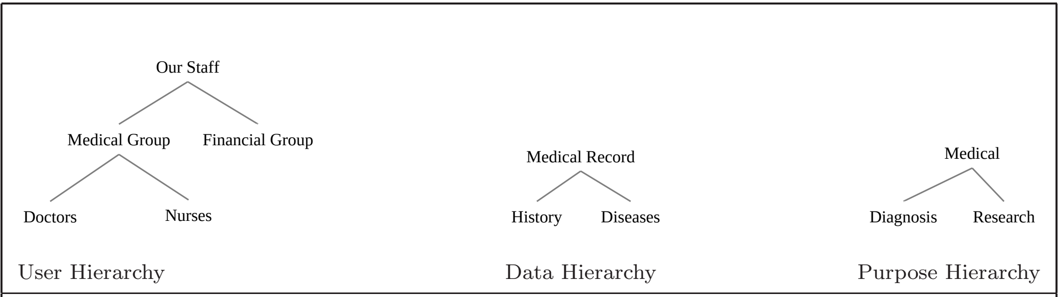
Redundancy Detection Framework

Our work takes the EP3P language as the starting point for writing formal privacy policies and provides a framework for detecting and resolving the redundant rules in an EP3P policy.

In this chapter by using a motivating example we will introduce our definition of redundancy with respect to the EP3P language. Further, we will present an abstraction for the situations in which a rule can become redundant in an EP3P policy and we will prove that, first, it is computationally efficient to detect these redundancies and second, we can detect all the redundant rules in an EP3P policy. Finally, we will provide an algorithm for detecting the classified redundant rules.

4.1 Illustrative Example

In this section we will introduce an example of a privacy policy of an arbitrary hospital, "Medi-Care Hospital". The MCH privacy policy is authored by more than one policy auditor and may change frequently to address the frequent changes to privacy legislations. The MCH privacy policy is based on a simplified vocabulary (Figure 4.1).



$actions = \{read, disclose\}$

$conditions = \{optin, emergency\}$

$obligations = \{notify, delete\ record\}$

Figure 4.1: Simplified Vocabulary

Let us assume that the current state of the MCH privacy policy contains three rules.

Similar to the previous chapter we represent these rules (i.e., in the ruleset) by a letter (e.g., A, B) and the rules derived from them by a combination of a letter and a number (e.g., A1, A2):

A $(\langle \textit{our staff}, \textit{read}, \textit{diseases}, \textit{diagnosis}\rangle\langle \textit{optin}, \textit{1}, +, \textit{notify}\rangle)$

B $(\langle \textit{doctors}, \textit{read}, \textit{diseases}, \textit{diagnosis}\rangle\langle \textit{optin}, \textit{5}, +, \textit{notify}\rangle)$

C $(\langle \textit{nurses}, \textit{disclose}, \textit{diseases}, \textit{research}\rangle\langle \emptyset, \textit{3}, +, \textit{notify}\rangle)$

Given the above rules, we can derive the following rules based on the vocabulary of the MCH policy and the semantics of EP3P language.

A1 $(\langle \textit{medical group}, \textit{read}, \textit{diseases}, \textit{diagnosis}\rangle\langle \textit{optin}, \textit{5}, +, \textit{notify}\rangle)$

A2 $(\langle \textit{financial group}, \textit{read}, \textit{diseases}, \textit{diagnosis}\rangle\langle \textit{optin}, \textit{5}, +, \textit{notify}\rangle)$

A3 $(\langle \textit{doctors}, \textit{read}, \textit{diseases}, \textit{diagnosis}\rangle\langle \textit{optin}, \textit{5}, +, \textit{notify}\rangle)$

A4 $(\langle \textit{nurses}, \textit{read}, \textit{diseases}, \textit{diagnosis}\rangle\langle \textit{optin}, \textit{5}, +, \textit{notify}\rangle)$

Hence, in total, the MCH policy contains six rules. Note that we have deliberately over-simplified the above example for the purpose of explaining the fundamental concepts in this chapter.

4.2 Definitions of Redundancy

Redundancy, in general terms, refers to the state of being *redundant*, that is: exceeding what is necessary, or containing an excess. It's used in different domains with different meanings. For example in information theory, redundancy is the number of bits used to transmit a message minus the number of bits of actual information in the message. In database theory, redundancy refers to duplicate records stored in multiple locations within a database. In languages, redundancy occurs when more than one statement is used to convey the same intent.

In all the definitions above, a redundant entity (eg. a record, a set of bits or a word) can be derived from another entity. For our purposes, we define redundancy as a state in which our policy has one or more redundant rules, with redundant rules defined as those that have no discernible impact on the current privacy policy.

Definition 9 A rule $\rho \in \mathcal{R}$ is *redundant* with respect to $\mathcal{R}$ iff the authorization decision returned for any access request is equal for $\mathcal{R}$ and $\mathcal{R} - \{\rho\}$.

This corresponds to the general notion of the term 'redundant'. However, the definition above shows that the redundancy, or otherwise, of a rule is directly related to the processing of privacy rules given the semantics of the EP3P language.

For example, given the policy of MCH, the rule "B" is redundant. Because by querying the policy with a request, $(doctors, read, diseases, diagnosis)$, the authorization response will be $(+, notify)$, regardless of whether the rule "B" exists in the policy or not. The reason is that there is a rule "A3" in the policy which is derived from rule "A". Since both rules are applicable for the access request (*doctors*, *read*, *diseases*, *diagnosis*), we can say rule "A3" is applicable in all situations where rule "B" is applicable. Hence, rule "B" can safely be removed from the policy and the *outcome* (i.e., authorization response) remains the same.

It can be observed from Definition 9 and the example above that only the rules within the ruleset ($\mathcal{R}$) can be redundant. Therefore we classified rule "B" as being redundant rather than rule "A3". There are two reasons for this:

- Classifying a rule in the ruleset as a redundant rule would mean that the rule written by a policy auditor does not add any value to the existing policy. On the other hand knowing a derived rule is redundant does not imply the rule from which it was derived is redundant too.
- Determining that a rule in the ruleset is redundant automatically implies that all the rules that are derived from this rule are also redundant.

However, Definition 9 ignores the dynamic nature of privacy policies. It is possible that a rule ρ_1 is redundant for $\mathcal{R}$, but is no longer redundant following the addition of some rule ρ_2. We wish to identify those rules that are redundant for the current policy and will remain redundant regardless of any rule additions.

Definition 10 A rule $\rho \in \mathcal{R}$ is *absolute-redundant* with respect to $\mathcal{R}$ iff ρ is redundant for any ruleset $\mathcal{R}' \supseteq \mathcal{R}$.

For example, let us assume that the policy auditor adds the following new rule to the MCH policy:

D (⟨*medical group*, *read*, *diseases*, *diagnosis*⟩⟨*optin*, *3*, $-$, $\emptyset$⟩)

Hence the following rules will also be derived and added to the policy:

D1 (⟨*doctors*, *read*, *diseases*, *diagnosis*⟩⟨*optin*, *3*, $-$, $\emptyset$⟩)

D2 (⟨*nurses*, *read*, *diseases*, *diagnosis*⟩⟨*optin*, *3*, $-$, $\emptyset$⟩)

Whilst the rule B is redundant with respect to the rule "A", it is not *absolute-redundant.* That is because by adding the rule "D" to the policy, the query $(doctors, read, diseases, diagnostic)$ will receive an authorization response $(+, notify)$ (i.e., from rule B), whilst without rule "B" in the ruleset the authorization response is $(-, \emptyset)$ (i.e., from rule D).

It is easy to see that absolute-redundancy implies redundancy; if a rule ρ is absolute-redundant in $\mathcal{R}$ then it is redundant for $\mathcal{R} \cup \{\}$. Definition 10 allows us to identify those rules which are redundant and will remain so for all future policy additions. This is a very important property which is specially necessary when building a mechanism for detecting redundant rules as they are added to a policy. Without Definition 10 the order in which the rules are added to the policy must be taken into account. This makes the process of constructing a privacy policy very complex. The following example will show a situation where the order by which rules are added to the policy needs to be taken into account:

For our example, assume the policy auditor of MCP would like to detect and remove redundant rules as they are added to the policy. Since there are three privacy rules, they can insert the rules in six different orders (e.g., D-B-A, B-D-A, D-A-B, etc.). If they do not have the concept of absolute-redundant then rule "B" will be labeled redundant for all the situations where "D" is inserted to the policy as the last rule (i.e., A-B-D, B-A-D)

There is another critical concept that the above notions of redundancy do not take into account: they can not determine what *causes* a rule to become redundant. It is not clear that a rule is made redundant by a single other rule, or by a collection of rules. This leads us to the third definition of redundancy that only compares a pair of rules.

Definition 11 Given $\rho, \rho' \in ruleset$, ρ is *pairwise-redundant* iff ρ is absolute-redundant w.r.t. $\{\rho, \rho'\}$

In previous examples we only considered two rules when determining redundancy. However, when a collection of rules co-exist in a policy they may make a rule redundant/ absolute-redundant. For example, let us assume the MCH privacy policy needs to address new legislation that allows patient's records to be revealed for the purpose of research, when there is an emergency situation. The policy auditor of MCH writes the following EP3P rules to address the new legislation:

E $(\langle \textit{medical group}, \textit{disclose}, \textit{diseases}, \textit{research}\rangle\langle \textit{emergency}, 5, +, \textit{notify}\rangle)$

F $(\langle \textit{medical group}, \textit{disclose}, \textit{diseases}, \textit{research}\rangle\langle \neg\textit{emergency}, 5, -, \emptyset\rangle)$

Each of the above rules in the ruleset can have two derived rules (based on the user hierarchy):

E1 $(\langle \textit{doctors}, \textit{disclose}, \textit{diseases}, \textit{research}\rangle\langle \textit{emergency}, 5, +, \textit{notify}\rangle)$

E2 $(\langle \textit{nurses}, \textit{disclose}, \textit{diseases}, \textit{research}\rangle\langle \textit{emergency}, 5, +, \textit{notify}\rangle)$

F1 $(\langle \textit{doctors}, \textit{disclose}, \textit{diseases}, \textit{research}\rangle\langle \neg\textit{emergency}, 5, -, \emptyset\rangle)$

F2 $(\langle \textit{nurses}, \textit{disclose}, \textit{diseases}, \textit{research}\rangle\langle \neg\textit{emergency}, 5, -, \emptyset\rangle)$

Recall that we have a rule "C" in the ruleset which has the same user, data, purpose and action as the rules "E2" and "F2". This means given an access request (*nurses*,*disclose*, *disease*, *research*), all the rules (C, E2, F2) are applicable. To see if there is any redundancy with respect to these rules we need to make sure that there are some situations in which the authorization decision of “C” is returned as an authorization response. Taking into account only two rules, “C” with “E2”, reveals that in circumstances where the condition *emergency* is not true, “C” is returned. Similarly, considering “C”with “F2”reveals that in circumstances where the condition *emergency* is false, the rule “C” is returned. However, since “E2” and “F2” are mutually exclusive and at any point in time at least one of the conditions are true, the coexistence of these three rules “C”, “E2” and “F2” in the policy make rule “C” redundant.

The assimilation of redundancy provide a means for determining the usability of the rules in a privacy policy. Hence, ultimately one can have a policy that only consists of the rules that add value (i.e., in terms of their contribution to the decision process) to the outcome (authorization response) of the policy (and future revisions of the policy). Such a policy is referred to as *minimal* policy.

Definition 12 An EP3P policy is *minimal* iff there exists no rule $\rho \in \mathcal{R}$ such that ρ is pairwise-redundant with respect to $\mathcal{R}$.

In the following sections we will firstly informally discuss the importance of detecting and removing redundancies. Further, we will describe specific instances of redundancy that can be efficiently tested by a computer. We will prove that the combination of these tests allows us to find all pairwise-redundant rules in a given policy. We will also describe the instances in which the pairwise definition of redundancy (and thus our efficient tests) differ from the absolute definition of redundancy.

4.3 Sources of Redundancy

In an EP3P policy, rules are prioritized by assigning an integer value. So, several rules may become redundant due to an improper priority assignments. The manual assignment of priorities to the rules can work when the number of rules is small. However, in privacy policies with a large number of rules, the priority assignment becomes very complex. If the assignment of priorities is not done with care, then the new rule may never fire.

In addition, redundancies can be introduced while merging privacy policies. Policies are usually merged to provide a uniform view of enterprise data practices. Merged privacy policies can also be considered as a policy that has been constructed by more than one policy auditor. In this case each policy auditor can see a segment of the global policy. Hence, it is impractical to assume that a policy auditor has a complete knowledge of the rules in that policy. In such a situation, there is always a likelihood that there are some rules that turn out to be redundant due to the existing general rules that were introduced by other policy auditors.

Another simplistic reason for the existence of redundant rules is to assume that a policy auditor deliberately introduces new rules with high priority to address the changes in legislation. Hence, new rules are added to make existing rules (i.e., old rules) redundant or in active. By taking this view, it is not important to identify/resolve redundant rules because such rules are made redundant deliberately. Therefore, they can safely stay in the policy. Obviously, for this view to be acceptable, one must also believe that the policy auditor is aware of redundancies that may occur at the time of introducing new rules. The assumption of such knowledge requires:

- the policy auditor to have a complete knowledge about the rules in the policy.
- the policy auditor to be able to determine the consequences of co-existence of several rules on the outcome (authorization response).

However, the belief that the above requirements can be satisfied is impractical because,

- The vocabulary of a privacy policy represents the internal structure of the organization which even for a medium sized enterprise can be very complicated.
- A privacy policy of a medium sized enterprise (e.g., hospital) may contain hundreds of rules.

Given the above incentives for the existence of redundant rules in a policy, it is essential to be able to identify and classify these rules. In other words, it is important to determine the consequences of co-existence of the rules on the outcome of the policy at the policy specification time.

4.4 Redundancy Classification

The Definitions 9, 10 and 11 provide us with the necessary tools for defining redundant rules in the context of EP3P policy. In this section, we will further classify redundancies and provide an answer to the following question:

> For $\rho' \in \mathcal{R}$, how can we determine if the rule $\rho \in \mathcal{R}$ or any rule derived from it makes ρ' redundant?

To answer the above question we need to take into account the relationships between the rules in a ruleset. Consider the following rules where the user, data and purpose hierarchies are as shown in Figure 4.2:

$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, +, 4, \{o_1, o_2\}\rangle)$ $\quad$ $\rho_3 = (\langle u_1, d_1, p_1, a\rangle\langle c, +, 2, \emptyset\rangle)$

$\rho_2 = (\langle u_3, d_1, p_2, a\rangle\langle c, -, 4, \emptyset\rangle)$ $\quad$ $\rho_4 = (\langle u_2, d_2, p_2, a\rangle\langle c, +, 1, \emptyset\rangle)$

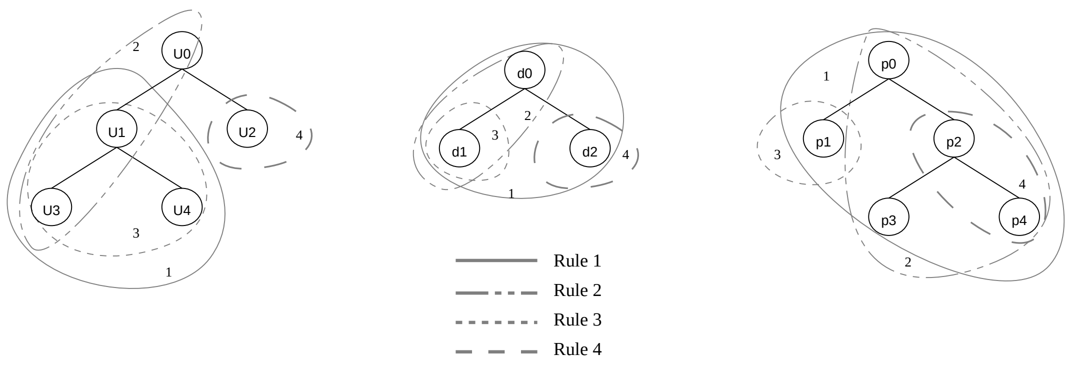


Figure 4.2: Interaction of rules with respect to user, data and purpose hierarchies

The circled areas on each hierarchy illustrate the derivation of rules with respect to upward and downward inheritance of rules in EP3P and to the position of the User, Data, Purpose elements, "udp" for short. For example, given a hierarchy (e.g., user), when one circle (group) subsumes another (e.g., ρ_1 and ρ_3) it means a rule (e.g., ρ_1) specifies an authorization for all the users that another rule (e.g, ρ_2) specifies an authorization for.

The following definitions allow us to identify pairs of rules that may introduce redundancy.

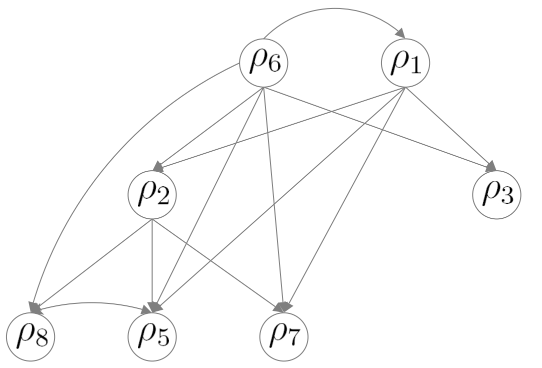
Definition 13 We write a rule $\rho \geq \rho'$ (read *covers*) iff: $u \geq_U u'$ and $d \geq_D d'$ and $p \geq_P p'$ and $a = a'$ and $c \subseteq c'$.

Definition 14 We write that $\rho > \rho'$ (read *strictly covers*) iff $\rho \geq \rho'$ and at least one of the following holds: $u >_U u'$ or $d >_D d'$ or $p >_P p'$ or $a = a'$ or $c \subset c'$.

Note that in the case where a more expressive language for conditions is used, the relation $c_1 \subseteq c_2$ can be casted to the logical entailment $c_2 \models c_1$.

By using Definition 13 and the rules shown in Figure 4.3, we observe that ρ_1 covers both ρ_2 and ρ_3. However, none of the other rules cover each other. ρ_1 does not cover ρ_4 because in the user hierarchy u_1 and u_2 are siblings. Hence, the group of users who are addressed by ρ_1 are mutually exclusive to those addressed by ρ_4. Similarly, ρ_2 does not cover rule three (or vice versa) because in the purpose hierarchy p_1 and p_2 are siblings.

Now we will construct a rule relationship model to show the *covering relationship* between the rules in a ruleset. Figure 4.3 depicts such a covering relation model where each node denotes a rule in the ruleset, each arrow in the model connects two rules, with the arrow going from the covering rule to the covered rule.



$\rho_6 = (\langle u_1, d_0, p_0, a\rangle\langle \emptyset, -, 5, \emptyset\rangle)$
$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 4, \{o_1, o_2\}\rangle)$

$\rho_3 = (\langle u_1, d_1, p_1, a\rangle\langle c, +, 2, \emptyset\rangle)$

$\rho_2 = (\langle u_3, d_1, p_2, a\rangle\langle c, -, 4, \emptyset\rangle)$
$\rho_7 = (\langle u_3, d_1, p_4, a\rangle\langle c, c_1, +, 4, \{o_1\}\rangle)$

$\rho_5 = (\langle u_3, d_1, p_3, a\rangle\langle c, -, 2, \emptyset\rangle)$
$\rho_8 = (\langle u_3, d_1, p_3, a\rangle\langle c, +, 3, \{o_1\}\rangle)$

$\rho_4 = (\langle u_2, d_2, p_2, a\rangle\langle c, +, 1, \emptyset\rangle)$

Figure 4.3: Covering Relation

Given our notion of covering, we are now in an position to classify the circumstances in which a rule in the policy makes another rule redundant. We choose these classifications because they are the obvious notions of redundancy and they identify all the pairwise-redundant rules.

4.4.1 Includes

First, we say one rule *includes* another rule if it already provides the same outcome (authorization decision) and fires in all the situations where the other rule fires. These rules add no value to the policy because even if we assume they fire, there will be no change in the outcome of the policy.

Definition 15 Given two rules ρ, ρ' we say, $\rho \succ_I \rho'$ (ρ includes ρ') when ρ covers ρ' and the priority of ρ is equal or higher than ρ' ($i \geq i', r = r'$ and $o \supseteq o'$)

In the rest of this section we will be using diagrams to describe some of the definitions that will be introduced. Diagrams implicitly assume there is a rule that the rules in the ruleset are checked against for redundancy and they show the rules that are redundant in a ruleset. Following are the notations used and their meanings.

$\mathcal{R}$: Ruleset

$P_=$: rules with equal priority.

$P_<$: rules with smaller priority.

$O_=$: rules with equal obligations.

$O_\subset$: rules with fewer obligations.

$r_=$: rules with equal ruling.

$r_{\neq}$: rules with opposite ruling.

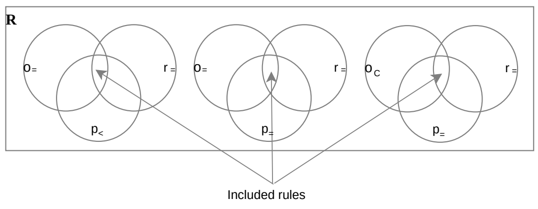


Figure 4.4: Redundancy Situation: Including

Included rules in the policy do not introduce unexpected (i.e., from policy auditor point of view) authorization, because the circumstances in which they could fire is already covered by a rule in the policy which has the same authorization response. For example, considering the rules in the Figure 4.3, we can say that both rules ρ_1, ρ_6 include ρ_5.

Covering Rule	Coverd Rule	Redundancy Type
$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 4, \{o_1, o_2\}\rangle)$	$\rho_5 = (\langle u_3, d_1, p_3, a\rangle\langle c, -, 2, \emptyset\rangle)$	Includes
$\rho_2 = (\langle u_3, d_1, p_2, a\rangle\langle c, -, 4, \emptyset\rangle)$	$\rho_5 = (\langle u_3, d_1, p_3, a\rangle\langle c, -, 2, \emptyset\rangle)$	Includes

4.4.2 Shadows

Second, we say a rule *shadows* another rule if that rule has strictly higher priority and fires in the same situations. The concept of 'shadowing' differs from inclusion in that the authorization decisions are different, but the difference in priority ensures the rule never fires [1].

Definition 16 Given two rule ρ, ρ', we can say $\rho \succ_S \rho'$ (ρ shadows ρ') when ρ covers ρ' and $(i > i')$.

[1]Note that the concept of shadowing is also used in firewall policies [4, 1, 32].

$\mathcal{R}$: Ruleset

$P_<$: rules with smaller priority.

$O_{\neq}$: rules with different obligations.

$r_{\neq}$: rules with opposite ruling.

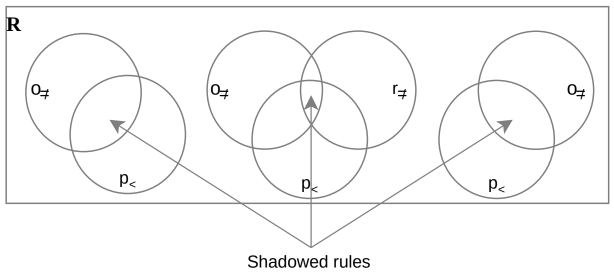


Figure 4.5: Redundancy Situation: Shadowing

Let us revisit the rules in Figure 4.2. By taking into account our definition of shadowing we can see that the following rules can be identified as being redundant (shadowed).

Covering Rule	Covered Rule	Redundancy Type
$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 4, \{o_1, o_2\}\rangle)$	$\rho_3 = (\langle u_1, d_1, p_1, a\rangle\langle c, +, 2, \emptyset\rangle)$	Shadows
$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 4, \{o_1, o_2\}\rangle)$	$\rho_5 = (\langle u_3, d_1, p_3, a\rangle\langle c, -, 2, \emptyset\rangle)$	Shadows
$\rho_6 = (\langle u_1, d_0, p_0, a\rangle\langle \emptyset, -, 5, \emptyset\rangle)$	$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 4, \{o_1, o_2\}\rangle)$	Shadows
$\rho_6 = (\langle u_1, d_0, p_0, a\rangle\langle \emptyset, -, 5, \emptyset\rangle)$	$\rho_3 = (\langle u_1, d_1, p_1, a\rangle\langle c, +, 2, \emptyset\rangle)$	Shadows
$\rho_6 = (\langle u_1, d_0, p_0, a\rangle\langle \emptyset, -, 5, \emptyset\rangle)$	$\rho_5 = (\langle u_3, d_1, p_3, a\rangle\langle c, -, 2, \emptyset\rangle)$	Shadows
$\rho_6 = (\langle u_1, d_0, p_0, a\rangle\langle \emptyset, -, 5, \emptyset\rangle)$	$\rho_7 = (\langle u_3, d_1, p_4, a\rangle\langle c, c_1, +, 4, \{o_1\}\rangle)$	Shadows
$\rho_6 = (\langle u_1, d_0, p_0, a\rangle\langle \emptyset, -, 5, \emptyset\rangle)$	$\rho_8 = (\langle u_3, d_1, p_3, a\rangle\langle c, +, 3, \{o_1\}\rangle)$	Shadows
$\rho_2 = (\langle u_3, d_1, p_2, a\rangle\langle c, -, 4, \emptyset\rangle)$	$\rho_5 = (\langle u_3, d_1, p_3, a\rangle\langle c, -, 2, \emptyset\rangle)$	Shadows
$\rho_2 = (\langle u_3, d_1, p_2, a\rangle\langle c, -, 4, \emptyset\rangle)$	$\rho_7 = (\langle u_3, d_1, p_4, a\rangle\langle c, c_1, +, 4, \{o_1\}\rangle)$	Shadows
$\rho_2 = (\langle u_3, d_1, p_2, a\rangle\langle c, -, 4, \emptyset\rangle)$	$\rho_8 = (\langle u_3, d_1, p_3, a\rangle\langle c, +, 3, \{o_1\}\rangle)$	Shadows

Shadowing is a critical problem in an EP3P privacy policy, as shadowed rules may cause a denied access request to be permitted and vice versa. In other words if we assign the priority to each shadowed rule such that they can be reached, then the outcome of the policy changes.

