PRIVACY APIS: FORMAL MODELS FOR ANALYZING LEGAL PRIVACY REQUIREMENTS

Michael J. May

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2008

Insup Lee
Supervisor of Dissertation

Carl A. Gunter
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

Acknowledgements

This work would not have been possible without the help and support of my advisors Carl A. Gunter and Insup Lee. Their advice has guided this work from its beginning all of the way to its final form. I am also grateful to my dissertation committee members whose input and corrections have been invaluable.

I also would like to acknowledge the support of the Ashton Fellowship at Penn which has helped support me financially for three years of my graduate studies.

Finally, I would like to thank my wife Raquel and daughter Nechama for their patience and support throughout my graduate studies. This work is dedicated to them.

Abstract

There is a growing interest in establishing rules to regulate the privacy of citizens in the treatment of sensitive personal data such as medical and financial records. Such rules must be respected by software used in these sectors. The regulatory statements are somewhat informal and must be interpreted carefully in the software interface to private data. Another issue of growing interest in establishing and proving that enterprises, their products, workflows, and services are in compliance with relevant privacy legislation. There is a growing industry in the creation of compliance tools that help enterprises self-examine to determine their status, but there is little formalization of what compliance means or how to check for it.

To address these issues, we present techniques to formalize regulatory privacy rules and show how we can exploit this formalization to analyze the rules automatically. Our formal language, *Privacy Commands* which combine to form *Privacy APIs*, is an extension of classical access control language to include operations for notification and logging, constructs that ease the mapping between legal and formal language, and a robust and expressive system for expressing references and constraints.

We develop constructs and evaluation mechanisms for the language which are specially suited to the modeling or legal privacy policies and show the usefulness of the language by developing several comparison metrics for Privacy APIs which let us compare the permissiveness of policies. We call the metrics *strong licensing* and *weak licensing* and show how they are useful in comparing Privacy APIs. To validate the robustness and flexibility of the language we show several involved case studies with a variety of policies including the US HIPAA Privacy Rule, the US Cable TV Privacy Act, and the Insurance Council of Australia's Privacy Code. To automate the evaluation of policy properties and comparison we develop and prove the correctness of a mapping from Privacy APIs to Promela, the input language for the SPIN model checker.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

To address the growing business of corporations collecting and using personal data, there are increased government attempts to create regulations that assure consumers their privacy will be respected. Privacy regulations are complex documents which often include requirements for enterprises who handle personal data to install safeguards and auditing controls to monitor access and use of personal information. The challenges posed by this arrangement are two fold.

First, since regulatory policies are structured in an idiosyncratic manner with differing levels of hierarchies, internal and external references, and implicit dependencies, it is difficult for non-experts to understand them. Key actions may be permitted or forbidden by a paragraph in the law if they are "permitted elsewhere" in the section or chapter, causing rules to have a complex reference and deference structure. Governing agencies and privacy activist groups frequently publish summaries of key privacy legislation to inform the public, but those summaries do not include all of the aspects of the legislation and are not designed for implementation. Enterprises need languages and automated tools to help them discover what actions are permitted and forbidden by the laws they are subject to.

Second, once an enterprise has implemented an internal policy, they may need to revisit it if the law changes, if they begin operating in a new legal jurisdiction, or if they enter new markets that are regulated differently. In such situations, the key questions then are whether their old policy is still valid under the new legal document and if not, what is it that they need to do differently.

The aim of this thesis is to construct a formalism which we call *Privacy APIs* that aids the modeling and understanding of regulatory rules for privacy as they appear in diverse contexts (such as countries or sectors) and use this formalization to analyze the impacts of changing or composing such policies. We develop Privacy APIs in a bottom up approach, from foundational theory to an implemented language. We begin with a theoretical model for privacy policies called Privacy Systems which is then concretized and applied to legal texts in the form of Auditable Privacy Systems. We then generalize Auditable Privacy Systems to Privacy APIs to better handle references and deferences. For the final language we provide a detailed syntax and semantics as well as a mapping from it to Promela, the input language for the model checker SPIN.

As a caveat, we stress that Privacy APIs are a tool to aid, but not replace, competent legal advice and domain expertise. We show the flexibility of the language through application to several real world legal policies, but since the formal language operates using the language of source text, its judgments are affected by how the source text is interpreted. For instance, rules which limit when information may be disclosed are subject to interpretation with respect as to what actions constitute disclosure. Also, while we introduce a flexible and expressive means for indicating precedence, we do not address the greater challenge of interpreting and integrating decisions from varying sources of law. For instance, in deciding whether a particular piece of information may be used for a purpose it may be necessary to examine rulings from privacy law in addition to tort, libel, and contracts laws.

As part of this introduction, we present more specific background on regulatory privacy policies in Section 1.1. We then outline notions of policy compliance in Section 1.2 and policy enforcement in Section 1.3 to motivate the work that we present in this dissertation. We discuss the methods used in this work and how we evaluate them in Section 1.4. We consider the challenges that we faced completing the work and its major contributions to the field of privacy policies and formalization in Section 1.5. We conclude the introduction with an outline for the rest of this dissertation in Section 1.6.

1.1 Regulatory Privacy Policies

An increasing number of government agencies and enterprises are finding a need to write down privacy rules for their handling of personal information of parties who entrust such information to them. These rules are derived from a complex set of requirements laid down by diverse stakeholders; they are often complex and may contain important ambiguities and unexpected consequences. Examples include sector-specific rules like the Health Insurance Portability and Accountability Act (HIPAA) [84] and Gramm-Leach-Bliley (GLB) Act [35] in the US and comprehensive privacy rules established by the European Economic Community [79, 80]. Increasing automation of data management using computer systems invariably means that these privacy rules become requirements for software systems that manage data affected by privacy rules. Analyzing conformance to these rules requires careful comparison of the permitted actions of a computer system with the (typically informal) regulatory rule sets.

Existing work in the area of legislative privacy policy is divided by discipline.

In the legal discipline, legal scholars, philosophers, and legislators discuss what are reasonable societal assumptions of privacy and confidentiality, what are fair information practices, and what sorts of disclosures and uses are appropriate for different contexts. Privacy policy formulations from the legal discipline involve information in all forms, whether digital, paper, or verbal. Legislators aim to create laws that are long lived, that is general enough to be relevant and prescriptive, but not too specific such that they become obsolete with changes in technology or business practices.

People who implement the legislative policy, those in the information processing discipline, including customer data brokers, computer scientists, information technology workers, and systems designers discuss what are reasonable levels of security for information, what are practical workflows for managing private data, and what sorts of safeguards are necessary for protecting information. Privacy policy formulations from the information processing discipline are more specific, focusing on information that can be easily transmitted, tracked, and audited. Policy specifications from this perspective make assumptions about technological and business process environments, exploiting features and norms that best fulfill their policy goals. The result of these assumptions is that implemented policy is often sector specific and not easily transferred to different problem domains.

The upshot of this division is that there is a need for a framework to understand both the legislative requirements and the information and workflow systems that are regulated by them. Such a framework will bridge the gap between the two views, giving clarity to both disciplines. Privacy APIs is a step in the necessary direction, work that will be aided by further developments in this area.

1.2 Compliance and Static Checking

There is a growing industry in regulatory compliance, ranging from software questionnaire based solutions for HIPAA (*e.g.,* GetHIP [67]) and the Sarbanes-Oxley Act (*e.g.,* Certus [31]) to private legal consultants. Compliance requires that enterprises create a large volume of paperwork: descriptions of their data practices, workflows, and security provisions, policy documents that describe their internal security and privacy policies, disclosure documents to provide to customers about their practices, access and disclosure histories for evaluation by compliance auditors, etc. At the heart of the large set of documents is the *essential policy* that the enterprise aims to implement—the coalescence of all the privacy and disclosure practices, workflows, security provisions, etc. of the enterprise.

Legislative privacy rules are designed to be general, but enforceable. This means that determining whether an enterprises' essential privacy policy is acceptable should be derivable from a comparison with the policy requirements as given in the relevant laws. The problem is that there is no common framework to compare the legal rules to the enterprise policies. A unified format that is flexible, generic, and formal enough for automated evaluation would greatly simplify deriving an answer to the question: *"Do the enterprise's current practices match with the law?"*

In addition to compliance, other static properties of legislation are of interest as well. Legal policy is complex as a rule, written by many loosely connected authors, and never created in a vacuum. Complexity is required in good legislation since comprehensive privacy policies are not easy to write. The downside of complexity is length and difficulty in gaining a big picture of what the ruling of a law really is. Evaluation of legal questions about privacy in complex real world situations requires investigations of terminology, norms, and many relevant clauses and restrictions. An additional source of complexity is resolving queries in the presence of preemption, deference to other laws, and resolution of jurisdiction.

Some static properties of laws such as cross-jurisdictional violations, relation to information safe-harbors, and the interface between criminal and civil laws are best left to legislators and legal experts. We can, however, gain traction in understanding questions of how a legal policy interacts with itself and other laws, what kind of references it makes to itself and other laws, when it explicitly defers to or preempts conflicting laws, what kind of loopholes it provides, whether it is internally consistent, and whether it maintains either its implicit and explicit invariants. Computational and automated formal methods for resolving such properties will greatly reduce the ambiguity and confusion created by newly proposed bills and freshly minted legislation by allowing legislator, lawyers, and lay people to see exactly what is being proposed.

1.3 Policy Enforcement

Existing languages and architectures focus on different areas of enforcement.

At the lower level are reference monitors and access control matrix systems that monitor access to objects in a single environment. Policy at this level is very specific and is enforced by the operating system or networked file system. Examples of this style include Harrison, Ruzzo, and Ullman's access control language framework (HRU) [56], Graham and Denning's access control matrix policy [49], the Chinese Wall security policy [25], and Role Based Access Control (RBAC) [39].

At the higher level are policy engines that answer queries about proposed actions in a distributed environment. Policy at this level is more general, describing combinations of actions and roles that are permitted or (less often) prohibited. Since the policy is designed for a distributed environment, enforcement of query results is left to the lower level. Examples of this style include W3C's Platform for Privacy Preferences (P3P) [93], the

Enterprise Policy Authorization Language (EPAL) [8], Cassandra [16], and some obligation systems.

Enforcement of regulatory policy requirements is a combination of regular distributed policy enforcement combined with a meta-policy enforcement. That is, regulatory policies often describe what sorts of policies must be enforced on the system. This higher level of enforcement requires a bigger picture view of system events than is classically available to policy enforcement engines. The development of an architecture to support a big picture view and enforce meta-policy requirements will be an enabler of enforcement of regulatory policy requirements.

1.4 Methods

Figure 1.1 gives an overview of the methods used in this dissertation. The figure shows the steps we take in adapting a legal text for formal analysis. We first translate the legal text by hand into a Privacy API which consists of commands and constraints in the Privacy Commands language. This step requires the input of legal professionals to indicate how the provisions in the law are to be interpreted. We then use the structured translation algorithm in Section 6.2 to translate the Privacy API into a Promela model. We use the Promela model, combined with initial state provided by the user as input to SPIN. SPIN uses the model, the initial state, and properties expressed in LTL to evaluate static properties of the models. The properties that we are most interested in are based on the relations developed in Section 5.5. Using the properties discovered by SPIN, we can then return to the legal text to map the results to the features of the source text. SPIN makes such mapping easier by providing full traces of events which lead to property violation.

In order to evaluate our work we develop several case studies of its usefulness and flexibility. Our choice of legal documents to study is affected by several factors: their complexity (*i.e.,* how interesting they are to study), their relevance, and their amenability to modeling (*i.e.,* they are specific enough to quantify objectively).

We have three particular interests to explore with these studies. The first is showing that our methods scale well to large policy sets. The second is using our tools to show

$$\begin{tikzcd}
\text{Legal Text} \arrow[rr, "\text{Hand Translation To Privacy Commands (Sections 5.2, 5.3)}"] & & \text{Privacy API} \arrow[d, "\text{Structured Translation (Section 6.2)}"'] \\
\text{Properties} \arrow[u, "\text{Checking Source}"'] & \text{SPIN} \arrow[l, "\text{Property Checking (Sections 7.1–7.3)}"'] & \text{Promela} \arrow[l, "\text{Model Adaptation (Section 6.3)}"'] \\
& \text{User input} \arrow[u, "\text{Invariants For Relations (Section 5.5)}"] \arrow[ur, "\text{Initial State}"'] &
\end{tikzcd}$$

Figure 1.1: Moving from text to Promela via Privacy Commands

compliance to both policies and meta-policies. The third is showing how policies interact, preempting, deferring to, and modifying each other's rulings.

HIPAA As part of our previous work we performed a case study on the HIPAA consent requirements. The case study, described in Chapter 7 compared two versions of the same subsection of the regulation, one from 2000 and one from 2003. Our results showed the usefulness of our approach and presented some interesting results on the comparison. In particular, we looked at three properties of the 2000 version that were pointed out by commentators as being inappropriate. We then used our representation to discover those issues in the 2000 version and to explore whether those properties were also true in the 2003 version.

Cable TV Privacy Act and TiVo As an exercise in the conformance determination features of Privacy APIs we develop a case study comparison between the US Cable TV Privacy Act of 1984 [48] (CTPA) and the TiVo corporation's (`tivo.com`) privacy policy [62] that is subject to it.

Australian Insurance Privacy Policy To exercise the comparison features of Privacy APIs we show how to compare two different privacy laws, HIPAA from the previous case study and the Insurance Council of Australia's privacy guideline [72].

1.5 Contributions and Challenges

We based the requirements for our work on the challenges faced by enterprises and individuals who must deal with legal policies. We sought a balance between descriptiveness, accuracy to the source text, and enabling automated formal analysis. Out fundamental goal was to allow better analysis, examination, and comparison of legal policies so we faced the following major challenges in doing so:

1. In order to enable analysis, we needed to design a policy representation suited to the structure of legal texts, including idiosyncratic and subtle methods of referencing other text locations as used in real world legal policies. We also needed to avoid "overfitting" the representation by tailoring it to one target policy to the exclusion of all others.

2. To enable quicker and analysis of legal policies we sought to enable automated exploration of properties.

3. To make comparison of legal policies easier and more objective, we sought mechanisms to compare and contrast policies.

We also faced numerous other smaller challenges in the design and execution of this work. We enumerate a detailed list of requirements later in Section 3.2 as part of our discussion of the complicating properties of legal policies. In meeting the above challenges we offer three major contributions to the area of privacy policy language design and analysis.

1. We present the Privacy Commands language as a representation suitable for capturing the permissions and structures of complex legal privacy policies. With proper expert advice, the formalism can be used to capture aspects of legal policies which are suitable for automated exploration. The evaluation engine for the language interprets

Privacy APIs (collections of commands and constraints in the Privacy Commands language) in a manner akin to how a person might interpret the source policy, making its results intuitive and readily understandable based on the source text.

2. We devise a methodology for modeling the various methods restrictions and constraints are stated in legal policies. Constraints may include references to other policies, invariants, situations where only one of a set of requirements need to be satisfied, and situations where a referenced constraint is only partially applicable. We devise a method for representing such constraints that is expressive, flexible, and subtle.

3. Since comparisons between privacy policies is not always straightforward, we devise objective comparison metrics. Our two fundamental metrics are *strong licensing* and *weak licensing* and they act as building blocks for derivation of higher level comparisons such as observational equivalence and comparison of permissiveness. We define our metrics with tunable granularity, giving us the flexibility to compare anything from single policy paragraphs in a specific situation to whole policies under any potential situation. Our metrics are based on the notions of strong and weak bisimulation from the process algebra community.

As a secondary contribution we offer a novel case study in the use of formal methods tools for the analysis of privacy policies. We devise a methodology for translating from the Privacy Command language to Promela, show its correctness, and offer three in depth examples of how we may use the SPIN state space analyzer to examine policy properties. This study is a strong indication of the usefulness of formal analysis tools in the examination of privacy policies. This observation is important since it may open up a new and fruitful avenue in the study, design, and management of privacy policies.

1.6 Organization

The rest of this dissertation is organized as follows. Chapter 2 discusses background and related work in access control, privacy policy formalization, and legal privacy policies.

Chapter 3 motivates the need for Privacy APIs by discussing the real world problems with respect to legal privacy policies which we assist in solving. Chapter 4 develops Auditable Privacy Systems and the Privacy Commands language, the tools that we develop in this dissertation. Chapter 5 develops the formal language that we use for Privacy APIs as well as the notions of strong and weak licensing. Chapter 6 develops the translation technique we use for moving from the formal language to verifiable code in Promela. Chapter 7 presents extensive case studies in the application and use of Privacy APIs. Chapter 8 concludes.

Chapter 2

Background and Related Work

This work bridges three areas of research: access control, privacy policy formalization, and legal privacy analysis. The purpose of this chapter is to provide background for the reader on the fundamentals of computer access control policies, privacy policies, and legal policy analysis and the relevant related work in those areas. Since each of the topics discussed here are the subject of decades of research, we briefly touch upon the topics at a general level and only discuss closely relevant material in depth.

The rest of this chapter is organized as follows. We begin with a summary of privacy policies as they have been implemented in legal documents in Section 2.1. We discuss several legal policies in Section 2.1.3 that we will use in later chapters and case studies as well as related work dealing with technical approaches to modeling them. We then present an overview of access control terms and policy formalizations in Section 2.2 as a lead in to the access control theory used in this work. Last, we discuss relevant related work in the area of privacy policy formalization in Section 2.3. Certain closely related work is discussed in depth with extensive examples. We conclude in Section 2.4.

2.1 Privacy in the Law

In this section we discuss some background for understanding legal privacy policies, present some particular attributes and complexities associated with them, and give some illustrative examples from the laws of the United States, the European Union, the United Kingdom, and Australia. We consider technical approaches to understanding legal texts in Section 2.1.4.

2.1.1 Legal Background

An understanding of the tradition of privacy in the laws and traditions of western countries is essential for an appreciation of the task addressed in this work. Certainly a presentation of legal minutia and litigation are beyond the scope of this work, so we focus on how relevant issues relate to the policies we are considering. Since this is not the work of a legal expert, the presentation here relies on the writings and analysis of legal scholars and historians and not from original scholarship by the author. The goal of this discussion is to motivate a computer science approach to this subject, so we will focus on the aspects of legal privacy most addressable using our methods.

Discussions of privacy in the United States' legal framework began well over a century ago in law review journals. Legal scholars debated the notions of privacy and confidentiality and where they belong in the considerations of society. The earliest discussions involved the definitions of privacy, as Warren and Brandeis' [95] formulation of privacy in terms of the "right to be let alone" as well as numerous other definitions by other scholars. Their discussion came at a time when the relatively new technique of photography was becoming widespread and so is in the context of the right of people to prevent unauthorized photographs of them being public circulated. The advent of computers and digital storage and retrieval has changed the scope of concern for legal privacy consideration, but the old definitions are still applicable with respect to their demand for people to have relief from the courts against abuses of information about them.

Privacy laws in the last several decades have focussed on the twin aspects of data collection and data use. As we discuss below, US regulations differ between industrial sectors while European and Australian privacy laws are cross-industry. Despite the difference of scope, there are several basic concepts and protections that appear in nearly all privacy laws.

One common set of regulations relating to data collection is provisions requiring collectors to disclose the purpose for which they are collecting the information. Regulations of this type appear, for instance in the Australian Market Research Privacy Principles Section E paragraph 1.5 [74] and the US Gramm-Leach-Bliley Act 15 USC, Subchapter I, Section 6803(a)–(b) [35]. They are requirements that data subjects be told what information about them is being collected and what types of procedures will be followed in the use the information. Such provisions are especially important in scenarios where people may not even be aware that information is being collected about them (*e.g.,* when storing a browser cookie that will be sent to a tracking server).

A second common set of regulations on collections is limitations on the *correlation and compilation of records* from multiple data sources. Such limitations are placed to prevent owners of one database from combining their records with another database to produce a more revealing picture than would be available otherwise from either database. Particular worry about government surveillance using multiple data source correlations led to the inclusion of such provisions in US law regarding government databases (*i.e.,* The US Privacy Act [73] 5 USC §522a(o) and note)), but not for commercial databases. There is a large industry in the collection and correlation of consumer information for businesses and retailers (*e.g.,* Acxiom (`acxiom.com`), Equitec (`equitec.com`)) that exists beyond the scope of the US Privacy Act. The issue of database correlation and combination has come to debate several times in the past few years, for example in 2000 with the merger of a web advertising company and a customer information company [29, 85, 82].

A third common set of regulations is requirements for safeguarding and securing collected information. Requirements for safeguarding typically include physical, technical, and administrative mechanisms. Technical safeguards are difficult to get right in legal documents since terms such as "encrypted" and "password protected" are ambiguous if not rigorously defined by law. Some specifications are couched with phrases such as "reasonable and appropriate" (*e.g.,* HIPAA [84] [§164.306(d)(3)(i)-(ii), v. 2003]) which places a burden of proof on the data holders that the safeguards they implemented in fact are so. There is certainly a desire to ensure that legislation is not tied to a particular software package or transient technology (*e.g.,* HIPAA Final Security Rule Section III paragraph 2, page 8336 [45]). With respect to our policy formalizations, we do not implement any details not specifically mentioned in the legal text. Thus, our policy analysis takes place at the level of specificity contained in the document.

A fourth common set of regulations are limitations on use. They include common restrictions such as rules about disclosures of information and restrictions of actions that may be performed based on information. More procedurally complicated restriction include requirements for formulating business and administrative procedures for the handling and approval of proposed uses and allowances for individuals to access and correct information about themselves. Since our policy formalization focusses on uses and disclosures of information, we have limited capability in reflecting rules about business processes and workflows.

Lastly, a fifth common set of regulations relate to the depersonalization or anonymization of records. Anonymized records are needed for statistical analysis and research, both clinical and otherwise so there has been research into mechanisms for achieving record anonymization while maintaining the properties of data (*e.g.,* privacy preserving data mining [4]) as well as studies showing some schemes that are not truly secure (*e.g.,* work by Malin [65, 66]). Such research helps inform technical discussions of what types of anonymization are effective and appropriate for different research and analysis needs.

2.1.2 Complexities in Legal Policies

Having discussed some common themes and requirements of legal policies we now discuss some aspects of legal policies that tend to make them complex to implement rigorously in policies derived by computer scientists. Legal policies include common access control concepts discussed above as well as complex dependency structures that often create a mesh of rules rather than a distinct policy. Policy interpretation then involves determining how a particular policy ruling fits into the big picture.

A first reading of legal policies shows that concepts in their technical and administrative safeguards have much in common with classical access control work. This should not be surprising because classical access control theory is based on existing human level controls in place in government, military, and commercial environments (*e.g.,* Bell-LaPadula [18], ORCON [50], Clark-Wilson [32], Chinese Wall [25], etc.). The commonalities include the concepts of agents in a system that are assigned roles, vocabularies of precisely defined terms, dependencies on purposes and obligations, and formulations for exceptions. These commonalities mean that legal policies share a large enough set of terms with access control systems that we can use one to enforce the other.

Despite the conceptual similarities between the two fields, legal policies have a complex dependency and decision mechanism not commonly seen in access control theories. Dependencies in policies range from simple references between paragraphs to wholesale deference to any other existing law (*e.g.,* such an exception in HIPAA [46] [164.512(c)(1)(iii), v.2003]). Simple references to other paragraphs are textual pointers that allow one paragraph to invoke the functionality of another. Another type of reference refers just to the condition of another paragraph, allowing one paragraph to condition its functionality on another paragraph's guard. In terms of deference, many documents explicitly delineate when they defer to or preempt competing or conflicting policies, but the delineations are not always symmetric (*e.g.,* A states that it preempts B, but B does not say that it defers to A). When laws do not explicitly state deference or preemption guidelines we leave the interpretation to legal experts.

Intertwined with issues of deference and preemption are laws with limited scope and coverage. For example, laws may be addressed to particular industrial sectors (*e.g.,* only financial services providers), specific categories of use (*e.g.,* only for market research), or even to specific media (*e.g.,* only electronic records). When a particular policy speaks to only a limited set of cases its impact on the mesh of legal policies is also limited and we must be careful in applying it.

A fourth, perhaps more subtle, point regarding management of legal policies is the difference between those laws that are *actively regulated* and those that are *reactively regulated*. An actively regulated law implies that an authority actively checks implementation of the legal policy and regularly reviews the actions of *covered entities* (*i.e.,* parties covered by the particular law) to ensure compliance. Under a reactively regulated law, however, covered entities may only be checked or reviewed for compliance when someone files a complaint. A reactively regulated law that is couched in terms of "reasonable and appropriate" places a larger burden on a plaintiff seeking relief from a covered entity than an actively regulated law that is specifically worded. The upshot of this is that in designing systems to support the enforcement of legal privacy policies we must tailor the system to the method of enforcement. A system to enforce an actively regulated law will require strong preventive enforcement guarantees that violations will not occur. A system to enforce a reactively regulated law will require weaker enforcement guarantees coupled with stronger audit and tracking facilities to enable after the fact investigations. As system and computer policy designers, not legislators, our goal is to make systems that respect the law as well as enable its enforcement appropriately.

2.1.3 Policy Examples

To help the reader understand the policies that we discuss in this work, we now present and briefly some example privacy policies, some of which we will use for examples and case studies in this work. An important contrast to note is that US privacy law is primarily sectoral, so particular policies are limited to industries and specialities. Thus, while the HIPAA law discussed below provides comprehensive protections for personal health information held by certain entities, it does not provide any protections from health information held by other entities.

HIPAA

The US Health Information Portability and Accountability Act of 1996 [84] obligates the Department of Health and Human Services (HHS) to define rules and regulations for the privacy, security, and portability of electronic health information. Pursuant to it, the HHS created documents called *Rules* to fulfill its new mandate. Of interest in this work is the Privacy Rule which contains rules regarding the use and disclosures of personal health information stored in electronic format. The Privacy Rule (as with the rest of the HHS proposed rules) went through a few revisions. The first version was released in 2000 [42].

Its release was followed by a comment period during which individuals and affected parties were allowed submit comments to the HHS. Relevant comments were summarized into a comments document [43] and published by the HHS. After the comment period, the Rules were revised and the new version of the Privacy Rule [46] (along with the other rules) was published in 2003. It has been amended several times since by various laws. Each Rule defines particular phase-in stages and deadlines by which covered entities must be compliant with the law.

Of interest in HIPAA is its limitation of covered entities to health plans, health providers, and heath care clearinghouses (*i.e.,* entities that convert health information from non-standard format to standard format or vice versa). Non-covered entities that maintain and use electronic health information in conjunction with covered entities (*e.g.,* a billing service) must be bound by *business associate contracts* between them. Such contracts need not have the same force as the HHS Rules, but must contain a level of privacy assurance and accountability to the covered entity. Such contractual relationships are of interest to this work since they provide a good example of policy conformance analysis. That is, it is required that business associate contracts limit the actions of associates to within a certain approximation of the HIPAA rules.

Colloquially, the term "HIPAA" is used to refer to the contents of the Privacy and Security Rules as issued and enforced by the HHS even though strictly it refers to the Act of 1996 and not the subsequent Rules. In this work we use the term "HIPAA" in its relaxed meaning, to refer to the law itself or the Rules. When needed for specificity we will clearly mention if we refer to the Act, the Privacy Rule, or the Security Rule.

The Rules are codified in the Code of Federal Regulations (CFR) and published by the Federal Register on a quarterly basis. The Rules are located in Title 45 of the CFR, Subtitle A, Part 164 and are further divided into subparts, sections, and paragraphs. Throughout this work we refer to text in the Rules by their section and paragraph headings as well as the year in which they are published and set the reference out in square brackets. For instance, the reference [§164.500(a)(1), v.2003] refers to the text published in 2003 in Part 164, section 500, paragraph (a)(1). The reference [§164.502(b)(2)–(4), v.2000] refers to the text published in 2000 in Part 164, section 502, paragraphs (2), (3), and (4).

Compliance with the Privacy Rule requires covered entities to appoint a Privacy Officer who oversees the design and implementation of policies to enforce the information security rules. When the Rule was first published in 2001, covered entities rushed to review their current practices, compile detailed lists of their information technology (IT) assets, and perform a gaps analysis to determine what practices, policies, and IT systems needed to be updated. Larger covered entities employed HIPAA consultants to or companies which specialize in compliance to aid them. Case studies from covered entities emphasize the review process and gap analysis performed during the compliance evaluation as a major burden in the process [26, 69].

In this dissertation we use selections from HIPAA's Privacy Rule as running examples. The sections that we select from generally concern the use and disclosure of protected health information for treatment, payment, and health care operations. We choose these sections since they cover many common situations of information use. The requirements that they include therefore represent very important aspects of the privacy protections included in the Privacy Rule.

The rules regarding the need for consent for treatment, payment, and health care operations from the 2000 version of the Privacy Rule have six major divisions (paragraphs):

§**164.506(a)** The first paragraph (a) contains the rules about when a covered health care provider must obtain consent from an individual prior to the use or disclosure of the individual's protected health information. The top level text gives a general constraint that unless permitted in the paragraph, a covered health care provider must obtain consent before use or disclosure for treatment, payment, or health care operations. Exceptions are made for health care providers who have indirect treatment relationships (*e.g.,* a radiologist), information regarding prison inmates, emergencies, health care providers required by law to provide treatment who attempt to obtain consent but fail, and health care providers who infer consent from an individual but cannot obtain written consent due to barriers in communication.

Paragraph (a) states that in cases where the health care provider is not required to obtain consent as per the exceptions above, it may do so if it is obtained appropriately as per the section (*i.e.,* 164.506). It also states that consent given to one health care provider does not permit another to use or disclose unless they are a joint entity as defined in paragraph (f).

§**164.506(b)** The second paragraph (b) contains the rules for the management of individual consent such as how the consent may be combined with other documents, revocation, and the retaining of signed consent.

§**164.506(c)** The third paragraph (c) contains the rules for the content that must be present in a signed consent form.

§**164.506(d)** The fourth paragraph (d) defines a "defective consent" as one that is either revoked or that does not fulfill the rules in (c).

§**164.506(e)** The fifth paragraph (e) defines how two conflicting consent documents signed by the same individual must be resolved. It prescribes that in case of conflicting statements, a health care provider must either follow the stricter of the two statements or communicate directly with the individual.

§**164.506(f)** The sixth paragraph (f) contains the rules for how health care providers in an organized arrangement may provide and receive joint consents from individuals. Joint consent documents must have the documentary content of paragraph (c) but may be modified to refer to all the parties in the arrangement. Revocation of a joint consent by an individual must be communicated to all parties in the arrangement.

The rules regarding the need for consent for treatment, payment, and health care operations from the 2003 version of the Privacy Rule have three paragraphs:

§**164.506(a)** The first paragraph (a) is a general rule permitting the use and disclosure of protected health information for treatment, payment, or health care operations without consent. It references paragraph (c) with respect to the details of consent. Exceptions to the rule include situations where an authorization is required as per §164.508(a)(2)–(3).

§**164.506(b)** The second paragraph (b) grants permission for covered entities to request consent from individuals for use and disclosure for treatment, payment, or health care operations. Such voluntary consent, however, does not permit use or disclosure when an authorization as per §164.508 is required.

§164.506(c) The third paragraph (c) details the cases where use and disclosure of protected health information for treatment, payment, and health care operations are permitted. Five subparagraphs detail the permitted cases.

The rules in §164.506 reference the requirements for special written authorizations in §164.508. The rules for authorizations there are given in three paragraphs:

§164.508(a) The first paragraph (a) states that authorizations must be obtained to use or disclose psychotherapy notes or to use protected health information for marketing. Some exceptions to each case are given with references to many other sections.

§164.508(b) The second paragraph (b) delineates general requirements for authorizations, how they are obtained, revoked, and combined with other documents when presented to individuals.

§164.508(c) The third paragraph (c) delineates the required contents of authorization documents and the requirement that individuals be given a copy of any consent signed.

Aside from rules regarding consent, HIPAA imposes many other rules regarding the management of health information. One important one relates to the logging of activity [§164.308(a)(ii)(D), v.2003]:

> (D) Information system activity review (Required). Implement procedures to regularly review records of information system activity, such as audit logs, access reports, and security incident tracking reports

The requirement to log all actions and operations is common enough that we include special operations to manage logs for auditing.

EU Privacy Directive

The European Union's 1995 Data Protection Directive 95/46/EC [79] is its main regulation for the protection of personal information. Unlike the US which has taken a sectoral approach, the European Parliament chose to devise a broad policy document that would cover all areas of information processing. The policy directive has been followed up with some additions to handle more recent technologies (*e.g.,* communication and location information in 2002 [80]), but the 1995 law sets the groundwork for all subsequent regulations. The document establishes privacy principles which generally limit categories and types of actions. Its principles include meta-policy to describe what laws member states should create to support it. It has a different structure than the US privacy laws, using less references and larger self-contained paragraphs.

We do not consider a case study involving the EU Privacy Directive in this work, although comparison between its privacy principles and US regulation would be an interesting direction for further research.

Insurance Council of Australia Privacy Code

The Australian Privacy Act of 1988 [71] sets National Privacy Principles (NPPs) for the protection of information about individuals which are binding on all sectors of industry in Australia. Section 18BG of the Privacy Act, as activated in 2001, permits private industry groups to compose sector specific privacy codes which, if approved by the Privacy Commissioner, may replace the NPPs for the constituent members of the group. As of this writing in October 2007, the Privacy Commissioner has approved three private sector privacy codes, two are under consideration, and one has been revoked.

The Insurance Council of Australia (ICA) (`ica.com.au`) issued a General Insurance Information Privacy Code (GIIPC) as per that permission that is applicable to all insurance providers who choose to belong to the ICA. The code was approved by the Privacy Commissioner in April 2002 and became effective as of that date. The code was revoked by the Privacy Commissioner on April 4, 2006 due to lack of industry uptake and several complaints from individuals. Even though it is no longer in active use, it provides a good example of a legal privacy policy and a contrast in writing style from the HIPAA rules. We consider a piece of ICA privacy code in a case study in Section 7.2. The full text of the relevant section for disclosure of personal information is included in Appendix B.3.

Cable TV Privacy

The Cable TV Privacy Act of 1984 [48] (CTPA) restricts the uses and disclosures that cable operators can perform on certain categories of subscriber information. The CTPA is recorded in US Code Title 47, Chapter 5, Subchapter V-A, Part IV, Section 551. Section 551(c) includes the rules regarding the disclosure of personally identifiable information which we consider as part of a case study in Section 7.3. In the case study we compare the permissions for disclosure in the CTPA with the privacy policy of TiVo.

TiVo Corporation (`tivo.com`) produces digital video recorders (DVRs) and service contracts that allow subscribers to download television show times, record shows, rewind shows during viewing, and other convenient features. When a service contract is in place, TiVo DVRs collect information about the viewing habits of their users—which shows they watch, when they change channels, *etc.*—and a portion of this information is sent back to TiVo each time the DVR "dials home" to receive new listing information [68, 89]. TiVo initially did not operate over cable lines and therefore was not subject to the Cable TV Privacy Act. However, they recently began offering their service for use over Comcast Cable's (`comcast.com`) network in certain areas [87] which may therefore place them under the requirements of CTPA.

The TiVo policy includes 9 sections, two of which are of interest to us in our case study in Section 7.3:

Section 1 "Our User Information Definitions" The section contains definitions for terms used in the policy, including definitions of personally identifiable information.

Section 3 "Disclosure of User Information" The section contains the policies for when TiVo discloses user information to others.

2.1.4 Related Technical Work

There has been work on implementing and analyzing legal privacy policies from the natural language, logic, formal methods, and artificial intelligence communities. Since this work presents a policy and formal analysis approach to legal policies, we consider related work that approaches the technical aspects of analyzing and implementing legal privacy policies. A full discussion of approaches from the other communities is beyond the scope of this work.

With respect to designing and enforcing health information, Anderson presents a list of requirements to secure a national patient health information database in the UK [5]. He considers features of the system that must be legislated to ensure patient privacy, such as sending automatic notifications of access by doctors to patients and strong audit controls. Becker and Sewell use Cassandra, a trust management language, to implement a system that is compliant with the UK's actual health information data Spine specification [17]. To be sure, their work is an implementation of a system that is *compliant* with the specification, not necessarily a close following of the specification.

Antón, *et al.* have developed a methodology for analyzing the different statements made in privacy policies. They take each statement in a privacy policy and figure out its goal. They then classify statements by their goals (*e.g.,* technical, legal, business) and as either privacy protections or vulnerabilities. A protection statement is a goal that will keep information safe. A vulnerability is a goal that may imply use or disclosure of information. [38]. Using their goal based analysis, they analyze how privacy policies have changed before and after HIPAA, based on the number and types of goals that policies contain [6]. Their metrics are rather vague since their analysis compares the number and category of goals in policies rather than what those goals really mean.

As a study of the power of EPAL, a policy language that we discuss in Section 2.3, Powers, *et al.* translate a section of Ontario's Freedom of Information an Protection of Privacy Act [75] to EPAL [81]. As with our model, their resulting policy closely mirrors the legal text, having one rule per textual paragraph. They show how the policy allows them to process a request and return a ruling in a similar manner to how a human would, but do not analyze or verify their policy formulation.

While it is more of technical discussion than a theoretical discussion, Hogben considers P3P's relation to EU's privacy directive in the context of cookies and other potentially privacy invading technologies [59].

The above works are united by their focus on implementing systems or policies that are influenced by legislation. They do not consider the problem of analyzing legislative policy themselves or comparing them to others. In contrast, this work explicitly takes that consideration and presents tools for better understanding and analyzing legislation in addition to implementing it.

There has been significant study of legal ontologies and structure of legal text in machine learning and natural language processing fields. As examples, Dinesh, *et al.* [37] use formal semantics to extract structure from regulations for blood banks. They consider the reference structure noted above in particular regarding global references. Breaux and Antón [24] analyze the formal semantics of a HIPAA privacy rule summary sheet [44], analyzing the phrases and structures that commonly appear in it as a model for other legal privacy policies. Interestingly, as part of their study they count the number of references in HIPAA's Privacy Rule, but do not consider their types. As we discuss in Section 4.1.2, one of the more complex and subtle aspects of modeling legal policies is capturing the intent of the many types of references used.

2.2 Access Control Formalization

The classic goal of access control is to maintain and enforce a set of *permissions* for *agents* over *objects* in a closed system. Access control systems were originally designed to protect computer system files and resources from unauthorized people. As policies became more sophisticated, notions of delegation, obligations, roles, and capabilities were added. Since our work builds on these fundamental concepts, we discuss some of the theoretical frameworks that have been designed to support them.

2.2.1 Access Control Terms

The basic problem of managing the rights that people have on files and resources in a computer system can be imagined as a mapping of people (agents) and files or resources (objects) to permissions. The key terms and concepts that we use in our models are as follows.

Objects Objects are abstractions of anything that can be affected, used, or modified by an agent in a computer system, including files, printers, storage media, and other agents.

Agents Agents are an abstraction of people who interact with other agents or object, often through a programmatic interface. Often it is useful to think of an agent as an operating system process that is performing the actions of the person controlling it. In our work, we use agents heavily, not distinguishing between people and the software or hardware agents that perform tasks on their behalf. Thus, if a doctor has the right to view a file, it does not matter whether the doctor sees it on paper or on her computer screen. This decision abstracts away the notions of authentication, user sessions, and other lower level systems details which are typically not considered in legal privacy policies.

The policies that we consider in this work are often concerned with two classes of agents in every action: the actor who performs the action, and the agent(s) who the action affects. If the action involves sending information to another agent, we also consider the agent who receives the information (*i.e.,* the recipient). If the action involves information about another agent, we consider the agent who the information is about (*i.e.,* the subject).

Rights Rights are abstractions of permissions that agents may hold over objects. When considering rights in policies we are normally not so concerned with the semantics of the rights so much as how they are created, transferred, and deleted by policies. We normally assume that all rights are stored in a central repository, an access control matrix, or some other data repository that is shielded from modification by others. Thus, policies that we create implicitly trust the integrity of the permissions repository.

Roles A role represents a set of authorizations that a given agent has. In this work we use roles as static descriptions of agent authorizations. This simplification is to the exclusion of a large body of work related to the treatment of roles as dynamic descriptors which are added, deleted, and modified. We make this simplification in order to simplify our policy language and formalism since the legal policies we consider do not explicitly discuss the mechanisms used to dynamically manage them.

Tags Tags are named boolean variables which we use to store meta-information about objects. They are used to keep track of information that the policy needs to be aware of, allowing decisions to be made about the properties of an object. Tags may be derived from the contents of an object (*e.g.,* personal information), its provenance (*e.g.,* file came from the CIA's secret records), or any other aspect. At the policy level we are normally concerned only with the truth values of the tags related to an object, however, not the manner of how tags are set by the environment or derived.

2.2.2 Access Control Policies

Fundamental research in computer access control systems took off in the 1970s as methods of protecting files and resources in operating systems. Many of the models were based on human level access control policies then enforced by the military, intelligence, and banking communities.

Graham and Denning

A representative example of early access control matrix policy and theory is the "protection system" of Graham and Denning [49]. In their system agents, objects, and permissions exist in the context of a system-wide access control matrix. The rows of the matrix are the agents, called *subjects* in their work. The columns of the matrix are the objects. The system is assumed to have a root subject who is the prime mover for all events in the system.

The system supports five permissions: read, write, execute, owner, and control. The first three in the list also have transferrable versions, indicated by the appending of an asterisk.

Actions in the system are called *events*. The events allowed by the system are: object and subject creation, object and subject deletion, granting and removal of permissions, transfer of permissions, and reading of permissions.

Events in the system are defined as complex entities and are allowed or disallowed by the protection system policy. For example, any agent p can create an object o by executing the event p **creates object** o. Upon completion of the event a new object o has been created and p owns it: $M(p, o) = \{\mathsf{owner}\}$. Only the owner of o, p in this instance, can later delete it with the command p **destroys object** o. The system verifies p's ownership by checking that $\mathsf{owner} \in M(p, o)$ and then removes it from the matrix. The policy regarding the creation and deletion of subjects is similar.

Agents are restricted from reading the access control matrix as a whole. Instead, an agent can request to read individual rows in the matrix and are allowed or denied by the policy in accordance with its permissions on the referenced agent. A reference monitor is used by the system to track requests for actions and to protect the access control matrix from unauthorized reading and modification.

Some examples of events in the Graham and Denning system are:

1. Root creates subject Alice
2. Root creates object File
3. Root grants read to Alice on File

After these commands execute the access control matrix looks as in Table 2.1. The matrix contains two rows, the principals Root and Alice. The matrix has three columns: Root, Alice, and File. Root owns both Alice and File. Alice has control on Alice (meaning that Alice can read her own entries in the matrix) and read on File.

Table 2.1: Access control matrix under Graham/Denning

	Root	Alice	File
Root	control	own	own
Alice		control	read

Graham and Denning consider issues of safety and correctness of their protection system and ways to implement it efficiently. They consider extensions to permissions that include management of delegation, transfer, and message passing. They relate their model to existing operating system implementations as well. The base policy and operations of the protection system can also be used by more complex policies to enforce higher level rules by breaking them down to simpler operations.

Our work is in the spirit of a protection system, but differs in an important way. The requirements for satisfying legal policies differ from those needed for operating systems, but we retain the spirit of identifying a useful base system that will support those requirements. We differ in that we decouple the policy for a system from its low level implementation. That is, we do not presume that there is any underlying policy built into the system that a higher level policy must respect. Instead, we create a language for defining policies and an evaluation engine which respects certain invariants of the language.

Harrison, Ruzzo, and Ullman

A more abstract language for writing access control policies is developed by Harrison, Ruzzo, and Ullman [56]. Like Graham and Denning, their system relies on an underlying access control matrix and reference monitor. Their contribution is in their formulation of events and their definition of a form of access control policy. Actions in the system are performed by "rules" which agents invoke.

Rules in the Harrison, Ruzzo, and Ullman work are transaction-style commands that execute in sequence on a single system. They discern two types of rules. *Primitive operations* of the system manage the reading, granting, and revocation of rights and the creation and deletion of principals and objects in the matrix. The set of *Commands* consist of (optional) conditions and a series of primitive operations that are executed transactionally.

Example 2.2.1 As an example of a rule, consider a command for creating an object in the Graham and Denning policy in the following command:

```
1   command   CreateObject (actor, object)
2             create object object
3       and   enter owner into (actor, object)
4       end
```

Line 1 declares the name of the rule and that it is of type command. It takes two parameters: the agent carrying out the action and the name of the object to create. Line 2 executes the primitive operation that creates a new row in the matrix. Line 3 executes the primitive operation that enters the owner right for the actor on the new object. Line 4 concludes. □

Harrison, Ruzzo, and Ullman's access control system structure of primitive operations and commands that execute them is a flexible framework that allows arbitrary policies to be composed.

As a caveat, Harrison, Ruzzo, and Ullman show that a specific issue of *safety*, whether granting a specific permission to another principal could lead to the permission being leaked to an untrusted principal, is in general undecidable since a Turing machine can be constructed whose termination depends on safety. They note, however, that some changes to their model would invalidate the undecidability result.

Other access control formulations such as the Chinese Wall [25] have addressed such issues as the expiration of rights in the context of rights becoming constantly more restrictive as time goes on. Policies in the model are designed to enforce separation of duty requirements commonly found in banking and investment companies. Analysis of the static properties of the policy lead to modifications in the policy to allow for the expiration of restrictions to enable greater flexibility and the notion that individuals may change positions and so lose previous restrictions with time.

Role Based Access Control

Roles were introduced to access control systems to make policies more scalable and manageable. In role based access control (RBAC) [39] there is a set of roles that are assigned rights in the policy. Rather than assigning individual rights to each agent independently, each agent is assigned a set of standard roles. Policy decisions are based on the roles that the requesting agent is assigned as in the following example.

Example 2.2.2 (Roles)

The access control system for an office has two roles: Secretary, Manager. Two people work in the office: Sam, Mary. The office policy allows Secretaries to only view and edit object that belongs to them, but Managers can view and edit any object. The role and policy tables are as follows:

Name	Role
Sam	Secretary
Mary	Manager

Role	Rights
Secretary	Read and Write own
Manager	Read and Write any

When Alice tries to view an object that belongs to Sam, she is allowed access. When Sam tries to view an object that belongs to Alice, he is denied access.

After the manager sees that Sam tried to access an object he was not allowed, he is fired. Mary hires a new secretary Tina. Tina is assigned the same role as Sam and thus the same permissions, but the policy table does not need to updated. □

Some policies that allow principals to have multiple roles require that actions be performed under only one role. Thus, a principal who has multiple roles must *activate* a role that is effective for some duration as in the following example.

Example 2.2.3 Activating Roles

The access control system for the office is extended with a third role: IT Staff. Members of the IT Staff role can view any object in the system, but can only edit objects they own. Ian, who is already a manager in the office, is assigned to the new role of IT Staff as well. The role and policy tables are extended as follows:

Name	Role
Ian	Manager, IT Staff

Role	Rights
IT Staff	Read and Write own, Read any

When Ian tries to view an object owned by Mary the policy requires him to activate either Manager or IT Staff. Either one will let him view the object. However, when he tries to edit a file that belongs to Mary, he can only do so if he chooses to activate the Manager role. □

Advanced policy systems (*e.g.,* [17]) include rules that limit role activation, for example limiting the number of principals who can activate a given role at any time. Complex systems for the management and delegation of roles are termed *trust management systems* [22].

Fisler, *et al.* [41] present *Margrave*, a tool for verifying properties of a role based access control policy and for performing change impact analysis on it. Policies for Margrave are encoded in XACML [70] as binary decision trees with optional constraints. They do not present a method for policy translation. Presumably the translation is done by hand. They present algorithms for combining two trees with similar constraints and for comparing differences between two similar decision trees. Their work is similar to EPAL in that they consider properties of policies that yield access control decisions, providing permit or deny decisions. Their method for discovering changes between policies in interesting in that their tree representation lets them verify all the differences between two policies, not just with respect to a particular property.

Our work differs from Fisler, *et al.* 's in terms of scope and approach. We are concerned with policies that include more than just permit and deny decisions and therefore their decision tree structure would not be appropriate. Their algorithm for comparing differences between policies is of interest, but only as an example to extracting differences between policies. The authors' consideration of complexity in choosing their decision tree structure is a good example of policy condensation, something we will need to consider as we develop our lower level representation of policy.

Originator Control

Originator Control (ORCON) policies [50] are a special kind of mandatory access control that give rights to the originators of objects even if those objects are no longer owned by them. For example, if Alice creates a file and Bob makes a copy of it, only Alice can grant new permissions on Bob's copy, even if Bob modifies it. The policy is mandatory since the system, not Bob, determines how the originator designation is assigned. The ORCON idiom is commonly used for policies where one principal holds or collects another principal's information. It becoming a common idiom for privacy policies as well, leading to techniques to maintain records of where data originates (*i.e.,* data provenance).

Usage Control and DRM

The spread of digital content and widespread file sharing brought the need for policies to control how files and digital objects are used, termed *Digital Rights Management* (DRM). Rich languages for access control systems have been designed recently to support DRM systems, including concepts of state, roles, and delegation. Languages for DRM policies include the Open Digital Rights Language (ODRL) [60] and eXtensible rights Markup Language (XrML) [61, 94], recently adopted as the Rights Expression Language of the Motion Picture Experts Group Multimedia Framework (MPEG-21) [23]. There has been work on the formal properties of both ODRL [53, 83] and XrML [55]. Both languages are designed to create digital contracts that are attached to digital media objects. Before a user can use (*i.e.,* play, print, view) the object, software must consult the contract and determine whether the proposed action should be allowed. Both languages make use of signed digital tokens from content owners as well as counters to regulate the rights that content users have.

Park and Sandhu's Usage Control (UCON) [76, 77] model is an extensive access control system. The system includes provisions for digital rights management through the use of updateable permissions and rights that can behave as counters. UCON allows a very rich combination of rights, conditions, and obligations to be included in a single access control system.

2.3 Privacy Policy Formalization

There is extensive work on fundamentals of privacy policies and privacy policy languages and a extensive discussion of the whole space is beyond the scope of this work. In this section we therefore survey related work and discuss in depth only work that is closely related to our own. We return to three of the policy languages below in Section 4.5 with a discussion of their suitability for modeling legal policies.

Let us first consider two languages similar in nature to our approach: P3P and EPAL.

P3P

The Platform for Privacy Preferences (P3P) [93, 36] is a World Wide Web Consortium (W3C) standard language for web information privacy policies. P3P is a browser-centric standard designed to put web site privacy policies in a machine readable format. P3P policies are XML documents that a web site can store in a well known location. A combination of descriptive tags and human readable text describe the promises that a web site owner makes about how data on the site is collected, used, and maintained.

An example P3P policy snippet is shown in Figure 2.1. For clarity we omit the required namespace declarations and policy URL declarations. Lines 1–8 describe the entity that is making the privacy promises. In this case, ABC Corporation is the owner of the web site hosting the policy, so it provides its address. Line 9 declares that data subjects will have access to all information about them. Lines 10–14 provide a method for users to challenge or dispute the way that information about them is handled. Line 11 describes that disputes will be resolved through the company's customer service policies. Line 12 is an English explanation of this method, meant to be displayed to the user. Line 13 explains that resolutions will be made by correcting the information about the user. Lines 15–23 are a policy "statement," a declaration about how particular types of data will be retained and used. Different data types may have their own, independent statements. Line 16 gives an English summary of the statement's thrust. Line 17 declares that data will be used for the purposes of administering the web site (`admin`), completing and supporting the purpose for which the data was provided (`current`), and research and development (`develop`). Line 18 declares that only ABC Corp or its will receive the data. Line 19 declares that data collected may be retained indefinitely. Lines 20–23 describe the data covered by the statement, in this case http connection and click stream information.

The authors of P3P see it as a machine version of a normal human-readable legal privacy policy. Before interacting with a web site, users can instruct their browsers download the site's policy, examine it, and present it to receive user approval to continue interaction with the site. P3P is a one way communication mechanism, so web users have an all or nothing choice. It is not clear, however, whether the promises in a P3P policy are as legally binding as standard corporate privacy policies. For this reason privacy activists initially critiqued

```
<ENTITY><DATA-GROUP>
<DATA ref="\#business.contact-info.postal.organization">ABC Corp</DATA>
<DATA ref="\#business.contact-info.postal.street">123 Avenue A</DATA>
<DATA ref="\#business.contact-info.postal.city">Rome</DATA>
<DATA ref="\#business.contact-info.postal.stateprov">ME</DATA>
<DATA ref="\#business.contact-info.postal.postalcode">04957</DATA>
<DATA ref="\#business.contact-info.postal.country">USA</DATA>
</DATA-GROUP></ENTITY>
<ACCESS><all/></ACCESS>
<DISPUTES-GROUP>
<DISPUTES resolution-type="service" service="example.com/pol.htm">
<LONG-DESCRIPTION>Ask our staff for help</LONG-DESCRIPTION>
<REMEDIES><correct/></REMEDIES>
</DISPUTES></DISPUTES-GROUP>
<STATEMENT>
<CONSEQUENCE>Our Web server collects access logs</CONSEQUENCE>
<PURPOSE><admin/><current/><develop/></PURPOSE>
<RECIPIENT><ours/></RECIPIENT>
<RETENTION><indefinitely/></RETENTION>
<DATA-GROUP>
<DATA ref="\#dynamic.clickstream"/>
<DATA ref="\#dynamic.http"/>
</DATA-GROUP></STATEMENT>
```

Figure 2.1: Sample P3P policy web tracking

the language for allowing policies that are perhaps misleading since they are unenforceable in court [30, 33, 90].