4.4.3 Contradicts

Third, we say a 'deny' rule *contradicts* an 'allow' rule if the 'deny' rule exists with equal or higher priority and fires in the same situations. This differs from 'inclusion' and 'shadowing' in that the rulings are accounted for in determining redundancy. This situation is not simply resolvable by changing the priority, because the difference in the ruling of the rules signifies two opposite authorization decisions, it flags the potential logical mistakes made while formulating the policy.

Definition 17 Given two rules ρ, ρ', we can say $\rho \succ_C \rho'$ (ρ contradicts ρ') when ρ covers ρ', the priority of ρ is equal or higher than ρ' ($i \geq i'$), and $r =$ '$-'$ and $r' =$ '$+'$.

$\mathcal{R}$: Ruleset

$P_=$: rules with equal priority.

$P_<$: rules with smaller priority.

$r_{\neq}$: rules with opposite ruling.

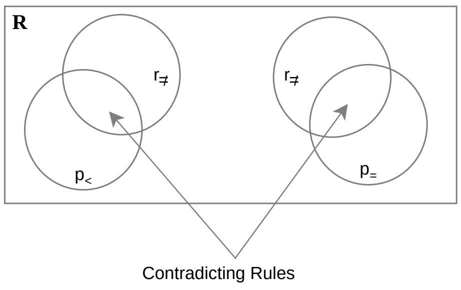


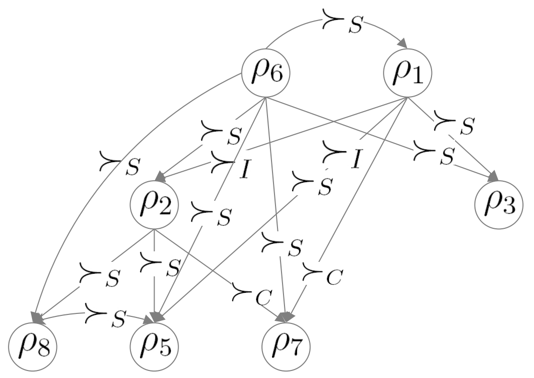
Figure 4.6: Redundancy Situation: Contradicts

By re-examining the rules in Figure 4.3 with respect to the above definition we can observe that the following rules are contradictory.

Covering Rule	Coverd Rule	Redundancy Type
$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 4, \{o_1, o_2\}\rangle)$	$\rho_7 = (\langle u_3, d_1, p_4, a\rangle\langle c, c_1, +, 4, \{o_1\}\rangle)$	Contradicts
$\rho_2 = (\langle u_3, d_1, p_2, a\rangle\langle c, -, 4, \emptyset\rangle)$	$\rho_7 = (\langle u_3, d_1, p_4, a\rangle\langle c, c_1, +, 4, \{o_1\}\rangle)$	Contradicts

4.5 Rule Redundancy Model

Now that the redundancies are classified, we are in a position to extend the rule covering model (Figure 4.3) to represent the *redundancies* between the covering rules as well. In the rule redundancy model, a node denotes a rule in the ruleset, each arrow in the model connects two rules, with the arrow going from the rule that is causing redundancy to the rule that is redundant. The labels on the arrow show the class of the redundancy. To show this we extend the rules used in the Figure 4.3 by four new rules and show the redundancy relationship between them. This is shown in the Figure 4.7.



$\rho_6 = (\langle u_1, d_0, p_0, a\rangle\langle\emptyset, -, 5, \emptyset\rangle)$
$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 4, \{o_1, o_2\}\rangle)$

$\rho_3 = (\langle u_1, d_1, p_1, a\rangle\langle c, +, 2, \emptyset\rangle)$

$\rho_2 = (\langle u_3, d_1, p_2, a\rangle\langle c, -, 4, \emptyset\rangle)$
$\rho_7 = (\langle u_3, d_1, p_4, a\rangle\langle c, c_1, +, 4, \{o_1\}\rangle)$

$\rho_5 = (\langle u_3, d_1, p_3, a\rangle\langle c, -, 2, o\rangle)$
$\rho_8 = (\langle u_3, d_1, p_3, a\rangle\langle c, +, 3, \{o_1\}\rangle)$

$\rho_4 = (\langle u_2, d_2, p_2, a\rangle\langle c, +, 1, \emptyset\rangle)$

Figure 4.7: Redundancy Relation

From the definitions 15, 16 and 17 and Figure 4.7 we can observe that the redundancy relations that we have defined are not mutually exclusive, (i.e., it is possible for a rule to be included and shadowed or contradicted and shadowed). This property is useful for us because it indicates that the reason for the existing redundancies are not only the improper assignment of priorities. We will discuss these issues in detail in the following sections, when we talk about how to resolve the redundancies in a ruleset.

4.6 Pairwise Redundancy Detection

The most important advantage of defining redundancy classification is the ability to detect them.

Definition 18 We define the relation $\succ$ (read *overruled*) as the union of the relations $\succ_I$, $\succ_S$ and $\succ_C$. That is, $\rho \succ \rho'$ iff $\rho \succ_I \rho'$ or $\rho \succ_S \rho'$ or $\rho \succ_C \rho'$.

We wish to prove that the above means of testing and classifying pairs of rules is equal to the pairwise-redundant definition. This property is important because it assures that the detection of the overruled rules in the policy are consequently equivalent to the detection of all the pairwise-redundant rules in the policy.

Theorem: 1 Given a rule $\rho \in \mathcal{R}$ where ρ is pairwise-redundant w.r.t. some other rule ρ', it is also true that ρ is overruled by ρ'.

Proof 1 We must show that there is no situation where a rule is pairwise-redundant but not overruled.
Given the definition of pairwise-redundant (Definition 11) we can assume there are only two rules in the ruleset, $\rho = (\langle u, a, d, p\rangle\langle c, i, r, o\rangle)$ and $\rho' = (\langle u', a', d', p'\rangle\langle c', i', r', o'\rangle)$. Now, for ρ' to be pairwise-redundant ρ must fire in all the situations that ρ' fires; by the concept of

covering (Definition 13), ρ must cover ρ'.
Assume, ρ covers ρ', given the semantics of EP3P (Definition 8), ρ' never fires iff:

1. ρ has a higher priority than ρ' or
2. ρ and ρ' have the same priority but ρ has a negative ruling while ρ' has a positive rulling.

Therefore, if any of the above criteria hold, ρ' is pairwise-redundant. In addition, the definition of pairwise-redundant entails that all rules with identical authorization decision are pairwise-redundant. From the redundancy classes (i.e., shadows (Definition 16), contradicts (Definition 17) and includes (Definition 15)) it follows that all the above mentioned situations for ρ to make ρ' pairwise-redundant are addressed in the relation $\succ$. Therefore, the above theorem holds.

It is also important to consider the following properties which ensure the efficient testability of the three redundancy classes.

Lemma 1 The relation 'covers' is transitive.

Proof 2 Given three rules ρ, ρ', ρ'' where ρ covers ρ' and ρ' covers ρ'', we must prove that ρ covers ρ''. We know $u \geq_U u' \geq_U u''$ and so $u \geq_U u''$. By identical processes we can show $d \geq_D d''$, $p \geq_P p''$, $a = a''$ and $c \subseteq c''$. Therefore ρ covers ρ''.

Lemma 2 The relation $\succ_I$ is transitive.

Proof 3 Assume $\rho \succ_I \rho'$ and $\rho' \succ_I \rho''$. By the definition of $\succ_I$, we know $o = o' = o''$ and so $o = o''$. By identical processes we can show $r = r''$ and $i \geq i''$. By Lemma 1 we know that ρ covers ρ''. Therefore $\rho \succ_I \rho''$.

Lemma 3 The relation $\succ_S$ is transitive.

Proof 4 Assume $\rho \succ_S \rho'$ and $\rho' \succ_S \rho''$. By the definition of $\succ_S$, we know $i \geq i' \geq i''$ and so $i \geq i''$. By Lemma 1 we know that ρ covers ρ''. Therefore $\rho \succ_S \rho''$.

Lemma 4 The relation $\succ_C$ is transitive.

Proof 5 Assume $\rho \succ_C \rho'$. We can prove that no ρ'' exists such that $\rho' \succ_C \rho''$, and $\rho \succ_C \rho''$ thus there are no circumstances for which $\succ_C$ is not transitive.
By the definition of $\succ_C$, we know $r = -$ and $r' = +$. Similarly for $\rho' \succ_C \rho''$ to hold, it must be true that $r' = -$, $r'' = +$. Therefore, $\rho' \succ_C \rho''$ is not true for any ρ'', and so $\succ_C$ is (trivially) transitive.

Lemma 5 The relation $\succ$ is transitive and antisymmetric.

Proof 6 Given three rules ρ, ρ', ρ'' where $\rho \succ \rho'$ and $\rho' \succ \rho''$ to prove transitivity we must show that $\rho \succ \rho''$.
Given three lemmas 2, 3, 4 we can derive that $\rho \succ \rho''$ for the following situations:

- $\rho \succ_S \rho'$ and $\rho' \succ_S \rho''$.
- $\rho \succ_I \rho'$ and $\rho' \succ_I \rho''$.

Now, we need to show that for the rest of situations (e.g., $\rho \succ_I \rho'$ and $\rho' \succ_S \rho''$) the relation $\succ$ is also transitive. Hence, we must consider the following relations:

- In the case where $\rho \succ_S \rho'$ regardless of the type of the relation between ρ' and ρ'', the relation $\rho \succ_S \rho''$ always holds. Therefore, $\rho \succ \rho''$ holds.
- In the case where $\rho \succ_I \rho'$ two types of circumstances may arise based on the relation between ρ' and ρ'':
  - $\rho' \succ_S \rho''$: the relation $\rho \succ_S \rho''$ holds because $i \geq i'$ and $i' \geq i''$.
  - $\rho' \succ_C \rho''$: the relation $\rho \succ_C \rho''$ holds because $c \subseteq c''$, $i \geq i''$ and $r = -$ and $r'' = +$.
- In the case where $\rho \succ_C \rho'$ two types of situations may arise based on the relation between ρ' and ρ'':
  - $\rho' \succ_S \rho''$: the relation $\rho \succ_S \rho''$ holds because $c \subseteq c'$, $c' \subseteq c''$, $i = i'$ and $i' \geq i''$.
  - $\rho' \succ_I \rho''$ at the relation $\rho \succ_C \rho''$ holds because $i = i'$, $i' \geq i''$ and $r = -, r' = r'' = +,$.

To prove antisymmetry we will show that given $\rho \succ \rho'$ the relation $\rho' \succ \rho$ can never be true. Based on the definition of $\succ$, we need to examine $\succ_S$, $\succ_C$, $\succ_I$; if all these relations are antisymmetric, so is $\succ$. Based on the definitions 9, 16, 17 and given two rules ρ, ρ' where:

- $\rho \succ_S \rho'$: it must be true that $i \geq i'$, therefore the relation $\rho' \succ_S \rho$ can never hold when $\rho \succ_S \rho'$.
- $\rho \succ_C \rho'$ it must be true that $r = -$ and $r' = +$, therefore the relation $\rho' \succ_C \rho$ can never hold when $\rho \succ_C \rho'$.
- $\rho \succ_I \rho'$ we know that $i \geq i'$ and $c \subseteq c'$ therefore the only situation where this relation $\rho' \succ_I \rho$ also holds is when $i = i'$ and $c = c'$. Hence two rules are identical. However, since $\mathcal{R}$ is a set, it must not contain ρ, ρ'. Therefore, $\rho \succ \rho'$ is transitive and antisymmetric.

Transitivity is important as it allows the pairwise definition of the relation $\succ$ to be extended to a global definition. That is, given a $\mathcal{R}$ we can determine that $\rho' \in \mathcal{R}$ is overruled iff there exists any rule $\rho \in \mathcal{R}$ such that $\rho \succ \rho'$. This definition continues to hold if ρ itself is overruled by a rule ρ'', as transitivity would imply that ρ'' also overrules ρ'.

Finally, the finiteness of $\mathcal{R}$ ensures there exists a unique minimal set $\mathcal{R}' \subseteq \mathcal{R}$ such that no rule $\rho' \in \mathcal{R}'$ is overruled by some rule $\rho \in \mathcal{R}$.

4.6.1 Undetected Redundancies

It is worthwhile considering those redundancies which are not detectable by pairwise-redundancy. From Definition 9 we can see that a rule $\rho' \in \mathcal{R}$ is redundant if and only if, for all queries and interpretation of conditions [2] there exists a rule that fires before ρ'.

By the requirement of "all interpretation" we know that, for any redundant rule, there must exist a set of rules $\{\rho_1, \ldots, \rho_n\} \subseteq \mathcal{R}$ such that $c' \Rightarrow \bigvee_{1 \le i \le n} c_i$. However, pairwise-redundancy provides only for a single rule ρ such that $c' \Rightarrow c$.

Therefore, a redundant rule will not satisfy the definition of pairwise-redundancy only when a collection of rules enumerates all possible conditions in which the rule would fire, in a piecewise fashion. We argue that this form of policy construction will be rare in practice as it is cumbersome and requires deliberate effort from the policy author. Any outcome which can be achieved in a piecewise fashion is more easily achieved by creating a general rule with enumeration of exceptions.

4.6.2 Redundancy Detection Algorithms

There are two different redundancy detection algorithms that we will discuss in this section. The first algorithm uses the entire ruleset as an input and determines if redundancy exists within the ruleset. The second algorithm takes a minimal ruleset $\mathcal{R}$ and a rule ρ and identifies the redundancies in the $\mathcal{R}$ with respect to the rule ρ.

Both algorithms record their search result in a table that we denote as *redundancy table* with three columns. Each row represents the redundancy relation between a pair of rules: the first column contains the covering rule, the second column contains the rule which is covered and the third column contains the redundancy type (e.g., shadows) between the rules in the first two columns.

Before we explain the main redundancy detection algorithms, we must describe the functions that are used by them. We provide the pseudo code for these functions and explain the outcome of each one of them.

The function *isCovering*(ρ, ρ') takes two rules and uses the vocabulary of the privacy policy to specify if one rule covers another. It may return any of the following flags:

- *BOTH*: ρ covers ρ' and similarly ρ' covers ρ. Based on the definition of covering this situation can arise when user, data, purpose, action and condition elements of both

[2] An interpretation I is a set of logical terms providing a single truth value for every atom.

rules are identical.

- *COVERS*: The first rule ρ covers ρ'.
- *COVERED*: The first rule ρ is covered by ρ'.
- *NONE*: There is no covering relation between the two rules.

Algorithm 1 isCovering(Rule ρ, Rule ρ')

```
1: if $\rho > \rho'$ then
2:     return COVERS.
3: else if $\rho' > \rho$ then
4:     return COVERED.
5: else if $\rho \geq \rho'$ AND $\rho' \geq \rho$ then
6:     return BOTH.
7: else
8:     return NONE
9: end if
```

The function *getSetRelation*(s, s') takes two obligation sets or two condition set and returns any of the following flags:

- *EQUAL*: If both sets are equal.
- *SUBSET*: If s is a strict subset of s'
- *SUPERSET*: If s is a superset of s'.
- *NONE*: If none of the above relation holds.

Algorithm 2 getSetRelation(Object[] set_1, Object[] set_2)

```
1: if (isSubset($set_1$,$set_2$)) then
2:     return SUBSET;
3: end if
4: if (isSubset($set_2$,$set_1$)) then
5:     return SUPERSET;
6: end if
7: if (isSubset($set_1$,$set_2$) and isSubset($set_2$,$set_1$)) then
8:     return EQUAL;
9: end if
10: return NONE;
```

The function *isSubset*(s, s') takes two sets and returns *true* if s is the subset of s', otherwise it returns *false*.

Algorithm 3 isSubset(Object[] set_1, Object[] set_2)

```
int i, j;
if ($set_1$==null) then
  return true;
end if;
if ($set_2$==null) then
  return false;
end if;
for (i=0;i¡$set_1$.length;i++) do
  for (j=0;j¡$set_2$.length;j++) do
    if ($set_1$[i]==$set_2$[j]) then
      break;
    end if
    if (j==$set_2$.length) then
      return false;
    end if
    return true;
  end for
end for
```

We use the function *populateTable*$(node)$ to store the detected redundancies in a table for later use (i.e., resolving them). Each redundancy is an instance of the *redundancyNode* class that encapsulate the rules (i.e., covering and covered) in each redundancy relations and the redundancy class.

```
public class RedundancyNode{
    Object parent, child;
    int redundancy;
}
```

The constants below are used in the following algorithms to specify the redundnacy class of a redundant rule:

- public static final int CONTRADICTS = 1;
- public static final int SHADOWS = 2;
- public static final int INCLUDES = 4;

Note that the constant INCLUDE is assigned value 4 (instead of 3) to allow us to represent the situations where a rule satisfies more than one class of redundnacy (e.g., when a rule CONTRADICTS and SHADOWS it will get the value of 3).

Algorithm 4 *populateTable(node)*

```
Require: redundancies {An ArrayList() is a already defined};
1: if (node.redundancy $\neq 0$) then
2:     redundancies.add(node);
3: end if
```

Finally, we introduce the algorithm that is the logic behind determining redundancy classes.

Algorithm 5 setRedundancyClass(Rule ρ, Rule ρ')

```
Require: RedundnacyNode node = new RedundnacyNode;
1: if ($\rho$.priority = $\rho'$.priority and $\rho$.ruling = "deny" and $\rho'$.ruling="authorize") then
2:     node.redundancy += CONTRADICTS;
3: end if
4: if ($\rho$.priority = $\rho'$.priority and $\rho$.ruling = $\rho'$.ruling and getSetRelation($\rho$.obligation, $\rho'$.obligation)=SUPERSET) then
5:     node.redundancy += INCLUDES;
6: end if
7: if ($\rho$.priority $\geq$ $\rho'$.priority and $\rho$.ruling = $\rho'$.ruling and getSetRelation($\rho$.obligation, $\rho'$.obligation)=EQUAL) then
8:     node.redundancy += INCLUDES;
9: end if
10: if ($\rho$.priority > $\rho'$.priority) then
11:     node.redundancy += SHADOWS;
12: end if
13: populateTable(node).
```

Now we are in the position to represent the algorithm that detect all the redundancies in a ruleset.

Algorithm 6 detectRedundancy(Ruleset $\mathcal{R}$)

```
1: int self;
2: for all $\rho \in \mathcal{R}$ do
3:   self = 0;
4:   for all $\rho' \in \mathcal{R}$ do
5:     switch($isCovering(\rho, \rho')$)
6:     case $COVERS$: call setRedundancyClass($\rho$,$\rho'$); break;
7:     case $COVERED$: call setRedundancyClass($\rho'$,$\rho$); break;
8:     case $BOTH$:
9:     if $\rho$.priority = $\rho'$.priority and $\rho$.ruling = $\rho'$.ruling and $getSetRelation(\rho.obligation, \rho'.obligation)$ then
10:      if self != 1 then
11:        call setRedundancyClass($\rho$,$\rho'$);
12:        call setRedundancyClass($\rho$,$\rho'$); break; self++
13:      end if
14:    else
15:      call setRedundancyClass($\rho$,$\rho'$); break;
16:    end if
17:    case $NONE$: break;
18:    end switch
19:  end for
20: end for
```

While Algorithm 6 evaluate the ruleset of redundancies. The following algorithm allows us to check all the rules in the ruleset against a new rules.

Algorithm 7 detectRedundancy(Rule ρ, Ruleset $\mathcal{R}$)

```
1: for all $\rho' \in \mathcal{R}$ do
2:   switch($isCovering(\rho, \rho')$)
3:   case $COVERS$: call setRedundancyClass($\rho$,$\rho'$); break;
4:   case $COVERED$: call setRedundancyClass($\rho'$,$\rho$); break;
5:   case $BOTH$: call setRedundancyClass($\rho$,$\rho'$); call setRedundancyClass($\rho$,$\rho'$); break;
6:   case $NONE$: break;
7: end for
```

By executing Algorithm 6 or Algorithm 7 a *redundancyTable* will be populated with all the detected redundancies along with their redundancy class, highlighting the reason to why they are considered as redundant. In the next chapter we will explain how can we resolve the redundancies in an EP3P policy.

Chapter 5

Redundancy Resolution Framework

In this chapter we will provide a method for *resolving* all the identified redundancies within an EP3P policy. In addition, we will show how our redundancy detection and resolution frameworks can assist policy auditor(s) in changing the rules in an existing policy. We begin with discussing what it means to resolve a redundancy and we introduce a pairwise redundancy resolution. We go on to discuss the difficulties that will arise when one wants to resolve all the redundancies in a policy. Finally, we explain the applicability of our framework to updating an EP3P policy.

5.1 Pairwise Redundancy Resolution

In this section we will investigate and resolve redundancies between a pair of rules[1]. For now, we ignore the complexities that may arise when more than two rules are taken into account while resolving redundancies (e.g., one rule makes two rules redundant or when the first rule makes the second and the second rule makes a third rule redundant). Such a narrow perspective assists us in explaining the intuition behind our approach for resolving a redundant rule.

The common notion for "redundancy resolution" denotes that redundant rules must be removed. This is based on the belief that all the redundant rules are *excessive* - they do not contain any information more than what has already been expressed by other rules. However, as we will show, not all of the redundant rules are excessive. Therefore, it is important to evaluate the rules with respect to the authorization information they provide to the policy. Knowing that, we can decide on an appropriate operation (referred to as resolution *suggestion*) for resolving the redundancy.

There are two criteria to decide if a rule provides any extra authorization information in comparison to another rule in the policy. First, the *situation* of the rule. Second, the *authorization decision* the rule provides. The situation is defined in terms of the followings:

- the User, the Data, and the Purpose elements (udp) of the rule.

[1]we assume that redundancies are already detected by executing the algorithm 6 and recorded in a *redundancyTable*.

- the circumstances (conditions) in which the rule is applicable.

A situation can be "a doctor (user) wants to read (action) the patients record (data) for treatment (purpose)". This is captured by the following definition:

Definition 19 Given ρ, ρ' we write $\rho' \cong \rho$ iff $\rho \geq \rho'$ and $\rho' \geq \rho$.

When two rules have the same situation and the same authorization decision we can safely believe that one rule is a duplicate of another.

Definition 20 A pairwise-redundant rule $\rho' \in \mathcal{R}$ is *duplicate* iff there exists another rule s.t., $\rho \cong \rho'$, $r = r'$ and $o = o'$.

Duplicate rules are excessive. To resolve such a redundancy one of the rules must be *removed* from the ruleset.

However, sometimes the authorization decision of one rule for a situation *contradicts* with the authorization decision that another rule provides for the same situation. By contradicts we mean, one authorizes an access whilst another denies the access.

Definition 21 A pairwise-redundant rule $\rho' \in \mathcal{R}$ is *opposing* iff there exists another rule $\rho \in \mathcal{R}$ s.t. and $r \neq r'$.

Obviously,the existence of opposing rules in the policy signifies a logical error made by the policy auditor. Hence, to resolve the redundancy either ρ or ρ' must be *removed* from the policy. Note that we assume these rules are presented to the policy auditor and they can make a decision on which rule should remain in the policy.

On the other hand, when two rules agree on whether an access is to be authorized or denied for the same situation, but they provide different obligations, they are considered to be two incomplete rules. In other words, both rules provide a part of the obligations of the authorization response that must be assigned to the situation.

Definition 22 A rule $\rho' \in \mathcal{R}$ is *incomplete* iff there exists another rule $\rho \in \mathcal{R}$ s.t. $\rho \cong \rho'$, $r = r'$, and $o \neq o'$.