Despite the declarative nature of P3P, policies written in it can be very complex, opaque, and difficult to understand. The difficulty is compounded by the possibility for web sites to have different P3P policies for different pages and the ability for policies to specify different rules for each type of data collected. To help get a handle on this complexity, a rule set language called "A P3P Preference Exchange Language" (APPEL) [92] was devised to enable users to specify their preferences so they can be matched against policies received used by web sites. In theory a user could specify what policies are acceptable using APPEL and then let the APPEL engine examine each site's policy before visiting. However, its use of rule sets with precedence by order and default policy rulings makes composing a correct policy in APPEL subtle and writing nontrivial policies using it is somewhat difficult [59]. Some alternative languages have been proposed, such as XPref [3], which is based on the XML query language XPath [91].

In terms of formalization of P3P, Yu, *et al.* present formal semantics for P3P [98]. Hayati and Abadi use certain P3P duration flags to create an information flow enforcing tool [57].

EPAL

The Enterprise Privacy Authorization Language [8, 10] is an XML based language to describe and enforce internal enterprise privacy rules. The language uses a client-server architecture where each action on private information must first be submitted as a formal request to a "privacy server" which processes the request and returns an answer. The request includes the following information:

1. Name or role of the requestor
2. Proposed action
3. Declared purpose of the proposed action
4. The class of data is being acted upon
5. An optional *container* which is a mapping of field names to values

The privacy server has a policy in the form of a set of rules. Rules include the following fields:

1. A ruling (deny or allow)
2. Restrictions on the values of the requestor's name, proposed action, purpose, and data class
3. Optional conditions which check boolean conditions based on the values in the container (*e.g.,* Bob may do action A on data D if field $D.name$ is blank)
4. Optional obligations to impose

The privacy server uses a matching algorithm to compare the rules to the requests. The EPAL authors propose two different matching algorithm:

- In their formal description of EPAL (there called E-P3P) [9], the authors propose a full search of the policy rule set to find all rules that match a given request. There is no ordering of rules in the policy. The rulings of the rules are then combined in some manner (either most restrictive or most lenient). The combined ruling is returned along with the union of all the obligations from the rules.

- In their proposed policy architecture and implementation [10, 8], the authors propose that the policy document impose ordering on the rules. The privacy server iterates down the list of rules and returns the ruling of the first one that matches.

Obligations define (normally) external requirements for the user to fulfill as a condition of doing the requested action (*e.g.,* data may be stored, but must be deleted after six months). The policy language is not concerned with the meaning of or enforcement of obligations. They are to be enforced by the requestor in accordance with the ruling given by the privacy server.

One interesting property of the EPAL framework is that a rule may impose obligations regardless of whether it responds with an approval or denial. This makes it easy for a rule to impose obligations on requests that fail. For example, a policy rule may require that failed access attempts be logged. It also introduces some complexity in automatically exploring the permissiveness of a policy since failure to gain permission and failure to make a request may have different outcomes.

In contrast to P3P which addresses web privacy statements and so has a fixed set of terms carefully defined to cover common web site usage, EPAL, is a generic architecture that requires policy writers to define a *vocabulary* of terms for the policy. Terms in the vocabulary can either be user categories, data categories, purposes, actions, data containers, or obligations. The first three types are hierarchical. A snippet of an example vocabulary is shown in the following example.

Example 2.3.1 EPAL Vocabulary

We present a snippet of an EPAL vocabulary for a policy about location information below. We omit details such as namespace declarations and some header clauses.

```
<user-category id="AnyUser">
   <short-description>Root user category</short-description>
</user-category>
<user-category id="Subscriber" parent="AnyUser">
   <short-description>Subscriber</short-description>
</user-category>
<data-category id="AnyCategory">
   <short-description>Root data category</short-description>
</data-category>
<data-category id="Location">
   <short-description>Location information</short-description>
</data-category>
<purpose id="AnyPurpose">
   <short-description>Root purpose category</short-description>
</purpose>
<purpose id="Advertising" parent="AnyPurpose">
   <short-description>Advertising</short-description>
</purpose>
<action id="PublishSubscribe">
   <short-description>Publish Subscribe event</short-description>
</action>
<container id="Permissions">
   <short-description>Container for information about the principal
   involved in the publishing event.</short-description>
     <attribute id="Owner" maxOccurs="1" minOccurs="1"
     simpleType="http://www.w3.org/2001/XMLSchema\#string">
     <short-description>Name of the principal that owns this permission
     </short-description>
```

```
29          </attribute>
30     </container>
31     <obligation id="ReduceAccuracy">
32        <short-description>Data must be reduced in accuracy.
33        </short-description>
34     </obligation>
```

Lines 1–3 defines a base user category called AnyUser. It functions as the root of the user hierarchy. Lines 4–6 defines a user category called Subscriber and makes its parent AnyUser. Lines 7–9 defines a base data category called AnyCategory. Lines 10–12 defines a location information data category and makes its parent AnyCategory. Lines 13–15 defines a base purpose called AnyPurpose. Lines 16–18 defines an advertising purpose and makes its parent AnyPurpose. Lines 19–21 defines an publish/subscribe action. Lines 22–30 define a permissions data container. Lines 22–24 defines its name and gives a short description of it. Lines 25–29 defines an attribute of the container called Owner. Owner must occur exactly once (since maxOccurs and minOccurs are both 1) and is a string type. Line 30 closes the container. Lines 31–34 defines an obligation to reduce the accuracy of data.

We have included only one base type for each of users, data categories, and purposes, but we could have created an arbitrary number of them if desired. We also have given the container only one attribute though we could have included an arbitrary number of them as well. □

Since EPAL is designed to be executable, they may include details of enterprise employee hierarchies that data subjects do not need to know. Instead, they would be better served with a conventional privacy policy that summarizes everything that is allowed by the executable policy. As Hayati and Abadi [57] note, the policy provided to the data subject is an upper bound on the actions that may be performed under the executable policy. In the specific case of web server privacy policies then, an enterprise may publish a policy in P3P for visitors and implement a compliant policy in EPAL. To that end the EPAL authors [86] suggest a method of translating from an EPAL policy to a P3P policy.

Their method is only semi-automated, however, since the EPAL policy rules must first be tagged by hand with P3P `purpose` tags. Since an EPAL policy contains more information than a P3P policy, there is no automated way to convert from P3P to EPAL.

In our study of legislative policies, our language has a similarity to the way an executable EPAL policy interacts with a descriptive P3P policy. We also begin with a descriptive policy document and produce an executable version of it. There, however, the similarity ends. We work on a direct translation from the descriptive policy to an executable one. Also, our policy language is more suitable for legislative policies as we shall discuss next.

Certain properties of EPAL's rules, conditions, system model, and vocabularies, however, made it difficult for us to use. Rules do not have an explicit representation of access to system and object state, so rules that inspect the state of objects and rights or modify state (*e.g.,* data anonymization, disclosure of minimum necessary information, etc.) must rely on complex conditions and obligations. They also do not include a parameter for "recipient," so rules that depend on the information recipient can not be written easily. They also can not query or invoke other rules, so they can not collaborate or query each other for rulings or obligations. Conditions can not access rule invocation parameters such as purpose, actor, and data. They are confined to flags and parameters included in a "container." There also is no provision for writing to or inspecting a system log.

There has been extensive work on the formal semantics of EPAL [9] (there called E-P3P) and translation to other languages [64, 11]. There has also been work on EPAL on policy composition [12, 14] and comparison and merging [13]. Since EPAL policies depend on the construction of customized vocabularies, it is only meaningful to compare and merge policies that have compatible vocabularies. As shown in the case studies in Chapter 7, the vocabulary overlap issue is important to consider when comparing policies. Two policies with non-overlapping roles, rights, and purposes are difficult to compare.

Other Related Work

XACML [70] is a privacy and access control authorization language that includes much of the functionality of our system. It does not, however, allow for conditions or obligations that depend on past actions or offer constructions for "permitted elsewhere" rules. The policy relations we consider in Section 5.5 may prove useful for analysis of XACML policies since they consider many of the same requirements such as roles, object types, and initial state. Jajodia *et al.* present an analysis of the provisional authorizations [63] which underly the XML access control language XACL [54] which is similar in nature to XACML.

Other recent work on formalizations of privacy policies include Bertino, *et al.* 's formal model for policy features and comparison [20], Fischer-Hübner and Ott's formalization of the Generalized Framework for Access Control [40], and Wijesekera and Jajodia's propositional policy model and algebra [97, 96].

There has been extensive work on the formalization of obligations in contexts including from businesses processes [2] and contracts [1], access control policies [58, 78, 21], and enterprise privacy policies [28, 27]. Obligations in our model are handled rudimentarily with flags and checks by policy rules. A full treatment of the classification, tracking, enforcement, and management of the different types of obligations that arise in privacy laws is beyond the scope of this work.

There has been work on the development of privacy policy languages for location based services. Some of those languages have taken considerations for policy merging and composition. The Geopriv working group (`ietf.org/html.charters/geopriv-charter.html`) from the Internet Engineering Task Force (`ietf.org`) has developed drafts of policy documents to describe rights granted to location based services providers. Their language proposal includes query based access control policies, but do not allow negative rules due to complexity restrictions. By removing negation, they also allow policies to be stored partially in different locations and combined later to form one large policy. Snekkenes [88] presents a lattice theoretic model of location privacy policies, parameterized by data accuracy allowance. Policies can be compared by where on the lattice they stand for each of several data fields (*e.g.,* identity, time, geographic location). Policy merging in his model would involve the consideration of each data field and adjusting the policy lattice value according to the input policies.

Barth, *et al.* [15] present a theoretical framework for privacy policies based on message passing. Their framework allows policies to predicate rulings on the total history of messages sent and to impose future obligations. They outline some results on the complexity of resolving the satisfiability of future obligations. Importantly, their policy evaluation framework handles negation properly, imposing the strictest of all possible rulings based on queries. They present some policy ruling examples from legal privacy policies, including HIPAA, GLB, and the Children's Online Privacy Protection Act (COPPA) [34]. Notably, policies in their framework do not include notions of the purpose of an action and their conditions to examination of message logs.

REALM [47] is a formulation for the expression of regulations as logical models. The focus of the work is on business processes and so focusses on the timing of obligations and actions, but it may be possible to extend their work to privacy requirements as well.

2.4 Conclusion

In conclusion, we have presented background work on the landscape of access control and privacy legislation. The two fields have long and separate histories, but share considerable concepts and goals. In this work we show how we can use access control techniques and formal model checking procedures to better understand what different legal policies. Our hope is to make the task of understanding complex legislation easier and more tractable using automated tools and analysis.

Chapter 3

Motivation for Formal Privacy

In this chapter we develop the conceptual framework of Formal Privacy to motivate our use of formal methods in the study of privacy preservation and policies. Broadly, we consider the benefits we are seeking to attain with formal privacy models and the advantages they have over other formulations. With the benefits in mind, we develop a set of desired properties for formal privacy models to allow evaluation of their various strengths and shortcomings. We use the desired properties developed in this chapter to give us direction in the development of our methodology in the chapters that follow.

The rest of this chapter is organized as follows. Section 3.1 motivates the use of formal privacy in the study of privacy preserving systems and privacy policies and the benefits we may desire from them. Section 3.2 enumerates the properties we desire for Formal Privacy models in light of the desired benefits. Section 3.3 concludes and leads into Chapter 4 which presents our methodology.

3.1 Formal Privacy

Information privacy policies are generally divided between two enforcement levels. At the human level, natural language policies describe procedures, actions, and protections that an entity will follow. At the data level, computer and network level policies enable, enforce, or restrict particular actions. When a human level policy governs what a data level policy must enforce, we face the challenge of mapping between them. Of course, the gap between the human and data policies serves as a layer of abstraction between them, letting one human level policy govern many potential data level policy implementations, so there is good reason to maintain the division.

With respect to data level policies, there has been considerable work in the access control and policy communities with respect to the construction, management, composition, comparison, and combination of policies. There has been some emerging work on the examination and analysis of human level privacy policies outside of the natural language processing community. The policy and formal methods communities have been exploring ways in which their tools can be adapted to the format, style, and structure of human level policies. Interest has come from a few directions, primarily from the significant amount of complex privacy legislation issued by governments globally. Even as new laws requiring enterprises to comply with standards for data privacy and protection are coming into force, a steady stream of news alerts concerning data theft and misdirection from companies has emerged as well.

The application of formal methods to access control policies is not new, as we have explored in the background sections of Chapter 2. The strength and novelty of our work stems from our approach to create a formal language that is a step towards bridging the gap between the natural language idiom and data driven analysis. Our goal is the creation of a formal language that is intuitive to read and is visually and semantically similar to the contents of the legal policy that it is supposed to represent. By letting the legal structure dictate the format of our language and evaluation tasks, we impose as little artificial structure on the law requirements as possible and thereby give us stronger confidence in the accuracy of our models.

Our focus on maintaining the structure of the legal text is one important feature of our formal privacy language and techniques for analysis, a feature often not present in purely logic-based or implementation-based solutions. Laws are written in a highly structured dialect of natural language, including repeating phrases, references, and deference to other parts of the document[19]. The structure helps in the management of the various laws by prescribing and proscribing actions based on multiple sections of text. Our formal language preserves the structure of the law, thereby reducing the size of the resulting policy and by enabling various types of common reference schemes.

One potential pitfall to maintaining the document structure, however, is in designing a language that is too close to the text to be implementable. That is, the derivation of a formal model which is too complex or abstract to be analyzed is not as useful as one that is. Furthermore, a formal language which is analyzable using existing and well understood formal methods tools has an even greater advantage since it permits quicker and more efficient analysis of policies.

With the above observations in mind, we advocate models in the rubric that we call Formal Privacy:

Definition: *Formal Privacy* is a framework for models which provide structured representations for privacy policies that enable the use of formal methods tools in the analysis and exploration of policy properties including aspects such as data use and the purposes of actions. □

When derived from a natural language text, Formal Privacy models should maintain the structure of the underlying document, acting as a bridge between natural language privacy policies and code level policy implementations. Aiming the formal representation towards formal methods analysis tools gives greater ease of use and enables quick, push-button exploration of policy properties.

Given the varying scope, depth, and structure of legal language and policy, we limit the focus of this work to common aspects of several specific, well known privacy laws. We develop the Formal Privacy framework to address the common aspects of the laws chosen and develops a formalism which may be readily adapted to similar aspects of other legal privacy policies as well. In narrowing our focus, we explicitly do not model aspects of many privacy laws which are beyond the scope of our expertise and ability to model. We note such limitations in the upcoming chapters and return to them briefly in the conclusion of this work.

3.2 Desired Properties of Formal Privacy

Our goal in using Formal Privacy as a model is to improve the current practices for achieving and evaluating compliance as well as providing a tool to let legal experts provide best practices guidance to non-experts regarding legal requirements. We foresee two prototypical usage scenarios for Formal Privacy which we explain next. The usage scenarios guide us by providing a source for properties that a Formal Privacy solution should provide.

Compliance Reviews Compliance reviews involve current practices evaluations and legal requirements reviews in enterprises. Current practices evaluations are performed by enterprises to gather their current business and technical practices in order to understand the way things are currently done and compare them against some metrics. Such metrics are derived by performing legal requirements reviews in which enterprise staff and legal experts together derive standards of what is required by the various laws impacting the enterprise. Given the current practices and the legal requirements metrics, enterprises chart out guidelines for the creation of documents, the modification of practices that must be changed, and the equipment or software required to support the requirements [26, 69].

Achieving compliance is a long and complex task for large enterprises [26] which is aided by many consulting companies specializing in particular classes of enterprises. As described in published compliance case studies [26, 69], compliance efforts normally begin with a gathering phase where enterprise staff members collect information about the current practices of the enterprise. This involves interviewing technical staff, managers, and business staff to establish precisely how the covered operations currently handled. For instance, for a requirement to enforce two-factor authentication on all network accesses, the technical staff would be interviewed about the current authentication mechanisms and the enterprise users would be queried about their typical uses of the system and how they would be affected by a change to two-factor authentication. Once the staff members have completed the gathering stage, they consult with legal experts to determine what of their current practices must be modified to fit the legal requirements. With those results, the staff produce recommendations for the parts of the enterprise that must modify their behaviors and produce whatever documentation is required to prove their compliance to the regulating body. After completing the review, enterprises must continue to evaluate the creation and deployment of new processes and services in the enterprise. Enterprise parlance is that "Compliance is a process"; a process which requires current practices reviews regularly and regular checkups with respect to the applicable laws and regulations. As new applications, systems, and requirements are adopted, enterprises must carefully evaluate how they fit with the relevant legislation and best practices.

Certainly the process of compliance reviews is and will remain a work intensive process at the human level. Automation will not relieve the burden of technical assessments, project management, and review by legal experts. Our language can aid the different steps in the process however, by creating a common language for the description of certain aspects of enterprise processes and legal requirements. For instance, in the process of performing a compliance review with respect to health care privacy legislation, a hospital needs to review its current practices and systems to identify areas which require review and potential by legal experts in the hospital's compliance office. To do that, the hospital information technology (IT) staff must gather and create a presentation of the ways its systems work for their meeting with the compliance office staff. Since the compliance office staff are experts in the law and not necessarily the workings of IT resources, the presentation needs to be placed at the correct level of detail to highlight the behaviors that are legally relevant. Unlike models standard software modeling languages such as the Unified Modeling Language (UML) or Business Process Execution Language (BPEL) which offer encodings for program structure, the IT staff are interested in system behavior as an executable property of their systems, asking "How does the system behave under the various circumstances in which the hospital uses the IT system?" A language that enables them to describe the behavior of their systems under various situations and circumstances enables the IT staff to create a clear description of IT system behavior. Additionally since it is executable, the both the IT and compliance staff members can experiment with the model to discover behaviors which warrant examination. Additionally, for the next compliance review cycle, even if the IT and compliance offices have seen significant changes, the model can be extended and adapted for reuse.

On the other side of the compliance evaluation, the legal staff must have a clear understanding of what the behaviors permitted by the legal texts are. While legal training is required for the correct interpretation of laws and statues, the job of an enterprise's compliance office is to evaluate business practices and potential liabilities, create enterprise policies, and act as a resource for other parts of the enterprise. The policies created by the compliance office staff are used by the enterprise to guide its behavior, so producing an executable version of an enterprise policy gives an additional dimension to the policy. For the benefit of the compliance staff which must validate that the executable model fits the policy, the executable version should closely mirror the structure of the source policy for easy comparison. By providing an executable model for the policy, the compliance office makes it easier for other enterprise departments to implement the policy guidelines since they can see precisely what behaviors are affected by the policy. As a caveat, however, since the executable model is written at the level of the policy language it must be carefully designed to capture the interpretation of the policy as relevant to the enterprise's IT staff. Greater precision in delineating purposes, actions, and data classifications make the executable policy more descriptive, complete, and useful for implementation comparison.

As an example, let us consider a case where a company which previously was not covered under any privacy legislation suddenly becomes subject to legislation due to a shift in business practices.

Example 3.2.1 (TiVo and Cable TV Privacy Act)

The Cable TV Privacy Act of 1984 [48] (CTPA) restricts the uses and disclosures that cable operators can perform on certain categories of subscriber information. TiVo Corporation (`tivo.com`) produces digital video recorders (DVRs) and service contracts that allow subscribers to download television show times, record shows, rewind shows during viewing, and other convenient features. When a service contract is in place, TiVo DVRs collect information about the viewing habits of their users—which shows they watch, when they change channels, *etc.*—and a portion of this information is sent back to TiVo each time the DVR "dials home" to receive new listing information [68, 89]. TiVo initially did not operate over cable lines and therefore was not subject to the Cable TV Privacy Act. However, they recently began offering their service for use over Comcast Cable's (`comcast.`

`com`) network in certain areas [87]. We are therefore interested in discovering whether policies and business practices of TiVo are sanctioned by the Cable TV Privacy Act. Our comparison should be at the policy level since we do not have access to the internal business practices of TiVo. We must therefore presume that TiVo behaves precisely according to its policy, doing no less and no more than what is written in it. Conversely, since our analysis remains at the textual level, results from the comparison must be validated by competent legal professionals. For instance, in the case we examine in Section 7.3, it is essential to determine what qualifies as a "disclosure" in the Cable TV Privacy Act and which companies fall under the "Corporate Family" of TiVo Corporation. □

Aside from the process of initial compliance review, as mentioned above, the creation of an executable version of enterprise behavior and legal requirements makes policy and model reuse simpler, especially in scenarios where current practices must be compared against new legal requirements. For instance, if a hospital management company has offices only in the United States and wishes to expand to new European markets, the compliance office must evaluate what changes must be made in order to comply with the requirements in the new area of jurisdiction. Such evaluations differ from basic compliance evaluations in that the business practices for the enterprise have already been evaluated for compliance with the existing legal regime and therefore the compliance staff must be careful to evaluate where the current policy subsumes the new policy to avoid wasting resources on them. The reevaluation process certainly requires legal expertise which no automated system can replace, however, an executable model can aid the compliance office staff in identifying differences between the new policy and the existing policy and practices. Importantly, the model can aid both staffs in finding instances in which no change is necessary.

As an example, let us consider a case where a company needs to explore whether its current practices under one privacy regulation must be changed to comply with a different one.

Example 3.2.2 (HIPAA and Australian Insurance)

The Privacy Rule of the US Health Insurance Portability and Accountability Act

(HIPAA) [84] governs the management, storage, and distribution of patient health information. Since hospital electronic health care records systems are normally complex and often nonuniform, assuring compliance with the rule required a significant effort by the entities affected by it. The primary changes brought about by the Privacy Rules are in the area of information security and protection. The rule requires health care providers, health insurance companies, and health care clearing houses (*i.e.,* the *covered entities*) to secure the electronic health information that they hold through physical, electronic, and social mechanisms. For instance, hospitals are required to protect their servers with locks (physical mechanisms), enforce access with passwords (electronic mechanisms), and ensure doctors do not haphazardly discuss patients in public places (social mechanisms).

Compliance with the Privacy Rule requires covered entities to appoint a Privacy Officer who oversees the design and implementation of policies to enforce the information security rules. When the Rule was first published in 2001, covered entities rushed to review their current practices, compile detailed lists of their information technology (IT) assets, and perform a gaps analysis to determine what practices, policies, and IT systems needed to be updated. Larger covered entities employed HIPAA consultants or companies which specialize in compliance to aid them. Case studies from covered entities emphasize the review process and gap analysis performed during the compliance evaluation as a major burden in the process [26, 69].

The Australian Privacy Act [71] grants permission for industries to develop self regulatory consortiums that enforce tailored privacy policies. The Insurance Council of Australia (ICA) (`ica.com.au`) issued a General Insurance Information Privacy Code (GIIPC) as per that permission that is applicable to all insurance providers who choose to belong to the ICA. For a hypothetical Australian health insurer interested in expanding to the US market, the insurer would need to evaluate whether its policies need to be changed to conform to the US' HIPAA Privacy Rule [46] which applies to health insurers. Let us assume that the insurance company has already established business practices compliant with the ICA's policy. The insurance company would then need to investigate any potential new requirements of HIPAA and modify its business practices to comply with them. It must be careful in doing so ensure that in changing its policies it does not violate the ICA's policy.

$\square$

Legal Guidance Providing legal guidance for enterprises and industries requires evaluation of the legal requirements in light of the business practices. Legal scholarship is required to give such guidance since the rules regarding interpretation, precedence, and application are complex. The recipients of guidance include legislative staffers, enterprises, and stakeholders for a particular area of law. We can aid legal experts in their efforts by letting them more quickly visualize and explore what new or existing laws require. Since the executable models are serving as summaries of a law or policy it is important that they be either created by or thoroughly checked by legal experts to ensure their accuracy. By creating and using Formal Privacy models, our goal is to allow providers and recipients of guidance to "query" a legal text as if it were a database of permitted and forbidden behaviors based on some set of circumstances. We are interested in queries that legislators, legal experts, and interested parties would normally perform by hand. We want to ask queries such as "Does this loophole exist in the law?" and "Can an enterprise perform action A before it does B?" The results we can receive from our formal analysis give useful indications about how such questions can be answered.

When performing such queries, we wish to permit the establishment and evaluation of invariants. Invariants represent assumptions by a stakeholder about what behaviors should be allowed or forbidden. When the model revels a violation of a stakeholder invariant, we call it a relative stakeholder problem since other stakeholders may have conflicting views of what the invariants should be. Queries may not need a large number of steps to be answered, but they are interesting because the stakeholder asking may not be able to look at or properly interpret the text directly due to its length or complexity. In this work we consider relative stakeholder invariants which operate at the level of the source text. Invariants which are subject to how particular words, phrases, or exceptions are interpreted are beyond the scope of this work.

As an example of a stakeholder invariant that we can examine, let us consider one of the legal policies discussed above in Section 2.1.3 and queries that stakeholders would be interested in.

Example 3.2.3 (HIPAA Consent)

As noted above in Section 2.1.3, the HIPAA Privacy Rules changed between its initial release in 2000 and its final release in 2003. Changes were made after a comment period in which stakeholders were given the opportunity to submit comments to HHS regarding the formulation of the rules. The comments summary as released by HHS reveals a list of the various, often conflicting, interests of the different stakeholders. Some items on the list are relative stakeholder problems, so other stakeholders might not consider them to be concerns. Others are concerns in that they forbid what is common industry practice.

The rules about when covered entities must get patient consent before performing using or disclosing health information were the subject of debate during the development of the Privacy Rule. The version of the rules from 2000 [§164.506, v.2000] requires that covered entities receive consent from individuals for the use and disclosure of their information for treatment, payment, and health care operations. There are many provisions and exceptions to the general rule, but for the majority of cases, it required covered entities to receive consent. Consent for treatment, payment, and health care operations was to be signed and revokable by the patient at any time. The general rules from [§164.506, v.2000] are as follows:

> §164.506 Consent for uses or disclosures to carry out treatment, payment, or health care operations.
>
> *(a) Standard: Consent requirement.*
>
> (1) Except as provided in paragraph (a)(2) or (a)(3) of this section, a covered health care provider must obtain the individual's consent, in accordance with this section, prior to using or disclosing protected health information to carry out treatment, payment, or health care operations.. . .
>
> (3)(i) A covered health care provider may, without prior consent, use or disclose protected health information created or received under paragraph (a)(3)(i)(A)–(C) of this section to carry out treatment, payment, or health care operations:
>
> (A) In emergency treatment situations, if the covered health care provider attempts to obtain such consent as soon as reasonably practicable after the delivery of such treatment;
>
> (B) If the covered health care provider is required by law to treat the individual, and the covered health care provider attempts to obtain such consent but is unable to obtain such consent; or . . .
>
> *(b) Implementation specifications: General requirements.*

> (1) A covered health care provider may condition treatment on the provision by the individual of a consent under this section.. . .
>
> (5) An individual may revoke a consent under this section at any time, except to the extent that the covered entity has taken action in reliance thereon. Such revocation must be in writing.
>
> (6) A covered entity must document and retain any signed consent under this section as required by §164.530(j).