To resolve the redundancy introduced by incomplete rules, the obligations of the rules must be merged. By doing so we will have a third rule ρ'' that is identical to ρ with $o'' = o \cup o'$. After performing such an operation ρ and ρ' must be *removed* from the policy. For example, let us assume the following two rules exist in a policy:

1) Patient's medical record can be disclosed to other hospitals if the patient is in a life threatening condition. However, the family of the patient must be notified about the disclosure.

2) Patient's medical record can be disclosed to other hospitals if the patient is in a life threatening condition. However, the hospitals that are receiving the data are obliged to remove the patient's record after the critical condition clears.

Since both of the above rules refer to the same situation and provide different obligations, it is intuitive to believe that both obligations must be fulfilled. In the following sections we will further explain why these rules are likely to exist in a privacy policy.

In addition to the above mentioned scenarios, sometimes a rule become redundant because there exists some other covering rules with a higher priority in the policy.

Definition 23 A pairwise-redundant rule $\rho' \in \mathcal{R}$ is *inactive* iff there exists another rule $\rho \in \mathcal{R}$ s.t. $\rho > \rho'$.

The fact that a rule is inactive signifies that the redundant rule is more specific than the rule that makes it redundant $(\rho > \rho')$. The inactive ρ' can be considered as an *exception* to the more general rule ρ. Hence, to resolve the redundancy we can *raise the priority* of the redundant rule.

Thus far we have classified the redundancies with respect to the authorization information they provide to an EP3P policy. However, it is important to distinguish the above classifications from the *shadowing, include*, and *contradicts*. In the previous chapter we were only concerned about how to detect *all* the pairwise-redundant rules, while the classification made in this chapter assists us to determine what needs to be done with detected redundant rules.

Theorem: 2 Given a rule ρ that is pairwise-redundant, ρ is either *duplicate, opposing, incomplete* or *inactive.*

Before we proceed further, we must address doubts that may arise for readers about Definition 20. One may argue that it is not necessary to restrict the definition of duplicates to pairwise-redundant rules where $\rho \cong \rho$. In other words one may argue that ρ' is also duplicate if $\rho \geq \rho'$ and $r = r', o = o'$. They may reason that although two rules have different udp's or conditions, since their authorization decisions are equal, the covered rule ρ' is duplicate. Further they could continue: even if the priority of ρ' is raised such that it fires, then the authorization response of the policy does not change. However, we can show that such a reasoning does not always hold by providing the following counter example. Let us assume there exists a ruleset with three rules:

$$\begin{tikzcd}
\rho_1 \arrow[d] \arrow[dd, bend right=40, "\succ_I"'] \\
\rho_2 \arrow[d, "\succ_C"] \\
\rho_3
\end{tikzcd}$$

$\rho_1 = (\langle u_0, d_0, p_0, a\rangle\langle c, +, 5, \{o_1\}\rangle)$
$\rho_2 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 7, \emptyset\rangle)$
$\rho_3 = (\langle u_3, d_0, p_0, a\rangle\langle c_0, c_1, +, 3, \{o_1\}\rangle)$

	Covering Rule	Covered Rule	Suggestion
2	ρ_1	ρ_3	Includes
2	ρ_2	ρ_3	Includes

Given definitions 20, 21, 22, we can see that ρ_3 is not duplicate nor opposing nor incomplete, because $\rho_1 > \rho_3$. Hence, to resolve the redundancy $\rho_1 \succ_I \rho_3$ the priority of ρ_3 will be raised to 8. Now, ρ_1 becomes a general rule (e.g., authorizes all the staff to perform a on d for p) and ρ_2 is an exception to ρ_1 (e.g., specifically denies the medical team access), and ρ_3 is also an exception to ρ_2 (e.g., authorizes only nurses, who are part of the medical team). It is clear that if our definition of duplicates had captured the rules with the same ruling and obligations, regardless of their specificity, then in the above example ρ_3 could have been removed.

5.2 Ruleset Redundancy Resolution

In section 4.6.2 we introduced algorithm 6 that detects all the pairwise-redundancies in a ruleset. Then, in the previous section we discussed how to resolve each identified redundancy in isolation - one redundancy relation at a time. In other words, we were not concerned if a redundant rule was made redundant by more than one rule. In this section we explain how to resolve all the pairwise redundant rules in a policy. Clearly stated we have three goals: First, to specify a redundancy resolution procedure. Second, to guarantee that the policy will be minimal (Definition 12) after performing the redundancy resolution procedure. Third, to ensure that our procedure for creating a minimal policy is efficient. To achieve these goal we are facing two obstacles:

- How to deal with contradictory resolution suggestions for resolving a redundant rule.
- In what order the redundancies must be resolved such that after resolving all the identified redundant rules, the ruleset will be minimal.

In the rest of this chapter we explain the sources of these obstacles and provide the necessary procedures to tackle them in order to achieve our goals.

5.2.1 Dealing With Contradictory Suggestions

When a rule is redundant by more than one rule in a ruleset, there are multiple resolution suggestions each addressing one redundancy relation. In an ideal case, all the suggestions are identical. For example, consider the following case:

$$\begin{tikzcd}
\rho_1 \arrow[dr, "\succ_C"'] & & \rho_2 \arrow[dl, "\succ_C"] \\
& \rho_3 &
\end{tikzcd}$$

$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c_1, +, 6, o_1\rangle)$
$\rho_2 = (\langle u_1, d_0, p_2, a\rangle\langle c_2, +, 8, o_2\rangle)$
$\rho_3 = (\langle u_3, d_1, p_3, a\rangle\langle c_1, c_2, -, 5, o_1, o_2\rangle)$

Covering Rule	Covered Rule	Suggestion
ρ_1	ρ_3	Raise Priority
ρ_2	ρ_3	Raise Priority

Since both covering rules ρ_1, ρ_2 have different situations and different authorization decisions with respect to ρ_3, the resolution suggestion is to increase the priority of ρ_3. By considering the first redundancy relation, the priority of ρ_3 will increase to 7. To resolve the second redundancy relation, the priority of ρ_3 will further increase to 9.

Clearly stated, the problem of contradictory resolution suggestions arise when a rule is made redundant by more than one rule, whereby the resolution suggestions provided for resolving the redundant rule are different. For example, consider the following rules:

$$\begin{tikzcd}
& \rho_1 \arrow[dl, "\succ_C"'] \arrow[dr, "\succ_C"] & \\
\rho_2 \arrow[rr, "\succ_S"] & & \rho_3
\end{tikzcd}$$

$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 7, \{o_0, o_1, o_2\}\rangle)$
$\rho_2 = (\langle u_1, d_0, p_2, a\rangle\langle c, +, 6, \{o_0\}\rangle)$
$\rho_3 = (\langle u_1, d_0, p_2, a\rangle\langle c, +, 5, \{o_1\}\rangle)$

	Covering Rule	Coverd Rule	Suggestion
1	ρ_1	ρ_2	Raise Priority
2	ρ_1	ρ_3	Raise Priority
2	ρ_2	ρ_3	Merge Obligations

As the diagram illustrates ρ_3 is made redundant by both ρ_1 and ρ_2. However, there are two different resolution suggestions for resolving the redundancy of ρ_3. One suggestion is "raise the priority" to resolve $\rho_1 \succ_C \rho_3$. Whilst the other suggestion is to "merge obligations" to resolve $\rho_2 \succ_I \rho_3$.

In the following section we will describe a two stage process for resolving all the redundancies in a ruleset. The first stage is denoted as *policy compaction*, and the second stage as *policy rectification*.

5.2.2 Policy Compaction

The aim of policy compaction is two fold. First, to remove the rules whose coexistence in the policy either does not add any authorization information (e.g., duplicate rules) or whose coexistence is logically contradictory (e.g., opposing rules). Second, to combine the incomplete rules, such that during the policy maintenance one can deal with only one rule instead of many incomplete rules.

Definition 24 An EP3P policy P is *compact* iff there exists no *duplicate, opposing or incomplete* rule $\rho \in \mathcal{R}_P$ where $\mathcal{R}_P$ is the ruleset of the policy P.

Recall that the resolution suggestion for resolving an opposing rule is to "remove either" of the rules. However, this suggestion is not specific enough. We need to be able to decide which rules shall be removed. As we will show, when there are more than two rules opposing one rule, we can provide a more specific resolution suggestion - determine which rule must be removed. Consider the following ruleset that we wish to compact:

$$\rho_1 = (\langle u_0, d_0, p_0, a\rangle\langle c, +, 5, \{o_1\}\rangle)$$
$$\rho_2 = (\langle u_0, d_0, p_0, a\rangle\langle c, -, 7, \emptyset\rangle)$$
$$\rho_3 = (\langle u_0, d_0, p_0, a\rangle\langle c, +, 3, \{o_3\}\rangle)$$

$$\begin{tikzcd}
\rho_1 \arrow[r, "\succ_C"] \arrow[rr, bend left, "\succ_S"] & \rho_2 & \rho_3 \arrow[l, "\succ_C"']
\end{tikzcd}$$

	Covering Rule	Coverd Rule	Suggestion
1	ρ_1	ρ_3	Merge Obligations
2	ρ_1	ρ_2	Remove either
3	ρ_3	ρ_2	Remove either

As the diagram shows, ρ_2 is opposing with respect to both rules ρ_1 and ρ_3, hence the resolution suggestion is to remove either ρ_2, or both ρ_1 and ρ_3. Since there are two rules opposing ρ_2, it is intuitive to believe that ρ_2 is most likely the rule that must be removed from the ruleset rather than ρ_1 and ρ_3.

In the above example, if we have first considered resolving the incomplete rules, ρ_1 and ρ_3 would have been merged to create a new rule ρ_2'. Hence, since there would be only two rules where one is opposing another ($\rho_2' \succ_C \rho_3$) we would not be able to make a specific suggestion to remove ρ_2'.

Here we shall describe our procedure for compacting a policy. The policy compaction procedure ensures that after its execution there remains no opposing, duplicate and incomplete rules in the policy.

1. For all the redundant rules (e.g., ρ, ρ') where $\rho \cong \rho'$, count the authorize rule as well as deny rule.
2. If the number of authorize rules is greater than the number of deny rules then the suggestion is to remove all the deny rules.
3. Otherwise if there are more deny rules than authorize rules, the suggestion is to remove authorize rules.
4. else the suggestion is to remove either of the rules (there is no preference).
5. Remove all the duplicate rules.
6. Merge the obligations of the opposing rules.

The above procedure shows that in the process of compacting a policy opposing rules must be resolved prior to merging the obligations of incomplete rules or removing duplicate rules. This allows us to make a decision of which opposing rules to remove based on the number of occurrence of the same rule with the same ruling.

5.2.3 Policy Rectification

Now that all the redundancies with resolution suggestions "remove rule" or "merge obligations" are resolved (during policy compaction), the priorities of the remaining redundant rules need to be adjusted such that they are not redundant any more.

Definition 25 An EP3P policy P is *rectified* if there exists no *inactive* rule $\rho \in \mathcal{R}_P$ (i.e., $\mathcal{R}_P$ is the ruleset of the policy P).

As the priority is bounded below by zero, the simplest uniform technique for adjusting priorities is to raise them. However, this can cause problems when:

- raising the priority of one rule causes further redundancies of other rules; and
- raising the priority of a rule interferes with any semantics the user associates with specific priority values.

It is therefore in our interests to minimize the amount of change that occurs to priority values.

Definition 26 We define the *rectification distance* for a rule $\rho \in \mathcal{R}$ as the absolute difference between the current priority of ρ and the minimum priority required to ensure it is active.

Definition 27 We define the *minimal rectification distance* for a rule $\rho \in \mathcal{R}$ as a recursive function *minrec*(ρ). We will use the notation $\hat{\rho}$ to denote those rules which strictly cover ρ (in Definition 14). We can thus present *minrec* as:

$$minrec(\rho) = \min_{\rho' \in \hat{\rho}} \begin{cases} minrec(\rho') & when\, \text{r'} = +\, and\, \text{r} = \text{-} \\ minrec(\rho') + 1 & otherwise \end{cases}$$

While the above definitions describe the rectification and minimal rectification distance for individual rules, there might be more than one inactive rule in a policy. Hence, we need a method for combining rectification distances, and a method of comparison to determine overall "minimal change" when rectifying the policy. There are several options available for combining rectification distances and corresponding methods of comparison.

Summation: A common method for combining measures of change is to use arithmetic summation. For example, if the rectification distance of two rules is 3 and 5 respectively, then it can be assumed that the combined rectification distance is $3 + 5 = 8$. We would say that we have performed less change if the combined rectification distance is lower according to the usual number ordering.

Maximum: An alternative is to take the maximum rectification distance for all rules. Using the same individual distances, we would calculate the combined rectification as $\max(3, 5) = 5$. Again, we would say that we have performed less change if the combined rectification distance is lower according to the usual number ordering.

Lexical: Finally, we can determine minimal change using lexical ordering of the individual rectification distances. The combined rectification distance would therefore be a vector $\langle 5, 3\rangle$. Comparison would then proceed in lexical ordering from left to right, using the usual number ordering. For example $\langle 4, 4\rangle$ is less than $\langle 5, 3\rangle$ which in turn is less than $\langle 5, 4\rangle$.

We can now describe a simple procedure for policy rectification, and describe how it achieves each of the above notions of minimal change. The following procedure ensures that through an iterative process, there will exist no inactive rule in the policy.

1. Let ρ be any inactive rule.
2. Raise the priority of ρ to the minimum required to make it active.
3. Go to 1.

Observe that the above procedure provides an iterative computation equivalent of *minrec* for *every* rule in the ruleset simultaneously. That is, each rule is raised only the minimum amount required. To raise any rule less would leave it inactive, but by line 2 we ensure we only raise it the minimum amount. As we have achieved absolute minimal change for each of the rules, it follows that we satisfy each of the above notions of minimal change (i.e., summation, maximum and lexical).

Theorem: 3 The policy rectification algorithm terminates and rectifies all redundancies.

Proof 7 Rectifying a rule will never induce new redundancies of rules that are not covered by it, as rectification does not impact on the covering relation (Definition 13) and the covering relation is key to the definition of redundancy. For a given set of inactive rules A, the algorithm ensures that in every iteration at least one redundant rule is given a priority such that changing the priority of any other rule in the set does not make it redundant. Such a rule(s) exists because there always exists at least one rule that is not covered by any rule in the set A. Hence, regardless of which inactive rules chosen from A, the number of inactive rules in A monotonically decrease. Therefore, the algorithm terminates.

5.2.4 Efficient Policy Rectification

As we showed in the previous section, when there is more than one inactive rule (ρ) in the policy the process of policy rectification is an iterative call to the function *minrec*(ρ) until there is no redundant rule to be rectified. However, as we will show in the following example, such a process may become very inefficient. To clearly specify the problem, let us assume there exists a policy where the following redundancies are identified:

$$\begin{tikzcd}
 & \rho_0 \arrow[d, "\succ S"] \arrow[ddl, "\succ S"'] \arrow[ddr, "\succ S"] & \\
 & \rho_1 \arrow[dl, "\succ S"] \arrow[dr, "\succ S"'] & \\
\rho_2 & & \rho_3
\end{tikzcd}$$

$$\rho_0 = (\langle u_0, d_0, p_0, a\rangle\langle c, -, 7, \emptyset\rangle)$$
$$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 5, \{o\}\rangle)$$
$$\rho_2 = (\langle u_1, d_0, p_2, a\rangle\langle c, c_1, +, 2, \emptyset\rangle)$$
$$\rho_3 = (\langle u_3, d_1, p_3, a\rangle\langle c, c_1, +, 3, \{o_1\}\rangle)$$

	Covering Rule	Coverd Rule	Suggestion
1	ρ_0	ρ_2	Raise Priority
2	ρ_1	ρ_2	Raise Priority
3	ρ_1	ρ_3	Raise Priority
4	ρ_0	ρ_3	Raise Priority
5	ρ_0	ρ_1	Raise Priority

To resolve any of the above redundancies, we must raise the priority of the redundant rule such that it becomes greater than the priority of the covering rule. Hence, to resolve the first redundancy, we may increase the priority of ρ_2 to 8. In the second redundancy, the current priority of ρ_2 is greater than the priority of ρ_1, hence the redundancy is already resolved. Similarly, the priority of ρ_3 is increased to 6. To resolve the forth redundancy, ρ_3 must be modified again such that $\rho_3 = (\langle u_3, d_1, p_3, a\rangle\langle c, c_1, +, 8, \{o_1\}\rangle)$. Finally, we modify $\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 8, \{o\}\rangle)$. Now, if we examine the ruleset for redundancy, the redundancies between ρ_1, ρ_2, ρ_3 are still unresolved even though we have individually resolved all the detected redundancies.

$$\begin{tikzcd} & \rho_1 \arrow[dl, "\succ_S"'] \arrow[dr, "\succ_S"] & \\ \rho_2 & & \rho_3 \end{tikzcd}$$

$$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 8, \{o\}\rangle)$$
$$\rho_2 = (\langle u_1, d_0, p_2, a\rangle\langle c, c_1, +, 8, \emptyset\rangle)$$
$$\rho_3 = (\langle u_3, d_1, p_3, a\rangle\langle c, c_1, +, 8, \{o_1\}\rangle)$$

	Covering Rule	Coverd Rule	Redundancy Type
1	ρ_1	ρ_2	Shadows
2	ρ_1	ρ_3	Shadows

As we can observe, to rectify a policy that contains multiple inactive rules, several priorities may need to be reassigned to the inactive rule(s) before the policy is rectified. Therefore, it is in our interests to ensure that a policy is rectified without multiple reassignment of priorities to inactive rules. The following procedure ensures that a policy is rectified and no inactive rule in the policy is assigned a new priority more than once.

1. For every inactive rule ρ s.t. there exists a rule $\rho' \succ_S \rho$ and $\rho'' \succ_S \rho'$, ignore the relation $\rho'' \succ_S \rho$ (Lemma 3).
2. Let ρ be an inactive rule with respect to ρ' s.t. there exists no other rule ρ'' in the ruleset that can make ρ' redundant.
3. Raise the priority of ρ to the minimum required to make it active (Based on the Definition 27).
4. Go to 2.

Theorem: 4 The algorithm terminates and rectifies all redundancies. Also, each inactive rule will be assigned one new priority, at most.

Proof 8 Given Theorem 3, the algorithm terminates and rectifies all the redundancies. For a given rule ρ, the above algorithm ensures that all rules covering ρ are rectified first by working top-down. Hence, no rules which have been inspected will be made redundant again by a *top-down* procedure. As all rules are inspected, and need only be inspected once, we know that the algorithm will terminate, rectify all redundancies and each rule will be not be assigned more than one new priority.

Hence, by performing the first step of the above procedure on the graph shown in the example in this section we can observe that the redundancy graph will be converted to the following:

$$\begin{tikzcd} & \rho_0 \arrow[d, "\succ_S"] & \\ & \rho_1 \arrow[dl, "\succ_S"'] \arrow[dr, "\succ_S"] & \\ \rho_2 & & \rho_3 \end{tikzcd}$$

$\rho_0 = (\langle u_0, d_0, p_0, a\rangle\langle c, -, 7, \emptyset\rangle)$
$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 5, \{o\}\rangle)$
$\rho_2 = (\langle u_1, d_0, p_2, a\rangle\langle c, c_1, +, 2, \emptyset\rangle)$
$\rho_3 = (\langle u_3, d_1, p_3, a\rangle\langle c, c_1, +, 3, \{o_1\}\rangle)$

	Covering Rule	Coverd Rule	Redundancy Type
1	ρ_0	ρ_1	Raise Priority
2	ρ_1	ρ_2	Raise Priority
3	ρ_1	ρ_3	Raise Priority

Now that we have the minimal required redundancy relations for rectifying the inactive rules in the policy, by taking a top-down approach the procedure ensures that all the nodes (rules) in every level of the hierarchy are resolved in turn. Hence, when an inactive rule (e.g., ρ_1) is resolved with respect to all its parent rules (nodes) (e.g., ρ_0) in the redundancy relation graph, there exists no situation whereby ρ_1 can become redundant again. Hence, each inactive rules needs to be assigned a new priority only once. This is regardless of the number of the rules and redundancy relations.

Note that there is another alternative approach that can ensure all the inactive rules are assigned a new priority only once. This can be achieved by starting from the leaf nodes (ρ_2, ρ_3) in the redundancy graph. For each inactive rule (sibling) reduce the priority of the direct parent rule (e.g., ρ_1) such that the rules become active.

Even though the above approach ensures that all the rules are assigned a new priority only once, this does not guarantee that the priority of rules stay above zero. Consider the the following example where by following the alternative approach we can reach a state where the priority of ρ_0 becomes -1.

$$\rho_0 = (\langle u_0, d_0, p_0, a\rangle\langle c, -, 7, \emptyset\rangle)$$
$$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 5, \{o\}\rangle)$$
$$\rho_2 = (\langle u_1, d_0, p_2, a\rangle\langle c, c_1, +, 1, \emptyset\rangle)$$
$$\rho_3 = (\langle u_3, d_1, p_3, a\rangle\langle c, c_1, +, 2, \{o_1\}\rangle)$$

Thus far we have described how the redundancy detection and resolution frameworks can assist a policy auditor in maintaining the minimality of an EP3P privacy policy. It is important to note that the proposed redundancy detection and redundancy resolution frameworks do not change the syntax or the operational semantics of the EP3P language. Therefore, the important properties of the language, such as uniqueness of authorization response will be preserved.

In the following section we shall explain how these frameworks can be used in performing management operations such as *policy update* and *policy revision.*

5.3 Maintaining an EP3P Privacy Policy

Within the artificial intelligence, psychology, database and philosophy communities, considerable attention has been paid to the study of *belief change*. Belief change occurs when new facts are introduced into an existing set of beliefs, called a *belief-base*.

Belief revision and *belief update* are two approaches used for maintaining a belief-base after introducing new beliefs. Belief revision is an approach for maintaining a belief-base when existing beliefs may be mistaken or incomplete. Belief update is used to accommodate the existing beliefs, that were correct at one time, but are now obsolete due to changes in the world.

Similarly, rules that are added to an EP3P policy can be classified in one of the following categories:

- The rule has not been specified before.
- The rule is a modification of an existing rule that is out-of-date due to external changes. Such changes include new legislation, change of business rules and data owners changing their preferences.
- The rule is a correction of an existing rule that was realized to be incorrect or incomplete. Such rules may be added due to errors made by policy auditor(s).

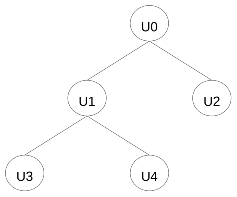
These cases, like those seen in belief update, can be consolidated into two approaches; *policy update* and *policy revision*. These will characterize the methods for maintaining an EP3P policy when a new rule is added. We consider these as heuristics for determining which rules must be modified when a new rule is added to an EP3P privacy policy. In the rest of this section we assume there exist a minimal privacy policy. It will be shown that how our redundancy detection and redundancy resolution frameworks can be used for updating or revising the EP3P privacy policy.

5.3.1 Policy Update

In Chapter 2 we mentioned that privacy policies are derived from legislation and business rules that may change over time. Hence, policies often need to be *updated* to (correctly) reflect newer legislation. *Policy update* refers to the process of changing an EP3P policy to accommodate these changes. The main property of this approach is that the new rules are considered as absolute rules, by which we mean they must remain in the policy and there must not exist an exception to them. Hence, if there exists exceptions to these rules, they need to be identified and presented to the policy auditor.

For example, the privacy policy of a hospital may specify that, during an epidemic, a patient's medical records can be disclosed for further research only if the patient has given consent for medical usage. However, new legislation is issued which specifies during an epidemic, a patient's records can be disclosed for research purposes regardless of consent. In this case the original policy of the hospital must be updated to address the requirements in the new legislation.

In the following example we will show how our redundancy detection framework can assist the policy auditor to detect the rules that may need to be removed, while updating the policy. For simplicity, let the EP3P vocabulary of the hospital have a user hierarchy, with the rest of the vocabulary (e.g., data type, purpose, action, etc.) being single elements.



$\rho_1 = (\langle u_1, d_0, p_0, a\rangle\langle c, +, 5, \{o\}\rangle)$

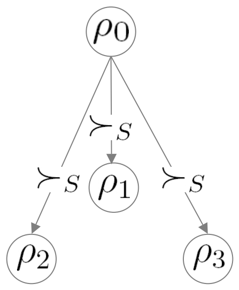
$\rho_2 = (\langle u_3, d_0, p_0, a\rangle\langle c, c_1, +, 6, \emptyset\rangle)$

$\rho_3 = (\langle u_4, d_0, p_0, a\rangle\langle c, c_1, +, 6, \{o_1\}\rangle)$

$\rho_2 = (\langle u_2, d_0, p_0, b\rangle\langle c, +, 6, \{o, o_1\}\rangle)$

$\rho_5 = (\langle u_0, d_0, p_0, a\rangle\langle c_1, +, 6, \{o_1\}\rangle)$

Further, assume that the above policy needs to be updated with $\rho_0 = (\langle u_1, d_0, p_0, a\rangle\langle c, -, 7, \emptyset\rangle)$. When the above rule is added to the policy, the rules ρ_1, ρ_2, ρ_3 become redundant and can be identified using algorithm 7 given in Chapter 4.