The shift to requiring explicit consent from patients was a major shift in approach from previous modes of medical and hospital operations and required that doctors offices, clinics, and hospitals create new methods for documenting the acquisition of and management of patient consent.

The response from stakeholders during the public comment period brought out many of the concerns that covered entities and health care organizations had over the new rules. Many pointed to the changes in hospital operating procedure as well as the burden of managing the new consent documents. Some comments pointed to particular aspects of the new consent requirements that interfered with the day to day operation of health care providers. Those comments pointed to specific concerns with respect to the content and requirements of the section. Three specific comments relating to the consent rules which we return to later in this section are as follows (paragraph numbers added for reference):

> [(1)] Emergency medical providers were also concerned that the requirement that they attempt to obtain consent as soon as reasonably practicable after an emergency would have required significant efforts and administrative burden which might have been viewed as harassing by individuals, because these providers typically do not have ongoing relationships with individuals.
>
> [(2)] The transition provisions would have resulted in significant operational problems, and the inability to access health records would have had an adverse effect on quality activities, because many providers currently are not required to obtain consent for treatment, payment, or health care operations.
>
> [(3)] Providers that are required by law to treat were concerned about the mixed messages to patients and interference with the physician-patient relationship that would have resulted because they would have had to ask for consent to use or disclose protected health information for treatment, payment, or health care operations, but could have used or disclosed the information for such purposes even if the patient said "no."

The HHS response to the comments was to admit that the concerns raised were troubling enough to warrant removal of mandatory consent for treatment, payment, and health care operations. The later version of the rules from 2003 was simplified, making consent for treatment, payment, and health care operations optional for most situations, but leaving intact restrictions for things such as marketing and the release of psychotherapy notes. The general rules for [§164.506, v.2003] are as follows:

> §164.506 Uses and disclosures to carry out treatment, payment, or health care operations.
>
> *(a) Standard: Permitted uses and disclosures.* Except with respect to uses or disclosures that require an authorization under §164.508(a)(2) and (3), a covered entity may use or disclose protected health information for treatment, payment, or health care operations as set forth in paragraph (c) of this section, provided that such use or disclosure is consistent with other applicable requirements of this subpart.
>
> *(b) Standard: Consent for uses and disclosures permitted.*
>
> (1) A covered entity may obtain consent of the individual to use or disclose protected health information to carry out treatment, payment, or health care operations. . . .
>
> *(c) Implementation specifications: Treatment, payment, or health care operations.*
>
> (1) A covered entity may use or disclose protected health information for its own treatment, payment, or health care operations

While the 2003 policy was written with the comments in mind, it is not clear if the new policy solved all of the concerns. To answer that question, we would like to take the expert discovered concerns, find them in the 2000 version, and query as to whether they are still present in the 2003 version. From the above three concerns we would therefore like to perform the following three queries:

(1) In general, are emergency workers who do not have ongoing treatment relationships required to acquire consent from individuals after the provision of services?

(2) In general, are health care providers required to acquire consent for access to records for treatment, payment, and health care operations after the transition period mentioned in the document?

(3) Are there situations where providers that are required by law to treat individuals are permitted to use or disclose health information for treatment, payment, or health care operations even after the individual has refused consent?

Given the large number of exceptions and circumstances discussed in the Privacy Rule, we must carefully choose the circumstances under which the queries are performed. That is, for the first two queries we are interested in examining the general circumstances for the emergency workers and health care providers, ignoring exceptional and extenuating circumstances. For the third query we are interested in the exploration of circumstances in which providers may ignore the refusal of consent by an individual so we are interested in any exceptional circumstances in which that is the case. □

Based on the above scenarios and examples we have the following requirements for what Formal Privacy models must provide.

1. **Confidence and Traceability.** Since the formal models will be used as representations of the regulation it is essential that it be straightforward to establish that models created are accurate to the content of the regulation. One way to aid in establishing confidence is to have the models be visibly and directly close to the text. With such models, we can ensure that the model correctly correlates to the text as required. Additionally, since the ones using the models may not be legal experts, it is essential that the properties discovered from the formal models be traceable to the part(s) of the original policy that are their source. This includes the requirement that the models be specific enough to show interesting properties of the regulation. Models that do not preserve the structure of the source text may also be used for such purposes, but the verification and traceability are likely to be significantly harder.

2. **Usability.** The formal models that we develop need to have a robust and flexible interaction mechanism. Legal texts are normally descriptive and the advantages in terms of usability and policy comparison that we seek come from creating representations with which experiments can be performed. Models with clear and intuitive means of for users to interact with policies aid non-experts in understanding complex policies since they allow for users to explore the properties of the model more

intuitively.

3. **Legal Properties.** Legal texts and privacy policies derived from them use standard privacy policy concepts such as actions, actors, objects, and purposes as well as specialized structures such as complex references, deference between rules, scopes, vagaries, and testimonials from actors. A formal model must include functionality to model the intent of the source text, whether or not it maintains the particular structure of the text. The design of an intuitive interaction paradigm for the model (see requirement 2) that produces the same response as the source text is required for accuracy.

4. **Extensibility.** Since regulations and policies change over time, Formal Privacy models must be easily extensible to new situations and environments. Formal representation which are as generic as possible will aid in this requirement since by making the formal model simple we ensure that new features can be added easily. Additionally, the formal language itself must be extensible to adapt to different legal requirements, regimens, and styles. Importantly, the language must be able to represent specifications from a variety of applications and jurisdictions.

In listing the requirements we emphasize the desired outcomes independent of the details of the particular solution. With them in mind we next develop the Formal Privacy model that is the result of this dissertation.

3.3 Conclusion

In this chapter we have motivated the problem space that we address in this work: the analysis and examination of legal privacy policies. Our goal is to develop a language that will aid us in comparing legal privacy policies and determining whether following one policy is sufficient to conform with another. We develop Formal Privacy as a framework for developing languages which offer the tools needed to meet our requirements. In the next chapter we describe the methodology that we use in this dissertation: the development of Auditable Privacy Systems and the Privacy Commands language.

Chapter 4

Methodology

The Formal Privacy language that we develop in this dissertation is Auditable Privacy Systems. Using the formal language we develop models that represent the requirements and properties of regulations and show how to use them to identify interesting properties. In chapter we develop the intuition behind the language, including some of its fundamental features and mechanisms. We more formally develop the language along with its syntax and semantics in Chapter 5.

The rest of this chapter is organized as follows. Section 4.1 presents Auditable Privacy Systems as a Formal Privacy framework and Privacy Commands as a formal language for the development of models of regulations, motivates its usefulness, and outlines the structures needed to use it. Section 4.2 outlines the translation steps we use to adapt legal privacy language to Auditable Privacy Systems. We discuss some of the challenges that we faced in doing so in Section 4.3. Section 4.4 describes the Privacy Commands language at a high level, using a running example to illustrate the features and constructs of the language. Section 4.6 concludes.

4.1 Auditable Privacy Systems

We base our development of Auditable Privacy Systems on the Privacy Systems work of Gunter, May, and Stubblebine [52]. Privacy Systems develop a generic set of policy and interaction atoms that can be used to capture the operational behavior permitted by privacy policies. Gunter, *et al.* show the formal properties of the Privacy Systems language and apply it to a scenario involving a privacy policy-controlled location based services framework. The salient features of the language are the inclusion of an object being about a particular principal (the *subject* of the object), a consideration of data crossing domains of control, and an explicit consideration of the contents and properties of objects. An model created in the Privacy Systems language describes the properties of a policy by dividing it into four mappings: a *transfer* function that prepares objects for transfer between agents, an *action* relation that permits actions on objects, a *data creation* relation that permits certain agents to create information about other agents, and *rights establishment* relation that permits certain agents to create or modify the rights that one agent holds over another.

The state of a Privacy System includes the sets of principals, objects, and the rights in the access control matrix that have been established. The policy of a particular Privacy System model is reflected in the rules that it enforces over the events. Rules in the system make decisions based on the state of the access control matrix and the properties of the system policy.

The intuition behind actors and subjects is that the actors are the parties to a transaction that concerns private information about the subject of the transaction. The actors initiate events through joint agreement subject to the privacy rules they have with respect to the subject of the event.

The Privacy Systems approach for the management of rights on private data gives a good model for the maintenance of data. It also lets policies explicitly examine and modify data objects, expanding the scope and specificity of policies. Because of these favorable properties we use Privacy Systems as a foundation for our formal language for regulatory policies.

Regulatory policies differ from many classical privacy and access control policies in their use of purposes, evidence, logging, and references. We consider therefore the following extended list as the primary events necessary for modeling regulations.

Transfer What is the right of an agent a to transfer an object o to another agent r where the object is about another agent s? In many policies this depends on the rights of both a and r relative to s, the features of o, and the purpose and circumstances of the transfer. After the transfer of an object is completed, the recipient may have different restrictions on its use than the sender.

Action What is the right of an agent a to carry out an action that affects the privacy of an agent s? This normally on the purpose and circumstances of the action as well as previous actions or interactions by a and s.

Data Creation Which agents are allowed to create an object o whose subject is s? The right to create objects about another agent is normally tied to the roles that the creator and subject are holding and the contents of the object being created.

Rights Establishment How are rights established for an agent s? Regulations often by default grant certain rights to agents in a particular role that may be modified by interactions with the subject or special circumstances.

Notification When an agent a carries out or attempts to carry out an action on or transfers an object o that is about a subject s, who must be informed of it? Commonly, notification is imposed as an enforcement mechanism for circumstances subject to the discretion of the actors, but it may be used in other circumstances as well.

Logging After the completion or attempted completion of an event initiated by agent s and potentially involving other objects or agents, what record must be kept of it? Like notification, logging is a requirement imposed by regulations often as an enforcement or auditing mechanism.

By including notification and logging we extend the Privacy Systems events with the ability to audit the events and actions that have been performed under the policy. We therefore term the framework Auditable Privacy Systems that we informally define as follows:

Definition: *Auditable Privacy Systems* is a formal model for privacy policies that is characterized by defining policies in terms of transfer, action, data creation, rights establishment, notification, and logging events. □

The auditing events differ from the other four in that their execution is not governed by the other aspects of the policy. For instance, if a a must inform s of something after an event affecting s has been performed by a, we do not consider the act of informing s to be a transfer event. Similarly, if a must add a note to the log of an action performed on o, it is not considered an action on o. This separation ensures that notification and logging may always be performed, easing their use as auditing tools.

4.1.1 Constraints

Policies restrict or permit the events described above based on a variety of circumstances and situations. Generically, we refer the circumstances, requirements, and situations that restrict the performance of events as constraints. Constraints may be simple with just one requirement or complex combinations of multiple requirements and circumstances. Since Auditable Privacy Systems are to be used for the modeling of regulations, we must consider the types of constraints they commonly employ.

Since regulatory privacy policies give rules for the interaction of users and information, they normally include constraints that can not be objectively verified. In some cases, the constraints require an expert to evaluate properly. A common example is a requirement that a particular action be reasonable or appropriate. For such constraints that are part of the requirements, but an automated policy can not properly evaluate, Auditable Privacy Systems rely upon assertions or environmental information to make decisions. By factoring out unenforceable constraints and placing trust in agents or the environment we achieve a model that is easier to design and verify. Since we are interested in creating models of regulations and policies, the unenforceable constraints are useful as an indication of the degree to which the regulation or policy trusts the agents it regulates.

It is helpful to delineate three types of constraints that regulations commonly include:

Type 1 are constraints that are immediately and objectively evident from the situation or information at hand.

Type 2 are constraints that are not objectively evident but may be resolved based on the input of some environmental or meta-information without reliance on the unverifiable judgment of a person.

Type 3 are constraints that can not be resolved without the subjective or unverifiable judgment of a (perhaps non-objective) person.

The makeup of the constraints imposed by a particular regulation varies widely based on the regulation's area of application and style. As an example of the three types of constraints, let us consider a sample quote.

Example 4.1.1 (Gramm-Leach-Bliley Constraints)

The following is a selection from the Financial Modernization Act of 1999 [35], commonly referred to as the Gramm-Leach-Bliley Act (GLB). It is codified in US Code Title 15, Subchapter I. GLB imposed regulations on the management of financial services companies and the information that they store. Included in GLB is a section on financial services privacy requirements in §6801–6803. Section 6802 "Obligations with respect to disclosures of personal information" regulates the disclosures that financial services companies may perform. The following is a quote from §6802(b)(1)(B) that relates to disclosures on personal information to third parties:

> A financial institution may not disclose nonpublic personal information to a nonaffiliated third party unless—
>
> (A) such financial institution clearly and conspicuously discloses to the consumer, in writing or in electronic form or other form permitted by the regulations prescribed under section 6804 of this title, that such information may be disclosed to such third party;
>
> (B) the consumer is given the opportunity, before the time that such information is initially disclosed, to direct that such information not be disclosed to such third party; and
>
> (C) the consumer is given an explanation of how the consumer can exercise that nondisclosure option.

Let us identify a few constraints in the above passage and classify them according to the types above.

The introductory sentence includes constraint that the rules apply to "nonpublic personal information." It is a type 1 constraint since the definition of nonpublic personal information is given in the GLB text (§6809(4)) and whether a particular piece of information falls into the categories defined there is objective and straightforward. Similarly, the introductory sentence constrains the applicability of the rules to agent acting in the roles of financial institutions and nonaffiliated third parties. They are also type 1 constraints since their precise meaning is defined in §6809(3) and §6809(5) respectively and so whether an agent occupies the given roles can be objectively determined.

Paragraph (A) constrains disclosures based on whether the financial institution has "clearly and conspicuously" disclosed to the consumer that the information may be disclosed to a third party. Whether the financial institution has sent any information to the consumer at all is a type 2 constraint since by checking the history of what the financial institution has sent in the past we may determine objectively whether the consumer was sent a notice. The constraint that the disclosure be clear and conspicuous is a type 3 constraint since the terms are not objectively defined. Instead the regulation relies on the financial institution to ensure its disclosures are clear and conspicuous.

Paragraph (B) constrains disclosures based on whether the financial institution has previously given the consumer the opportunity to prevent (opt-out of) the disclosure. The constraint of previously offering the consumer an opt-out opportunity is a type 2 constraint since it may be resolved objectively by checking the history what communications were sent to the consumer.

Paragraph (C) constrains disclosures based on whether the financial institution has given the consumer an explanation of how to opt-out of the disclosure. As in (B), the constraint is type 2 since we it may be objectively resolved by examining the communications sent to the consumer. □

The variety of constraints contained in the quote in Example 4.1.1 shows the diversity of constraints that regulations commonly impose. In order to properly model diverse regulations, an Auditable Privacy Systems language or model must include mechanisms to account for them. From a software engineering point of view, outside constraints are environment information that is invisible to the system specification [51]. The system specification needs assurance that the environmental variables are correct, however, so we must create a bridge between them. Tags as defined in Section 2.2 let agents make assertions about the environment that the automated system can check to impose outside constraints. The tags can be tied to particular principals in the system (*e.g.,* patient gives a consent form that is signed and dated) or be non-principal-specific conditions (*e.g.,* disclosure document reserves the right to change).

4.1.2 References

Regulations use references to allow paragraphs in one text to affect paragraphs in other places, most often as a constraint that is to be imposed by the referenced paragraph. The manner and diversity of references varies by regulation, so Auditable Privacy Systems models must offer mechanisms to support the various types. In order to introduce the reader to the different types of references that are found in regulations we introduce a series of quotes from the HIPAA Privacy Rule that employ a range of references. After the series of examples we summarize the types of references and how they can be modeled.

We begin with an example of a reference where a child paragraph clarifies the permissions of a parent paragraph.

Example 4.1.2 (2003 HIPAA Disclosure Rule)

The following example section is a quote from the disclosure rules of HIPAA [§164.506(c)(1), v.2003]. Here our goal is to delineate its references as constraints. We shall return to it in later chapters to describe how we capture its intent in our formal language.

> §164.506(c) Implementation specifications: Treatment, payment, or health care operations.
>
> (1) A covered entity may use or disclose protected health information for its own treatment, payment, or health care operations. [or]
>
> (2) A covered entity may disclose protected health information for treatment activities of a health care provider.

The parent paragraph (c) provides a heading for the implementation paragraphs that follow and so it implicitly refers to its child paragraphs (1) and (2) to delineate the what kinds of actions are to be permitted or forbidden. The child paragraphs are independent since they do not refer to each other.

Paragraph (1) permits use and disclosure of protected health information under three constraints: (1) that the action be conducted for the purpose of treatment, payment, or health care operations and (2) that the intended treatment, payment, or health care operations be for a covered entity and (3) its own. The result, therefore, is a constraint on usage and disclosure for treatment, payment, and health care operations to situations with the specified properties. Note that all three constraints are of type 1 since the definitions of what purposes constitute treatment, payment, and health care operations and what entities are covered entities are defined in the [§164.501, v.2003] and it is straightforward to determine whether the agent has the correct role, whether the intended action falls into the categories listed, and whether the action is intended for the use of the covered entity.

Paragraph (2) is similarly structured, but refers to just disclosure. It constrains disclosure to be only for treatment purposes and for the benefit of a health care provider.

In summary, the top level sentence (c) permits treatment, payment, and health care operations only as delineated in its child paragraphs. Paragraphs (1) and (2) do not refer to each other and so offer independent permissions. A reference to the permissions of §164.506(c) therefore permits any actions permitted by its child paragraphs. □

In the above example, the top level sentence references to the constraints since its body inherently depends on its children's constraints. We next consider direct references between paragraphs where one paragraph constrains an action based on another, non-child paragraph.

Example 4.1.3 (2000 HIPAA Consent Requirement)

Direct references to other paragraphs often appear as exceptions or constraints to a paragraph's actions. For example, the paragraph quoted here from HIPAA [§164.506(a), v.2000] concerns when consent is required for treatment, payment, or health care operations:

> (a) Standard: Consent requirement.
>
> (1) Except as provided in paragraph (a)(2) or (a)(3) of this section, a covered health care provider must obtain the individual's consent, in accordance with this section, prior to using or disclosing protected health information to carry out treatment, payment, or health care operations. (2) A covered health care

> provider may, without consent, use or disclose protected health information to carry out treatment, payment, or health care operations, if:
>
> (i) The covered health care provider has an indirect treatment relationship with the individual; or
>
> (ii) The covered health care provider created or received the protected health information in the course of providing health care to an individual who is an inmate.
>
> (3)(i) A covered health care provider may, without prior consent, use or disclose protected health information created or received under paragraph (a)(3)(i)(A)–(C) of this section to carry out treatment, payment, or health care operations:...

Here paragraph (1) permits usage and disclosure of protected health information by a covered entity provided that it has the individual's consent and that the situation is not subject to two other paragraphs: (2) and (3). This means that paragraph (1)'s constraints are not applicable if either (2) or (3)'s constraints are satisfied. Intuitively, (1) refers to (2)–(3) and so their applicability must be determined before (1) yields a decision. In this case, (2)–(3) give exceptions to the consent requirement, so they provide a more lenient ruling than (1) does. □

Constraint references may also be implicit, where constraints limit the actions of other paragraphs without the action paragraph acknowledging the limitation. Such constraints are invariants that must be maintained by any applicable action paragraph.

Example 4.1.4 (2000 Transferring Consent)

As an example of an invariant constraint, the following is a limitation from HIPAA on the transferability of consent from one covered entity to another [§164.506(a)(5), v.2000]:

> (5) Except as provided in paragraph (f)(1) of this section, a consent obtained by a covered entity under this section is not effective to permit another covered entity to use or disclose protected health information.

Ignoring for a moment the exception to (f)(1), the invariant in (5) limits how the permission from a consent is granted, but does not provide any specific action that it is limiting. As an invariant, any action that is predicated on a consent derived from the section must first determine if (5) is applicable and satisfied. □

The final type of constraint reference that we consider is a “permitted elsewhere” constraint:

Definition: A *permitted elsewhere* constraint is a condition on an action that depends upon a permission derived from another policy document or document fragment (*i.e.,* paragraph, sentence). □

In a permitted elsewhere constraint, a constraint is imposed if an action is permitted by some other paragraph(s). Paragraphs may permit an action if it is permitted elsewhere (in which case the permission is a duplicate) or require an action if it or another action is permitted elsewhere. Modeling such constraints are interesting in their own right, but also are of interest since they lead to an intuition of legal policy comparison. We develop this concept further in Section 5.5. As an example of a permitted elsewhere constraint, we quote the following example from HIPAA.

Example 4.1.5 (Authorization Requirement 2003)

As an example of a constraint of the form “required unless permitted elsewhere”, consider the beginning of the HIPAA rules requiring authorizations for usage and disclosures of protected health information [§164.508(a), v.2003]:

> Uses and disclosures for which an authorization is required.
>
> (a) Standard: authorizations for uses and disclosures.
>
> (1) Authorization required: general rule. Except as otherwise permitted or required by this subchapter, a covered entity may not use or disclose protected health information without an authorization that is valid under this section. When a covered entity obtains or receives a valid authorization for its use or disclosure of protected health information, such use or disclosure must be consistent with such authorization.

Here (1) requires that an authorization be acquired from individuals before protected health information can be used or disclosed unless the action is permitted by some other part of the subchapter. The constraint for the paragraph is a resolution of whether there exists some other paragraph in the subchapter that permits the intended action. Resolving the constraint requires a method for determining the applicability of a number of paragraphs. □

From the above examples we see four types of constraint references:

1. Implicit references between parent and child paragraphs.
2. Direct references between independent paragraphs.
3. Indirect references where one paragraph limits the actions of another without the limited paragraph mentioning the limitation.
4. Permitted elsewhere references where one paragraph constrains actions based on a permission granted by another paragraph(s).

Each of the four types of constraint references mentioned behave differently and therefore will need to be addressed separately. We must therefore include support for all four types in the Auditable Privacy Systems framework and the formal language that we devise in it.

4.2 Methodology of Translation

As discussed above, we are interested in translating legal privacy documents into commands and constraints. Doing so, we strive to stay as close to the structure of the text as possible. This is in order to allow quicker translation, easier verification of the relationship between the source text and the derived rules, and to preserve the particular structure of legal texts. To this end, we designed a methodology which made translation straightforward and observably close to the text. Our methodology also has the advantage of yielding a command set that is linear the number of paragraphs in the text.

Each paragraph in the text has one or more commands that execute it. Paragraphs are translated into multiple commands when they allow or deny multiple actions (*e.g.,* use and disclose). Paragraphs may also include constraints that limit the execution of commands. Commands and constraints yield judgments based on user input, including the purpose(s) for the proposed action and any relevant environmental information.

When paragraphs reference each other, whether directly or implicitly, we locate and translate the referenced clause and include an explicit reference to it.

Parent paragraphs that refer to their children include their children as references. Where appropriate, child paragraphs have their own commands. A child paragraph may reference its own specific constraints, its parent's constraints, or both.

Each condition or obligation in a paragraph is included in the paragraph's constraints. If the condition or obligation is unrelated to access control (*e.g.,* HIPAA's [§164.520] which has typographic rules for privacy practices disclosures documents) then it is elided unless relevant to actions. Relevant rules are then included through a combination of meta-information tag checks (see Section 5.1). We follow the language of the legal text, so unless two conditions or obligations are phrased very similarly or have obviously the same intent they have separate tags or rights associated with them. The number of tags, roles, and rights included in each policy model is therefore dependent on the writing style of the legal text.

4.3 Challenges in Translation

It is not surprising that legal privacy policies differ in style from standard computer systems access control policies, but two common features we came upon made the distinction very sharp.

The first distinguishing feature is the way that references are used. Commonly, a paragraph refers to the conditions of another paragraph independent of its body. For example in HIPAA [§164.506(a), v.2003] "Except with respect to uses or disclosures that require an authorization under §164.508(a)(2)" is a condition that points to the conditions of the referenced paragraph, but does not intend to activate the functionality of it. This is akin to a procedure creating a condition out of the precondition of another procedure without executing the referenced procedure. Because this is not a common programming language idiom, we implement it by dividing each paragraph into two parts: a constraint which contains the conditions and one or more regular commands that reference the associated constraint. This separation allows us to keep the reference structure of the law.

Legal references also vary in specificity. Most are unambiguous, but some are global pointers that refer to a large body of law. For example in HIPAA [§164.520(b)(3), v.2000]

"Except when required by law, a material change to any term..." is a deference to any other relevant legal requirement. We deal with this and other kinds of ambiguous references by using tags and rights to assert that the condition is satisfied. Non-monotonic default logic (for example, [7]) could perhaps be used here instead.

The second distinguishing feature is the use of testimonials in resolving conditions. Many environmental conditions are resolved by a testimonial from a principal in the system. For example, the following quote from HIPAA [§164.506(a)(3)(c), v.2000] grants a permission based on a health care provider's judgment:

> If ... the covered health care provider determines, in the exercise of professional judgment, that the individual's consent to receive treatment is clearly inferred from the circumstances."

Our model handles testimonials by placing them in tags to associate them with objects in the system knowledge, but a practical deployment of a policy system would need to track the testimonials that allowed a command to execute and log who asserted them.

We present several examples of privacy commands later in Section 5.2.6 after we develop the syntax and semantics for Privacy Commands and Privacy APIs.

4.4 Privacy Commands and APIs

The Auditable Privacy Systems framework delineates the events and atoms that are needed for the modeling of regulatory policies. In order to create concrete models for regulations, we devise a formal language for encoding the rules, conditions, and references that make up the body of the regulation. We call the language *Privacy Commands* due to the emphasis on commands as we shall explain below:

Definition: *Privacy Commands* is a formal language that is in the Formal Privacy framework and that satisfies the Auditable Privacy Systems requirements. It is characterized by defining policies in terms of "commands" and "constraints." □

Using the Privacy Commands language we devise models for regulations that implement the body of the regulation. We call such models Privacy Auditable Policy Interfaces or

Privacy APIs. We now discuss the fundamental properties of the formal language as an introduction to the formal presentation of the language in Chapter 5.

We devise a formal language that concretizes the theoretical constructs of Auditable Privacy Systems by creating a rule based language for the description of actions. The atoms of the language are *commands* that perform actions and *constraints* that limit the behavior of commands by imposing constraints.

Definition: A *command* is a small program that checks some conditions and then executes some instructions that may modify the knowledge state. □

Definition: A *constraint* is a collection of conditions that are evaluated to derive a judgment that indicates when and where some action should be permitted or forbidden. □

Both commands and constraints operate over an implicit collection of information called a *knowledge state*, which we define in Section 4.4.1. Both commands and constraints may receive input information before they are evaluated (*i.e.,* parameters) and inspect the knowledge state to make decisions. Importantly, constraints include an explicit indication as to their *scope*, the commands which they limit.