	Covering Rule	Coverd Rule	Suggestion
1	ρ_0	ρ_1	Remove
1	ρ_0	ρ_2	Remove
2	ρ_0	ρ_3	Remove

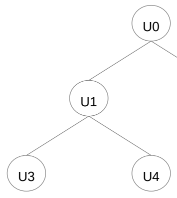
Note that in previous sections we assumed that we were resolving redundancies in an existing policy where none of the rules were considered a *new* rule, so the default suggestions for resolving redundancies was to increase the priorities to make all the inactive rules active. This approach could turn some of the existing rules to an exception of the new rule. Therefore, the resolution suggestion to deal with such redundancies at the time of policy update is to remove all the rules that are made redundant by the new rule. As the above table illustrates ρ_1, ρ_2, ρ_3 shall be removed from the policy.

When updating a policy, we roll-out the possibility of simultaneously adding more than one rules to the policy. In addition, the priority of the new rule must be higher than any rule in the policy. This is is very important, as it enables us to detect all the potential exceptions (i.e., a 'existing covered rule' with a higher priority than the new rule is not redundant, but it is an exception to the new rule). However, after removing the redundant rules, the priority of the new rule can be reduced to the maximum required. To do that we must follow the same (policy rectification) scheme that we introduced in the previous section. So in the case of the above example, the minimum priority for resolving ρ_0 can be determined by $minrec(\rho_0)$.

5.3.2 Policy Revision

It is common that a policy auditor writes incorrect rules while creating an EP3P policy. These rules are usually introduced to an EP3P policy due to the mis-interpretation of the informal privacy policy by the policy auditor. In such situations, although the sources of the rules in the policy has not change, the rules may need to be modified. The process of performing such modifications is referred to as *policy revision*.

Clearly stated, the difference between policy revision and policy update is this: policy update postulates that the new rule is *absolute*, hence it is necessary to remove (or at least double check) all the existing rules that are made redundant by the new rule, whilst policy revision only modifies an incorrect rule by the new rule (i.e., it is not concerned with the exceptions to the new rule). To clarify the difference we use an example which is very close to the example given in the previous section. Similarly, we will show that our redundancy detection/resolution framework can be used during policy revision.



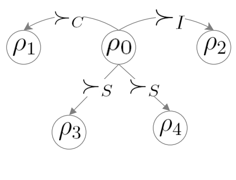
$\rho_1 = (\langle u_0, d_0, p_0, a\rangle\langle c, +, 5, \{o\}\rangle)$

$\rho_2 = (\langle u_0, d_0, p_0, a\rangle\langle c_1, -, 5, \emptyset\rangle)$

$\rho_3 = (\langle u_1, d_0, p_0, a\rangle\langle c_2, +, 6, \{o_1, o_2\}\rangle)$

$\rho_4 = (\langle u_2, d_0, p_0, b\rangle\langle \emptyset, +, 6, \{o_1\}\rangle)$

Now assume that the policy auditor(s) realize that their interpretation of the hospital's informal privacy policy with respect to that rule was incorrect. Hence, the policy needs to be revised with $\rho_0 = (\langle u_1, d_0, p_0, a\rangle\langle \emptyset, -, 7, \emptyset\rangle)$. So, when the above rule is added to the policy, the rules $\rho_1, \rho_2, \rho_3, \rho_4$ become redundant.



	Covering Rule	Covered Rule	Suggestion
1	ρ_0	ρ_1	Remove
1	ρ_0	ρ_2	Remove
2	ρ_0	ρ_3	Raise Priority
2	ρ_0	ρ_4	Raise Priority

However, despite the update operation, while revising the policy we remove the ρ_1 and ρ_2 because intuitively, ρ_1 and the new rule ρ_0 logically contradict each other. In addition, ρ_2 becomes duplicate as there is no situation where ρ_2 is application but not ρ_0. From what we have shown we can conclude that policy revision should endeavor to achieve *minimal change* of the policy.

5.3.3 Changes in a Vocabulary

While policy update and policy revision address the changes of the rules in an EP3P privacy policy, sometimes vocabulary needs to be updated, mainly to reflect the changes in the policy model of an enterprise. For example, the introduction of a new department which has access to the private data adds a new *data user* element to the vocabulary. Similarly, an enterprise may need to perform additional operations on the data, which will lead to the introduction of a new *action* element.

Updating a vocabulary is a trivial process. The main concern when adding or removing an element of the vocabulary is to ensure that the existing rules are derived for the newly added elements. In addition, the rules which use any of the removed vocabulary elements must be removed from the ruleset. For example, given the vocabulary shown in Figure 4.1 if the purpose category *research* is removed from the vocabulary of the policy, the authorization rule $(\langle \mathit{nurses}, \mathit{disclose}, \mathit{disease}, \mathit{research}\rangle\langle \emptyset, \mathit{2}, +, \mathit{notify}\rangle)$ must also be removed. Hence, the main issue is to keep the rules and vocabulary in sync.

Chapter 6

EP3P Policy Management Console

In this chapter, we will describe the *EP3P Policy Management Console (EPMC)*, a policy management tool that we have developed based on the frameworks presented in previous chapters. The purpose of this tool is to assist a policy auditor to *create, update and revise* an EP3P privacy policy while maintaining four properties:

1. To allow policy auditors to specify complex EP3P privacy policies.
2. To ensure that EP3P policies are in accordance with the EP3P specification presented in Chapter 3.
3. To allow policy auditors to realize the consequences of their actions (i.e., adding or removing rules) at specification time.
4. To ensure all the privacy policies are minimal.

The first property ensures that the tool is usable by policy auditors who are unfamiliar with XML language. This can reduce the potential errors made by policy auditors while formalizing privacy policies. By satisfying the second property we ensure that the EP3P policies can then be exchanged with any application or organization, which conforms to the EP3P specification. The third property ensures that the policy auditor is informed about the existing rules in the policy that may be affected by the new rule while updating or revising a policy. Further, it allows more than one policy auditors to edit an EP3P policy, without having a complete knowledge of the rules in the policy. Finally, the fourth property ensures that there are no pairwise-redundant rules in the policy. In other words, it ensures that EP3P policies are minimal.

In the first section, we will briefly describe how *EPMC* achieves all the above mentioned properties/goals. In the second section, we will use *EPMC* to create an EP3P privacy policy. Then, we will show how *EPMC* assists a policy auditor to detect and resolve the redundancies. Finally, we will discuss the future development of *EPMC*.

6.1 System Design

The design of *EPMC* constitute three three main components:

1) Graphical User Interface: The GUI is responsible for collecting and displaying data for creating and updating privacy policies, hence it supports the following functionalities:

- Allows a policy auditor to create/update Policy Vocabulary (i.e., User Category, Data Category, Purpose, Action, Conditions and Obligation).
- Allows a policy auditor to create, update or revise the rules in an EP3P privacy policy. However, the vocabulary for the rules must be specified prior to creating the privacy policy.
- Represents the detected redundancies in the existing privacy policy.
- Allows a policy auditor to resolve the detected redundancies.

The look and feel of the GUI is tailored to help policy auditors overcome any technical difficulties while writing and editing a privacy policy. The GUI is developed using Java Swing (JDK 5.0).

2) Redundancy Detection/Resolution Engine: It is responsible for detecting redundancies and providing suggestions for resolving them. When updating a policy or revising a policy, the engine detects redundancies as the rules are added to the policy by using Algorithm 7. In addition, when an existing EP3P policy is loaded, the engine first examines the policy to ensure that it is minimal by using Algorithm 6.

3) XML Transformer Engine: It is responsible for collecting the data from the GUI and generating the EP3P privacy policy in XML format that is in accordance with EP3P policy specification. It interacts with the redundancy detection/resolution engine as well as the host file system to create and edit (XML) policy files. The engine uses Xerces[1] to parse and extract data from an XML file. The data is then passed to the GUI for further processing.

Since all the created/edited EP3P privacy policies are in accordance with the EP3P specification, they can be used by any access control system that enforces EP3P privacy policies. The high level block diagram in Figure 6.1 illustrates the functionalities that we have described. However note that the diagram does not show all the functionalities of *EPMC*, (interested readers can refer to the appendix B.2 for more technical information regarding the design and the implementation of *EPMC*).

[1]Apache Xerces is a high-performance XML parser. For more information refer to http://xerces.apache.org/

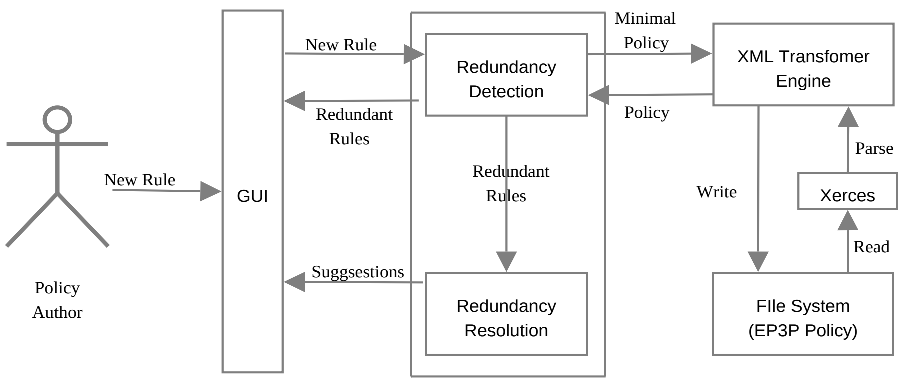


Figure 6.1: Policy Management Console Block Diagram

6.2 Graphical User Interface

The graphical user interface consists of two windows, each window constitutes relevant tabs that are designed to provide an effective access to the relevant tools and information:

- *policy window:* Provides all the necessary functions for the creation/modification of vocabulary or rules of an EP3P privacy policy.
  - *vocabulary tab:* Provides the components required for specifying vocabulary elements such as user, data and purpose hierarchies as well as action, condition, obligation.
  - *rules tab:* It uses the vocabulary that has already been defined and allow the policy auditor to create authorization rules.
- *redundancy management window* Provides the components required for the representing the detected redundancies. In addition, it enables policy auditor to resolve these redundancies.
  - *compaction tab:* Represent the duplicate, contradicting and incomplete rules that were detected by redundancy detection engine. In addition, it represent the resolution suggestions that are provided by redundancy resolution engine for resolving such redundancies.
  - *rectification tab:* Shows the inactive rules that were detected by redundancy detection engine. Similar to compaction tab, it represent the resolution suggestions that are provided by redundancy resolution engine for resolving the detected redundancies.

In the following sections we will demonstrate the functionality of *EPMC* by taking an stepwise approach for formalizing the privacy policy of *Medi-Care Hospital* that were introduced in the previous Chapters.

6.3 Formalizing A Privacy Policy

In this section we assume *Alice* is Chief Privacy Officer (CPO) of *Medi-Care Hospital*. The hospital has many branches in different countries, with thousands of patients and hundreds of staff. Respectively, it has a considerable amount of private data that must be protected against privacy invasive access.

Currently, the hospital has an access control system in place, which satisfies the security requirements and protects records from being accessed by unauthorized users. In addition, the hospital has an informal privacy policy that has been specified in alignment with HIPPA guidelines. However, the staff of the hospital can access private data even though accessing such data may not be required for completing their tasks. Also, sometimes private data are transfered from one branch of the hospital to another branch that has a different privacy policy. The transfered data may then be used for purposes other than what they were originally collected for. In such a setting, the hospital can not ensure that the privacy requirements are satisfied.

To ensure that privacy policy of the hospital is enforced, the hospital is planning to implement an *E-P3P* privacy control system which can ensure that the private data is used in accordance with the hospital's privacy policy. Since Alice is the policy auditor of the hospital, they are responsible for performing two tasks to make the new privacy control system operational.

1. Create an EP3P vocabulary that represents the concepts of the natural language text policy.

2. Specify EP3P policy rules to express the data handling practices that are either allowed or disallowed by the natural language text policy.

In the following sections we will demonstrate how our application can be used to assist Alice in formalizing an EP3P privacy policy that can be enforced by different branches of *Medi-Care Hospital*.

6.3.1 Constructing EP3P Vocabulary

The first step in creating an enforceable privacy policy is to formalize the policy such that it can be enforced. The vocabulary of an EP3P privacy policy forms the privacy model of the enterprise (e.g., *Medi-Care Hospital*). To specify such a model, Alice needs to know what private data is collected and stored by the hospital. Usually this data is divided into several categories that are hierarchically organized to ease the data handling[2]. Figure 6.2 represents the privacy sensitive data that is collected and used within *Medi-Care Hospital*.

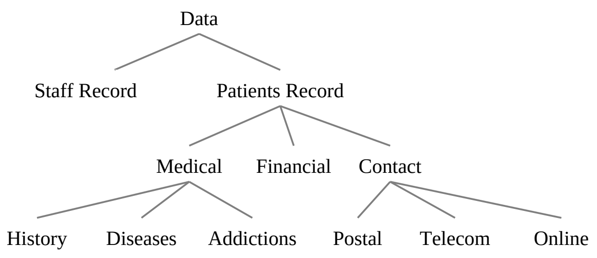


Figure 6.2: Data Categories

Further, Alice identifies who the users are within the hospital that may be either permitted to access, or denied from accessing such data. Figure 6.3 illustrates the users in *Medi-Care Hospital*.

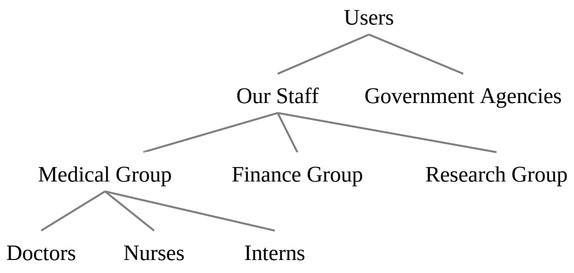


Figure 6.3: User Categories

Alice also identifies the possible purposes for which the users in the hospital may need to access the personal data. Figure 6.4 illustrates the purposes that were identified for *Medi-Care Hospital*.

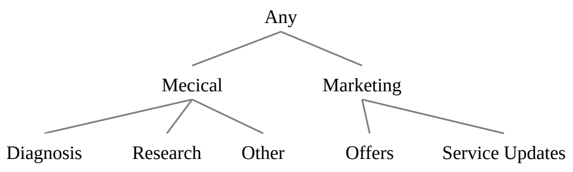


Figure 6.4: Purpose Categories

[2]As we have explained in Chapter 3, such categorizations are usually context dependent. It is out of the scope of our work to discuss how or why such elements are structured in such a way. In real world applications, these categorization are derived from the rigorous data flow analysis that are conducted prior to the implementation of an access control system

Data, users and purposes are three vocabulary elements that are represented as hierarchies in EP3P language. Our application provides an easy to use interface that allows a policy auditor (e.g., Alice) to create hierarchies. They can also delete or modify any of the vocabulary elements an any time. Figure 6.5 illustrates the part of the application that provides this functionality:

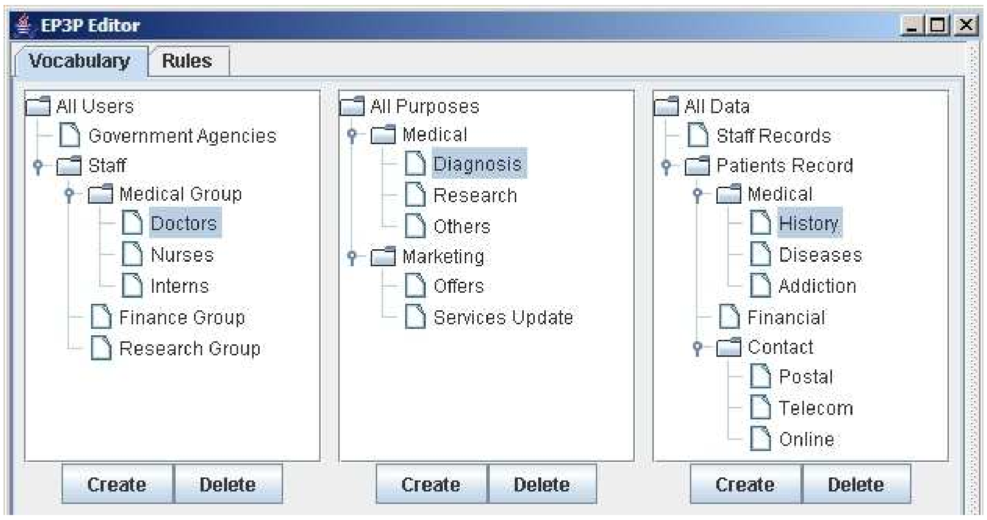


Figure 6.5: Specifying User, Data and Purpose Hierarchies

After specifying data, user and purpose hierarchies, Alice specifies four actions that an authorized user may perform on the identified data:

Action	**Description**
read	To read the record, (i.e., this action can be specialized to distinguish between reading the data on the screen or allowing to take a print. However, we don't make such a distinction)
write	To make changes to the record. This operation concerns the integrity of the data.
delete	To remove the record from the data storage (Database).
disclose	To execute an operation that transfers the protected record to the outside of the enterprise boundaries.

Further, she needs to identify the conditions that may need to be taken into account while providing an authorization. For example, for some access to be permitted the data owner must have explicitly given a consent to the hospital. In addition, sometimes the medical situation of the patient may induce exceptions on the existing privacy rules in order to provide safety. For example, when a patient is in a life threatening condition, their medical record can be accessed without their consent. These conditions are propositions whose evaluation is dependent on the enforcement system. Alice identifies the conditions shown in the following table to be taken into account for by the authorization system in *medi-care hospital*:

Condition	**Description**
minor(d)	returns true nif the owner of the data *d* is less than 13, otherwise the condition is false
optin(d,p)	returns true if the owner of the data *d* is has given the consent for the *d* to be used for purpose *p*.
optout(d,p)	returns true if the owner of the data *d* has not specified that *d* shall not be used for purpose *p*.
guardian(u,d)	returns true if the user *u* is the guardan of the data owner of data *d*.
consent(u,d)	returns true if the user *u* has been given permission by the data owner of data *d* to use *d*.

Finally, Alice identifies the obligations that the hospital may need to satisfy after accessing some of the private records:

Action	**Description**
notify	To inform the data owner about the date, the purpose and the type of action that were performed on their data.
anonymize	To ensure that all the references to the identitiy of the data owner are removed from the record.
delete	To ensure that the record is rmoved from the database within the time frame that is specified by the data owner.

Figure 6.6 shows the part of the *EPMC* that allows policy auditors to specify actions, conditions and obligations, for an EP3P policy.

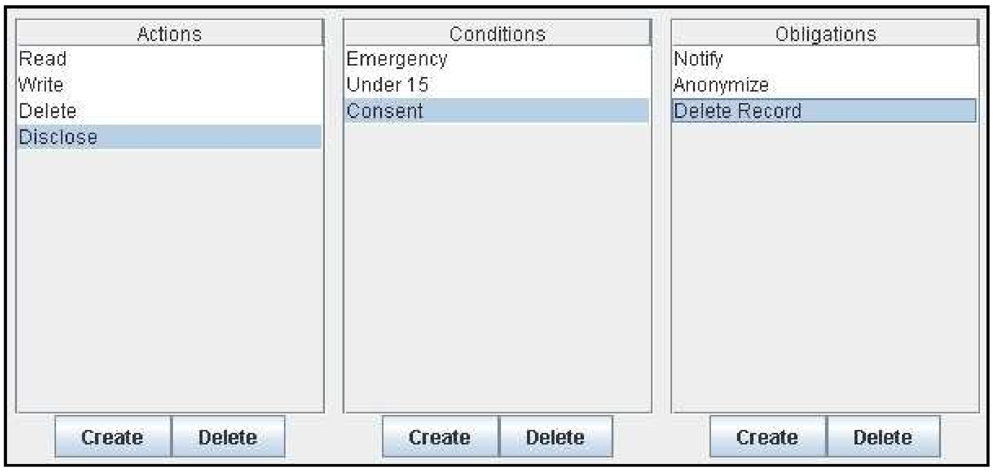


Figure 6.6: Specifying Action, Condition and Obligations

The presented GUI allows policy auditors to specify an EP3P vocabulary using a natural language construct. Further, it ensures that the created vocabulary is valid and well formed with respect to the EP3P language schema. The XML representation of the above vocabulary that Alice has created is shown in the Appendix A.1. Now that the vocabulary of the policy is specified, Alice is in a position to specify privacy rules for the hospital.

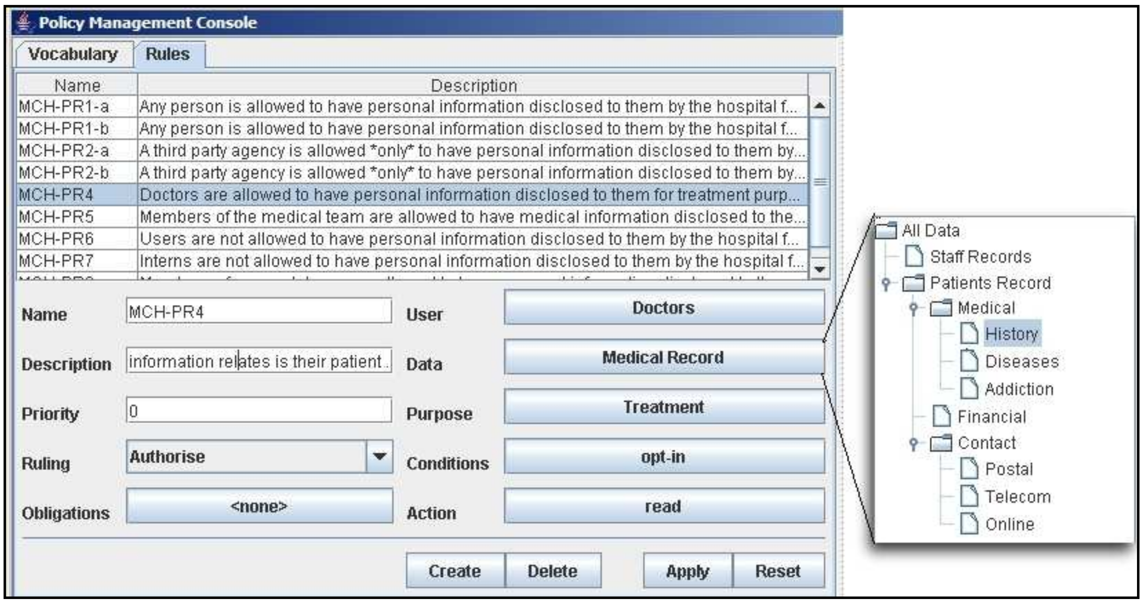
6.3.2 Constructing EP3P Rules

Given the vocabulary that was specified in the previous section Alice shall specify a set of EP3P rules that reflect the informal privacy policy of the *MCH* hospital. A segment of the informal policy is shown below.

Rule ID	**Rule Description**
MCH-PR1	Any person is allowed to have personal information disclosed to them by the hospital for any purpose if the recipient of the disclosure is next of kin and the individual is ill, or deceased or a minor.
MCH-PR2	A third party agency is allowed *only* to have personal information disclosed to them by the hospital for marketing purpose if the person to whom the information relates has approved the disclosure.
MCH-PR3	Doctors are allowed to have personal information disclosed to them for treatment purposes if the person to whom the information relates is their patient (i.e., by default patients optin to such access).
MCH-PR4	Members of the medical team are allowed to have medical information disclosed to them for treatment purposes. However, for all unconsented access, the patient must be notified about the usage.
MCH-PR5	Users are not allowed to have personal information disclosed to them by the hospital for any purpose, unless other rules in the policy are satisfied that allow such a disclosure.
MCH-PR6	Interns are not allowed to have personal information disclosed to them by the hospital for any purpose.
MCH-PR7	Members of research team are allowed to have personal information disclosed to them by the hospital for research purposes, when there is an epidemic. However, patient's record must be anonymized.

As we have discussed in previous chapters, formalizing an informal policy is a non-trivial task because it is very common to write a set of rules that does not reflect the informal policy.

Here we will assume that Alice has decided on a set of authorization rules. Our concern in this section is to show how our tool enables Alice to write these rules using a natural constructs instead of directly manipulating XML files. Following rules constitute the informal privacy policy of the *MCH* hospital. To create an XML EP3P policy, Alice can use the GUI shown in Figure 6.7.