Definition: The *scope* of a constraint is the set of commands to that it is applicable. □

4.4.1 Knowledge State

The knowledge state is a universe based on standard access control and trust management concepts as discussed in Section 2.2. In particular we use agents to represent people and objects to represent resources or files. As in standard access control models, the relationships between agents and objects are described using rights. *Rights* are flags that indicate the presence of a relationship between agents and objects. The rights for a given model are derived from the source regulation and may include standard operating system permissions such as read/write/execute as well as more abstracted concepts such as "primary care physician of." Agents and objects are augmented with descriptive properties that describe their contents and disposition. Agents have special properties called roles that describe the social or formal positions that they occupy. Both agents and objects are annotated with tags that indicate their properties. For simplicity we simplify roles and tags to boolean properties that may be true or false for any particular agent or object. Roles differ from tags in that they are read only to the model (*i.e.,* commands can not modify an agent's roles). As in many privacy policy languages, whenever a person performs an action he includes information about the purpose of the action. The purposes that are in a model's vocabulary are determined by the language of the regulation. Thus, the universe for a given model includes a vocabulary of purposes extracted from the source regulation.

An instance of a knowledge state is a snap shot of the evolving universe. It includes information about the agents, objects, and rights that exist as well as their role and tag properties. As explained below, we include mechanisms for logging and sending information to agents, so a knowledge state includes information about the log and what information has been sent. The set of roles, tags, rights, and purposes used by a model are a static vocabulary and so are not included in the description of the universe.

As noted above in Section 4.1.1, we need to implement mechanisms for the encoding of diverse types of constraints. We use combinations of input parameters, tags, roles, purposes, and rights to encode the three different types of constraints as follows. Resolving each type of constraint requires using information present in the knowledge state in combination with parameter information. Type 1 can be resolved normally by inspection of the knowledge state to determine fundamental properties of the knowledge state such as the presence or absence of roles and rights. The parameters passed in indicate which agents or objects to inspect, but the information required to resolve the constraint is normally already present. Type 2 and type 3 constraints require more information to resolve. Both are resolved by relying more on the input parameters such as information about the purpose of an action. Tags also play a major role in the evaluation of type 2 and type 3 constraints. The difference between the two is that type 3 constraints rely on tags that are subjective and not as readily verifiable as those used for resolving type 2 constraints.

4.4.2 Evaluation Model

Since Privacy APIs are meant to closely model the structure of legal texts, we base our evaluation model on how a non-expert reader might interpret a legal policy. When a non-expert wants to perform an action that is restricted by law, the first step he would take is to search the applicable law to find the paragraph(s) or section that deals with a case most similar to the one in which he is interested. Having found the correct paragraph(s) or section, he may find that the text permits the action under a set of specific circumstances or conditions. The applicable paragraph(s) may include some conditions that are mentioned only by reference in the paragraph so he must find the referenced paragraphs to see what conditions are mentioned there. He also must check the context of the applicable paragraph(s) by skimming the surrounding sections to see if they impose any additional restrictions or mention any mitigating circumstances. Having collected all of the conditions mentioned in surrounding paragraphs, referenced paragraphs, and in the applicable text itself, he can finally evaluate whether all of the applicable conditions are satisfied. If they are, he may decide that the ruling from the law is to permit the action. Otherwise, he may decide that the ruling from the law is to forbid the action.

Let us consider the procedure taken by the hypothetical reader above to see how we may apply it to Privacy APIs.

Step 1: Searching the law to find the applicable paragraph(s) or section.

Step 2: Examining the conditions and requirements mentioned in the applicable text.

Step 3: Examining all other locations or rulings mentioned in the applicable text.

Step 4: Examining the surrounding text for any applicable restrictions or mitigating circumstances.

Step 5: Combining all applicable restrictions and mitigating conditions to derive a final ruling.

Step 6: Apply the ruling from the text to the case.

We adapt the above process to the evaluation of Privacy APIs as follows. First, to set up the initial conditions for the action, we create a knowledge state for the commands and constraints to use. As noted in Section 4.4.1, the knowledge state is the universe over which the commands and constraints operate and it includes information such as the agents that may interact with each other and other objects. Since the commands as derived from the regulation are the actions that the model allows, progress occurs by agents running commands in series. By *running* we mean that a list of parameters (including one for the agent who is performing the action, the *actor*) along with the knowledge state and command name are provided to an evaluation engine. The evaluation engine "runs" the command by using the parameters and knowledge state to check the constraints for the command. We discuss the constraint evaluation procedure in the next paragraph in depth, so for the moment let us just say that the engine evaluates the constraints and arrives at a decision, a *judgment*, as to whether they are satisfied. If the constraints are satisfied, the updates to the knowledge state included in the command are executed. We refer to the updates as the *true* branch for the command. Additionally, some commands include updates to perform in case the constraints are not satisfied. Those updates are the *false* branch for the command. Due to limitations that we discuss later in Section 5.2.3, some types of updates can only appear in the true branch while others can appear in either branch. Upon the completion of a command's updates, the knowledge state is updated and another command can be executed with the same or different parameters.

Commands behave in a manner consistent with a standard transition system. We may consider the knowledge state as representative of the state of the transition system and the commands as transitions on the knowledge state. The transitions are conditioned by guards and constraints that determine whether a transition may be taken. We discuss the transition system in more details in Section 5.2.3.

Constraints on commands In order to handle the diverse constraints and references mentioned above in Section 4.1.1 and Section 4.1.2 in a manner similar to the human level procedure described above, we implement a multi-step constraint evaluation procedure. Since the result is a logical AND of all of the conditions, they may be collected in any order, so the steps below could be ordered differently or performed concurrently without affecting the final decision. The procedure followed by the evaluating engine follow for checking the constraints of commands is as follows:

1. The engine first checks to see what constraints have the command to run in their scope. This is done to check for invariants as shown above in Example 4.1.4 and corresponds to Step 4 above. We call this process the *constraint search.* In order to do the constraint search, the name of the command to be run is checked against the scopes for all of the constraints. Any constraints that have the command in their scopes are collected in a list for checking. Figure 4.1 illustrates how the invariants structure works with two constraints including Command1 in their scope. The constraints are invariants on Command1 and must run before the command can be run.

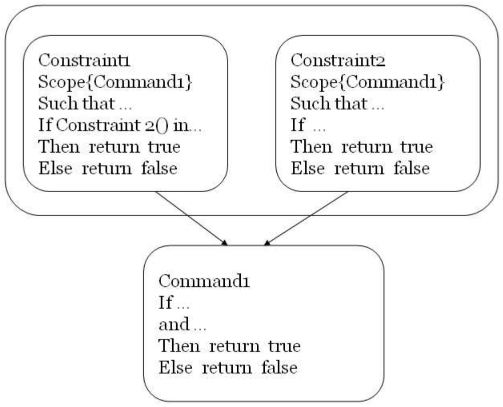


Figure 4.1: Invariants on a command

2. With the list, the engine needs to check which constraints are applicable to the given knowledge state and parameters. Constraints that are not applicable may be removed from the list. For instance, if a constraint limits some an action related to children under 18, if the situation involves a 30 year old, the constraint is not applicable. We call such checks *such that* guards.

3. The engine then evaluates all of the constraints on the list, that is the ones which have

the command in scope and are applicable. Each constraint is resolved to a boolean result that indicates its decision. We call the result the constraint's *judgment.* True means that the command should be allowed to run. False means that it should not be allowed to run. If the judgment of any constraint on the list is false, the resulting judgment is false (*i.e.,* a most-strict combination of judgments), the command is forbidden to run and the updates in the false branch of the command are performed. If there are no constraints on the list, the default judgment is to allow the command. This derives a result for Step 4 so it can be combined with the results from the other steps.

4. If the judgment from the constraint search is true (*i.e.,* allow the command to execute), the evaluation engine proceeds to examine the constraints contained in the command itself. We call the constraints inside a command its *guards.* Included in the command's guards are properties of the knowledge state, properties of the parameters, or the names of constraints. Names of constraints appear in cases such as shown above in Examples 4.1.2 and 4.1.3. The evaluation engine checks the properties directly against the knowledge state and parameters. It evaluates any constraints named by running them and deriving their judgments. Since any constraint that is run is mentioned explicitly by the command, its judgment is derived ignoring whether the command appears in the constraint's scope.

5. Using the combined results from the constraint search and the guards, the evaluation engine decides which branch of the command to execute. If the constraint search result permits the command and the guards all permit the command as well, the updates in the *true* branch of the command are performed. Otherwise, the updates in the *false* branch are performed.

The result of performing the constraint search steps above is that we have a list of the constraints that are applicable to the command since we have filtered out all non-applicable ones based on their *such that* guards and scopes. We have also derived a judgment from the applicable constraints that we can use for deriving a final judgment as mentioned in Step 5 above.

Intuitively there are two ways that we might combine the results from multiple constraints. Using a *most strict* algorithm, a forbidding result from a constraint overrides any permitting results. If we represent a permitting judgment as “true” and a forbidding judgment as “false”, the logic for such an algorithm would parallel a logical AND of the results. Using a *most lenient* algorithm, a permitting result from a constraint overrides any forbidding results. Using true and false representations, the logic for such an algorithm would parallel a logical OR of the results. We use both types of algorithms, most strict and most lenient, in different scenarios of constraint combination as we shall discuss.

In order to concretize the steps discussed we give a simple example of how the evaluation engine works.

Example 4.4.1 (Helping Part 1)

Since this example is meant to illustrate the evaluation process we use a simplified version of the Privacy Commands language.

Let us consider a store that has many employees. Some employees are designated to help customers with their shopping while others are assigned other duties. The store’s policy is that any employee may say hello to a customer, but only those designated as helpers may offer to help them. The policy is expressed in the following rules:

1. Employees may greet customers.
2. Employees who are designated Helpers may offer to assist customers.

Let us design a few commands and constraints to describe the store’s policy. Let the knowledge state be three agents: Harry, Ed, and Roy. There are no other objects or tags. There are two roles that the agents may hold: Helper and Employee. Let Harry have the roles Helper and Employee. Let Ed have the role Employee. Let us consider the following constraint and command that are formatted similarly to Privacy Commands.

```
 CMD  CmdHelping(actor, recipient)
  If  true
Then  actor sends “Can I help you?” to recipient
Else
```

The first command is executed by someone who wants to offer help to someone in the store. Line 1 begins with the abbreviation “CMD” that indicates that the text to follow is a command. The name of the command follows: CmdHelping. In parentheses are the parameter values that are passed to CmdHelping. The parameter “actor” indicates which agent is to perform the action. The parameter “recipient” indicates which agent is to receive the action. Line 2 is the guard for the command. In this case it declared “true”, so there is nothing to check. Line 3 begins the true branch for the command. It sends a message from the actor to the recipient offering help. Line 4 is the false branch for the command and is empty.

```
1   CMD   CmdGreeting(actor, recipient)
2    If   true
3  Then   actor sends “Hello” to recipient
4  Else
```

The second command is similar to CmdHelping. Its true branch, line 3, however, sends a “Hello” greeting to the recipient.

```
1        CST   CstHelping(actor, recipient)
2      Scope   {CmdHelping}
3  Such That   actor in role Employee
4         If   actor in role Helper
5       Then   Allow
6       Else   Forbid
```

The constraint places a limit on which employees can offer help in the store. Line 1 begins with the abbreviation “CST” that indicates that the text to follow is a constraint. The name of the constraint follows: CstHelping. The parameters are the same as in CmdHelping. Line 2 declares the commands to which the constraint are applicable, its scope. Since CmdHelping deals with offering help, it is included in the scope. Line 3 limits the applicability of the constraint to employees since they are ones to whom the policy applies. If the actor isn’t an employee, the command isn’t applicable. The check is the constraint’s *such that* guard. Line 4 checks that the actor is a helper. It is the guard for the constraint. Line 5 is the “true” branch for the constraint and allows the action. Line

6 is the “false” branch and forbids the action.

Let us now consider the following scenarios:

CmdGreeting(Harry, Roy) When the evaluation engine attempts to run the command CmdGreeting(Harry, Roy), it first checks to see if any constraints are applicable. Since CmdGreeting is not in the scope of CstHelping, the constraint search results in a default judgment that allows it. The evaluation engine then checks the guard for CmdGreeting on line 2. Since the guard is “true”, it is automatically satisfied. Since the judgment from the constraint search (Allow) and the guard both allow the action, the true branch is run and Harry sends “Hello” to Roy.

CmdHelping(Harry, Roy) When the evaluation engine attempts to run the command CmdHelping(Harry, Roy), it first checks to see if any constraints are applicable. Since CmdHelping is in the scope of CstHelping, it is run. Its *such that* guard checks that Harry has the role Employee. Since he is, the constraint is applicable and so the guard on line 4 is checked. It checks that Harry has the role Helper. Since he does, it is satisfied and the true branch is run which permits the action. The resulting judgment from CstHelping is therefore Allow. The evaluation engine then checks the guard for CmdHelping on line 2. Since it is “true”, it is automatically satisfied. Since the judgment from the constraint search (Allow) and the guard both all the action, the true branch is run and Harry send “Can I help you?” to Roy.

CmdHelping(Ed, Roy) When the evaluation engine attempts to run the command CmdHelping(Ed, Roy), it performs the same steps as in the previous case. It first checks to see if the command is in the scope of CstHelping. Since it is, it checks the *such that* guard. Since Ed has the role Employee, it is satisfied and therefore CstHelping is applicable. The guard for the constraint on line 3, however, is false since Ed does not have the role Helper. The false branch for the constraint is therefore run and the resulting judgment is Forbid. The evaluation engine then checks the guard for CmdHelping and since it is “true”, it is automatically satisfied. Since the constraint search judgment is Forbid, the false branch for the command is run and no message is sent.

Given the above policy, it may be interesting to evaluate whether Roy may offer to help others. That is, since Roy does not have the role Employee, the policy is unclear about whether he may greet or offer to help others. We return to this point later in Example 4.4.3.

□

In the commands in Example 4.4.1, CstHelping is an invariant since it includes CmdHelping in its scope without CmdHelping acknowledging it. Let us next consider a modified version with an explicit reference.

Example 4.4.2 (Helping Part 2)

The store discovered that even though it did not allow non-helpers to offer help, employees who said “Hello” to customers were often asked for help in response to their greeting. In order to prevent the situation from arising again, the store decided to change its policy to only allow employees who are permitted to offer help to say “Hello” to customers. The policy rules are as follows:

1. Employees who are permitted to offer to help customers may greet customers.
2. Employees who are designated Helpers may offer to help customers.

Since the policy already includes a constraint CstHelping that determines who can offer help, the greeting command can be rewritten in terms of the helping requirements without changes to CmdHelping and CstHelping.

```
1  CMD   CmdGreeting(actor, recipient)
2    If  CstHelping(actor, recipient)
3  Then  actor sends “Hello” to recipient
4  Else
```

The new guard on line 2 is a reference to CstHelping. By including it as a guard, CmdGreeting now allows the actor to say “Hello” only if CstHelping permits the action.

Let us now reconsider the first scenario from Example 4.4.1 as well as a variation of it:

CmdGreeting(Harry, Roy) When the evaluation engine attempts to run the command CmdGreeting(Harry, Roy), it first checks to see if any constraints are applicable.

Since CmdGreeting is not in the scope of CstHelping, the constraint search results in a default judgment that allows it. The evaluation engine then checks the guard for CmdGreeting on line 2. The guard indicates to check the judgment from CstHelping and so it is run even though CmdGreeting does not appear in its scope. The *such that* guard for CstHelping on line 3 checks that Harry is an employee. Since he is, the command is applicable. The guard on line 4 checks that Harry has the role Helper. Since he does, the true branch is taken and the judgment from the constraint is Allow. Since the judgment from the constraint search (Allow) and the guard both allow the action, the true branch is run and Harry sends "Hello" to Roy.

CmdGreeting(Ed, Roy) When the evaluation engine attempts to run the command CmdGreeting(Ed, Roy), it first checks to see if any constraints are applicable. As above, CmdGreeting is not in the scope of CstHelping and so the result is a default Allow judgment. As above, the evaluation engine then checks the judgment from CstHelping as per line 2 even though CmdGreeting does not appear in its scope. The *such that* guard for CstHelping on line 3 checks that Ed is an employee. Since he is, the command is applicable. The guard on line 4 checks that Ed has the role Helper. Since he does not, the false branch is taken and the judgment from the constraint is Forbid. Since the judgment from the constraint search is Allow and the guard Forbids the action, the false branch is run and nothing is sent.

□

The above example illustrates the difference between how constraints are treated when they are evaluated as invariants and when they are evaluated via direct reference. In the former case the evaluation engine considers the scope of the constraint. In the latter case it may ignore it. The inclusion of the *such that* guards introduces another dimension for constraints, however, which we discuss next.

Intuitively, the scheme for constraint evaluation will terminate if we ensure that there are no circular references between constraints. We formalize this intuition in Section 5.4.5 with Lemma 5.4.1.

4.4.3 Judgments and Derivation of Judgments

The addition of *such that* guards and scope makes the judgments derived from constraints more complex than just Allow and Forbid. Recall that scope and *such that* guards are designed to filter the applicability of a constraint, but they operate at different levels. Scope operates at the command level, declaring whether particular commands are subject to the constraint irrespective of the knowledge state or parameters. Conversely, *such that* guards operate at the input level, examining the knowledge state and parameters for a given situation. In combination, they create a flexible mechanism for focusing the applicability of constraints.

We use judgments to summarize the decision of constraints in particular situations. In designing the derivation mechanism for judgments, it is essential to distinguish between the two different scenarios in which constraints are run:

Constraint Search Before any command is executed, the evaluation engine examines which constraints are applicable to it and the current situation. For the search, since we are interested in just discovering invariants which may not be mentioned explicitly in the command's guards, we strictly are interested in constraints that are directly applicable to the command and the situation. Therefore, any constraint for which the command is not in scope or the *such that* guards are not applicable may be ignored. Only constraints that have the command in scope and for which the *such that* guards are satisfied need be consulted for their judgment. The final decision from the constraint search is then the combination of the judgments from the applicable constraints using a most strict algorithm. We use a most strict algorithm since if any of the constraints forbids the command to be run, its judgment should be respected.

Explicit Reference A command's guards may include a constraint name as a guard. By including the constraint's name, the guard is introducing the judgment of the constraint as a condition on the command. When the evaluation engine sees the constraint name, it runs the constraint to derive its judgment. Since the command has explicitly referenced the constraint, the scope of the constraint is ignored in deriving the judgment. The command's inclusion of the constraint as being applicable is sufficient for the evaluation engine. Even though the scope is ignored, the *such that* guards are still evaluated to determine whether the particular situation is applicable. If they are not satisfied, the constraint issues a *don't care* judgment since the situation is not applicable according to the constraint. The *don't care* judgment is issued in combination with the result from the other guards for the constraint. The final judgment as received by the evaluation engine must then be checked against the expectations of the command.

The scenarios above lead to the derivation of five judgments that may result from a constraint in different situations. Their names, a short description, and applicability are shown in Table 4.1. The judgments *Allow* and *Forbid* directly permit or forbid the running of a command. An *Ignore (Allow)* (sometimes shortened to *Ignore*) judgment is the result of a constraint during a constraint search when the command in question is not in scope or the *such that* guards are not satisfied. When an explicit reference is the source of a constraint being run, if the *such that* guards are not satisfied the result is a *don't care* prefix for the result of the rest of the guards. Therefore, if the *such that* guards are not satisfied, but the other guards are, the judgment is *Don't Care/Allow*. If the other guards are also not satisfied, the judgment is *Don't Care/Forbid*. The implication of Don't Care/Allow and Don't Care/Forbid is that the case presented to the constraint is not directly applicable, but ignoring the differences, the constraint would have permitted (or forbid) the action. We postpone a fuller discussion of the derivation of judgments from constraints and their combination algorithm to the formal discussion of constraints in Section 5.2.4.

In order to give a better intuition for how the different judgments are used in policies let us consider an example from the scenario we have discussed previously.

Example 4.4.3 (Helping Part 3)

The store's policy relating to employees offering help and greeting customers was working. The store posted its policies on the wall so that customers would understand why only certain employees would greet them or offer help. After posting the policy, however, customers began to wonder whether they were included in the prohibition to help or greet

Table 4.1: Names, description, and applicability of judgments

Judgment	Description	Search	Reference
Allow	Permit the command to be run.	✓	✓
Forbid	Forbid the command to be run.	✓	✓
Ignore (Allow)	Constraint ignores the command.		✓
Don't Care/Allow	Situation not applicable. It would otherwise permit.		✓
Don't Care/Forbid	Situation not applicable. It would otherwise forbid.		✓

other customers since the policy didn't make any indication about non-employees. The store decided that for liability reasons everybody would be subject to the restriction on offering help, however the greeting policy would be applicable only to employees. The policy rules are as follows:

1. Employees are restricted to greeting customers only if they may offer to help.

2. Only employees who are designated Helpers may offer to help customers.

The resulting policy requires some tuning of the commands using the judgments that may be derived from the constraints, but no additional commands. For reference, CstHelping is shown below as from Example 4.4.1.

```
1        CST   CstHelping(actor, recipient)
2      Scope   {CmdHelping}
3  Such That   actor in role Employee
4         If   actor in role Helper
5       Then   Allow
6       Else   Forbid
```

```
CMD   CmdGreeting(actor, recipient)
  If  CstHelping(actor, recipient) ∈ {Allow, Don't Care/Allow,
          Don't Care/Forbid}
Then  actor sends "Hello" to recipient
Else
```

The greeting policy is amended to indicate that unless the constraint issues a Forbid judgment, the actor may say "Hello" to a customer. This is shown on line 2 where the name of the constraint is followed by the set inclusion symbol ∈ and a set of judgments to examine {Allow, Don't Care/Allow, Don't Care/Forbid}. The evaluation engine processes the guard by first deriving the judgment from CstHelping and then comparing it against the judgments listed afterwards. If the judgment from CstHelping is included in the set, the guard is satisfied. Otherwise, it is not satisfied.

```
1  CMD   CmdHelping(actor, recipient)
2    If  CstHelping(actor, recipient) ∈ {Allow}
3  Then  actor sends "Hello" to recipient
4  Else
```

The helping policy remains unchanged since it remains applicable to everyone. We update the syntax with the expected judgment from the constraint. As above, if the judgment from CstHelping is Allow, the guard on line 2 is satisfied. Otherwise, it is not satisfied.

Let us consider now four scenarios for the new policy. Let the knowledge state include four agents: Harry, Ed, and Roy, and Nancy. Harry, Ed, and Roy are as in Example 4.4.2. Nancy is a non-employee and holds no roles. The first two scenarios are as above in Examples 4.4.1 and 4.4.2 and the last two are a new variation.

CmdHelping(Harry, Roy) As in Example 4.4.1, the evaluation engine first performs a constraint search. Since CmdHelping is in the scope of CstHelping and its *such that* guards and other guards are all satisfied, the resulting judgment is Allow. When the evaluation engine then checks the guard in line 2, it runs CstHelping to derive its judgment (an optimization would be to cache the results from CstHelping) of Allow. Since Allow is in the set judgment set {Allow}, the true branch is taken and Harry offers to help Roy.

CmdGreeting(Ed, Roy) As in Example 4.4.2, the evaluation engine first performs a constraint search. Since CmdGreeting is not in the scope of CstHelping, its judgment is Ignore (Allow), permitting the command to run. When the evaluation engine runs CstHelping on line 2, however, it ignores the scope and derives a judgment based on the *such that* guard and other guards. Since the *such that* guards are satisfied and the other guard is not, the resulting judgment is Forbid. Since Forbid is not in the judgment set on lines 2–3, the false branch is taken and no communication occurs, forbidding Ed to greet Roy.

CmdGreeting(Nancy, Roy) The evaluation steps for CmdGreeting(Nancy, Roy) begin similarly to those for (Ed, Roy). The constraint search results in an Ignore (Allow) judgment as above. However, when the guard on line 2 is run, the *such that* guard on line 3 for CstHelping is not satisfied since Nancy does not hold the role Employee. The other guard is also not satisfied since Nancy does not hold the role Helper. The resulting judgment from CstHelping is thus Don't Care/Forbid. However, since Don't Care/Forbid is included in the judgment set on lines 2–3, the guard is satisfied. The true branch of CmdGreeting is therefore run and Nancy says "Hello" to Roy.

CmdHelping(Nancy, Roy) The evaluation steps for CmdHelping(Nancy, Roy) begins with a constraint search. Even though CmdHelping is in the scope for CstHelping since Nancy does not have the role Employee, the *such that* guard on line 2 is not satisfied and the resulting judgment is Ignore (Allow). When the evaluation engine checks CstHelping on line 2 of CmdHelping, the resulting judgment is Don't Care/Forbid as for CmdGreeting(Nancy, Roy). Since Don't Care/Forbid is not in the judgment set {Allow}, the guard is not satisfied and the false branch is taken. No communication occurs and therefore Nancy may not offer help to Roy.

□

4.4.4 Overloading

For simplicity we have restricted constraints and commands to using only AND combinations of guards. This may be overly restrictive for constraints that offer multiple options for fulfillment. For instance, a constraint that allows an action to proceed under one of two circumstances would require the maintenance of two copies of each referring command to check either of the two resulting constraints as a guard. This is unacceptable since it is common for regulatory paragraphs to mention multiple ways of fulfilling a constraint and creating disparately referenced constraints that differ from the text would conflict with requirements 1 and 2 as mentioned above in Section 3.2.

In order to better model such constraints with multiple options for fulfillment we introduce a limited OR mechanism for constraints that we call *overloading*. For instance, when a regulatory paragraph offers two conditions for fulfillment A and B, we create two *overloaded constraints* for the paragraph with identical names, one for A and the other for B. When the paragraph name is then referenced by a command, both A and B are run and the most lenient judgment of the two is selected by the evaluation engine. This corresponds to the intuition that if either A or B are satisfied, the paragraph's restriction has been met. Since the judgment derivation is performed by the evaluation engine, the resulting judgment appears the same as from a regular constraint. Commands can therefore reference overloaded constraints identically to non-overloaded constraints. The operation of overloaded constraints is reminiscent of overloaded functions in a C when the compiler selects the appropriate function to run from a list of identically named functions. Overloaded constraints differ from overloaded functions, however, in that all of the overloaded constraints are run in order to derive a judgment instead of C's requirement that a maximum of one function be executed.

Figure 4.2 illustrates how overloaded constraints are run. First, the command includes a reference to run the constraint. The evaluation engine then runs all of the instances of the constraint and combines their judgments. After combining the judgments, it returns the resulting judgment to the command.

While we create separate constraints for each situation in a paragraph, we place one restriction on overloaded constraints: that they must all share the same scope. This

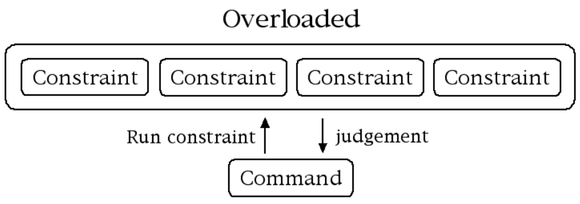


Figure 4.2: Running an overloaded constraint

restriction is practical since paragraphs normally declare their applicability without respect to the options for fulfillment. It also simplifies the constraint search somewhat.