Name	Description
MCH-PR1-a	Any person is allowed to have personal information disclosed to them by the hospital f...
MCH-PR1-b	Any person is allowed to have personal information disclosed to them by the hospital f...
MCH-PR2-a	A third party agency is allowed *only* to have personal information disclosed to them by...
MCH-PR2-b	A third party agency is allowed *only* to have personal information disclosed to them by...
MCH-PR4	Doctors are allowed to have personal information disclosed to them for treatment purp...
MCH-PR5	Members of the medical team are allowed to have medical information disclosed to the...
MCH-PR6	Users are not allowed to have personal information disclosed to them by the hospital f...
MCH-PR7	Interns are not allowed to have personal information disclosed to them by the hospital f...

Name	MCH-PR4	User	Doctors
Description	information relates is their patient .	Data	Medical Record
Priority	0	Purpose	Treatment
Ruling	Authorise	Conditions	opt-in
Obligations	<none>	Action	read

Figure 6.7: Specifying Authorization Rules

In the "Rules" tab of the application there are three *text fields*[3] that allow Alice to specify a name, an informal description and a priority for the selected rule. In addition to text fields, there are six *drop down menus* [4](e.g., users, condition, obligations, etc.) that will be populated with the vocabulary of the policy that was specified using the "Vocabulary" tab. Drop down menus can not be modified, this will restrict all the elements of the rules to be taken from the existing vocabulary.

When Alice has specified all the authorization rules, in the case there is no redundancy, she will be given an option to *confirm* the policy. By doing so, vocabulary and authorization rules will be written into two XML files, respectively *vocab.xml* and *policy.xml*. These XML files form the enforceable privacy policy of the hospital.

The following table represents the set of rules that Alice has specified as the EP3P policy of the hospital. The first column shows the code[5] that we have devoted to the informal policy rules. The second column shows the EP3P rules using the syntax that we used in this thesis.

[3]A text field is a basic text control that lets the user enter a small amount of text.
[4]A drop down menu allows a user to select a value from the list.
[5]These codes are only for readability purposes in this work.

Rule ID	Formal Rule
MCH-PR1-a	$(\langle \textit{next of kin}, \textit{patient record}, \textit{read}, \textit{any}\rangle\langle \textit{emergency}, 1, +, \emptyset\rangle)$
MCH-PR1-b	$(\langle \textit{next of kin}, \textit{patient record}, \textit{read}, \textit{any}\rangle\langle \textit{minor}, 1, +, \emptyset\rangle)$
MCH-PR2-a	$(\langle \textit{other agencies}, \textit{contact}, \textit{read}, \textit{marketing}\rangle\langle \textit{optin}), 1, +, \emptyset\rangle)$
MCH-PR2-b	$(\langle \textit{other agencies}, \textit{data}, \textit{read}, \textit{any}\rangle\langle \emptyset), 1, -, \emptyset\rangle)$
MCH-PR3	$(\langle \textit{doctors}, \textit{read}, \textit{medical}, \textit{treatment}\rangle\langle \textit{optin}, 1, +, \emptyset\rangle)$
MCH-PR4	$(\langle \textit{medical team}, \textit{read}, \textit{medical}, \textit{medical}\rangle\langle \emptyset, 1, +, \textit{notify}\rangle)$
MCH-PR5	$(\langle \textit{users}, \textit{read}, \textit{data}, \textit{any}\rangle\langle \emptyset, 1, -, \emptyset\rangle)$
MCH-PR6	$(\langle \textit{interns}, \textit{read}, \textit{medical}, \textit{treatment}\rangle\langle \emptyset, 1, -, \emptyset\rangle)$
MCH-PR7	$(\langle \textit{researchgroup}, \textit{read}, \textit{personal}, \textit{research}\rangle\langle \textit{epidemic}, 1, +, \textit{anonymize}\rangle)$

Thus far we have showed that our application simplifies formalizing a privacy policy by two means. First it provides a user interface that allows policy auditors to specify a policy vocabulary as well as policy rules without directly manipulating XML files. Second, it ensures that the constructed EP3P policies are in alignment with EP3P XML schema. Appendix A.1 and A.2 respectively show vocabulary and authorization rules of *Medi-care Hospital's* EP3P privacy policy.

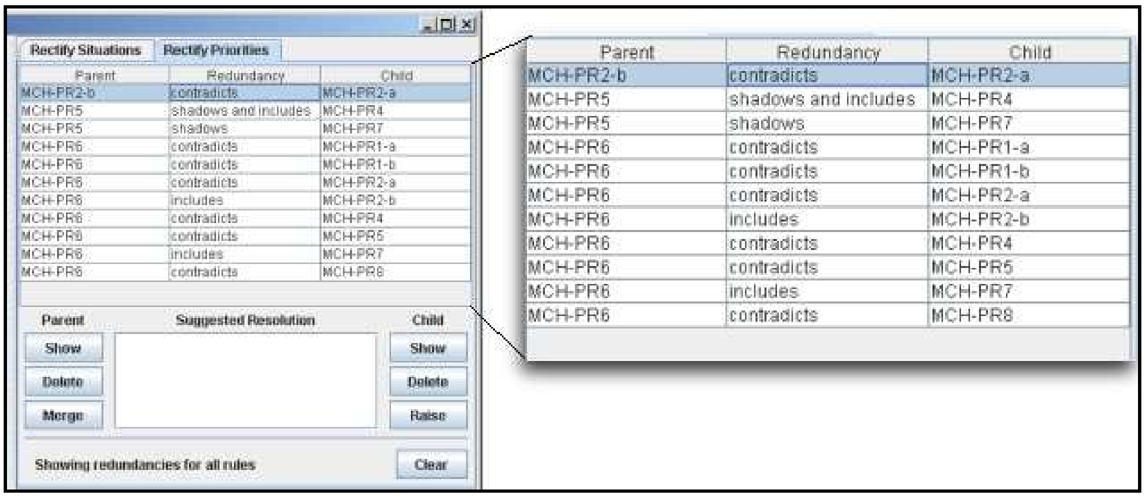
6.4 Redundancy Detection

In the previous section we skipped many details regarding how our application assists Alice to ensure the EP3P policy that she specified is minimal. In other words we were not concerned about the consequences of the co-existence of several rules on the outcome (authorization response) of the policy. As we have discussed in Chapter 4, such a knowledge during policy specification time enables a policy auditor (e.g., Alice) to determine what rules need to be modified, removed or merged while update or revising a policy.

In this section we will show how our application employs the redundancy detection framework to allow policy auditors to realize how one or more rules make another rule(s) redundant. Given such a functionality, a policy auditor is not required to have a complete knowledge of the rules in the policy in order perform policy management operations (e.g., update, revision, merge). In addition, this allows policy management tasks to be performed by more than one policy auditor.

Recall that in Chapter 4 we divided redundancies into three classes: includes, shadows and contradicts. We follow the same convention in the implementation of *EPMC*. Whenever a rule changes or a new rule is added to the rule table (shown in Figure 6.7), the redundancy detection engine evaluates the existing covering rules. In the case where a redundant rule is found, it is shown along with the rules that makes it redundant as well its type of redundancy.

Let us consider the EP3P rules that Alice has specified for the *MCH* privacy policy. After confirming the changes (i.e., using the apply button in rule window) the following redundancies are detected and shown in the redundancy detection table (Figure 6.8).



Parent	Redundancy	Child
MCH-PR2-b	contradicts	MCH-PR2-a
MCH-PR5	shadows and includes	MCH-PR4
MCH-PR5	shadows	MCH-PR7
MCH-PR6	contradicts	MCH-PR1-a
MCH-PR6	contradicts	MCH-PR1-b
MCH-PR6	contradicts	MCH-PR2-a
MCH-PR6	includes	MCH-PR2-b
MCH-PR6	contradicts	MCH-PR4
MCH-PR6	contradicts	MCH-PR5
MCH-PR6	includes	MCH-PR7
MCH-PR6	contradicts	MCH-PR8

Figure 6.8: Redundancies

As you can observe Alice can easily determine which rules are redundant, why they are redundant, and what are the other rules that make them redundant. For example, the second row of the table shows that *MCH-PR5* shadows and includes *MCH-PR4*. By referring to the actual rules (i.e., clicking on the *show* button), Alice can observe that in theory, *MCH-PR4* allows patient's doctor to read their records without being obliged to notify the patient. However, *MCH-PR4* is made redundant by *MCH-PR5* that has a higher priority and is applicable in all the situations that *MCH-PR4* is applicable.

In the following sections we will show how the application can assist policy auditors by providing resolution suggestions and assist them in resolving the identified redundancies.

6.5 Redundancy Resolution

After identifying redundant rules, the redundancy resolution module assists Alice in resolving redundancies. There are two options regarding *when* to resolve the detected redundancies.

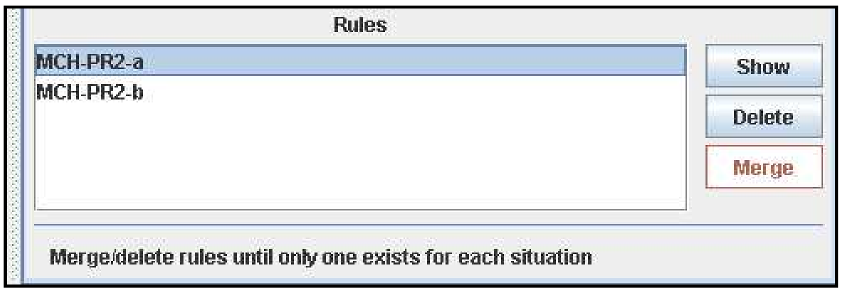
- They can be resolved as they occur (at the time of adding a new rule).
- They can be dealt with when all the rules have been specified.

These options are provided to avoid interrupting the policy auditor, hence improving the usability of the application.

The redundancy resolution module is based on the resolution framework that was introduced in the previous chapter. The application divides the resolution procedure into two stages (i.e., they are represented as two tabs in the resolution window): *Policy Compaction* and *Policy Rectification*.

In the policy compaction module Alice is provided with a list (a table) of rules that are either *duplicate, incomplete* or *contradicting*. By selecting any of the rules, all the redundancy relations related to the selected rule will be shown in another list (Figure 6.9).

Alice has two options on *how* to resolve these redundancies: *automatic* or *manual*. By choosing the automatic option she allows *EPMC* to decide on how to resolve the redundancies. The decision that *EPMC* takes is based on the default resolution suggestions that were described in the previous chapter. However, in the case of choosing the manual option, Alice will be provided with a list of redundant rules and options highlighting possible resolution action. Figure 6.9 illustrates a segment of policy compaction tab. As you can see the *merge* bottom is disabled. This is because merging the obligations of these rules (contradicting rules) does not resolve the redundancy.



Merge/delete rules until only one exists for each situation

Figure 6.9: Redundancies

The policy rectification module is also designed with similar functionalities. However, there, Alice is provided with a list of *inactive* rules in the policy. In contract to the policy compaction module where Alice were able to resolve the redundancies manually, in this module she is not give the option of manually modifying priorities. Hence, given the existing inactive rules, Alice can either delete them or ask *EPMC* to resolve the redundancies. In the case where latter is chosen, the application assigns a priority to the inactive rules such that it resolve all the redundancies. By using the efficient rectification procedure that were shown in the previous chapter *EPMC* can inherit all the properties that were proved for that procedure.

By performing these two resolution procedures Alice can resolve all the redundancies that were identified and ensure that the EP3P of *Medi-care Hospital* is minimal.

Chapter 7

Conclusion

The main contribution of this thesis was the identification of the problem of *redundancy(s)* in EP3P privacy policy(s) and the providing of a method for solving this problem. The *detection* and *resolution* of redundant rules is an area in which no prior work has been done thus far. This task is becoming more important for the following reasons: 1) The number of enterprises that seek to enforce privacy policies is increasing. 2) These enterprises need to exchange their privacy policies. 3) The management of a privacy policy is not necessarily centralized, as an enterprise may have multiple CPO's. 4) In any medium sized enterprise there may be tens or hundreds of rules. 5) As privacy guidelines or business rules are refined, privacy policies must also change. 6) There is no systematic approach for formalizing/managing privacy policies. 7) Redundant rules may misdirect policy auditors while writing policies, because although they exist in the policy, they can never be used. 8) In addition, such rules reduce the efficiency of policy transfers and operations such as merge, refinement, and update.

Hence, preventing, detecting and resolving redundancies as they are introduced to a policy has become an important problem. Redundancies can be prevented if, at rule specification-time, the consequences of adding the rule to the policy can be determined. The detection and resolution of redundancies within an existing policy is important and as such, once a redundancy is detected, resolution is the approach that must be taken. Hence, we proposed a redundancy detection and resolution framework that evaluates the rules that have a chance of making one another redundant.

As a way of solving the redundancy detection problem, we formally defined three different concepts of redundancy: *Redundancy*, *Absolute Redundancy* and *Pair-wise Redundancy*. The first one captures a common notion of redundancy: a rule is redundant if removing it from the policy does not change the authorization response of the policy. The second is that a redundant rule always remains redundant regardless of how many other rules are added to the policy. This ensures that non-redundant rules are not mistakenly labeled as redundant. In addition, it ensures that redundancy detection is not dependent on the order by which the rules are added to the policy. Hence, redundancies can be detected incrementally as they are being added to the policy. In the first two notions of redundancy it is not clear whether a rule is made redundant by more than one rule. This lead us to the third notion of redundancy that only compares a pair of rules.

We have classified pairwise-redundant rules into three general categories, *include, contradict* and *shadow.* First we say one rule includes another rule if it already provides the same outcome (authorization decision) and fires for all the access requests where the other rule fires. Second, we say a rule shadows another rule if that rule has strictly higher priority and fires in the same situations. Third, we say a "deny" rule contradicts an "authorize" rule if the "deny" rule has an equal or higher priority and fires in the same situations.

We mathematically proved that by detecting pairwise-redundant rules we would be in a position to detect all the redundancies in an EP3P privacy policy, except for the one instance that has been shown in the thesis previously. We developed an algorithm based on that proof that allows us to detect all the pair-wise redundant rules in an EP3P privacy policy and proved the algorithm to be sound and complete.

We took the concept of redundancy even further and provided a redundancy resolution technique that can assist a policy auditor in resolving the detected redundancies. Our main concern was to provide a resolution method that is intuitive and as close as possible to a decision that a human author of the policy may make when confronted with the redundancy. To do that, we took into account the reasons that a rule may become redundant. We took an information theory approach and evaluated the rules based on the authorization information they provide to the policy. In other words, we differentiate between redundant rules by determining: 1) Whether a rule is redundant because the authorization decision that it provides for a situation has already been specified by another rule. 2) Whether there is a rule that contradicts this rule. 3) Whether the rule has been made redundant because the EP3P evaluation engine can never reach a rule due to an improper priority assignment. 4) In addition, we identified the rules that are incomplete and need to be merged.

These classifications allow us to decide whether a redundant rule shall be *removed*, whether to *raise its priority*, such that it is not redundant any more, or whether to *merge* the authorization decision of the two rules. In addition, we have investigated the problem of conflicting resolution suggestions. This occurs when a rule is made redundant by more than one rule and the resolution suggestions are different. We continued by providing a procedure we deemed to be the most efficient way for resolving all the detected redundancies in an EP3P policy.

As a proof of concept, we presented an integrated implementation of a *Policy Management Console* (*EPMC*) based on the EP3P language. We showed how *EPMC* provides a graphical editor that allows policy auditors to use natural language constructs for specifying policies. *EPMC* also has an engine for translating policies into XML formats used by policy enforcement systems (e.g., *E-P3P*). It allows management operations such as *update, revision* and *evaluation* of already existing policies to ensure that there are no redundant rules in the policy. The application assists policy auditors in resolving the redundancies that may occur during any of the management operations (i.e., update, revision and merge)

Throughout this thesis, we have mostly used examples relating to the privacy policies of a hospital to describe the concepts of our redundancy detection/resolution framework. However, while the fine-grained concepts that were described are based on the EP3P language, they are independent of the policies written using EP3P language.

Chapter 8

Future Work

The work presented in this thesis has been motivated by the need to support privacy policy specification for enterprises by using the EP3P language. In this chapter, we examine the limitations of our proposed framework and discuss some of the open issues that offer directions for future investigations.

The EP3P language that were used in this thesis has limitations for expressing conditions and obligations. The language assumes that obligations and conditions are atomic. It only allows an "and" relation to be used in a condition or obligation of a rule. This restriction significantly reduces the expressiveness that can be provided when the "implication" relation is used between the condition elements or obligation elements. Given such a restriction, we can not determine, for example, "the obligation to remove data within 10 days implies that the data is deleted within 20 days".

Future work includes extending the redundancy detection and resolution frameworks that were presented in this thesis such that we can detect the redundancies of a more expressive language such as EPAL (i.e., the language that allows implication relation for representing conditions and obligations). To make such an improvement to our frameworks we need to modify two definitions, Definition 13 ("covers") as well as the Definition 15 "includes". The covering relation is affected because it uses the condition of the rules to determine if a rule has a potential to make another redundant. The definition of include must be enhanced because currently it only considers a subset relation of obligations to determine if a covering rule includes another.

In this thesis we have only dealt with the problem of redundant rules, hence, our framework can not determine if one policy refines[1] another, it can only reduce the number of rules in the policy and improves the efficiency of the existing policy refinement techniques. In the future we would like to investigate the possibility of determining the refinement of one policy by another, using the pair-wise comparison of rules in the policies.

As we have mentioned in Chapter 2, it is important for enterprises to determine if their privacy promises are in alignment (synchronized) with their privacy practices. We already know that removing redundancies can improve the efficiency of these techniques. However,

[1]For more information on refinement refer to Chapter 2.

we plan to investigate whether removing redundancies from an EP3P policy improves the effectiveness of the existing techniques for evaluating the alignment of EP3P policy and P3P policy. In addition, we would like to use our redundancy detection framework to perform such an evaluation.

Finally, future work could also include an empirical testing of the application that is based on the proposed frameworks. This would demonstrate that redundancy detection and resolution frameworks perform well in practice as well as in theory. This would ideally be done with a real size enterprise with a privacy policy consisting of many rules.

Bibliography

[1] D. Agrawal, J. Giles, K.-W. Lee, and J. Lobo. Policy ratification. In *Policy*, pages 223–232, 2005.

[2] R. Agrawal, R. Cochrane, and B. G. Lindsay. On maintaining priorities in a production rule system. In *VLDB*, pages 479–487, 1991.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. An xpath based preference language for p3p. In *12th Int'l World Wide Web Conf.*, 2003.

[4] E. S. Al-Shaer and H. H. Hamed. Discovery of policy anomalies in distributed firewalls. In *IEEE INFOCOM*, pages VOL 4, pages 2605–2616. Institute of electrical engineers (IEEE), 2004.

[5] R. Alcock, D. Lee, and P. Forseth. Matters related to the review of the office of the privacy commissioner. Technical report, House Of Commons Canada, 2003.

[6] A. Anderson. A comparison of two privacy policy languages: Epal and xacml. Technical Report TR-2005-145, Sun Micosystems Inc., 2005.

[7] A. Antón, J. Earp, D. Bolchini, C. J. Q. He, and W. Stufflebeam. The lack of clarity in financial privacy policies and the need for standardization. In *IEEE Security & Privacy*, pages 36–45, 2004.

[8] A. Antón, J. Earp, and A. Reese. Goal mining to examine health care privacy policies. Technical report, 2001.

[9] P. Ashley, S. Hada, G. Karjoth, and M. Schunter. E-p3p privacy policies and privacy authorization. In *WPES '02: Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society*, pages 103–109, 2002.

[10] P. Ashley, C. Powers, and M. Schunter. From privacy promises to privacy management: a new approach for enforcing privacy throughout an enterprise. In *NSPW '02: Proceedings of the 2002 workshop on New security paradigms*, pages 43–50, New York, NY, USA, 2002. ACM Press.

[11] M. Backes, M. Durmuth, and G. Karjoth. Unification in privacy policy evaluation - translating epal into prolog. In *Fith International Workshop on Policies for Distributed Systems and Networks (POLICY*. IEEE Computer Society, 2004.

[12] M. Backes, G. Karjoth, W. Bagga, and M. Schunter. Efficient comparison of enterprise privacy policies. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 375–382, 2004.

[13] M. Backes, B. Pfitzmann, and M. Schunter. A toolkit for managing enterprise privacy policies. In *ESORICS*, pages 162–180, 2003.

[14] Y. Bai. *On Formal Specification of Authorization Policies and their Transformation.* PhD thesis, Univeristy of Western Sydney, 2000.

[15] A. Barth and J. C. Mitchell. Enterprise privacy promises and enforcement. In *WITS '05: Proceedings of the 2005 workshop on issues in the theory of security*, pages 58–66, New York, NY, USA, 2005. ACM Press.

[16] A. Barth, J. C. Mitchell, and J. Rosenstein. Conflict and combination in privacy policy languages. In *WPES '04: Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 45–46, 2004.

[17] E. Bertino, B. Catani, E. Ferrari, and P. Perlaska. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 2003.

[18] D. Bolchini, Q. He, A. Antn, and W. Stufflebeam. I need it now: Improving website usability by contextualizing privacy policies. In *The 4th International Conference on Web Engineering (ICWE 2004)*, pages 28–30, Munich, Germany, 2004.

[19] P. Bonatti, S. D. C. di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, 5(1):1–35, 2002.

[20] T. D. Breaux and A. I. Antón. Analyzing goal semantics for rights, permissions, and obligations. In *RE*, pages 177–188, 2005.

[21] S. Byers, Cranor, and D. L., Kormann. Automated analysis of p3p-enabled web sites. In *ICEC*, Pittsburn, PA, 2003.

[22] S. Byers, L. F. Cranor, D. P. Kormann, and P. D. McDaniel. Searching for privacy: Design and implementation of a p3p-enabled search engine. In *Privacy Enhancing Technologies*, pages 314–328, 2004.

[23] L. Cranor, M. Langheinrich, M. Marchiori, and M. Presler-Marshall. The platform for privacy preferences 1.0 (p3p1.0) specification. Technical report, 2002.

[24] L. F. Cranor. Requirements for a p3p query language. In *QL*, 1998.

[25] L. F. Cranor, M. Arjula, and P. Guduru. Use of a p3p user agent by early adopters. In *WPES*, pages 1–10, 2002.

[26] L. F. Cranor, M. Langheinrich, and M. Marchiori. A p3p preference exchange language 1.0 (appel1.0). In *W3C Working Draft*, 2002.

[27] F. Cuppens and C. Saurel. Specifying a security policy: a case study. In *CSFW*, page 123, 1996.

[28] N. Damianou, A. K. Bandara, M. Sloman, and E. C. Lupu. A survey of policy specification approaches. *Lecture Notes in Computer Science*, 2002.

[29] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 2001.

[30] S. Fischer-Hbner and A. Ott. From a formal privacy model to its implementation. In *Proc. of the 21st National Information Systems Security Conference*. Oct. 5-8, 1998.

[31] S. Fischer-Hubner. *IT-security and privacy: design and use of privacy-enhancing security mechanisms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[32] A. Graham, T. Radhakrishnan, and C. Grossner. Incremental validation of policy-based systems. In *Proceedings of the fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*. IEEE Computer Society, 2004.

[33] G. Hogben. Suggestions for long term changes to p3p. Position paper for W3C Workshop on the Long Term Future of P3P.

[34] G. Hogben. A technical analysis of problems with p3p v1.0 and possible solutions. Available at http://www.w3.org/2002/p3p-ws/pp/jrc.html., 2002.

[35] S. Jajodia. Flexible support for multiple access control policies. *ACM Transactions on Dababase Systems*, 26:214–260, 2001.

[36] S. Jajodia, M. Kudo, and V. Subrahmanian. Provisional authorization. 2000.

[37] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 31, Washington, DC, USA, 1997. IEEE Computer Society.

[38] G. Karjoth, M. Schunter, and E. V. Herreweghen. Translating privacy practices into privacy promises - how to promise what you can keep. In *Proceeding of the forth International workshop on policies for Distributed Systems and Networks*, 2003.

[39] G. Karjoth, M. Schunter, and M. Waidner. The platform for enterprise privacy practices - privacy-enabled management of customer data‘. In *In 2nd Workshop on Privacy Enhancing Technologies (PET)*. ACM Press, 2002.

[40] G. Karjoth, M. Schunter, and M. Waidner. Privacy-enabled services for enterprises. In *DEXA Workshops*, pages 483–487, 2002.

[41] M. Langheinrich. A privacy awareness system for ubiquitous computing environments. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 237–245, London, UK, 2002. Springer-Verlag.

[42] J. Lobo, R. Bhatia, and S. A. Naqvi. A policy description language. In *AAAI/IAAI*, pages 291–298, 1999.