Overloading constraints is a syntactic simplification meant to reduce the number of references from commands. Any set of overloaded constraints could be transformed into a set of non-overloaded constraints provided that all referencing commands are changed to refer to each constraint independently. Overloading is a form of "syntactic sugar" to make Privacy APIs simpler and easier to read but does not add any new semantic constructs to the language.

In order to give an intuition for how overloaded constraints work, let us continue with the example from the previous subsections.

Example 4.4.4 (Helping Part 4)

The store's restriction of who can offer help to customers hit a snag when customers wanted to ask the store manager for help in special cases. Since the manager is also an employee she was not permitted to offer help as per the policy, placing her in a difficult situation. In order to resolve this, the store changed its policy to permit Managers and Helpers to offer help to customers. The policy rules are then as follows:

1. Employees are restricted to greeting customers only if they may offer to help.
2. Only employees who are designated Helpers or Managers may offer to help customers.

Rule 1 is the same as in Example 4.4.3, but rule 2 differs by offering two options for fulfillment. Under rule 2, if the actor has the roles Helper or Manager the constraint is satisfied. Also note that as a consequence of the change in rule 2, that managers are now permitted to greet customers since the greeting right is defined in terms of the right to offer help. Let us first consider how the policy would appear without using overloading.

```
      CST   CstHelping1(actor, recipient)
    Scope   {CmdHelping1, CmdHelping2}
Such That   actor in role Employee
       If   actor in role Manager
     Then   Allow
     Else   Forbid
```

```
      CST   CstHelping2(actor, recipient)
    Scope   {CmdHelping1, CmdHelping2}
Such That   actor in role Employee
       If   actor in role Manager
     Then   Allow
     Else   Forbid
```

Since there are now two options for fulfilling the Helping constraint, we implement the two as separate constraints. Now in order to reference both options for the Helping constraints, we must create multiple instances of the Greeting and Helping commands:

```
 CMD   CmdGreeting1(actor, recipient)
  If   CstHelping1(actor, recipient) ∈ {Allow, Don't Care/Allow,
       Don't Care/Forbid}
Then   actor sends "Hello" to recipient
Else
```

```
 CMD   CmdGreeting2(actor, recipient)
  If   CstHelping2(actor, recipient) ∈ {Allow, Don't Care/Allow,
       Don't Care/Forbid}
Then   actor sends "Hello" to recipient
Else
```

```
1  CMD    CmdHelping1(actor, recipient)
2    If   CstHelping1(actor, recipient) ∈ {Allow}
3  Then   actor sends “Hello” to recipient
4  Else
```

```
1  CMD    CmdHelping2(actor, recipient)
2    If   CstHelping2(actor, recipient) ∈ {Allow}
3  Then   actor sends “Hello” to recipient
4  Else
```

Now in order to check whether an agent can send a “Hello” greeting to another, we must select from either CmdGreeting1 or CmdGreeting2. The two are identical except with respect to the constraint that they reference, so they are somewhat redundant and reduce the clarity of the model. The same is true of the CmdHelping1 and CmdHelping2 commands. For each additional option for CstHelping we would need to add more commands linearly. For each additional constraint not related to CstHelping we would need to add more commands exponentially, quickly leading to an explosion in the size of the model.

The key observation here is that the problem of multiple references is caused by the OR logic in the Helping rule. We can use overloading to better model OR constraint choices and thereby collapse the size of the model. By collapsing CstHelping1 and CstHelping2 into one name CstHelping that can be referenced as one constraint, we translate the helping rule into two constraints as follows:

```
1        CST   CstHelping(actor, recipient)
2      Scope   {CmdHelping}
3  Such That   actor in role Employee
4         If   actor in role Helper
5       Then   Allow
6       Else   Forbid
```

```
        CST  CstHelping(actor, recipient)
      Scope  {CmdHelping}
  Such That  actor in role Employee
         If  actor in role Manager
       Then  Allow
       Else  Forbid
```

Now we can use the same CmdGreeting and CmdHelping commands from Example 4.4.3 above. For reference, they are as follows.

```
  CMD  CmdGreeting(actor, recipient)
   If  CstHelping(actor, recipient) ∈ {Allow, Don't Care/Allow,
       Don't Care/Forbid}
 Then  actor sends "Hello" to recipient
 Else
```

```
  CMD  CmdHelping(actor, recipient)
   If  CstHelping(actor, recipient) ∈ {Allow}
 Then  actor sends "Hello" to recipient
 Else
```

Note that since the constraints have the same name as in the previous policies, we do not need to revise the commands at all. The evaluation engine takes care of the reference by combining the judgments from the CstHelping constraints so that the command does not care whether it is invoking an overloaded constraint. Let us consider a few scenarios for the above policy. Let the knowledge state include four agents: Harry, Ed, and Roy, and Mary. We augment the roles that the agents may hold to be: Employee, Helper, and Manager. Harry, Ed, and Roy are as in Example 4.4.2. Mary is a manager who holds the roles Employee and Manager.

CmdHelping(Harry, Roy) When the evaluation engine runs CmdHelping(Harry, Roy), it first performs a constraint search as mentioned in the scenarios in previous examples. Since the evaluation engine sees that CstHelping has CmdHelping in its scope and CstHelping is overloaded, it runs both versions of CstHelping and combines their judgments. The first CstHelping yields a judgment of Allow since Harry has the roles Employee and Helper. The second CstHelping evaluates to Forbid since Harry has the role Employee (thereby satisfying the *such that* guard on line 3) but does not have the role Manager (violating the guard on line 4). Since the two constraints are options for fulfilling the requirement, the evaluation engine combines the Allow and Forbid judgments using a *most lenient* algorithm. The final judgment is therefore Allow since it is the most lenient result. The command CmdHelping is then run as above in the scenarios in Example 4.4.3.

CmdHelping(Mary, Roy) The evaluation engine runs CmdHelping(Mary, Roy) in a manner similar to the previous scenario of CmdHelping(Harry, Roy). However, during the constraint search the first CstHelping instance yields Forbid since Mary has the role Employee but not the role Helper and the second CstHelping instance yields Allow since Mary has the roles Employee and Manager. As before, the evaluation engine uses a most lenient algorithm to combine the judgments (yielding Allow) and Mary offers to help Roy as above in Example 4.4.3.

CmdGreeting(Mary, Roy) When the evaluation engine runs CmdGreeting(Mary, Roy), it first performs a constraint search. Since CmdGreeting is not in the scope of the CstHelping constraints, the resulting judgment from the constraint search is Ignore (Allow). When the evaluation engine runs CmdGreeting, it checks the reference to CstHelping on line 2. Since CstHelping is overloaded, both instances of CstHelping are run. The first instance yields Don't Care/Forbid since Mary has the role Employee but not Helper. The second instance yields Allow since Mary has the roles Employee and Manager. As during the constraint search, the evaluation engine combines the two judgments using a most lenient algorithm, yielding Allow. Since Allow is in the set of judgments on lines 2–3, the guard in CmdGreeting is satisfied and Mary is permitted to greet Roy.

In this example, we could perhaps have avoided using overloading by introducing an OR guard in line 4 of CstHelping, for instance changing it to "actor in role {Helper, Manager}" to indicate that it should be satisfied if the actor has either role. We then could collapse

CmdHelping into a single constraint and not need to use overloading. We use the example as given primarily for illustrative reasons since more complex policies can not be easily collapsed as such, but also to retain the atomicity of guards as much as possible and thereby making policies slightly easier to compose, model, and understand. □

From the scenarios we see that since the evaluation engine takes care of the running and judgment combination of overloaded constraints, references for overloaded and non-overloaded constraints are handled identically. We use overloading to let constraints offer multiple options for fulfillment without creating an explosion of references, but any collection of overloaded constraints can be rewritten without using overloading at the price of complexity. Example 4.4.4 shows this by offering the same policy with and without the use overloading.

4.5 Alternate Approaches

The Privacy Commands language and evaluation engine structure, while complex, enables an expressive and flexible modeling of privacy policies. Constraints in particular are treated in a complex way with subtle combinations or scopes, *such that* guards, and overloading. The most important design decision made in the design of the language and the evaluation engine is the strict adherence to the structure of the legal text. As shown in Section 4.1.1 and Section 4.1.2, legal policies use a variety of complex constraint and reference mechanisms. Rather than designing simpler mechanisms with similar semantics, we enable the implementation of policies which maintain the original structure and style of the source text's constraints and references even at the expense of simplicity. As we explore in Section 5.5, our representation enables the evaluation of interesting policy comparison metrics. Used in combination with our representation, we can more easily explore how the properties and permissiveness of a policy is affected by its use of references and constraints.

To contrast our approach, let us consider other languages and systems from the privacy policy literature as discussed in Section 2.3. Three policy systems mentioned there are very similar to our work so we shall describe them in depth. In order to demonstrate how they differ from our work, we use an example policy fragment, derived from the HIPAA Privacy Rules. The 2000 HIPAA Privacy Rule [§164.506(a)(1)-(2), v.2000] permits use and disclosure of protected health information under certain circumstances. Two simplified conditions from the text are as follows:

1. A covered entity may use protected health information for its own treatment.
2. Unless permitted by 506(a)(2) or (3), a covered entity must receive consent from a patient before using protected health information.

The source text for the above rules is in Appendix B.1.1 and is considered in depth in Example 7.1.1. The commands and constraints for the quote are shown in Appendix C.1.2. Let us now examine how the above policy would be modeled by three other languages from the literature.

P3P

A policy fragment for the permissions 1 and 2 in P3P is as follows:

```
<purpose>
<other-purpose required="always">treatment</other-purpose>
<other-purpose required="opt-in">any</other-purpose>
</purpose>
```

The "required" attribute indicates whether actions for a purpose need no consent (always), opt-in consent, or opt-out consent. The statement does not identify the agents performing the actions, but by specifying the purposes separately we can capture the notion of requiring consent using the required attribute. If the obligation had been something else, requiring patient notification for example, it would not have been expressible.

Contextual Integrity Barth, *et al.* [15] model contextual integrity based using temporal logic formulas called norms to describe allowed and forbidden actions. Formulas consider a message's sender, its recipient, the agent who is its subject, and meta-information about the content of the message. Positive norms (permissions) and negative norms (requirements) may include an additional obligation clause which can inspect additional properties and require past or future messages. As an aside, the symbol used in Barth, *et al.* for past requirements is a diamond with a bar through it. For technical reasons, we use $\blacklozenge$ instead.

A positive norm for 1 must adapt a "use" to be the sending of a message, in this case a message the sender sends to herself:

$$\text{inrole}(p_1,\ \textit{covered-entity}) \wedge (p_2 = p_2) \wedge \text{inrole}(p_3,\ \textit{patient}) \wedge (t \in \textit{protected-health-information}).$$

The norm checks that the sender (p_1) is a covered entity, that the recipient (p_2) is the same as the sender, that the message subject (p_3) is a patient, and that the message (t) is protected health information. We can not include purposes such as treatment in clauses because they are not supported, however it would not be difficult to add them. There also is no way to express the notion of "otherwise" using the formulas since all negative norms must be satisfied before any positive ones can be exercised. . The closest we could get for 2 is a negative norm:

$$\text{inrole}(p_1,\ \textit{covered-entity}) \wedge (p_2 = p_2) \wedge \text{inrole}(p_3,\ \textit{patient}) \wedge (t \in \textit{protected-health-information}) \rightarrow \blacklozenge\ \text{send}(p_3, p_1,\ \textit{usage-opt-in})$$

The norm, however, does not capture the concept of the use being for a different purpose or that it is exercised only if the permissions in the other location are not applicable.

EPAL

EPAL rules are similar to constraints in that they return judgments based on parameter value, but don't perform state updates. An EPAL rule for 1 is:

```
<rule id="OwnTreatment" ruling="allow">
      <user-category refid="Covered-Entity"/>
      <data-category refid="Protected Health Information"/>
      <purpose refid="Treatment"/>
```

```
        <action refid="Own-Use"/>
</rule>
```

The rule allows covered entities access to protected health information for its own treatment use. There is no corresponding notion of permitted elsewhere, but since EPAL policies are evaluated in order, an enterprise could place the following rule at the end of its policy to emulate 2:

```
<rule id="OtherUse" ruling="allow">
        <user-category refid="Covered-Entity"/>
        <data-category refid="Protected Health Information"/>
        <purpose refid="Any"/>
        <action refid="Use"/>
        <condition refid="Granted-Consent"/>
</rule>
```

The rule adds a condition that consent had been previously granted. The body of the condition evaluates whether consent had been granted previously and that the granter had a particular name.

Aside from the hazards of creating ordered rule sets [3], EPAL is limited in that its rules can not directly examine or update state, include the recipient of messages, or impose obligations that are linked to concrete actions or conditions. EPAL policies depend on well defined vocabularies and systems that enforce the meaning of vocabulary terms.

Discussion

The limitations in the above systems center around the management of purposes and constraints. All three models are based fundamentally on an access control paradigm where policies are compartmentalized, giving simple combinations of rules which much be applied to specific circumstances. Barth, *et al.* 's provides a robust mechanism for the writing of policies by including the possibility for negative norms to override positive ones, however its LTL logic based approach does give it the flexibility to deal with contingencies and references which commonly appear in larger legal policies. A representation suited for modeling legal policies must be adapted to their idioms and structure in order to be accurate and representative. Through the examples and case studies in the chapters that follow we show that the structures and evaluation engine algorithms in Privacy Commands are necessary and sufficient for the task.

4.6 Conclusion

In this chapter we developed Auditable Privacy Systems and how they fulfill the goals of the Formal Privacy framework described in Chapter 3. Since the framework is focussed on the development of regulatory models, we describe two usage cases and requirements for languages which are designed in the framework. We then informally describe the Privacy Commands language and show how its features are used and can address the requirements. Our informal presentation is meant to provide an intuition for Chapter 5 which goes more deeply into syntax and semantics.

Chapter 5

Formal Language and Framework for Privacy APIs

In the previous chapters we have developed an informal description of the requirements for our formal privacy language. In this chapter we give technical details about our formal Privacy Commands language including its syntax and semantics. As shown in examples in Section 3.2, legal privacy documents generate a combination of commands and constraints that interact through references and deferences. As discussed previously in Section 4.4, Privacy APIs use commands and constraints to model the permissions and references that are combined to make policy decisions. In this chapter we develop the formal model for the Privacy Commands language, discuss its syntax and semantics, and develop relations that let us analyze their properties.

The rest of this chapter is organized as follows. In Section 5.1 we discuss the fundamental types and sets that we use in the formal model for Privacy Commands. In Section 5.2 we introduce the syntax for guards and operations and how they are used to create commands and constraints. In Section 5.2.2 we introduce the typing system used for Privacy Commands. In Section 5.2.5 we present a grammar for Privacy APIs and provide descriptive examples to show its use in Section 5.2.6. In Section 5.3 we present the operational semantics for Privacy Commands. In Section 5.4 we discuss the evaluation engine for Privacy Commands and how it interacts with the various features of the language which may manifest in a Privacy API. In Section 5.5 we develop the formal relations for strong and weak licensing, policy relations defined in the same vein as strong and weak bisimulation from process calculi, and relate them to the evaluation structures defined in this chapter. We conclude in Section 5.6.

5.1 Fundamental Types and Sets

Before presenting the formal model for Privacy Commands, we must explore the fundamental sets, operations, and atoms which the language uses. We base our formal model on the fundamental models for access control and privacy policies discussed above in Section 2.2 using the concepts of agents, objects, rights, and actions to model the people, resources, and behaviors that models may make decisions about. We extend the standard access control model with the privacy policy concept of purposes for actions. We annotate objects and agents with tags to indicate meta-data and use a log to record evidence. We also include a construct for the sending of messages to agents to inform them of events. Since such communication is out of the scope of the policy and its evaluation, we formally model such messages by recording evidence in the log of each message and its recipient.

The type and sets in the universe are thus as follows. As a rule we use small caps to indicate type names (*e.g.,* Type) and italics for set names (*e.g., Set*). To indicate that a variable is of a given type we use the colon notation. For instance, v :V means that variable v has type V.

Agent is the type for agents. The set *Agent* is a finite set of objects of the type.

Object is the type for objects. The set *Object* is a finite set of objects and agents. The type Agent is a subtype of Object and so we enforce that *Object* includes all agents ($\mathit{Agent} \subseteq \mathit{Object}$).

Role is the type for roles. The set *Role* is a finite, non-hierarchical set of roles. Roles are implemented as properties of agents which are not updateable by commands. For a : Agent, we use the function $\mathit{Roles}(a)$ to extract the set of roles that an agent holds. For $k_1 \in \mathit{Role}$ and using $a.k_1 = \mathit{true}$ to denote that a holds the role k_1,

$Roles(a) = \{k \in Role | a.k = true\}$.

RIGHT is the type for rights which represent relations between agents and objects. The finite set *Right* are the rights recognized by a policy. A knowledge state stores the rights in a matrix $Matrix \subset Agent \times Object \times \mathsf{pwr}(Right)$. For example, if $(a, f, \{r, w\}) \in Matrix$ then we say a has rights r and w over f.

TAG is the type for meta-data tags. Tags are boolean flags which indicate properties of agents and objects. The finite set *Tag* are the tags recognized by a policy. For an agent a and tag $t \in Tag$, we write $a.t = true$ to indicate that the tag is true (or *set*) for a. We use the function $tags(a)$ to extract the set of tags that are true of an object, $tags(a) = \{t \in Tag | a.t = true\}$.

Purpose a finite hierarchical set of purposes for actions as included in a policy. They include general categories of purposes for actions (*e.g.,* use, disclosure) as well as specific purposes which fall under the general categories (*e.g.,* use for prevention of loss). For compactness of regulatory texts, enumerations of specific purposes for actions are classified under a more general heading and rules are given with respect to the general headings. To support this, we require that agents include a purpose set P whenever running a command.

Since they are normally hierarchical, we implemented a standard partial ordering on purposes based on their specificity and legal definition. We define the partial order in terms of parent and child relationships where parents are general terms which encompass their children. Thus, a purpose may have a maximum of one parent, but many children. The relation parent: $Purpose \rightarrow Purpose$ yields the parent element for a given $p \in Purpose$. If p has no parents (*i.e.,* it is a *root*), $\mathsf{parent}(p) = \emptyset$. The relation $\mathsf{children}$: $Purpose \rightarrow \mathsf{pwr}(Purpose)$ yields the set of direct descendants of p. If p has no children (*i.e.,* it is a leaf), $\mathsf{children}(p) = \emptyset$. Let $\mathsf{ancestors}$ be the transitive closure of parent and let $\mathsf{descendants}$ be the transitive closure of $\mathsf{children}$. The intuition of the hierarchy is that for a purpose p, its ancestors are more general and its descendants are more specific.

Agents provide a set of purposes $P \subset Purpose$ as a parameter for each command (*i.e.,* action or combination of actions) to run. Commands and constraints use P for the evaluation of guards to do with purpose by evaluating set membership guards in one of two mechanisms. Under *allowed semantics*, the guard is interested in evaluating whether P contains a particular purpose or any of its descendants, denoted $p \text{ in}_a P$. It is true if $\exists p' \in P \,.\, p' \in \mathsf{descendants}(p)$. Allowed semantics are used in cases such as when an action is permitted if it is for purpose p (*e.g.,* You may perform the action if it is for p). Thus, p or any of its descendants being included in P is sufficient to satisfy the guard. Under *forbidden semantics*, the guard is interested in whether the P contains a particular purpose p or any of its descendants or ancestors, denoted $p \text{ in}_f P$. It is true if $\exists p' \in P \,.\, p' \in \{\mathsf{ancestors}(p) \cup \mathsf{descendants}(p)\}$. Forbidden semantics are used in cases such as when an action is required if p is included (*e.g.,* You must perform the action if it is for p). Thus, p or any of its descendants or ancestors being included in P is sufficient to satisfy the guard. Note that if p has no ancestors (*i.e.,* it is a root), $p \text{ in}_a$ P is true iff $p \text{ in}_f$ P since $\mathsf{ancestors}(p) = \emptyset$. Other applications of "permit down" (allowed semantics) and "forbid up and down" (forbidden semantics) are found in privacy policy literature (*e.g.,* EPAL [9]).

Log An append-only log. It is stored as a series of strings and can be updated by commands, but it can not be inspected by guards. The log includes messages sent to agents. Storage of a message for $a \in \mathit{Agent}$ in *Log* is the equivalent of sending the message to a in the real world.

State As described above in Section 4.4.1, the knowledge state represents the state at a given moment in time. The set $\mathit{State} \subseteq \mathit{Agent} \times \mathit{Object} \times \mathit{Matrix} \times \mathit{Log}$. An individual s has members $s = (A, O, m, l)$ for $A \subseteq \mathit{Agent}, O \subseteq \mathit{Object}, m \in \mathit{Matrix}, l \in \mathit{Log}$.

Parameters We define a tuple type $\textsc{Parameters} = (a : \textsc{Agent}, s : \textsc{Agent}, r : \textsc{Agent}, P : \mathit{Purpose}^*, f : \textsc{Object}, f' : \textsc{Object}, \mathit{msg} : \textsc{String})$. For the rest of this work we do not include the type annotations for parameters, instead using a short hand notation for a member of $\textsc{Parameters}$ as $(a, s, r, P, f, f', \mathit{msg})$. Tuples such that $a, s, r \in \mathit{Agent}$, $P \subseteq \mathit{Purpose}$, $f \in \mathit{Object}$, $f' \notin \mathit{Object}$ if not null, and msg a string are members of the valid parameters set $\mathit{ParametersE}$. We use parameter lists to allow agents to pass information to the evaluation engine to aid in processing commands. We define a parallel tuple type $\textsc{ParametersE} = (a : \textsc{Agent}, s : \textsc{Agent}, r : \textsc{Agent}, P : \textsc{Purpose}^*, f : \textsc{Object}, f' : \textsc{Object}, msg : \textsc{String}, e : \textsc{Command})$ which are written without type annotations as $(a, s, r, P, f, f', msg, e)$. The members of $\textsc{ParametersE}$ such that $a \in \mathit{Agent}$, $s \in \mathit{Agent}$, $r \in \mathit{Agent}$, $P \subseteq \mathit{Purpose}$, $f \in \mathit{Object}$, $f' \notin \mathit{Object}$ if not null, $msg \in \mathit{String}$, and $e \in \mathit{Command}$ are members of the valid constraint parameter set $\mathit{ParametersC}$. They are used to pass information to constraints. We use the following names for the parameters throughout this work: *actor* for a, *subject* for s, *recipient* for r, *purpose set* for P, *object* or *file* for f, *new object* or *new file* for f', *message* for msg, and *current command* for e.

Judgments As noted above in Section 4.4.3 and informally described in Table 4.1, we summarize the policy decision of constraints using *judgments*. The judgment set *Judgment* contains the following members: *Judgment* = {Allow, Forbid, Ignore, Don't Care/Allow, Don't Care/Forbid}. We discuss the usage and derivation of the judgments below in Section 5.2.4. As a notational convention, we capitalize the names of the judgments when referring to them as the results of some judgment derivation algorithm. We refer to the judgments Ignore, Don't Care/Allow, and Don't Care/Forbid as a class of *don't care* results, italicizing the name. We use *don't care* to refer to all members of the class. We write Don't Care/* when we refer to Don't Care/Allow and Don't Care/Forbid equally. Since Ignore is given a semantics similar to Allow in some situations, we often write it as "Ignore (Allow)".

5.1.1 Purpose Examples

In order to give an intuition for the use of the partial order of *Purpose* let us consider two examples, one contrived to illustrate the differences between the usage of in_a and in_f and the other from HIPAA.

Example 5.1.1 (Surgery Purposes)

In order to illustrate the use of the *Purpose* partial order and allowed and forbidden semantics, let us consider an example of four purposes: Treatment is the parent of Surgery, Surgery is the parent of Oral Surgery and Eye Surgery. The hierarchy is shown in Figure 5.1. We include index numbers in parentheses next to the names of purposes for brevity in discussing the hierarchy.

$$\begin{tikzcd}
& \text{Treatment (0)} \arrow[d] & \\
& \text{Surgery (1)} \arrow[dl] \arrow[dr] & \\
\text{Oral Surgery (2)} & & \text{Eye Surgery (3)}
\end{tikzcd}$$

Figure 5.1: Hierarchy for surgery example

For allowed semantics, we are interested in the descendants of a purpose while for forbidden semantics we are interested in the descendants and ancestors. Table 5.1 enumerates the members of the sets under both semantics. For compactness, we use the numbering in Figure 5.1 to refer to purposes.

Table 5.1: Purposes included in allowed and forbidden semantics for surgery example

Purpose	in$_a$	in$_f$
Treatment (0)	{0, 1, 2, 3}	{0, 1, 2, 3}
Surgery (1)	{1, 2, 3}	{0, 1, 2, 3}
Oral Surgery (2)	{2}	{0, 1, 2}
Eye Surgery (3)	{3}	{0, 1, 3}

The following example guards are then evaluated as follows:

You may do A if it is for Treatment This guard uses allowed semantics and is equivalent to Treatment in$_a$ P. It is true if Treatment, Surgery, Oral Surgery, or Eye Surgery are in P.

You may not do B for Oral Surgery This guard uses forbidden semantics and is equivalent to Oral Surgery in$_f$ P. It is true if Treatment, Surgery, or Oral Surgery are in P.

□

Example 5.1.2 (Marketing Purposes)

In regulatory documents, purposes are often defined in a hierarchical format. Before being used in the regulatory text, general purpose terms are defined precisely. For example, consider the definition of "marketing" in HIPAA [§164.501, v. 2003]:

> Marketing means:
>
> (1) To make a communication about a product or service that encourages recipients of the communication to purchase or use the product or service, unless the communication is made ...
>
> (2) An arrangement between a covered entity and any other entity whereby the covered entity discloses protected health information to the other entity, in exchange for direct or indirect remuneration, for the other entity or its affiliate to make a communication about its own product or service that encourages recipients of the communication to purchase or use that product or service.

Marketing therefore includes two separate actions - (1) communication about a product or service to recipients and (2) directly selling information to another entity who will then make a communication to recipients. For clauses that then discuss marketing, the purpose for the action is examined using the permitted or forbidden semantics as described in the clause. Let us consider one usage of Marketing as a purpose from [§164.508(a)(3)(i), v.2003]:

> (i) Notwithstanding any provision of this subpart, other than the transition provisions in §164.532, a covered entity must obtain an authorization for any use or disclosure of protected health information for marketing, except if the communication is in the form of:
>
> (A) A face-to-face communication made by a covered entity to an individual; or
>
> (B) A promotional gift of nominal value provided by the covered entity.