[43] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, November/December 1999.

[44] C. M. M., B. A., and G. C. Power prototoype: Towards integrated policy-based management. 1999.

[45] P. B. Marco Cassasa Mont, Siani Piarson. An adaptive privacy management system for data repositories. 2004.

[46] R. S. Michael Backes, Markus Durmuth. An algebra for composing enterprise privacy policies. ESORCS, pages 33–52, Berlin Heidelberg, 2004. Springer-Verlag.

[47] J. D. Moffett and M. S. Sloman. Policy conflict analysis in distributed system management. *Journal of Organizational Computing*, 4(1):1–22, 1994.

[48] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[49] C. Powers, S. Adler, and B. Wishart. Epal translation of the the freedom of information and protection of privacy act. In *Tivoli Sofware*, March 11, 2004.

[50] C. S. Powers, P. Ashley, and M. Schunter. Privacy promises, access control, and privacy management. In *Proc. of the 3rd International Symposium on Electronic Commerce*, pages 13–21, IEEE, 2002.

[51] Research Report 3485, IBM Research. *Enterprise Privacy Authorization Language (EPAL)*, 2003.

[52] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. Security policy consistency. *CoRR*, cs.LO/0006045, 2000.

[53] M. Schunter, E. V. Herreweghen, and M. Waidner. Expressive privacy promises - how to improve the platform for privacy preferences (p3p). In *Position paper for W3C Workshop on the Future of P3P*, 2002.

[54] W. H. Stufflebeam, A. I. Antón, Q. He, and N. Jain. Specifying privacy policies with p3p and epal: lessons learned. In *WPES*, page 35, 2004.

[55] S. Warren and L. Brandeis. *The Right To Privacy.* Harward Law Review, 1890.

[56] A. F. Westin. *Privacy and freedom.* London:Bodley Head, 1970.

[57] D. Wijesekera and S. Jajodia. Policy algebras for access control the predicate case. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 171–180, New York, NY, USA, 2002. ACM Press.

[58] T. Yu, N. Li, and A. I. Anton. A formal semantics for p3p. ACM Workshop on Secure Web Services, 2004.

Appendix A

Medi-Care Hospital's EP3P Policy

In the following section we will show the XML version for *Medi-Care Hospital* privacy policy that is divided into vocabulary and rules.

A.1 EP3P Vocabulary

```
<?xml version="1.0" encoding="UTF-8"?>
<ep3p-vocabulary>
    <vocabulary-information id="Vocabulary">
        <short-description language="en"/>
        <long-description language="en"/>
        <issuer>
            <name>Medi-Care Hospital (MCH)</name>
            <Email>Alice@medi-care.com.au</Email>
            <address/>
            <country/>
        </issuer>
        <location/>
        <version-info
                valid-until="2006-12-12"
     last-modified="2005-12-12"
     revision-number="1"
                start-date="2004-12-12"
        superseded-by-revision="1"/>
    </vocabulary-information>
    <user-category id="Users" parent="">
        <short-description language="en">
            All the users that may somehow use
    the stored private data.
</short-description>
        <long-description language="en">
</long-description>
    </user-category>
    <user-category id="Our staff" parent="Users">
        <short-description language="en">
    Medi-care hospital employees.
</short-description>
```

```
        <long-description language="en"/>
    </user-category>
    <user-category id="Government Agencies" parent="Users">
        <short-description language="en">
    Organizations that may need to access patient’s
    information for legal reasons.
</short-description>
        <long-description language="en"/>
    </user-category>
    <user-category id="Medical Group" parent="Our staff">
        <short-description language="en">
    All the staff that are somehow involved in the
    process of patient’s treatment.
</short-description>
        <long-description language="en"/>
    </user-category>
    <user-category id="Finance Group" parent="Our staff">
        <short-description language="en">
    All the staff that are somehow involved in the
    process of patient’s financial handling.
</short-description>
        <long-description language="en"/>
    </user-category>
    <user-category id="Research Group" parent="Our staff">
        <short-description language="en">
    All the staff/academics that are somehow involved in
    research activities.
</short-description>
        <long-description language="en"/>
    </user-category>
    <user-category id="Doctors" parent="Medical Group">
        <short-description language="en"/>
        <long-description language="en"/>
    </user-category>
    <user-category id="Nurses" parent="Medical Group">
        <short-description language="en"/>
        <long-description language="en"/>
    </user-category>
    <user-category id="Interns" parent="Medical Group">
        <short-description language="en"/>
        <long-description language="en"/>
    </user-category>
    <data-category id="Data" parent="">
        <short-description language="en"/>
        <long-description language="en"/>
    </data-category>
```

```
<data-category id="Staff Record" parent="Data">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
<data-category id="Patient Record" parent="Data">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
<data-category id="Medical" parent="Patients Record">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
<data-category id="Financial" parent="Patients Record">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
<data-category id="Contact" parent="Patients Record">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
<data-category id="Hisorory" parent="Medical">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
<data-category id="Diseases" parent="Medical">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
<data-category id="Addictions" parent="Medical">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
<data-category id="Postal" parent="Contact">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>

<data-category id="Telecom" parent="Contact">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
<data-category id="Online" parent="Contact">
    <short-description language="en"/>
    <long-description language="en"/>
</data-category>
```

```
    <purpose id="Any" parent="">
        <short-description language="en"/>
        <long-description language="en"/>
    </purpose>
    <purpose id="Medical" parent="any">
        <short-description language="en"/>
        <long-description language="en"/>
    </purpose>
    <purpose id="Marketing" parent="any">
        <short-description language="en"/>
        <long-description language="en"/>
    </purpose>
    <purpose id="Diagnosis" parent="Medical">
        <short-description language="en"/>
        <long-description language="en"/>
    </purpose>
    <purpose id="Research" parent="Medical">
        <short-description language="en"/>
        <long-description language="en"/>
    </purpose>
    <purpose id="Other" parent="Medical">
        <short-description language="en"/>
        <long-description language="en"/>
    </purpose>
    <purpose id="Offers" parent="Marketing">
        <short-description language="en"/>
        <long-description language="en"/>
    </purpose>
    <purpose id="Service Updates" parent="Marketing">
        <short-description language="en"/>
        <long-description language="en"/>
    </purpose>
    <action id="read">
        <short-description language="en">
</short-description>
        <long-description language="en">
</long-description>
    </action>
    <action id="disclose">
        <short-description language="en"/>
        <long-description language="en"/>
    </action>
    <action id="write">
        <short-description language="en"/>
        <long-description language="en"/>
    </action>
    <condition id="emergency">
        <short-description language="en"/>
        <long-description language="en"/>
    </condition>
    <condition id ="age">
        <short-description/>
<long-description/>
    </conditoin>
    <condition id ="consent">
        <short-description/>
<long-description/>
    </conditoin>
    <obligation id="notify">
        <short-description language="en"/>
        <long-description language="en"/>
    </obligation>
    <obligation id="delete">
        <short-description language="en"/>
        <long-description language="en"/>
    </obligation>
    <obligation id="anonymize">
        <short-description language="en"/>
        <long-description language="en"/>
    </obligation>
</ep3p-vocabulary>
```

A.2 EP3P Rules

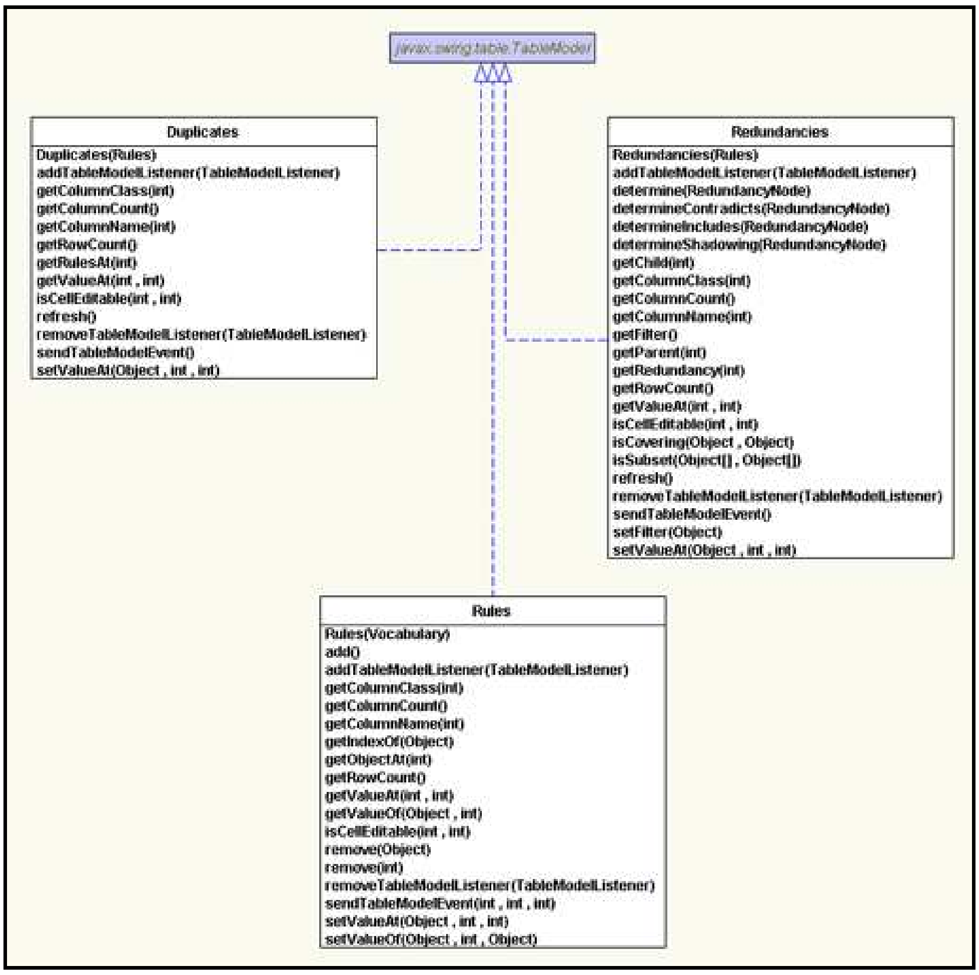
```
<?xml version="1.0" encoding="UTF-8"?>
<ep3p-policy default-ruling="deny" version="1.0">
<ep3p-vocabulary-ref id=""
    location="D:\documents\research\applications\vocabulary.xml" revision="1"/>
    <rule id="rule1" priority="low" ruling="allow">
        <short-description language="en">No one from the financial team
can read patient's infomation</short-description>
        <long-description language="en"/>
        <user-category refid="medicalTeam"/>
        <data-category refid="medicalRecord"/>
        <purpose refid="any"/>
        <action refid="read"/>
        <condition refid="emergencyCondition"/>
        <obligation refid="notify"/>
    </rule>
    <rule id="rule2" priority="low" ruling="deny">
        <short-description language="en">Rule 2</short-description>
        <long-description language="en"/>
        <user-category refid="medicalTeam"/>
        <data-category refid="medicalRecord"/>
        <purpose refid="any"/>
        <action refid="read"/>
        <condition refid="emergencyCondition"/>
        <condition refid="consent"/>
        <obligation refid="notify"/>
    </rule>
    <rule id="Rule3" priority="low" ruling="allow">
        <short-description language="en"/>
        <long-description language="en"/>
        <user-category refid="medicalTeam"/>
        <data-category refid="medicalRecord"/>
        <purpose refid="medicalpurposes"/>
        <action refid="read"/>
        <condition refid="emergencyCondition"/>
        <obligation refid="notify"/>
    </rule>
........
</ep3p-policy>
```

Appendix B

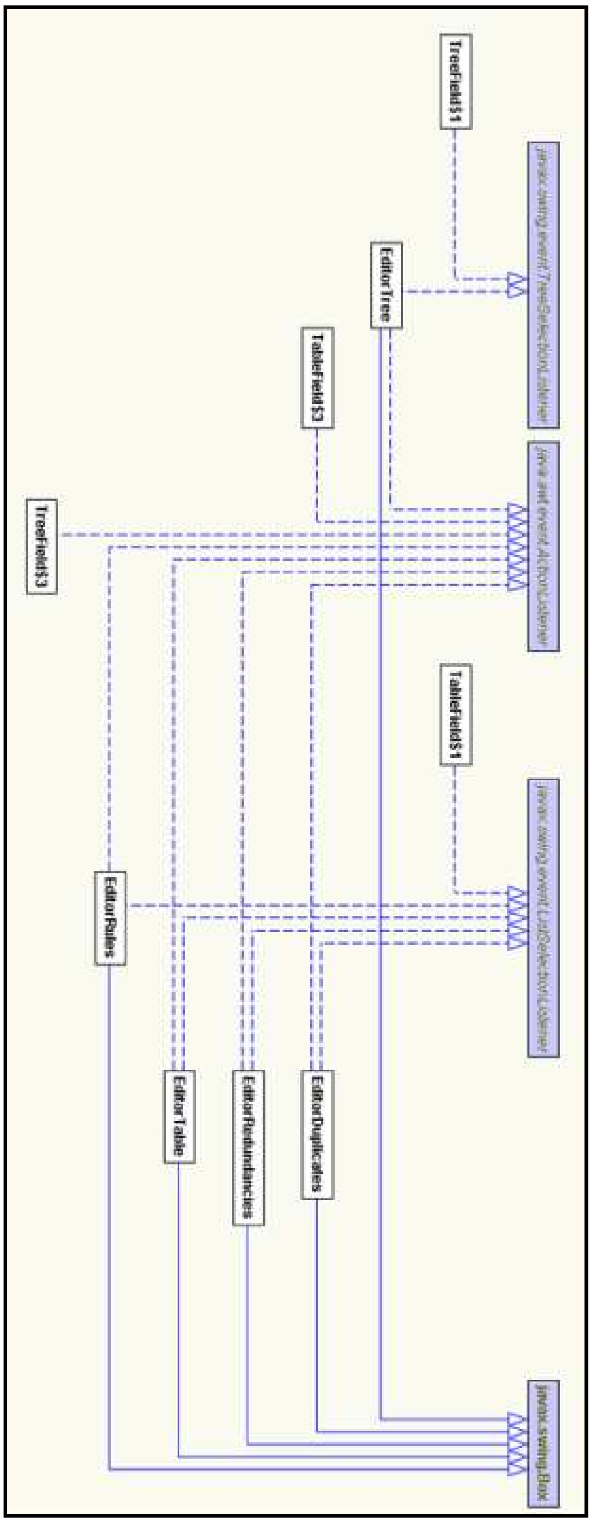
Medi-Care Hospital Application

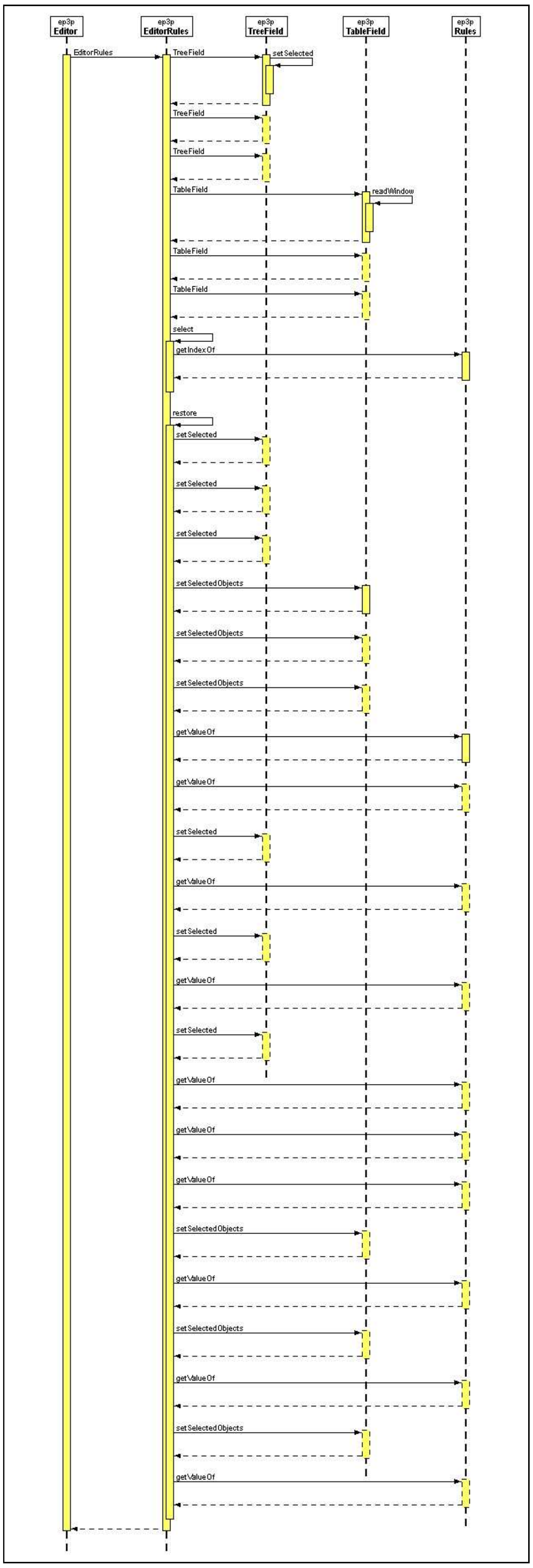
B.1 Class Diagrams & Sequence Diagrams

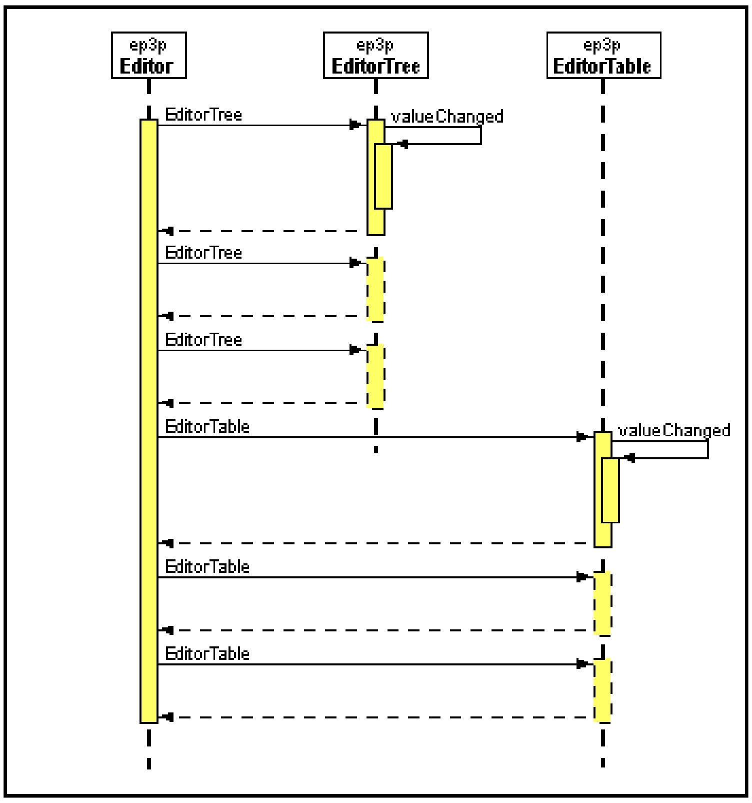
This section provides the class diagram, and sequence diagram of some of the important classes and methods in the application. This is followed by the actual source code of the application.

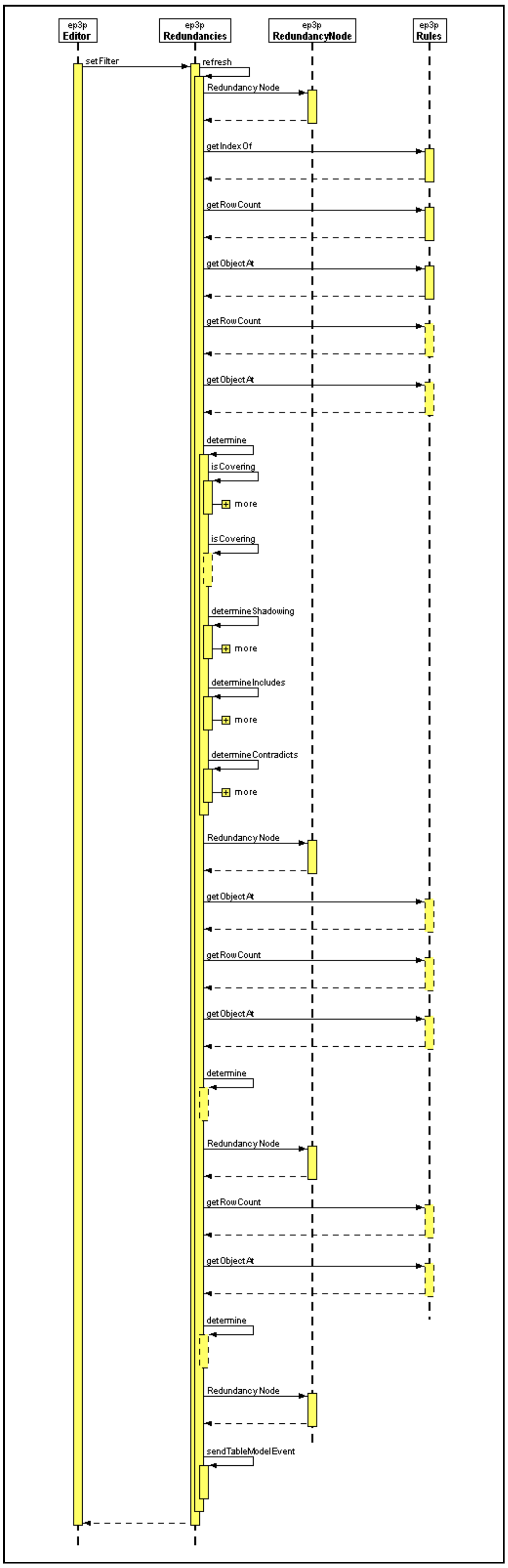


javax.swing.table.TableModel
Duplicates
Duplicates(Rules)
addTableModelListener(TableModelListener)
getColumnClass(int)
getColumnCount()
getColumnName(int)
getRowCount()
getRulesAt(int)
getValueAt(int , int)
isCellEditable(int , int)
refresh()
removeTableModelListener(TableModelListener)
sendTableModelEvent()
setValueAt(Object , int , int)
Redundancies
Redundancies(Rules)
addTableModelListener(TableModelListener)
determine(RedundancyNode)
determineContradicts(RedundancyNode)
determineIncludes(RedundancyNode)
determineShadowing(RedundancyNode)
getChild(int)
getColumnClass(int)
getColumnCount()
getColumnName(int)
getFilter()
getParent(int)
getRedundancy(int)
getRowCount()
getValueAt(int , int)
isCellEditable(int , int)
isCovering(Object , Object)
isSubset(Object[] , Object[])
refresh()
removeTableModelListener(TableModelListener)
sendTableModelEvent()
setFilter(Object)
setValueAt(Object , int , int)
Rules
Rules(Vocabulary)
add()
addTableModelListener(TableModelListener)
getColumnClass(int)
getColumnCount()
getColumnName(int)
getIndexOf(Object)
getObjectAt(int)
getRowCount()
getValueAt(int , int)
getValueOf(Object , int)
isCellEditable(int , int)
remove(Object)
remove(int)
removeTableModelListener(TableModelListener)
sendTableModelEvent(int , int , int)
setValueAt(Object , int , int)
setValueOf(Object , int , Object)


TreeField$1
javax.swing.event.TreeSelectionListener
EditorTree
TableField$3
java.awt.event.ActionListener
TreeField$3
TableField$1
javax.swing.event.ListSelectionListener
EditorRules
EditorTable
EditorRedundancies
EditorDuplicates
javax.swing.Box


ep3p Editor
ep3p EditorRules
ep3p TreeField
ep3p TableField
ep3p Rules
EditorRules
TreeField
setSelected
TreeField
TreeField
TableField
readWindow
TableField
TableField
select
getIndexOf
restore
setSelected
setSelected
setSelected
setSelectedObjects
setSelectedObjects
setSelectedObjects
getValueOf
getValueOf
setSelected
getValueOf
setSelected
getValueOf
setSelected
getValueOf
getValueOf
getValueOf
setSelectedObjects
getValueOf
setSelectedObjects
getValueOf
setSelectedObjects
getValueOf


ep3p
Editor
ep3p
EditorTree
ep3p
EditorTable
EditorTree
valueChanged
EditorTree
EditorTree
EditorTable
valueChanged
EditorTable
EditorTable


ep3p
Editor
ep3p
Redundancies
ep3p
RedundancyNode
ep3p
Rules
setFilter
refresh
RedundancyNode
getIndexOf
getRowCount
getObjectAt
getRowCount
getObjectAt
determine
isCovering
more
isCovering
determineShadowing
more
determineIncludes
more
determineContradicts
more
RedundancyNode
getObjectAt
getRowCount
getObjectAt
determine
RedundancyNode
getRowCount
getObjectAt
determine
RedundancyNode
sendTableModelEvent