Combining the purpose definitions, we have a hierarchy as shown in Figure 5.2.

$$\begin{tikzcd}
& \text{Marketing} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Promotional gift} & \text{Communication to recipient} \arrow[d] & \text{Sell to another who will communicate} \\
& \text{Face-to-face} &
\end{tikzcd}$$

Figure 5.2: Hierarchy for marketing purpose example

Using P as the purpose set provided by the agent, the guard in §164.508(a)(3)(i) requires an authorization for a use or disclosure of protected health information if (Marketing $\text{in}_f P \land !(\text{Face-to-face in}_a P \lor \text{Promotional gift in}_a P)$). □

5.2 Syntax for Privacy Commands

Commands and constraints operate over the fundamental sets by using *guards* to inspect the knowledge state and parameters and *operations* to modify the knowledge state. We first discuss the syntax of the guards and operations since they are building blocks for commands and constraints. We then discuss the syntax for commands and constraints followed by their semantics and interaction model. The following sections build on the overview and informal development of Privacy Commands in Section 4.4. We begin with a formal discussion of the structure and syntax of commands in Section 5.2.3 and constraints in Section 5.2.4 and follow with a grammar in Section 5.2.5. We resent several illustrative examples in Section 5.2.6.

5.2.1 Guards and Operations

Commands are combinations of guards (ψ) and operations (ω) that inspect and modify the knowledge state while constraints consist solely of guards and so do not modify the knowledge state. Since guards and operations are the fundamental atoms for both commands and constraints we first list them and informally describe their purposes. We postpone the discussion of their operational semantics to Section 5.3, after we have discussed the syntax and grammar for commands and constraints.

As shown in Table 5.2, guards inspect the knowledge state and parameters to extract their properties. Guards do not cause state updates and always yield a boolean result. The first guard d **in** (a, o) checks the existence of rights in the matrix. The second guard $o.t = b$ checks whether a tag t has the boolean value b on an object o. The third guard k **in** $Roles(a)$ checks whether a holds the role k. The fourth guard $p\ \mathbf{in}_a\ P$ checks set membership of p in P using allowed semantics. The fifth guard $p\ \mathbf{in}_f\ P$ checks set membership for p in P using forbidden semantics. The sixth guard $e(args) \in J$ instructs the evaluation engine

Table 5.2: Guards (ψ)

d **in** (a, o)	Checks for the presence of a right.
$o.t = b$	Checks the boolean value of a tag
k **in** $\mathit{Roles}(a)$	Checks an agent's holding a role
p $\mathbf{in}_a$ P	Checks if a purpose is *allowed* by a purpose set
p $\mathbf{in}_f$ P	Checks if a purpose is *forbidden* by a purpose set
$c(a, s, r, P, f, f', msg) \in J$	Runs a *constraint* in a command
$a_1 = a_2$	Compares agent identity
$!g$	Negation of guard g

Table 5.3: Operations (ω)

create object o	Create a fresh object
delete object o	Delete an object
set $o.t = b'$	Set a tag
insert d **in** (a, o)	Insert a right
delete d **from** (a, o)	Delete a right
insert s **in log**	Insert a note in the log
inform a **of** msg	Send a message to an agent
invoke $e(a, s, r, P, f, f', msg)$	Execute a command and waits for its completion
return b	End a command and returns a boolean value

to run the constraint e with the arguments $args$. The resulting judgment as derived by the evaluation engine is then compared against the set of judgments $J \subseteq \mathit{Judgment}$. If the resulting judgment is in J, the guard's result is true. Otherwise, its result is false. The seventh guard $a_1 = a_2$ checks the equality of two of the agent parameters (*i.e.,* a, s, r). If a_1's value coincides with a_2's value, the result is true. Otherwise it is false. The last guard $!g$ is the logical negation of another guard $!g$. We forbid negation of negation to prevent guards of unbounded length. We formalize this restriction in the grammar in Table 5.6.

As shown in Table 5.3, operations cause updates to the knowledge state. Operations include implicit checks of the knowledge state on which they operate which if not satisfied, result in a type error. Non-well typed operations are not well formed and so commands which contain them have undefined behavior at run time. We discuss typing for operations and guards and provide well-formedness definitions for them in Section 5.2.2. The first operation **create object** o creates a fresh object in *Object* with the fresh name o. The second operation **delete object** o removes o from *Object*. The third operation **set** $o.t = b$ sets the tag t for o to the boolean value b. The fourth operation **insert** d **in** (a, o) and fifth operation **delete** d **from** (a, o) update the rights matrix by inserting or deleting d for a on o. The sixth operation **insert** s **in log** inserts a string s in the knowledge state's log. The seventh operation **inform** a **of** *msg* sends a message to a by noting it in the log. If $a \notin Agent$, the logged message will indicate a recipient which does not exist. The eighth operation **invoke** $e(args)$ runs the command e with the parameters $args$. When the evaluation engine reaches an invoke operation it suspends running the current command to run e. Upon completion of running e, the evaluation engine resumes running the current command. The final operation **return** b is special in that it is the only operation that is included in constraints as well as commands. It indicates to the evaluation engine that the command or constraint is finished running. The value of b is the *return value* and the evaluation engine provides it to any command or constraint which had run the currently running command or constraint.

Let us consider the subset of operations which only perform logging operations, denoted $Operation_d$. The members are $Operation_d = \{$**inform** a of *msg*, **insert** s **in log**, **return** $b\}$. We use the set $Operation_d$ below for situations in commands where only logging operations are appropriate.

5.2.2 Typing

As defined in Section 5.1, we define several fundamental sets that guards and operations use. We use the sets as the basis for a typing system for guards and operations which lets us define well typed guards and operations. Based on the fundamental sets we define the sets shown in Table 5.4

The types shown in Table 5.4 include simple types such as Purpose, Right, Judgment, String, and Bool as well as composite types. Objects are represented as functions from a tag to a boolean since guards may use them in only two ways: (1) testing for existence and (2) testing or setting the value of a tag on the object. Agents are an extension of objects in that they can be tested for whether they have a role set. We assume

Table 5.4: Types

Type	Set	Description
$\textsc{Agent}$	*Agent*	$(\textsc{Tag} \cup \textsc{Role}) \rightarrow \textsc{Bool}$
$\textsc{Object}$	*Object*	$\textsc{Tag} \rightarrow \textsc{Bool}$
$\textsc{Role}$	*Role*	Atomic
$\textsc{Purpose}$	*Purpose*	Atomic
$\textsc{Right}$	*Right*	Atomic
$\textsc{Tag}$	*Tag*	Atomic
$\textsc{Judgment}$	*Judgment*	Atomic
$\textsc{String}$	*String*	Atomic
$\textsc{Bool}$	*Boolean*	Atomic
$\textsc{Log}$	*Log*	$\textsc{String}$
$\textsc{State}$	*State*	$\textsc{Agent}^* \times \textsc{Object}^* \times \textsc{Matrix} \times \textsc{Log}$
$\textsc{Guard}$	*Guard*	$\textsc{State} \rightarrow \textsc{Judgment}$
$\textsc{Operation}$	*Operation*	$\textsc{State} \rightarrow \textsc{State}$
$\textsc{Command}$	*Command*	$\textsc{Agent} \times \textsc{Agent} \times \textsc{Agent} \times \textsc{Purpose}^* \times \textsc{Object}$ $\times \textsc{Object} \times \textsc{String} \times \textsc{State} \rightarrow \textsc{State}, \textsc{Bool}$
$\textsc{Constraint}$	*Constraint*	$\textsc{Agent} \times \textsc{Agent} \times \textsc{Agent} \times \textsc{Purpose} \times \textsc{Object}$ $\times \textsc{Object} \times \textsc{String} \times \textsc{Command} \times \textsc{State} \rightarrow \textsc{Judgment}$
$\textsc{Policy}$	*Policy*	$\textsc{Constraint}^* \times \textsc{Command}^* \times \textsc{Role}^* \times \textsc{Tag}^* \times$ $\textsc{Purpose}^*$

that all members of the sets in Table 5.4 are well typed, but that the sets may not contain all members of the type. So, for instance all members of *Agent* are of type $\textsc{Agent}$, however there may be objects of type $\textsc{Agent}$ not present in *Agent*. We use these assumptions to make assertions about the variables used in guards and operations which we discuss next.

Guards and operations use variables of particular types. We enforce typing for the variables for each guard and operation to ensure that they are well formed. We use the colon notation : to annotate the types of variables. For instance $a : \textsc{Agent}$ indicates that a is of type $\textsc{Agent}$. Figures 5.3 and 5.4 show the typing derivation trees for all variables used in operations and guards respectively. Figure 5.3 does not include a rule for the **return** operation since it must be placed only at the end of a command branch. We therefore provide its typing in Figure 5.5 and Figure 5.6 as appropriate.

Using the typing from the tables we define well formed operations and guards as follows.

$$\frac{o : \textsc{Object}}{\textbf{create object } o : \textsc{Operation}} \text{ [T-Create]}$$

$$\frac{o : \textsc{Object}}{\textbf{delete object } o : \textsc{Operation}} \text{ [T-DeleteO]}$$

$$\frac{o : \textsc{Object} \quad t : \textsc{Tag} \quad b : \textsc{Bool}}{\textbf{set } o.t = b : \textsc{Operation}} \text{ [T-TagSet]}$$

$$\frac{d : \textsc{Right} \quad a : \textsc{Agent} \quad o : \textsc{Object}}{\textbf{insert } d \textbf{ in } (a, o) : \textsc{Operation}} \text{ [T-InsertR]}$$

$$\frac{d : \textsc{Right} \quad a : \textsc{Agent} \quad o : \textsc{Object}}{\textbf{delete } d \textbf{ from } (a, o) : \textsc{Operation}} \text{ [T-DeleteR]}$$

$$\frac{a : \textsc{Agent} \quad msg : \textsc{String}}{\textbf{inform } a \textbf{ of } msg : \textsc{Operation}} \text{ [T-Inform]}$$

$$\frac{\begin{array}{c} e : \textsc{Command} \quad a, s, r : \textsc{Agent} \quad P : \textsc{Purpose}^* \\ f, f' : \textsc{Object} \quad msg : \textsc{String} \end{array}}{\textbf{invoke } e(a, s, r, P, f, f', msg) : \textsc{Operation}} \text{ [T-Invoke]}$$

Figure 5.3: Typing rules for operations

Definition 5.2.1 (Well Formed Operation) An operation ω is well formed if its structure can be inferred from the typing rules in Figures 5.3 and 5.4. $\square$

Definition 5.2.2 (Well Formed Guard) A guard ψ is well formed if its structure can be inferred from the typing rules in Figures 5.3 and 5.4. $\square$

For the rest of this work we now presume that all operations and guards are well formed. As such, evaluation for non-well formed guards and operations is undefined.

A Privacy API ϕ is a collection of commands and constraints along with the purposes, tags, and roles which support them. We denote the set of Privacy APIs as *Policy*. Let us use a shorthand notation for : to apply to all members of a set such that V :V is a shorthand for $\forall v \in V, v$:V. Let C : $\textsc{Constraint}, E : \textsc{Command}, R : \textsc{Role}, T : \textsc{Tag}, P : \textsc{Purpose}$. A Privacy API is then a tuple $\phi = (C, E, R, T, P)$. Intuitively, in order to ensure that a Privacy API is well formed, we check that the roles, tags, and purposes used in E and C

$$\frac{d : \textsc{Right} \quad a : \textsc{Agent} \quad o : \textsc{Object}}{d \ \mathbf{in} \ (a, o) : \textsc{Guard}} \ [\text{T-}\textsc{CheckRight}]$$

$$\frac{o : \textsc{Object} \quad t : \textsc{Tag} \quad b : \textsc{Bool}}{o.t = b : \textsc{Guard}} \ [\text{T-}\textsc{CheckTag}]$$

$$\frac{k : \textsc{Role} \quad a : \textsc{Agent}}{k \ \mathbf{in} \ \mathit{Roles}(a) : \textsc{Guard}} \ [\text{T-}\textsc{CheckRole}]$$

$$\frac{p : \textsc{Purpose} \quad P : \textsc{Purpose}^*}{p \ \mathbf{in}_a \ P : \textsc{Guard}} \ [\text{T-}\textsc{PurposeA}]$$

$$\frac{p : \textsc{Purpose} \quad P : \textsc{Purpose}^*}{p \ \mathbf{in}_f \ P : \textsc{Guard}} \ [\text{T-}\textsc{PurposeF}]$$

$$\frac{\begin{array}{c} c : \textsc{Constraint} \quad a, s, r : \textsc{Agent} \quad P : \textsc{Purpose}^* \quad f', f : \textsc{Object} \\ \mathit{msg} : \textsc{String} \quad J : \textsc{Judgment}^* \quad e : \textsc{Command} \end{array}}{c(a, s, r, P, f, f', \mathit{msg}, e) \in J : \textsc{Guard}} \ [\text{T-}\textsc{Reference}]$$

$$\frac{a_1, a_2 : \textsc{Agent}}{a_1 = a_2 : \textsc{Guard}} \ [\text{T-}\textsc{Agent}] \qquad \frac{g : \textsc{Guard}}{!g : \textsc{Guard}} \ [\text{T-}\textsc{Neg}]$$

Figure 5.4: Typing rules for guards

are contained in R, T, and P and that the references in C and E are only to commands in E and constraints in C.

To concretize well formed-ness let us first define some shorthand notation. Let ω : Operation, ψ : Guard, e : Command, c : Constraint, and $\phi = (C, E, R, T, P) \in$ *Policy*. To avoid confusion we use the triple equal sign $\equiv$ to indicate structural equivalence to differentiate from $=$ which we use in the syntax of guards.

Definition 5.2.3 (Roles of)

- $\mathsf{roles}(\psi) = \begin{cases} \psi \equiv k \ \mathbf{in} \ \mathit{Roles}(a), & k; \\ \text{otherwise}, & \emptyset. \end{cases}$

- $\mathsf{roles}(e) = \displaystyle\bigcup_{\psi \in \overline{\psi}} \mathsf{roles}(\psi)$

- $\mathsf{roles}(c) = \displaystyle\bigcup_{\psi \in (\overline{\psi^{st}} \cup \overline{\psi^{r}})} \mathsf{roles}(\psi).$

- $\mathsf{roles}(\phi) = \bigcup_{c \in C} \mathsf{roles}(c) \cup \bigcup_{e \in E} \mathsf{roles}(e)$.

$\square$

Definition 5.2.4 (Tags of)

- $\mathsf{tags}(\omega) = \begin{cases} \omega \equiv \textbf{set } o.t = s, & t; \\ \text{otherwise}, & \emptyset. \end{cases}$

- $\mathsf{tags}(\psi) = \begin{cases} \psi \equiv o.t = b, & t; \\ \text{otherwise}, & \emptyset. \end{cases}$

- $\mathsf{tags}(e) = \bigcup_{\psi \in \overline{\psi}} \mathsf{tags}(\psi) \cup \bigcup_{\omega \in \overline{\omega^t}} \mathsf{tags}(\omega)$.

- $\mathsf{tags}(c) = \bigcup_{\psi \in (\overline{\psi^{st}} \cup \overline{\psi^r})} \mathsf{tags}(\psi)$.

- $\mathsf{tags}(\phi) = \bigcup_{c \in C} \mathsf{tags}(c) \cup \bigcup_{e \in E} \mathsf{tags}(e)$.

$\square$

Definition 5.2.5 (Purposes of)

- $\mathsf{purposes}(\psi) = \begin{cases} \psi \equiv p \textbf{ in}_a\ P, & p; \\ \psi \equiv p \textbf{ in}_f\ P, & p; \\ \text{otherwise}, & \emptyset. \end{cases}$

- $\mathsf{purposes}(e) = \bigcup_{\psi \in \overline{\psi}} \mathsf{purposes}(\psi)$

- $\mathsf{purposes}(c) = \bigcup_{\psi \in (\overline{\psi^{st}} \cup \overline{\psi^r})} \mathsf{purposes}(\psi)$.

- $\mathsf{purposes}(\phi) = \bigcup_{c \in C} \mathsf{purposes}(c) \cup \bigcup_{e \in E} \mathsf{purposes}(e)$.

$\square$

Definition 5.2.6 (References of)

- $\mathsf{references}(\omega) = \begin{cases} \omega \equiv \textbf{invoke } e(a, s, r, P, f, f', msg), & e; \\ \text{otherwise}, & \emptyset. \end{cases}$

- $\mathsf{references}(\psi) = \begin{cases} \psi \equiv c(a,s,r,P,f,f',msg), & c; \\ \text{otherwise}, & \emptyset. \end{cases}$

- $\mathsf{references}(e) = \bigcup_{\psi \in \overline{\psi}} \mathsf{references}(\psi) \cup \bigcup_{\omega \in \overline{\omega^t}} \mathsf{references}(\omega)$.

- $\mathsf{references}(c) = E \cup \bigcup_{\psi \in (\overline{\psi^{st}} \cup \overline{\psi^r})} \mathsf{references}(\psi)$.

- $\mathsf{references}(\phi) = \bigcup_{c \in C} \mathsf{references}(c) \cup \bigcup_{e \in E} \mathsf{references}(e)$.

□

Using the above shorthand, we define well formed *Privacy API* as follows:

Definition 5.2.7 (Privacy API) $\phi = (C, E, R, T, P)$ is a well formed *Privacy API* iff $\mathsf{roles}(\phi) \subseteq R$, $\mathsf{tags}(\phi) \subseteq T$, $\mathsf{purposes}(\phi) \subseteq P$, and $\mathsf{references}(\phi) \subseteq (C \cup E)$. □

5.2.3 Command Syntax

Legal privacy documents describe the types of actions, behaviors, and protections related the handling of protected information. We use *commands* to perform the actions permitted in policies, becoming the verbs which cause the knowledge state to evolve. Intuitively, a command is a button which can be pushed by an agent to perform an action. That is, when a command is *run*, there is an agent who actively intends to perform the updates contained in the command and provides information about its intent in the form of parameters.

For the following definition, let $a, s, r \in \mathit{Agent}$, $f \in \mathit{Object}$, f' a fresh name not in Object, $P \subset \mathit{Purpose}$, and msg a string. We say that e is the name of the command, a is the actor in the request, r the information recipient, s the information subject, P the purpose set for the command, f the object of the command, f' a fresh name for creating new objects, and msg a message string to be sent or logged. Let $\psi \in \mathit{Guard}$ be a well formed guard and $\omega \in \mathit{Operation}$ be a well formed operation as defined above in Section 5.2.1.

Definition 5.2.8 (Command) A command e has the following structure:

$$e(a,s,r,P,f,f',msg) = \textbf{if } \overline{\psi} \textbf{ then } (\overline{\omega^t}) \textbf{ and return true else } (\overline{\omega^f}) \textbf{ and return false}$$

where $\overline{\psi} = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m$ for $m \geq 0$, $\overline{\omega^t} = \omega_1^t \wedge \omega_2^t \wedge \ldots \wedge \omega_n^t$ for $n \geq 0$, $\overline{\omega^f} = \omega_1^f \wedge \omega_2^f \wedge \ldots \wedge \omega_k^f$ for $k \geq 0$, $\overline{\psi} \subset \mathsf{pwr}(\psi)$, $\overline{\omega^t} \subset \mathsf{pwr}(\mathit{Operation})$, and $\overline{\omega^f} \subset \mathsf{pwr}(\mathit{Operation}_d)$. When $\overline{\psi} = \emptyset$, the true branch of the command is always executed, so $\overline{\omega^f}$ may be ignored.

The case where $m = k = n = 0$ is the trivial *nil command.* A command e is a *valid command* if it has the above structure and $\nexists \omega \in \overline{\omega^t} \,.\, \omega \equiv \textbf{invoke}\ e(a, s, r, P, f, f', msg)$ (*i.e.,* no self-invocation). We forbid self invocation to avoid non-terminating commands. When building policies with more than one command we may still have non-termination issues due to circular references, an issue we address in Section 5.4.5.

A similar, less compact representation for commands which we use for representation of commands in examples throughout this work is:

CMD	$e(a, s, r, P, f, f', msg)$
if	ψ_1
and	$\ldots$
then	ω_1^t
and	$\ldots$
and	**return true**
else	ω_1^f
and	$\ldots$
and	**return false**

□

Intuitively, a command e accepts parameters $e(a, s, r, P, f, f', msg)$ and modifies the knowledge state, producing side effects. If the boolean AND of all guards in $\overline{\psi}$ is true, we say that the command *returns true.* Otherwise, we say that the command *returns false.* For convenience we indicate the boolean return value in the command with the syntax **return true** and **return false**.

Since commands are comprised of operations and guards, we define well formedness for them in terms of guards and operations. We show the typing derivation rule for commands in Figure 5.5 and use it for the following definition. We force that return operations only appear at the end of command branches using the type CmdBranch. Branches may be empty of other operations (T-CmdBnch2) or have a sequence of one or more operations

(T-CmdBnch1).

$$\frac{\overline{\omega} : \textsc{OpSeq} \quad b : \textsc{Bool}}{\overline{\omega}; \mathbf{return}\ b : \textsc{CmdBranch}}\ [\text{T-CmdBnch1}] \qquad \frac{b : \textsc{Bool}}{\mathbf{return}\ b : \textsc{CmdBranch}}\ [\text{T-CmdBnch2}]$$

$$\frac{\omega : \textsc{Operation} \quad \overline{\omega} : \textsc{OpSeq}}{\omega; \overline{\omega} : \textsc{OpSeq}}\ [\text{T-OpSeq1}] \qquad \frac{\omega : \textsc{Operation}}{\omega : \textsc{OpSeq}}\ [\text{T-OpSeq2}]$$

$$\frac{\psi \in \overline{\psi} \quad \psi : \textsc{Guard} \quad \overline{\omega^t}, \overline{\omega^f} : \textsc{CmdBranch}}{\mathbf{CMD}\ e(a, s, r, P, f, f', msg)\ \mathbf{If}\ \overline{\psi}\ \mathbf{Then}\ \overline{\omega^t}\ \mathbf{Else}\ \overline{\omega^f} : \textsc{Command}}\ \text{T-Cmd}$$

Figure 5.5: Typing rule for commands

Definition 5.2.9 (Well Formed Command) A command e is well formed if its structure matches the typing rule T-Cmd in Figure 5.5. □

For the rest of this work we now presume that all commands are well formed. As such, evaluation for non-well formed commands is undefined.

5.2.4 Constraint Syntax

The permissions offered by legal documents are constrained by requirements for when and how they may be exercised. Such constraints are a form of obligations, static external limitations imposed on actions, as opposed to classical obligations which impose or restrict future actions until fulfilled. An important aspect of regulatory constraints is their scope of applicability. Rules may explicitly invoke or refer to a set of constraints or a hierarchical structure (*e.g.,* a top level paragraph imposing constraints on all rules in sub-paragraphs). Alternatively, constraints may be formulated such that the constraining rule defines its scope of applicability (*e.g.,* must be enforced for all rules in a section) while the referenced rules contain no explicit reference to the constraint. In all cases, constraints must explicitly specify their *scope*, the commands that they apply to.

Constraints implement these requirements by including a set of commands which is the constraint's *scope* as well as two sets of guards to determine applicability. The first set of guards determine the *applicability* of the constraint and are denoted *such that* guards. The second set of guards, denoted "regular guards", determine the judgment of the constraint with respect to allowing or forbidding the intended action. The different contexts when constraints are executed and the interpretations of the judgments in those contexts are noted in Tables 5.10 and 5.8 in our discussion of the evaluation engine in Section 5.4.

Similar to our definition of commands, we formally define constraints as follows.

Definition 5.2.10 (Constraint) A constraint c has the following structure:

$$c(a,s,r,P,f,f',msg,e) = \textbf{Scope: } E \textbf{ Such That } \overline{\psi^{st}} \textbf{ if } \overline{\psi^{r}} \textbf{ then return true else return false}$$

where $E = \{e_1, \ldots\}$ for all $e_i \in \mathit{Command}$, $\overline{\psi^{st}} = \psi^{st}_1 \wedge \psi^{st}_2 \wedge \ldots \wedge \psi^{st}_m$ for $m \geq 0$, $\overline{\psi^{r}} = \psi^{r}_1 \wedge \psi^{r}_2 \wedge \ldots \wedge \psi^{r}_n$ for $n \geq 0$. When $\overline{\psi^{r}} = \emptyset$, the true branch of the constraint is always executed, so the return value for the regular guards is always true.

The case where $m = n = 0$ is the trivial *nil constraint* which is always true. A constraint c_1 is a *valid constraint* if it has the above structure and $\nexists \psi \in \{\overline{\psi^{st}} \cup \overline{\psi^{r}}\}$. $\psi \equiv c_1(a,s,r,P,f,f',msg,e) \in J$ (*i.e.*, no self referencing).

A similar, less compact representation which we use for representation of constraints in examples throughout this work is:

CST	$c(a,s,r,P,f,f',msg,e)$
Scope	$\{e_1, \ldots\}$
Such That	ψ^{st}_1
and	$\ldots$
if	ψ^{r}_1
and	$\ldots$
then	**return true**
else	**return false**

□

Intuitively, a constraint is a function which takes input parameters (a, s, r, P, f, f', msg, e), including a *command* parameter e which indicates which command was responsible for the constraint being run. The constraint checks the command parameter against a possibly empty set of commands in scope to determine whether the command is in scope. It then runs a (possibly empty) set of *such that* guards followed by a (possibly empty) set of regular guards. The return value for the constraint is determined by the regular guards alone, not the *such that* guards. The evaluation engine uses the results from the scope, *such that* guards, and regular guards to determine a judgment as we discuss below in Section 5.4.

The combination of scope, such that guards, and regular guards means that constraints issue three results: whether the command parameter is in scope, whether *such that* guards are satisfied, and whether the regular guards are satisfied. The intuitive use of the three results is as follows.

Scope The scope judgment is relevant only during the search for applicable constraints before the execution of a command. During the search for applicable constraints, if a constraint includes the proposed command in its scope set then it is executed and its outcome is noted. When a constraint is explicitly invoked by a command or another constraint, the scope judgment is irrelevant and so is set to null. Since the constraint has been asked for a judgment, its outcome will be noted.

Such That Guards The such that judgment relates to the particular cases for which a constraint is to be applicable. For example, a constraint relating to the processing of information about prison inmates (as in Example 5.2.2 below) would not be applicable to information about non-inmates and therefore would include a *such that* check to see if the information is about an inmate. If the constraint is queried about a case relating to non-inmates, it would indicate that its *such that* guard is not fulfilled and it is not applicable to the given query, the equivalent of a *don't care* decision.

Regular Guards The rest of the guards in a constraint, those that relate to the examination of the query and the issuing of the allow/forbid ruling, are consulted when the constraint is to be executed. They examine the particulars of the query and determine Allow/Forbid based on them.