B.2 Application's Source Code

B.2.1 Editor.java

```
package ep3p;

import java.awt.Component;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSeparator;
import javax.swing.JSplitPane;
import javax.swing.JTabbedPane;
import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JTree;
import javax.swing.ListSelectionModel;
import javax.swing.SwingConstants;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableColumnModel;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.MutableTreeNode;
import javax.swing.tree.TreeNode;
import javax.swing.tree.TreePath;
import javax.swing.tree.TreeSelectionModel;
```

```
class EditorTree extends Box implements ActionListener, TreeSelectionListener {
DefaultTreeModel model;
JTree tree;

JButton create;
JButton delete;

public EditorTree(DefaultTreeModel _model) {
super(BoxLayout.Y_AXIS);

model = _model;

tree = new JTree(model);
tree.getSelectionModel().addTreeSelectionListener(this);
tree.getSelectionModel().setSelectionMode(TreeSelectionModel.SINGLE_TREE_SELECTION);
tree.setEditable(true);

Box box = new Box(BoxLayout.X_AXIS);
create = new JButton("Create");
delete = new JButton("Delete");
create.addActionListener(this);
delete.addActionListener(this);

box.add(create);
box.add(delete);

add(new JScrollPane(tree));
add(box);

valueChanged(null);
}

public void actionPerformed(ActionEvent e) {
String action = e.getActionCommand();
TreePath path = tree.getSelectionPath();
if("Create".equals(action)) {
DefaultMutableTreeNode node = new DefaultMutableTreeNode("");
model.insertNodeInto(node, (MutableTreeNode)path.getLastPathComponent(), 0);
tree.startEditingAtPath(new TreePath(node.getPath()));
}
else if("Delete".equals(action)) {
    model.removeNodeFromParent((MutableTreeNode)path.getLastPathComponent());
}
}

public void valueChanged(TreeSelectionEvent e) {
```

```
boolean enabled = (tree.getSelectionPath()!=null);
create.setEnabled(enabled);
delete.setEnabled(enabled && tree.getSelectionPath().getPathCount()>1);
}
}

class EditorTable extends Box implements ActionListener, ListSelectionListener {
DefaultTableModel model;
JTable table;

JButton create;
JButton delete;

public EditorTable(DefaultTableModel _model) {
super(BoxLayout.Y_AXIS);

model = _model;

table = new JTable(model);
table.getSelectionModel().addListSelectionListener(this);
table.getSelectionModel().setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
table.setShowGrid(false);

Box box = new Box(BoxLayout.X_AXIS);
create = new JButton("Create");
delete = new JButton("Delete");
create.addActionListener(this);
delete.addActionListener(this);

box.add(create);
box.add(delete);

add(new JScrollPane(table));
add(box);

valueChanged(null);
}

public void actionPerformed(ActionEvent e) {
Object source = e.getSource();
if(source == create) {
model.addRow(new Object[] {"test"});
}
else if(source == delete) {
model.removeRow(table.getSelectedRow());
}
```

```
}

public void valueChanged(ListSelectionEvent e) {
boolean enabled = (table.getSelectionModel().getAnchorSelectionIndex()!=-1);
create.setEnabled(true);
delete.setEnabled(enabled);
}
}

class EditorRedundancies extends Box implements ActionListener, ListSelectionListener {
    Editor editor;

    JTable table;

    JButton clear;
    JLabel filter;

    JButton view1;
    JButton view2;
    JButton delete1;
    JButton delete2;
    JButton merge;
    JButton raise;

    JTextArea suggestion;

    public EditorRedundancies(Editor _editor)
    {
        super(BoxLayout.Y_AXIS);

        editor = _editor;

        table = new JTable(editor.redundancies);
        table.getSelectionModel().addListSelectionListener(this);

        TableColumnModel columns = table.getColumnModel();
        while(columns.getColumnCount()>3)
            columns.removeColumn(columns.getColumn(3));

        clear = new JButton("Clear");
        filter = new JLabel("Showing redundancies for all rules");

        editor.redundancies.addTableModelListener
        (new TableModelListener() {
            public void tableChanged(TableModelEvent e) {
                updateFilter();
            }});

        view1 = new JButton("Show");
        view2 = new JButton("Show");
        delete1 = new JButton("Delete");
        delete2 = new JButton("Delete");
        merge = new JButton("Merge");
        raise = new JButton("Raise");
        suggestion = new JTextArea();
        suggestion.setEditable(false);
        suggestion.setLineWrap(true);
        suggestion.setWrapStyleWord(true);

        clear.addActionListener(this);
        view1.addActionListener(this);
        view2.addActionListener(this);
        delete1.addActionListener(this);
        delete2.addActionListener(this);
        merge.addActionListener(this);
        raise.addActionListener(this);

        Box box = new Box(BoxLayout.X_AXIS);

        box.add(Box.createHorizontalStrut(10));
        box.add(filter);
        box.add(Box.createGlue());
        box.add(clear);

        JPanel actionPanel = new JPanel(new GridBagLayout());
        Insets insets = new Insets(2, 5, 2, 5);
        GridBagConstraints con1 = new GridBagConstraints(0,0,1,1,0,0,GridBagConstraints.CENTER,
GridBagConstraints.BOTH,insets,0,0);
        GridBagConstraints con2 = new GridBagConstraints(1,0,1,1,1,0,GridBagConstraints.CENTER,
GridBagConstraints.BOTH,insets,0,0);
        GridBagConstraints con3 = new GridBagConstraints(2,0,1,1,0,0,GridBagConstraints.CENTER,
GridBagConstraints.BOTH,insets,0,0);

        actionPanel.add(new JLabel("Parent", SwingConstants.CENTER), con1);
        actionPanel.add(new JLabel("Suggested Resolution", SwingConstants.CENTER), con2);
        actionPanel.add(new JLabel("Child", SwingConstants.CENTER), con3);
        con1.gridy++; con2.gridy++; con3.gridy++;

        con2.gridheight = 4;
        con2.weighty = 1;
        actionPanel.add(view1, con1);
        actionPanel.add(new JScrollPane(suggestion), con2);
        actionPanel.add(view2, con3);
        con1.gridy++; con3.gridy++;

        actionPanel.add(delete1, con1);
        actionPanel.add(delete2, con3);
        con1.gridy++; con3.gridy++;

        actionPanel.add(merge, con1);
        actionPanel.add(raise, con3);
        con1.gridy++; con3.gridy++;

        con1.gridy++;
        con1.insets = new Insets(5, 5, 5, 5);
        con1.fill = GridBagConstraints.HORIZONTAL;
        con1.gridwidth = 3;
        con1.anchor = GridBagConstraints.SOUTHEAST;
        actionPanel.add(new JSeparator(), con1);
        con1.gridy = con1.gridy+1;
        actionPanel.add(box, con1);

        add(new JScrollPane(table));
        add(actionPanel);

        valueChanged(null);
    }

    public void updateFilter()
    {
        if(editor.redundancies.getFilter() == null)
            filter.setText("Showing redundancies for all rules");
        else
            filter.setText("Showing redundancies for rule "+
                    editor.rules.getValueOf(editor.redundancies.getFilter(), Rules.NAME));
    }

    public void raiseRules(Object[] rules1, Object[] rules2)
    {
    }

    public void mergeRules(Object[] rules1, Object[] rules2)
    {
    }

    public void deleteRules(Object[] rules)
    {
        int i;
        for(i=0; i<rules.length; i++)
            editor.rules.remove(rules[i]);
    }

    public void actionPerformed(ActionEvent e) {
        JButton button = (JButton)e.getSource();
        int i;

        if(button == clear)
        {
            editor.redundancies.setFilter(null);
            return;
        }

        int row = table.getSelectedRow();
        Object rule1 = editor.redundancies.getParent(row);
        Object rule2 = editor.redundancies.getChild(row);

        int rows[] = table.getSelectedRows();
        Object rules1[] = new Object[rows.length];
        Object rules2[] = new Object[rows.length];

        for(i=0; i<rows.length; i++)
        {
           rules1[i] = editor.redundancies.getParent(rows[i]);
           rules2[i] = editor.redundancies.getChild(rows[i]);
        }

        if(button == view1) {
            editor.showRule(rule1);
            System.out.println("Selecting "+rule1);
        }
        else if(button == view2) {
            editor.showRule(rule2);
        }
        else if(button == delete1) {
            deleteRules(rules1);
        }
        else if(button == delete2) {
            deleteRules(rules2);
        }
        else if(button == merge) {
            mergeRules(rules1, rules2);
        }
        else if(button == raise) {
            raiseRules(rules1, rules2);
        }
    }

    public void valueChanged(ListSelectionEvent e) {
        int row = table.getSelectedRow();
        boolean enabled = (row != -1);

        view1.setEnabled(enabled);
        view2.setEnabled(enabled);
        delete1.setEnabled(enabled);
        delete2.setEnabled(enabled);
        raise.setEnabled(enabled);
        merge.setEnabled(enabled);
    }

}

class EditorDuplicates extends Box implements ActionListener, ListSelectionListener {
    Editor editor;

    JTable table;
    JList list;

    JButton view;
    JButton delete;
    JButton merge;

    public EditorDuplicates(Editor _editor)
    {
        super(BoxLayout.Y_AXIS);

        editor = _editor;

        table = new JTable(editor.duplicates);
        table.getSelectionModel().addListSelectionListener(this);

        list = new JList();
        list.getSelectionModel().addListSelectionListener(this);

        editor.duplicates.addTableModelListener
        (new TableModelListener() {
            public void tableChanged(TableModelEvent e) {
                valueChanged(null);
            }});

        view = new JButton("Show");
        delete = new JButton("Delete");
        merge = new JButton("Merge");

        view.addActionListener(this);
        delete.addActionListener(this);
        merge.addActionListener(this);

        Box box = new Box(BoxLayout.X_AXIS);

        box.add(Box.createHorizontalStrut(10));
        box.add(new JLabel("Merge/delete rules until only one exists for each situation"));
        box.add(Box.createGlue());

        JPanel actionPanel = new JPanel(new GridBagLayout());
        Insets insets = new Insets(2, 5, 2, 5);
        GridBagConstraints con1 = new GridBagConstraints(0,0,1,1,0,0,GridBagConstraints.CENTER,
GridBagConstraints.BOTH,insets,0,0);

        actionPanel.add(new JLabel("Rules", SwingConstants.CENTER), con1);
        con1.gridx++;
        con1.gridy++;
        actionPanel.add(view, con1);
        con1.gridy++;
        actionPanel.add(delete, con1);
        con1.gridy++;
        actionPanel.add(merge, con1);
        con1.gridy++;

        con1.gridx = 0;
        con1.gridy = 1;
        con1.gridheight = 4;
        con1.weightx = 1;
        con1.weighty = 1;
        actionPanel.add(new JScrollPane(list), con1);

        con1.gridy += con1.gridheight;
        con1.insets = new Insets(5, 5, 5, 5);
        con1.fill = GridBagConstraints.HORIZONTAL;
        con1.gridwidth = 3;
        con1.gridheight = 1;
        con1.weightx = 0;
        con1.weighty = 0;
        con1.anchor = GridBagConstraints.SOUTHEAST;
        actionPanel.add(new JSeparator(), con1);
        con1.gridy++;
        actionPanel.add(box, con1);
        add(new JScrollPane(table));
        add(actionPanel);

        valueChanged(null);
    }

    public void mergeRules(Object[] rules)
    {
    }

    public void deleteRules(Object[] rules)
    {
        int i;
        for(i=0; i<rules.length; i++)
            editor.rules.remove(rules[i]);
    }

    public void actionPerformed(ActionEvent e) {
        JButton button = (JButton)e.getSource();
        int i;

        Object rules[] = list.getSelectedValues();

        if(button == view) {
            editor.showRule(rules[0]);
        }
        else if(button == delete) {
            deleteRules(rules);
        }
        else if(button == merge) {
            mergeRules(rules);
        }
    }

    public void valueChanged(ListSelectionEvent e) {
     if(e == null || e.getSource() == list.getSelectionModel())
     {
     boolean enabled = (list.getSelectedIndex() != -1);
             view.setEnabled(enabled);
             delete.setEnabled(enabled);
             merge.setEnabled(enabled);
     }
     else
     {
        int row = table.getSelectedRow();
        if(row == -1)
        {
         list.setModel(new DefaultListModel());
        }
        else
        {
         list.setModel(editor.duplicates.getRulesAt(row));
        }
     }
    }

}

class EditorRules extends Box implements ActionListener, ListSelectionListener {
Editor editor;

JTable table;
Object selected;

JButton apply;
JButton reset;
JButton delete;
    JButton create;
    JButton show;

    JComboBox ruling;
JTextField name;
JTextField description;
JTextField priority;
TreeField user;
TreeField data;
TreeField purpose;
TableField action;
TableField conditions;
TableField obligations;

public EditorRules(Editor _editor) {
super(BoxLayout.Y_AXIS);

editor = _editor;

table = new JTable(editor.rules);
table.getSelectionModel().addListSelectionListener(this);

TableColumnModel columns = table.getColumnModel();
while(columns.getColumnCount()>2)
```

```
columns.removeColumn(columns.getColumn(2));

Box box = new Box(BoxLayout.X_AXIS);
        show = new JButton("Show Redundancies");
        apply = new JButton("Apply");
reset = new JButton("Reset");
delete = new JButton("Delete");
create = new JButton("Create");

        show.addActionListener(this);
apply.addActionListener(this);
reset.addActionListener(this);
delete.addActionListener(this);
        create.addActionListener(this);

box.add(Box.createGlue());
//        box.add(show);
//        box.add(Box.createHorizontalStrut(30));
box.add(create);
        box.add(delete);
        box.add(Box.createHorizontalStrut(30));
box.add(apply);
box.add(reset);

JPanel detailPanel = new JPanel(new GridBagLayout());
Insets insets = new Insets(5,5,5,5);
GridBagConstraints con1 = new GridBagConstraints(0,0,1,1,0,0,GridBagConstraints.SOUTH,
 GridBagConstraints.HORIZONTAL,insets,0,0);
GridBagConstraints con2 = new GridBagConstraints(1,0,1,1,1,0,GridBagConstraints.SOUTH,
 GridBagConstraints.HORIZONTAL,insets,0,0);
GridBagConstraints con3 = new GridBagConstraints(2,0,1,1,0,0,GridBagConstraints.SOUTH,
 GridBagConstraints.HORIZONTAL,insets,0,0);
GridBagConstraints con4 = new GridBagConstraints(3,0,1,1,1,0,GridBagConstraints.SOUTH,
 GridBagConstraints.HORIZONTAL,insets,0,0);

name = new JTextField();
description = new JTextField();
user = new TreeField(editor.vocabulary.treeUser);
data = new TreeField(editor.vocabulary.treeData);
purpose = new TreeField(editor.vocabulary.treePurpose);
action = new TableField(editor.vocabulary.tableAction, false);
priority = new JTextField();
conditions = new TableField(editor.vocabulary.tableConditions, true);
obligations = new TableField(editor.vocabulary.tableObligations, true);
ruling = new JComboBox(new Object[] {"Authorise", "Deny"} );
```

```
detailPanel.add(new JLabel("Name"), con1);
detailPanel.add(name, con2);
detailPanel.add(new JLabel("User"), con3);
detailPanel.add(user, con4);
con1.gridy = con2.gridy = con3.gridy = con4.gridy = con1.gridy+1;

detailPanel.add(new JLabel("Description"), con1);
detailPanel.add(description, con2);
detailPanel.add(new JLabel("Data"), con3);
detailPanel.add(data, con4);
con1.gridy = con2.gridy = con3.gridy = con4.gridy = con1.gridy+1;

detailPanel.add(new JLabel("Priority"), con1);
detailPanel.add(priority, con2);
detailPanel.add(new JLabel("Purpose"), con3);
detailPanel.add(purpose, con4);
con1.gridy = con2.gridy = con3.gridy = con4.gridy = con1.gridy+1;

detailPanel.add(new JLabel("Ruling"), con1);
detailPanel.add(ruling, con2);
detailPanel.add(new JLabel("Conditions"), con3);
detailPanel.add(conditions, con4);
con1.gridy = con2.gridy = con3.gridy = con4.gridy = con1.gridy+1;

detailPanel.add(new JLabel("Obligations"), con1);
detailPanel.add(obligations, con2);
detailPanel.add(new JLabel("Action"), con3);
detailPanel.add(action, con4);
con1.gridy = con2.gridy = con3.gridy = con4.gridy = con1.gridy+1;

con1.fill = GridBagConstraints.HORIZONTAL;
con1.gridwidth = 4;
con1.anchor = GridBagConstraints.SOUTHEAST;
detailPanel.add(new JSeparator(), con1);
con1.gridy = con1.gridy+1;
detailPanel.add(box, con1);

add(new JScrollPane(table));
add(detailPanel);

select(null);
restore(null);
}

void store(Object rule) {
editor.rules.setValueOf(rule,Rules.NAME,name.getText());
editor.rules.setValueOf(rule,Rules.DESCRIPTION,description.getText());

editor.rules.setValueOf(rule,Rules.USER,user.getSelected());
editor.rules.setValueOf(rule,Rules.DATA,data.getSelected());
editor.rules.setValueOf(rule,Rules.PURPOSE,purpose.getSelected());

editor.rules.setValueOf(rule,Rules.PRIORITY,new Integer(priority.getText()));
editor.rules.setValueOf(rule,Rules.CONDITIONS,conditions.getSelectedObjects());
editor.rules.setValueOf(rule,Rules.OBLIGATIONS,obligations.getSelectedObjects());
editor.rules.setValueOf(rule,Rules.ACTIONS,action.getSelectedObjects());

editor.rules.setValueOf(rule,Rules.RULING,new Integer(ruling.getSelectedIndex()==0?1:-1));
}

void restore(Object rule) {
if(rule==null) {
apply.setEnabled(false);
delete.setEnabled(false);
            reset.setEnabled(false);
            show.setEnabled(false);

name.setText("");
description.setText("");

user.setSelected((TreeNode)editor.vocabulary.treeUser.getRoot());
data.setSelected((TreeNode)editor.vocabulary.treeData.getRoot());
purpose.setSelected((TreeNode)editor.vocabulary.treePurpose.getRoot());
ruling.setSelectedIndex(0);

priority.setText("0");
conditions.setSelectedObjects(new Object[0]);
obligations.setSelectedObjects(new Object[0]);
action.setSelectedObjects(new Object[0]);
} else {
apply.setEnabled(true);
delete.setEnabled(true);
reset.setEnabled(true);
show.setEnabled(true);

name.setText((String)editor.rules.getValueOf(rule, Rules.NAME));
description.setText((String)editor.rules.getValueOf(rule, Rules.DESCRIPTION));

user.setSelected((TreeNode)editor.rules.getValueOf(rule, Rules.USER));
data.setSelected((TreeNode)editor.rules.getValueOf(rule, Rules.DATA));
purpose.setSelected((TreeNode)editor.rules.getValueOf(rule, Rules.PURPOSE));
ruling.setSelectedIndex(((Integer)
 editor.rules.getValueOf(rule, Rules.RULING)).intValue()<0?1:0);

priority.setText(editor.rules.getValueOf(rule, Rules.PRIORITY).toString());
conditions.setSelectedObjects((Object[])editor.rules.getValueOf(rule, Rules.CONDITIONS));
obligations.setSelectedObjects((Object[])editor.rules.getValueOf(rule, Rules.OBLIGATIONS));
action.setSelectedObjects((Object[])editor.rules.getValueOf(rule, Rules.ACTIONS));
}

}

void select(Object rule) {
int index = editor.rules.getIndexOf(rule);
if(index==-1) table.getSelectionModel().clearSelection();
else table.getSelectionModel().setSelectionInterval(index,index);
}

public void actionPerformed(ActionEvent e) {
Object source = e.getSource();
if(source == apply) {
store(selected);
}
else if(source == reset) {
restore(selected);
}
else if(source == delete) {
editor.rules.remove(selected);
}
else if(source == create) {
Object rule = editor.rules.add();
store(rule);
select(rule);
}
        else if(source == show) {
            editor.showRedundancies(selected);
        }
}

public void valueChanged(ListSelectionEvent e) {
int index = table.getSelectedRow();
Object rule = (index==-1)?null:editor.rules.getObjectAt(index);

if(rule != selected) {
selected = rule;
restore(selected);
}
```

```
}

}

public class Editor {
    Vocabulary vocabulary;
Rules rules;
    Redundancies redundancies;
    Duplicates duplicates;

private JFrame mainFrame;
private JTabbedPane tabbedPane1;
private JTabbedPane tabbedPane2;

private Component vocabularyComponent;
private Component rulesComponent;
private Component redundanciesComponent;
private Component duplicatesComponent;

private void buildVocabularyComponent() {
JPanel panel = new JPanel();
Insets insets = new Insets(5,5,5,5);
GridBagConstraints constraints = new GridBagConstraints(0,0,1,1,1,1,GridBagConstraints.CENTER,
GridBagConstraints.BOTH,insets,0,0);
panel.setLayout(new GridBagLayout());
constraints.gridx = 0;
panel.add(new EditorTree(vocabulary.treeUser), constraints);
constraints.gridx = 1;
panel.add(new EditorTree(vocabulary.treePurpose), constraints);
constraints.gridx = 2;
panel.add(new EditorTree(vocabulary.treeData), constraints);
constraints.gridy++;
constraints.gridx = 0;
panel.add(new EditorTable(vocabulary.tableAction), constraints);
constraints.gridx = 1;
panel.add(new EditorTable(vocabulary.tableConditions), constraints);
constraints.gridx = 2;
panel.add(new EditorTable(vocabulary.tableObligations), constraints);
vocabularyComponent = panel;
}

private void buildRedundanciesComponent() {
redundanciesComponent = new EditorRedundancies(this);
}

private void buildDuplicatesComponent() {
```

```
duplicatesComponent = new EditorDuplicates(this);
}

    private void buildRulesComponent() {
        rulesComponent = new EditorRules(this);
    }

public Editor() {
        vocabulary = new Vocabulary();
        rules = new Rules(vocabulary);
        redundancies = new Redundancies(rules);
        duplicates = new Duplicates(rules);

mainFrame = new JFrame("Policy Management Console");
tabbedPane1 = new JTabbedPane();
tabbedPane2 = new JTabbedPane();

        rules.setValueOf(rules.add(),Rules.NAME,"rule1");
        rules.setValueOf(rules.add(),Rules.NAME,"rule2");

buildVocabularyComponent();
        buildRulesComponent();
buildDuplicatesComponent();
buildRedundanciesComponent();

tabbedPane1.add("Vocabulary", vocabularyComponent);
tabbedPane1.add("Rules", rulesComponent);
tabbedPane2.add("Rectify Situations", duplicatesComponent);
tabbedPane2.add("Rectify Priorities", redundanciesComponent);

mainFrame.getContentPane().add(new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, tabbedPane1, tabbedPane2));
mainFrame.setSize(1000,500);
mainFrame.show();
}

    public void showRule(Object rule) {
        ((EditorRules)rulesComponent).select(rule);
        tabbedPane1.setSelectedComponent(rulesComponent);
    }

    public void showRedundancies(Object rule) {
        redundancies.setFilter(rule);
        tabbedPane2.setSelectedComponent(redundanciesComponent);
    }

public static void main(String[] args) {
```

```
new Editor();
}
}
```

B.2.2 TreeField.java

```
package ep3p;

import java.awt.Component;
import java.awt.Container;
import java.awt.Point;
import java.awt.Window;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ComponentEvent;
import java.awt.event.ComponentListener;
import java.awt.event.HierarchyEvent;
import java.awt.event.HierarchyListener;
import java.util.LinkedList;

import javax.swing.JScrollPane;
import javax.swing.JToggleButton;
import javax.swing.JTree;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.TreeModel;
import javax.swing.tree.TreeNode;
import javax.swing.tree.TreePath;
import javax.swing.tree.TreeSelectionModel;

public class TreeField extends JToggleButton {
private TreeNode selected;
private TreeModel model;

    private Component ancestor;
    private ComponentListener clistener;

    private JTree tree;
private Window window;

public TreeField(TreeModel _model) {
model = _model;

tree = new JTree(model);
        tree.getSelectionModel().setSelectionMode(TreeSelectionModel.SINGLE_TREE_SELECTION);

        tree.addTreeSelectionListener(new TreeSelectionListener() {
            public void valueChanged(TreeSelectionEvent e) {
                readWindow();
            }
        });

        addHierarchyListener(new HierarchyListener() {
            public void hierarchyChanged(HierarchyEvent e) {
                if(getModel().isSelected()) doClick();
            }
        });

        addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                toggleWindow();
            }
        });

        clistener = new ComponentListener() {
            public void componentHidden(ComponentEvent e) {
                if(getModel().isSelected())
                    doClick();
            }

            public void componentMoved(ComponentEvent e) {
                if(getModel().isSelected())
                    positionWindow();
            }

            public void componentResized(ComponentEvent e) {
                if(getModel().isSelected())
                    positionWindow();
            }

            public void componentShown(ComponentEvent e) {
            }
        };

        addComponentListener(clistener);
setSelected((TreeNode)model.getRoot());
}

public TreeNode getSelected() {
return selected;
}

public void setSelected(TreeNode node) {
```

```
LinkedList list = new LinkedList();
if(node == null) {
node = (TreeNode)model.getRoot();
}
while(node!=null)
{
list.addFirst(node);
node = node.getParent();
}
tree.setSelectionPath(new TreePath(list.toArray()));
}

    private void positionWindow() {
        Point p = getLocationOnScreen();
        p.y += getHeight();
        window.setLocation(p);
        window.setSize(getWidth(),window.getHeight());
    }

public void toggleWindow() {
if(window != null) {
window.dispose();
window = null;
}
else {
            Container container = this.getTopLevelAncestor();
            if(container!=ancestor) {
                if(ancestor != null) ancestor.removeComponentListener(clistener);
                container.addComponentListener(clistener);
            }

window = new Window((Window)container);
window.add(new JScrollPane(tree));
            window.setFocusableWindowState(true);
window.setSize(getWidth(),getHeight()*10);
            positionWindow();

setSelected(selected);
tree.grabFocus();

window.show();
window.toFront();
}
}

public void readWindow() {
```

```
TreePath path = tree.getSelectionPath();
selected = (TreeNode)((path!=null)?path.getLastPathComponent():model.getRoot());
setText(selected.toString());
}
}
```

B.2.3 TableField.java

```
package ep3p;

import java.awt.Component;
import java.awt.Container;
import java.awt.Point;
import java.awt.Window;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ComponentEvent;
import java.awt.event.ComponentListener;
import java.awt.event.HierarchyEvent;
import java.awt.event.HierarchyListener;

import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JToggleButton;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import javax.swing.table.TableModel;

public class TableField extends JToggleButton {
private TableModel model;
private Object[] selected;

    private Component ancestor;
    private ComponentListener clistener;

    private JTable table;
private Window window;

public TableField(TableModel _model, boolean multiple) {
model = _model;

table = new JTable(model);
table.setTableHeader(null);
table.setShowGrid(false);
if(multiple) table.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
        else table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        table.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent e) {
                readWindow();
            }
        });

        addHierarchyListener(new HierarchyListener() {
            public void hierarchyChanged(HierarchyEvent e) {
                if(getModel().isSelected()) doClick();
            }
        });

addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        toggleWindow();
            }
});

        clistener = new ComponentListener() {
            public void componentHidden(ComponentEvent e) {
                if(getModel().isSelected())
                    doClick();
            }

            public void componentMoved(ComponentEvent e) {
                if(getModel().isSelected())
                    positionWindow();
            }

            public void componentResized(ComponentEvent e) {
                if(getModel().isSelected())
                    positionWindow();
            }

            public void componentShown(ComponentEvent e) {
            }
        };

        addComponentListener(clistener);
        readWindow();
}

public Object[] getSelectedObjects() {
return selected;
}
```

```
public void setSelectedObjects(Object[] selection) {
table.getSelectionModel().clearSelection();
if(selection != null) {
ListSelectionModel sel = table.getSelectionModel();
Object object;
int i, j;

for(i=0;i<model.getRowCount();i++) {
object = model.getValueAt(i,0);
for(j=0;j<selection.length;j++)
if(object == selection[j])
sel.addSelectionInterval(i,i);
}
}
}

    private void positionWindow() {
        Point p = getLocationOnScreen();
        p.y += getHeight();
        window.setLocation(p);
        window.setSize(getWidth(),window.getHeight());
    }

    private void toggleWindow() {
        if(window != null) {
            window.dispose();
            window = null;
        }
        else {
            Container container = this.getTopLevelAncestor();
            if(container!=ancestor) {
                if(ancestor != null) ancestor.removeComponentListener(clistener);
                container.addComponentListener(clistener);
            }

            window = new Window((Window)container);
            window.add(new JScrollPane(table));
            window.setFocusableWindowState(true);
            window.setSize(getWidth(),getHeight()*10);
            positionWindow();

            table.grabFocus();

            window.show();
            window.toFront();
        }
    }

    private void readWindow() {
        int[] indices = table.getSelectedRows();
        String label = "";
        int i;

        selected = new Object[indices.length];
        for(i=0;i<indices.length;i++) {
            selected[i] = model.getValueAt(indices[i],0);
        }

        if(selected.length==0) {
            label = "<none>";
        }
        else {
            for(i=0;i<selected.length;i++) {
                if(i>0) label = label+", ";
                label = label+selected[i].toString();
            }
        }
        setText(label);
    }



}
```

B.2.4 Rules.java

```
package ep3p;

import java.util.ArrayList;

import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.TableModel;

class RuleNode {
Object[] properties = new Object[Rules.PROPERTY_COUNT];

public String toString()
{
return (String) properties[Rules.NAME];
}
}
```

```
public class Rules implements TableModel {
public static final int NAME = 0;
public static final int DESCRIPTION = 1;

public static final int USER = 2;
public static final int DATA = 3;
public static final int PURPOSE = 4;
public static final int ACTIONS = 5;

public static final int PRIORITY = 6;
public static final int CONDITIONS = 7;
public static final int OBLIGATIONS = 8;
public static final int RULING = 9;

public static final int PROPERTY_COUNT = 10;

private ArrayList rules;
private ArrayList listeners;

    private Vocabulary vocabulary;

public Rules(Vocabulary _vocabulary) {
        vocabulary = _vocabulary;
rules = new ArrayList();
listeners = new ArrayList();
}



public int getColumnCount() {
return Rules.PROPERTY_COUNT;
}

public int getRowCount() {
return rules.size();
}

public boolean isCellEditable(int rowIndex, int columnIndex) {
return false;
}

public Class getColumnClass(int columnIndex) {
return String.class;
}

public String getColumnName(int columnIndex) {
```

```
switch(columnIndex) {
case Rules.NAME: return "Name";
case Rules.DESCRIPTION: return "Description";
case Rules.USER: return "User";
case Rules.DATA: return "Data";
case Rules.PURPOSE: return "Purpose";
case Rules.ACTIONS: return "Action";
case Rules.PRIORITY: return "Priority";
case Rules.CONDITIONS: return "Condition";
case Rules.OBLIGATIONS: return "Obligation";
case Rules.RULING: return "Ruling";
}
return null;
}



public Object getValueAt(int rowIndex, int columnIndex) {
return ((RuleNode)rules.get(rowIndex)).properties[columnIndex];
}

public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
((RuleNode)rules.get(rowIndex)).properties[columnIndex] = aValue;
sendTableModelEvent(rowIndex, columnIndex, TableModelEvent.UPDATE);
}

public void remove(int rowIndex) {
rules.remove(rowIndex);
sendTableModelEvent(rowIndex, TableModelEvent.ALL_COLUMNS, TableModelEvent.DELETE);
}



// Used to get a fixed object reference for a row.
// This will not change even if the table is re-oredered
public Object getObjectAt(int rowIndex) {
return rules.get(rowIndex);
}

public int getIndexOf(Object rowObject) {
return rules.indexOf(rowObject);
}

public Object getValueOf(Object rowObject, int propertyIndex) {
return ((RuleNode)rowObject).properties[propertyIndex];
}

public void setValueOf(Object rowObject, int propertyIndex, Object aValue) {
```

```
int index = rules.indexOf(rowObject);
if(index!=-1) setValueAt(aValue, index, propertyIndex);
}

public Object add() {
RuleNode node = new RuleNode();
        node.properties[NAME] = "";
        node.properties[DESCRIPTION] = "";
        node.properties[USER] = vocabulary.treeUser.getRoot();
        node.properties[DATA] = vocabulary.treeData.getRoot();
        node.properties[PURPOSE] = vocabulary.treePurpose.getRoot();
        node.properties[CONDITIONS] = new Object[0];
        node.properties[OBLIGATIONS] = new Object[0];
        node.properties[ACTIONS] = new Object[] {vocabulary.tableAction.getValueAt(0,0)};
        node.properties[PRIORITY] = new Integer(0);
        node.properties[RULING] = new Integer(-1);
        rules.add(node);

sendTableModelEvent(rules.size()-1, TableModelEvent.ALL_COLUMNS, TableModelEvent.INSERT);
return node;
}

public void remove(Object rowObject) {
int index = getIndexOf(rowObject);
remove(index);
}



public void addTableModelListener(TableModelListener l) {
listeners.add(l);
}

public void removeTableModelListener(TableModelListener l) {
int index = listeners.lastIndexOf(l);
if(index!=-1) listeners.remove(index);
}

private void sendTableModelEvent(int rowIndex, int columnIndex, int type)
{
TableModelEvent event = new TableModelEvent(this, rowIndex, rowIndex, columnIndex, type);
TableModelListener[] array = new TableModelListener[listeners.size()];
int index;

array = (TableModelListener[])listeners.toArray(array);
for(index=0; index<array.length; index++)
array[index].tableChanged(event);
```

```
}
}
```

B.2.5 Vocabulary.java

```
package ep3p;

import javax.swing.table.DefaultTableModel;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.MutableTreeNode;
import javax.swing.tree.TreePath;

public class Vocabulary {
    DefaultTreeModel treeUser;
    DefaultTreeModel treeData;
    DefaultTreeModel treePurpose;

    DefaultMutableTreeNode allUsers;
    DefaultMutableTreeNode allPurposes;
    DefaultMutableTreeNode allData;

    MutableTreeNode staff, medicalTeam,doctors,nurses;
    MutableTreeNode medicalRecord, disease;
    MutableTreeNode treatment, research, marketing;

    DefaultTableModel tableAction;
    DefaultTableModel tableConditions;
    DefaultTableModel tableObligations;

    Vocabulary()
    {
     allUsers  = new DefaultMutableTreeNode("All Users");
     medicalTeam = new DefaultMutableTreeNode("Medical Team");
     staff  = new DefaultMutableTreeNode("Staff");
     doctors  = new DefaultMutableTreeNode("Doctros");
     nurses  = new DefaultMutableTreeNode("Nurses");

     allData   = new DefaultMutableTreeNode("All Data");
     medicalRecord  = new DefaultMutableTreeNode("Medical Record");
     disease = new DefaultMutableTreeNode("Disease");

     allPurposes = new DefaultMutableTreeNode("All Purposes");
     treatment = new DefaultMutableTreeNode("Treatment");
     research = new DefaultMutableTreeNode("Research");

     medicalTeam.insert(doctors,0);
     medicalTeam.insert(nurses,1);
     staff.insert(medicalTeam,0);
     allUsers.add(staff);

     allData.add(medicalRecord);
     allData.add(disease);

     allPurposes.add(treatment);
     allPurposes.add(research);

     treeUser  = new DefaultTreeModel(allUsers);
treeData  = new DefaultTreeModel(allData);
        treePurpose = new DefaultTreeModel(allPurposes);

        tableAction = new DefaultTableModel(new Object[] {"Actions"},0);
        tableConditions = new DefaultTableModel(new Object[] {"Conditions"},0);
        tableObligations = new DefaultTableModel(new Object[] {"Obligations"},0);

        tableAction.addRow(new Object[] {"read"});
        tableAction.addRow(new Object[] {"write"});
        tableAction.addRow(new Object[] {"delete"});

        tableObligations.addRow(new Object[] {"notify"});
        tableObligations.addRow(new Object[] {"anonymize"});
        tableObligations.addRow(new Object[] {"delete record"});

        tableConditions.addRow(new Object[] {"emergency"});
        tableConditions.addRow(new Object[] {"under 15"});
        tableConditions.addRow(new Object[] {"concent"});
    }
}
```

B.2.6 Redundancies.java

```
package ep3p;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.TableModel;
import javax.swing.tree.DefaultMutableTreeNode;
```

```
class RedundancyNode
{
    Object parent, child;
    int redundancy;
}

public class Redundancies implements TableModel {
    public static final int CONTRADICTS = 1;
    public static final int SHADOWS = 2;
    public static final int INCLUDES = 4;

    public static final int SUGGEST_MERGE = 1;
    public static final int SUGGEST_RAISE = 2;

    private ArrayList redundancies;
    private ArrayList listeners;

    private Rules rules;
    private Object filter;

    Redundancies(Rules _rules)
    {
        rules = _rules;
        redundancies = new ArrayList();
        listeners = new ArrayList();

        rules.addTableModelListener(new TableModelListener() {
            public void tableChanged(TableModelEvent e) { refresh(); }
            });
        refresh();
    }

    public boolean isSubset(Object[] array1, Object[] array2)
    {
        int i, j;

        if(array1==null) return true;
        if(array2==null) return false;

        for(i=0;i<array1.length;i++)
        {
            for(j=0;j<array2.length;j++)
                if(array1[i]==array2[j])
                    break;
            if(j==array2.length)
            return false;
        }
        return true;
    }

    public boolean isCovering(Object rule1, Object rule2)
    {
        DefaultMutableTreeNode node1, node2;
        Object[] array1, array2;

        // If node1 is not an ancestor of node2, then not covering
        node1 = (DefaultMutableTreeNode)rules.getValueOf(rule1, Rules.USER);
        node2 = (DefaultMutableTreeNode)rules.getValueOf(rule2, Rules.USER);
        if(!node2.isNodeAncestor(node1)) return false;

        // If node1 is not an ancestor of node2, then not covering
        node1 = (DefaultMutableTreeNode)rules.getValueOf(rule1, Rules.DATA);
        node2 = (DefaultMutableTreeNode)rules.getValueOf(rule2, Rules.DATA);
        if(!node2.isNodeAncestor(node1)) return false;

        // If node1 is not an ancestor of node2, then not covering
        node1 = (DefaultMutableTreeNode)rules.getValueOf(rule1, Rules.PURPOSE);
        node2 = (DefaultMutableTreeNode)rules.getValueOf(rule2, Rules.PURPOSE);
        if(!node2.isNodeAncestor(node1)) return false;

        // If array1 is not a subset of array2, then not covering
        array1 = (Object[])rules.getValueOf(rule1, Rules.CONDITIONS);
        array2 = (Object[])rules.getValueOf(rule2, Rules.CONDITIONS);
        if(!isSubset(array1, array2)) return false;

        // If object1 is not equal to object2, then not covering
        array1 = (Object[])rules.getValueOf(rule1, Rules.ACTIONS);
        array2 = (Object[])rules.getValueOf(rule2, Rules.ACTIONS);
        if(!Arrays.equals(array1, array2)) return false;

        return true;
    }

    public void determineContradicts(RedundancyNode node)
    {
     Integer int1, int2;

        int1 = (Integer)rules.getValueOf(node.parent, Rules.RULING);
        int2 = (Integer)rules.getValueOf(node.child, Rules.RULING);
        if(int1.intValue()>0 || int2.intValue()<0) return;
        node.redundancy |= CONTRADICTS;
    }

    public void determineShadowing(RedundancyNode node)
    {
     Integer int1, int2;

        int1 = (Integer)rules.getValueOf(node.parent, Rules.PRIORITY);
        int2 = (Integer)rules.getValueOf(node.child, Rules.PRIORITY);
        if(int1.compareTo(int2)<=0) return;

        node.redundancy |= SHADOWS;
    }

    public void determineIncludes(RedundancyNode node)
    {
     Integer int1, int2;
     Object[] array1, array2;

        int1 = (Integer)rules.getValueOf(node.parent, Rules.PRIORITY);
        int2 = (Integer)rules.getValueOf(node.child, Rules.PRIORITY);
        if(int1.compareTo(int2)<0) return;

        int1 = (Integer)rules.getValueOf(node.parent, Rules.RULING);
        int2 = (Integer)rules.getValueOf(node.child, Rules.RULING);
        if(int1.compareTo(int2)!=0) return;

        // If array1 is not a subset of array2, then not covering
        array1 = (Object[])rules.getValueOf(node.parent, Rules.CONDITIONS);
        array2 = (Object[])rules.getValueOf(node.child, Rules.CONDITIONS);
        if(!isSubset(array1, array2)) return;

        array1 = (Object[])rules.getValueOf(node.parent, Rules.OBLIGATIONS);
        array2 = (Object[])rules.getValueOf(node.child, Rules.OBLIGATIONS);
        if(!isSubset(array2, array1)) return;

        node.redundancy |= INCLUDES;
    }

    public void determine(RedundancyNode node)
    {
        node.redundancy = 0;
        if(node.child == node.parent) return;

        if(isCovering(node.parent, node.child) &&
   !isCovering(node.child, node.parent))
```

```
      {
determineShadowing(node);
determineIncludes(node);
determineContradicts(node);
      }
  }

  public void refresh()
  {
      RedundancyNode node;
      int i, j;

      redundancies = new ArrayList();
      node = new RedundancyNode();

      if(filter != null && rules.getIndexOf(filter) == -1)
      {
          filter = null;
      }

      if(filter == null)
      {
          for(i = 0; i < rules.getRowCount(); i++)
          {
              node.parent = rules.getObjectAt(i);
              for(j = 0; j < rules.getRowCount(); j++)
              {
                  node.child = rules.getObjectAt(j);
                  determine(node);
                  if(node.redundancy != 0)
                  {
                      redundancies.add(node);
                      node = new RedundancyNode();
                      node.parent = rules.getObjectAt(i);
                  }
              }
          }
      }
      else
      {
          node.parent = filter;
          for(i = 0; i < rules.getRowCount(); i++)
          {
              node.child = rules.getObjectAt(i);
              determine(node);
              if(node.redundancy != 0)
```

```
            {
                redundancies.add(node);
                node = new RedundancyNode();
            }
        }
        node.child = filter;
        for(i = 0; i < rules.getRowCount(); i++)
        {
            node.parent = rules.getObjectAt(i);
            if(node.child == node.parent) continue;
            determine(node);
            if(node.redundancy != 0)
            {
                redundancies.add(node);
                node = new RedundancyNode();
            }
        }
    }

    sendTableModelEvent();
}

public void setFilter(Object _filter)
{
    filter = _filter;
    refresh();
}

public Object getFilter()
{
    return filter;
}

public int getRedundancy(int rowIndex)
{
    return ((RedundancyNode)redundancies.get(rowIndex)).redundancy;
}

public Object getParent(int rowIndex)
{
    return ((RedundancyNode)redundancies.get(rowIndex)).parent;
}

public Object getChild(int rowIndex)
{
    return ((RedundancyNode)redundancies.get(rowIndex)).child;
}

public int getColumnCount() {
    return 3;
}

public int getRowCount() {
    return redundancies.size();
}

public boolean isCellEditable(int rowIndex, int columnIndex) {
    return false;
}

public Class getColumnClass(int columnIndex) {
    return String.class;
}

public Object getValueAt(int rowIndex, int columnIndex) {
    RedundancyNode node = (RedundancyNode)redundancies.get(rowIndex);
    switch(columnIndex)
    {
    case 0:
        return rules.getValueOf(node.parent,Rules.NAME);
    case 1:
        switch(node.redundancy)
        {
        case CONTRADICTS: return "contradicts";
        case INCLUDES: return "includes";
        case SHADOWS: return "shadows";
        case SHADOWS|CONTRADICTS: return "shadows and contradicts";
        case SHADOWS|INCLUDES: return "shadows and includes";
        default: return "";
        }
    case 2:
        return rules.getValueOf(node.child,Rules.NAME);
    default:
        return null;
    }
}

public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
}

public String getColumnName(int columnIndex) {
    switch(columnIndex)
    {
    case 0: return "Parent";
    case 1: return "Redundancy";
    case 2: return "Child";
    default: return null;
    }
}

public void addTableModelListener(TableModelListener l) {
    listeners.add(l);
}

public void removeTableModelListener(TableModelListener l) {
    int index = listeners.lastIndexOf(l);
    if(index!=-1) listeners.remove(index);
}

private void sendTableModelEvent()
{
    TableModelEvent event = new TableModelEvent(this);
    TableModelListener[] array = new TableModelListener[listeners.size()];
    int index;

    array = (TableModelListener[])listeners.toArray(array);
    for(index=0; index<array.length; index++)
        array[index].tableChanged(event);
}
}
```

B.2.7 Duplicates.java

```
package ep3p;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Iterator;
import javax.swing.ListModel;
import javax.swing.event.ListDataListener;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.TableModel;

class DuplicateNode implements ListModel
{
    ArrayList listeners = new ArrayList();
ArrayList duplicates;
```

```
Object user;
Object data;
Object purpose;
Object[] conditions;
Object[] actions;

public void refresh(Rules rules) {
Object rule;
int i;

duplicates = new ArrayList();
for(i=0; i<rules.getRowCount(); i++)
{
rule = rules.getObjectAt(i);

        if(!user.equals(rules.getValueOf(rule, Rules.USER))) continue;
        if(!data.equals(rules.getValueOf(rule, Rules.DATA))) continue;
        if(!purpose.equals(rules.getValueOf(rule, Rules.PURPOSE))) continue;
        if(!Arrays.equals(conditions,(Object[])rules.getValueOf(rule, Rules.CONDITIONS))) continue;
        if(!Arrays.equals(actions,(Object[])rules.getValueOf(rule, Rules.ACTIONS))) continue;

        duplicates.add(rule);
}
}

public int getSize() {
return duplicates.size();
}

public Object getElementAt(int index) {
return duplicates.get(index);
}

public void addListDataListener(ListDataListener l) {
        listeners.add(l);
    }

public void removeListDataListener(ListDataListener l) {
        int index = listeners.lastIndexOf(l);
        if(index!=-1) listeners.remove(index);
}
}

public class Duplicates implements TableModel {
    private ArrayList duplicates;
    private ArrayList listeners;
    private Rules rules;

    Duplicates(Rules _rules)
    {
        rules = _rules;
        duplicates = new ArrayList();
        listeners = new ArrayList();

        rules.addTableModelListener(new TableModelListener() {
            public void tableChanged(TableModelEvent e) { refresh(); }
            });
        refresh();
    }

    public void refresh()
    {
     HashSet used = new HashSet();
        DuplicateNode node;
        int i;

        for(i = 0; i < duplicates.size(); i++)
        {
         node = (DuplicateNode)duplicates.get(i);
         node.refresh(rules);
         used.addAll(node.duplicates);
         if(node.getSize()<2)
         {
         duplicates.remove(i);
         i--;
         }
        }

        node = new DuplicateNode();
        for(i = 0; i < rules.getRowCount(); i++)
        {
         if(used.contains(rules.getObjectAt(i))) continue;

     node.user = rules.getValueAt(i, Rules.USER);
     node.data = rules.getValueAt(i, Rules.DATA);
     node.purpose = rules.getValueAt(i, Rules.PURPOSE);
     node.conditions = (Object[])rules.getValueAt(i, Rules.CONDITIONS);
     node.actions = (Object[])rules.getValueAt(i, Rules.ACTIONS);
     node.refresh(rules);
         used.addAll(node.duplicates);
         if(node.getSize()>=2)
         {
         duplicates.add(node);
         node = new DuplicateNode();
         }
        }

        sendTableModelEvent();
    }

    public int getColumnCount() {
        return 5;
    }

    public int getRowCount() {
        return duplicates.size();
    }

    public boolean isCellEditable(int rowIndex, int columnIndex) {
        return false;
    }

    public Class getColumnClass(int columnIndex) {
        return String.class;
    }

    public ListModel getRulesAt(int rowIndex) {
     return (DuplicateNode)duplicates.get(rowIndex);
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        DuplicateNode node = (DuplicateNode)duplicates.get(rowIndex);

        switch(columnIndex)
        {
        case 0:
            return node.user.toString();
        case 1:
            return node.data.toString();
        case 2:
            return node.purpose.toString();
        case 3:
     return node.conditions.toString();
        case 4:
     return node.actions.toString();
        default:
            return null;
        }
    }

    public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
    }

    public String getColumnName(int columnIndex) {
        switch(columnIndex)
        {
        case 0: return "User";
        case 1: return "Data";
        case 2: return "Purpose";
        case 3: return "Conditions";
        case 4: return "Action";
        default: return null;
        }
    }

    public void addTableModelListener(TableModelListener l) {
        listeners.add(l);
    }

    public void removeTableModelListener(TableModelListener l) {
        int index = listeners.lastIndexOf(l);
        if(index!=-1) listeners.remove(index);
    }

    private void sendTableModelEvent()
    {
        TableModelEvent event = new TableModelEvent(this);
        TableModelListener[] array = new TableModelListener[listeners.size()];
        int index;

        array = (TableModelListener[])listeners.toArray(array);
        for(index=0; index<array.length; index++)
            array[index].tableChanged(event);
    }
}
```