Since constraints are comprised of operations and guards, we define well formedness for them in terms of guards and commands. We show the typing derivation rule for constraints in Figure 5.6 and use it for the following definition.

$$\frac{b : \textsc{Bool}}{\textbf{return } b : \textsc{CstBranch}} \;[\text{T-}\textsc{CstBnch}]$$

$$\frac{e' \in E \quad e' : \textsc{Command} \quad \psi \in \overline{\psi^{st}} \cup \overline{\psi^{r}} \quad \psi : \textsc{Guard} \quad b_1, b_2 : \textsc{CstBranch}}{\begin{array}{c}\textbf{CST } c(a, s, r, P, f, f', msg, e) \textbf{ Scope } E \textbf{ Such that } \overline{\psi^{st}} \textbf{ If } \overline{\psi^{r}} \\ \textbf{Then } b_1 \textbf{ Else } b_2 : \textsc{Constraint}\end{array}} \;\text{T-}\textsc{Cst}$$

Figure 5.6: Typing rule for constraints

Definition 5.2.11 (Well Formed Constraint) A constraint c is well formed if its structure matches the typing rule T-Cst shown in Figure 5.6. □

For the rest of this work we now presume that all constraints are well formed. As such, evaluation for non-well formed constraints is undefined.

Overloading As discussed above in Section 4.4.4, we implement a limited form of OR logic for constraints using overloading. Overloaded constraints have identical names and scopes, but different *such that* and regular guards. The evaluation engine treats overloaded constraints differently than regular constraints for judgment derivation as discussed below in Section 5.4. Formally, we require that for such $c_1, c_2, \ldots \in$ *Constraint* such that $c_1, c_2, \ldots$ have the same name, we require that no two constraints have syntactically equivalent *such that* and regular guard sets $(\overline{\psi^{st}_{c_1}} \cap \overline{\psi^{r}_{c_2}}) \cup (\overline{\psi^{r}_{c_1}} \cap \overline{\psi^{r}_{c_2}}) \neq \emptyset$ to avoid duplicates and that $E_{c_1} = E_{c_2} = \ldots$.

5.2.5 Grammar for Privacy Commands

We now formally describe the privacy commands language using a BNF grammar. The simple types for the language are shown in Table 5.5. The sets in Table 5.5 are as described above in Section 5.1. The last two sets, *Command* and *Constraint* are the collection of commands and constraints in the Privacy API. We have defined the syntax for commands in Section 5.2.3 and constraints in Section 5.2.4.

In Section 5.2.1 we listed, described, and informally defined operations and guards. Guards as listed in Table 5.2 inspect the knowledge state and parameter values passed to commands or constraints and assume boolean result values. The knowledge state updates

Table 5.5: Simple and complex types

Agent	$a \in Agent$
Object	$o, f \in Object$
Tag	$t \in Tag$
Rights	$d \in Right$
Matrix	$m \in Matrix$
Role	$k \in Role$
Log	$l \in Log$
Purpose	$p \in Purpose, P \subseteq Purpose$
String	$s \in String$
Boolean	$b \in Boolean$
Judgment	$j \in Judgment, J \subseteq Judgment$
Command	$e \in Command, E \subseteq Command$
Constraint	$c \in Constraint, C \subseteq Constraint$

via the operations listed in Table 5.3. The reader is directed to Section 5.2.1 for specifics on each operation and guard.

The top level terms for the Privacy Commands language are in Table 5.6. Commands and constraints are built using guards and operations which are listed as the first two sets in the table. Guards include all of the inspectors from Table 5.2. As noted above in Table 5.2, the negation guard can negate any guard other than another negation. This limitation is enforced by separating the negatable terms (NonNeg) from the term which contains the negation guard (Neg). The last guard in the NonNeg term is just a boolean value b which is interpreted at face value. If $b = true$, the guard is always satisfied. If $b = false$, the guard is never satisfied. Operations include all modifiers from Table 5.3 with the exception of the return operation which is reserved for use at the end of operation groups in command and constraints.

The constraints term in the grammar consists of a scope, *such that* guards, and regular guards. The scope is a series of command names as defined in the Scope term. The *such that* and regular guards can accept any guard in the Guard term. The difference between the two sets comes from the differing interpretations given them by the evaluation engine as discussed below in Section 5.4. We conclude with the Commands and Constraints terms

Table 5.6: Terms

NonNeg	::=	$o.t = b$ | d **in** (a, o) | k **in** Roles(a) | p $\textbf{in}_a$ P
		| p $\textbf{in}_f$ P | $c(a_1, a_2, a_3, P, o_1, o_2, s) \in J$ | $a_1 = a_2$ | b
Neg	::=	!(NonNeg)
ψ	::=	NonNeg | Neg
ψ^*	::=	$\bullet$ | ψ | ψ **and** ψ^*
ω	::=	**create object** o | **delete object** o | **set** $o.t = b$
		| **insert** d **in** (a, o) | **delete** d **from** (a, o)
		| **invoke** $e(a_1, a_2, a_3, P, o_1, o_2, s)$
ω^*	::=	$\bullet$ | ω | ω **and** ω^*
cmdbr	::=	ω^* **and return** b
e	::=	**if** ψ^* **then** $cmdbr_t$ **else** $cmdbr_f$
E	::=	$\{e_1, \dots\}$ | $\emptyset$
cstbr	::=	**return** b
c	::=	**Scope** $\{E\}$ **Such That** ψ_1^* **if** ψ_2^* **then** $cstbr_1$ **else** $cstbr_2$
C	::=	$\{c_1, \dots\}$ | $\emptyset$
ϕ	::=	$C \cup E$

which indicate how commands and constraints combine and a Policy term to indicate how whole Privacy APIs are constructed structurally using the grammar.

5.2.6 Privacy Commands Examples

We now present some illustrative examples using the Privacy Commands language to give an intuition for its syntactical use in addition to the previous informal examples above in Section 4.4. The purpose of these examples is to introduce the usage of the operations, guards, and terms introduced in this section. We begin with a simple example of a paragraph and its derived constraints and commands. The example shows the use of constraints and commands, including scope, *such that* guards, regular guards, and references along with several of the guards and operations shown above in this section.

Example 5.2.1 (Own Use)

As a first example of the use of Privacy Commands, we examine and translate the following quote from the US Health Insurance Portability and Accountability Act Privacy Rule [§164.506(c)(1), v.2003]:

> (c) Implementation specifications: Treatment, payment, or health care operations.
>
> (1) A covered entity may use or disclose protected health information for its own treatment, payment, or health care operations.

The quote contains six permitted actions: use and disclose for treatment, payment, and health care operations. We separate the cases allowed by the paragraph (only for a covered entity's own usage) of the paragraph into a constraint which checks the particulars of the case. Below is Permitted506c1, the constraint derived from the paragraph.

```
 1          CST   Permitted506c1 (a, s, r, P, f, f', msg, e)
 2        Scope   {TreatmentUse506c1, PaymentUse506c1,
 3                HealthCareOperationsUse506c1, TreatmentDisclose506c1,
 4                PaymentDisclose506c1, HealthCareOperationsDisclose506c1}
 5    Such That   f.protected-health-information = true
 6          and   coveredEntity in Roles(a)
 7          and   individual in Roles(s)
 8           if   own in_a P
 9         then   return true
10         else   return false
```

Line 1 declares the name of the constraint Permitted506c1 and the parameters that it receives. Lines 2–4 are the scope of the constraint, the commands for which it is an invariant. Only two of the commands in its scope are included in this example for brevity. The rest are similar and are shown in Section C.2.2 beginning on page 387. Lines 5–7 are the *such that* guards for the constraint. They limit the applicability of the constraint to cases where the object is protected health information (line 5), the actor is a covered entity (line 6), and the subject is an individual (line 7). Line 8 is the regular guard for the constraint which checks that the action is for the actor's own use. If the regular guard is true, line 9 returns true (Allow). Otherwise, line 10 returns false (Forbid).

In the above constraint we separate the covered entity role and protected health information checks as *such that* guards since they are conditions on the action which if false may not necessarily imply that the action should be forbidden based on the paragraph. For instance, if the information is unprotected health information, the paragraph is not necessarily forbidding use or disclosure of the information for treatment, payment, or health care operations, rather it is simply not discussing it. The policy may impose a separate set of guidelines for such information which are located elsewhere. We can make a similar argument for the check that the actor is a covered entity and the subject is an individual. If the actor is not a covered entity or the subject is not an individual, the paragraph is simply not applicable. We therefore separate off the three conditions as *such that* guards to make Permitted506c1 offer a *don't care* judgment in such cases which may be overridden by more applicable constraints. This is a simple case where we use *such that* guards to allow deference, but it illustrates the importance of maintaining a separate set of guards and *don't care* judgments.

We create two commands, TreatmentUse506c1 and TreatmentDisclose506c1, parameterized by purpose, as samples. Four other parallel commands for the other purposes are elided.

```
1   CMD   TreatmentUse506c1 (a, s, r, P, f, f', msg)
2    if   Permitted506c1 (a, s, r, P, f, f', msg) ∈ {Allow}
3   and   local in (a, f)
4   and   use $\mathbf{in}_a$ P
5   and   treatment $\mathbf{in}_a$ P
6  then   insert treatment in (a, s)
7   and   return true
8  else   return false
```

TreatmentUse506c1 begins on line 1 with the name of the command and its parameters. On line 2, it includes a reference to Permitted5061 which indicates that the evaluation engine should run Permitted506c1 to derive its judgment. If its judgment is Allow, the guard on line 2 is satisfied. Otherwise it is not. On line 3, there is a guard to check that the actor has the right "local" on the object. We use "local" to indicate that an actor has physical or logical access to an object even if the actor is not necessarily permitted to access the object based on the policy. Lines 4–5 check that the purpose of the action is for use and treatment. We use the allowed semantics for the purpose check since the command is permitting the action if use or treatment or any more specific (*i.e.,* child) purposes of the two are present. It is not permitting the action for more generic (*i.e.,* parent) purposes of use or treatment. If all of the guards are satisfied, line 6 grants the right for treatment to the actor by inserting the right "treatment" in the actor's rights over the subject and line 7 returns true. Otherwise, line 8 returns false.

We grant the right to treatment on the subject rather than on the object because the treatment relationship is between the covered entity and the individual, not with respect to a particular object. Thus, once the treatment right has been granted to the actor on the subject, any other object of protected health information may be accessed by the actor, not just the one identified with f in the command's parameters.

```
1  CMD   TreatmentDisclose506c1 (a, s, r, P, f, f', msg)
2    if  Permitted506c1(a, s, r, P, f, f', msg) $\in$ {Allow}
3   and  coveredEntity in Roles(a)
4   and  local in (a, f)
5   and  treatment $\mathbf{in}_a$ P
6   and  disclose $\mathbf{in}_a$ P
7  then  insert local in (r, f)
8   and  return true
9  else  return false
```

TreatmentDisclose506c1 is similar to TreatmentUse506c1 except that if the guards on lines 2–6 are satisfied, the "local" right is granted to the recipient on the object. Since disclosure is an action with respect to a particular object, we grant the "local" right to the actor on the object identified in the command's parameters f. If further objects are to be disclosed, TreatmentDisclose506c1 or another disclosure command must be run for the other objects. □

The above example illustrates the use and derivation of a constraint and commands from a policy paragraph. It shows some of the close decisions that policy writers must make to create properly working Privacy APIs. Specifically, the decision for when to assign a guard in a constraint as a *such that* or regular guard as well as a careful decision about how rights are granted (*i.e.,* on an object or on an agent) requires care. The terms and structures of Privacy Commands enable policy authors flexibility in designing and implementing policies, however it is still up to the policy author to make careful decisions when implementing a Privacy API.

In order to illustrate some of the power of the judgment structure in Privacy Commands, we next develop a more complex example which exercises *don't care* judgments from constraints.

Example 5.2.2 (Inmates)

HIPAA provides exceptions when protected health information may be used without consent from the individual. One exception is provided for inmates in a correctional institution [§164.512(k)(5), v.2003]:

> (5) Correctional institutions and other law enforcement custodial situations.
>
> (i) Permitted disclosures. A covered entity may disclose to a correctional institution or a law enforcement official having lawful custody of an inmate or other individual protected health information about such inmate or individual, if the correctional institution or such law enforcement official represents that such protected health information is necessary for:
>
> (A) The provision of health care to such individuals;
>
> (B) The health and safety of such individual or other inmates;
>
> (C) The health and safety of the officers or employees of or others at the correctional institution;
>
> (D) The health and safety of such individuals and officers or other persons responsible for the transporting of inmates or their transfer from one institution, facility, or setting to another;
>
> (E) Law enforcement on the premises of the correctional institution; and
>
> (F) The administration and maintenance of the safety, security, and good order of the correctional institution.
>
> (ii) Permitted uses. A covered entity that is a correctional institution may use protected health information of individuals who are inmates for any purpose for which such protected health information may be disclosed.

The first paragraph permits the disclosure of protected health information about an inmate for various reasons, including health care for the individual (*i.e.,* the inmate). The second paragraph permits the use of protected health information for any reason permitted in first paragraph for disclosure. The structure of the constraints is as follows.

The top level constraint for paragraph §164.512(k)(5)(i) permits the disclosure of protected health information with some restrictions as to the recipient and subject to the purposes enumerated in (A)–(F). For brevity in this example we show one instantiation of InmateDisclose512k5i, the one that references InmateHealthCare512k5iA. Since the subparagraphs are combined using logical or, InmateDisclose512k5i would be overloaded with one version of InmateDisclose512k5i for each of the subparagraphs (A)–(F).

```
1           CST   InmateDisclose512k5i (a, s, r, P, f, f', msg, e)
2         Scope   {}
3     Such That   inmate in Roles(s)
4           and   f.protected-health-information = true
5           and   subject in (s, f)
6           and   coveredEntity in Roles(a)
7           and   disclose $\mathbf{in}_a$ P
8           and   correctionalInstitution in Roles(r)
9            if   inLawfulCustodyOf in (r, s)
10          and   r.represents-necessary = true
11          and   InmateHealthCare512k5iA(a, s, r, P, f, f', msg) in {Allow}
12         then   return true
13         else   return false
```

InmateDisclose512k5i begins on line 1 with its name and parameter list. Line 2 declares the scope for the constraint. For simplicity, here it is empty since it does not explicitly claim coverage over any other paragraphs listed here. Commands which implement the disclosure rule would be included in the scope. Lines 3–8 are the *such that* guards for the paragraph. They check that the subject has the role Inmate (line 3), the information is protected health information (line 4), that the subject has the right "subject" on the object (*i.e.,* s's information is contained in f) (line 5), and that the actor has the role Covered Entity (line 6). Line 7 checks that the purpose of the action is disclosure. Line 8 checks that the recipient has the role Correctional Institution. If the *such that* guards are not fulfilled, the constraint is not directly applicable and may result in the judgments Ignore (Allow), Don't Care/Allow, or Don't Care/Forbid as discussed above in Section 5.2.4. The guards are placed as *such that* guards rather than regular guards since the intent of the paragraph is to give a ruling for the disclosure of protected health information about an inmate for use by a correctional institution. If those conditions are not met (*i.e.,* it's not for disclosure, the subject is not an inmate, the object is not protected health information, the object is not about the inmate, or the actor is not a covered entity) then the paragraph is not directly applicable. The regular guards on lines 9–11 check the remainder of the requirements from the top level paragraph: that the recipient has the right inLawfulCustodyOf on the subject (*i.e.,* the subject is in lawful custody of the recipient) (line 9), and that the recipient represents that receiving the information is necessary (line 10). The last guard on line 11 contains a reference to the child constraint InmateHealthCare512k5iA. If its resulting judgment is Allow, the guard is satisfied. If all of the regular guards are satisfied, the return value is true. Otherwise it is false. The evaluation engine uses the results from the *such that* guards and regular guards to derive the judgment.

```
1        CST   InmateHealthCare512k5iA(a, s, r, P, f, f', msg, e)
2      Scope   {}
3  Such That
4         if   provision-of-health-care $\mathbf{in}_a$ P
5        and   s.target-of-provision = true
6       then   return true
7       else   return false
```

The lower level constraint for the sentence §164.512(k)(5)(i)(A) is a permission for the disclosure of protected health information by a covered entity for the use of a correctional institution and is implemented in the constraint InmateHealthCare512(k)(5)(i)(A). It operates a sub-constraint, only checking the for the guards listed in the sentence: that the purpose of the action be for provision of health care (line 4) and that the target of the provision be the inmate (line 5).

```
1          CST   CorrectionalUse512k5ii (a, s, r, P, f, f', msg, e)
2        Scope   {}
3    Such That   correctional-institution in Roles(a)
4          and   inmate in Roles(s)
5          and   f.protected-health-information = true
6          and   use $\mathbf{in}_a$ P
7           if   InmateDisclose512k5i(a, s, r, P, f, f', msg) in {Allow,
8                Don't Care/Allow}
9          and   covered-entity in Roles(a)
10        then   return true
111       else   return false
```

The top level command for the paragraph §164.512(k)(5)(ii) permits correctional institutions which are a covered entities to use protected health information for any reason that would have permitted disclosure. The permission refers to another constraint, so we simplify it by including a reference to the intended paragraph: §164.512(k)(5)(i) on lines 7–8. The guard is satisfied if the resulting judgment is either Allow or Don't Care/Allow. The guard accepts the Don't Care/Allow judgment since the case is not directly applicable since the purpose of the action is use (line 6), not disclosure. Since line 7 of InmateDisclose512k5i will not be satisfied unless the purpose of the action is disclosure, the judgment Don't Care/Allow is likely to be returned. In order to accept such cases, CorrectionalUse512k5ii allows the Don't Care/Allow ruling.

In the above constraints, the second paragraph §164.512(k)(5)(ii) was checking for a permission from paragraph §164.512(k)(5)(i). If the constraint for paragraph (i) did not return Allow, that is if it returned Forbid or Don't Care/Forbid, paragraph (ii) would not permit the intended use. Conversely, if paragraph (ii) had instead been looking for a direct permission, it would not satisfied with a Don't Care/Allow judgment; it would have been the equivalent of a Forbid. This illustrates the flexibility and fine grained control given to commands and constraints in using the various judgments that the evaluation engine derives. It also illustrates the necessity of using the two different types of guards in constraints to better capture the intent of the policy authors. □

Examples 5.2.1 and 5.2.2 illustrate two common idioms for command and constraint interaction: commands referencing parent constraints and constraints referencing children.

In Example 5.2.1, the constraint Permitted506c1 defined the cases where actions in the paragraph could be performed. The guards in Permitted506c1 are applicable to any command which is created from the paragraph and so its scope includes all children commands derived from §164.506. The example is a good illustration of a structure that is repeated in many other contexts: commands reference their parent constraints for applicable restrictions.

Conversely, in Example 5.2.2 the parent constraint InmateDisclose512k5i references its children, including InmateHealthCare512k5iA, by creating overloaded copies of the parent paragraph to reference each of the child constraints. The parent constraint is then used for guards such as "if A is permitted by §164.512(k)(5)(i)." The evaluation engine can then determine whether §164.512(k)(5)(i) permits the action by running the overloaded versions of the top level constraint InmateDisclose512k5i which effectively runs all of its children. If any of the children permit the action (*e.g.,* if §164.512(k)(5)(i)(A) permits A), then the corresponding version of InmateDisclose512k5i will yield an Allow judgment, satisfying the guard. If none of the overloaded versions of InmateDisclose512k5i permit the action, the guard will not be satisfied. The example is a good illustration a common constraint structure: constraints referencing their children. Using the idiom, we enable parent constraints to "summarize" their children by creating overloaded copies of the parent which reference the children.

5.3 Operational Semantics

We developed the syntactical structures for privacy commands above in Section 5.2. In the following section we develop an operational semantics for the Privacy Commands language. We first consider the operational semantics for well formed guard and operations as per Definitions 5.2.2 and 5.2.1 using a small step semantics. We then use them to develop the operational semantics for well formed commands and constraints using a big step semantics.

Let us consider the operational semantics for guards and operations as shown below.

We first show the operational semantics for evaluating guards in Figures 5.7 and 5.8 and show how we evaluate guard sequencing in Figure 5.9. We show the operation semantics for operations in Figures 5.10 and 5.11 and show how we evaluate operation sequencing in Figure 5.12. As above, to avoid confusion we use the triple equal sign $\equiv$ to indicate structural equivalence to differentiate from $=$ which we use in the syntax of guards. Since operations modify the state, we frame their semantics in terms of transitions of a knowledge state $S = (A, O, m, l)$ using small step semantics. Where appropriate, we indicate the updates to the state using the modified versions of the state, $S' = (A', O', m', l')$. We indicate the state in which an operation is evaluated using the $\vdash$ operator.

The sequence indicator **and** is used to create sequences of guards and operations. We summarize sequences of guards and operations using the bar notation, for instance summarizing ψ_1 **and** ψ_2 **and** ψ_3 as $\overline{\psi}$. The evaluation of guards occurs in order, but since their evaluation does not change state, their order is not semantically significant. Operations are also evaluated in order, but since they update state, their order is semantically significant. We include the rules E-Ref1 and E-Ref2 which create a layer of abstraction in the evaluation of constraints, enabling overloaded and non-overloaded constraints to be referenced identically, letting the evaluation engine determine the results. We present the rules for the evaluation of overloaded constraints in Figure 5.14. The invoke rule E-Invoke relies on the evaluation of commands as shown in Figure 5.15 shown below. The precondition $(A, O, m, l) \vdash e(a, s, r, P, f, f', msg) \rightarrow (A', O', m', l')$ relies on the evaluation of e as per the rules shown there.

Using the semantics for guards and operations we define the operational semantics for constraints and commands as shown in Figure 5.13 and 5.15.

The evaluation of constraints is made slightly more complex through the use of overloading. Overloaded constraints are evaluated and combined using the most-lenient combination algorithms described in Table 5.9 for constraint reference and Table 5.11 for constraint search. We present the operational semantics for evaluating references to overloaded constraints in Figure 5.14. Due to space considerations we use a more compact representation for constraints, writing $\mathbf{CST}c(a, s, r, P, f, f', msg, e)(E, \overline{\psi^{st}}, \overline{\psi^{r}})$ in place of the normal representation **CST** $c(a, s, r, P, f, f', msg, e)$ **Scope** E **Such That** $\overline{\psi^{st}}$ **If** **If** $\overline{\psi^{r}}$ **Then** b_1

$$\frac{S = (A, O, m, l) \quad a \in A \quad o \in O \quad d \in m(a, o)}{S \vdash d \textbf{ in } (a, o) \rightarrow \mathit{true}} \text{[E-CheckRight1]}$$

$$\frac{S = (A, O, m, l) \quad a \in A \quad o \in O \quad d \notin m(a, o)}{S \vdash d \textbf{ in } (a, o) \rightarrow \mathit{false}} \text{[E-CheckRight2]}$$

$$\frac{S = (A, O, m, l) \quad o \in O \quad o.t \rightarrow b}{S \vdash o.t = b \rightarrow \mathit{true}} \text{[E-CheckTag1]}$$

$$\frac{S = (A, O, m, l) \quad o \in O \quad o.t \nrightarrow b}{S \vdash o.t = b \rightarrow \mathit{false}} \text{[E-CheckTag2]}$$

$$\frac{S = (A, O, m, l) \quad a \in A \quad k \in \mathit{Roles}(a)}{S \vdash k \textbf{ in } \mathit{Roles}(a) \rightarrow \mathit{true}} \text{[E-CheckRole1}$$

$$\frac{S = (A, O, m, l) \quad a \in A \quad k \notin \mathit{Roles}(a)}{S \vdash k \textbf{ in } \mathit{Roles}(a) \rightarrow \mathit{false}} \text{[E-CheckRole2]}$$

$$\frac{S \vdash g \rightarrow \mathit{false}}{S \vdash !g \rightarrow \mathit{true}} \text{[E-Neg1]} \qquad \frac{S \vdash g \rightarrow \mathit{true}}{S \vdash !g \rightarrow \mathit{false}} \text{[E-Neg2]}$$

Figure 5.7: Operational semantics for guards, part 1

Else b_2. In the representation we use c as the name of the constraint, so for overloaded constraints *Constraint* contains multiple constraints with the name c. We use C_c to denote the set of constraints with name c. For a simple constraint, the cardinality of the set $|C_c|$ is 1. For overloaded constraints, $|C_c| > 1$. To identify which constraint member of C_c we mean in a given rule we use indexes such as $c_1(E, \overline{\psi_1^{st}}, \overline{\psi_1^{r}})$. We write $c(a, s, r, P, f, f', msg, e)(E, \overline{\psi^{st}}, \overline{\psi^{r}}) \rightarrow j$ for the result of applying one of the evaluation rules in Figure 5.13 as appropriate.

The evaluation of commands is performed in two steps. We details the evaluation steps in Section 5.4, but we briefly outline them here as well and show their operational semantics. First, before being run, a constraint search is performed for the command. If any constraint in the set *Constraint* yields a Forbid judgment, the command's false branch will be run. The truth table for the evaluation of the constraint search is shown in Table 5.7. Second, the guards $\overline{\psi}$ are evaluated for the command. If any of them yield false, the false branch of the command is run. Otherwise, the true branch for the command is

$$\frac{p' \in P \quad p \in \mathsf{descendants}(p')}{S \vdash p \ \mathbf{in}_a \ P \rightarrow \mathit{true}} \ [\text{E-PurposeA1}]$$

$$\frac{\nexists p' \in P \ . \ p \in \mathsf{descendants}(p')}{S \vdash p \ \mathbf{in}_a \ P \rightarrow \mathit{false}} \ [\text{E-PurposeA2}]$$

$$\frac{p' \in P \quad p \in \mathsf{descendants}(p') \cup \mathsf{ancestors}(p')}{S \vdash p \ \mathbf{in}_f \ P \rightarrow \mathit{true}} \ [\text{E-PurposeF1}]$$

$$\frac{\nexists p' \in P \ . \ p \in \mathsf{descendants}(p') \cup \mathsf{ancestors}(p')}{S \vdash p \ \mathbf{in}_f \ P \rightarrow \mathit{false}} \ [\text{E-PurposeF2}]$$

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \\ S \vdash c(a, s, r, P, f, f', msg, null)(E, \overline{\psi^{st}}, \overline{\psi^{r}}) \rightarrow j \quad j \in J \end{array}}{S \vdash c(a, s, r, P, f, f', msg, null) \in J \rightarrow \mathit{true}} \ [\text{E-Ref1}]$$

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \\ S \vdash c(a, s, r, P, f, f', msg, null)(E, \overline{\psi^{st}}, \overline{\psi^{r}}) \rightarrow j \quad j \notin J \end{array}}{S \vdash c(a, s, r, P, f, f', msg, null) \in J \rightarrow \mathit{false}} \ [\text{E-Ref2}]$$

$$\frac{S = (A, O, m, l) \quad a_1, a_2 \in A \quad a_1 \equiv a_2}{S \vdash a_1 = a_2 \rightarrow \mathit{true}} \ [\text{E-Agent1}]$$

$$\frac{S = (A, O, m, l) \quad a_1, a_2 \in A \quad a_1 \not\equiv a_2}{S \vdash a_1 = a_2 \rightarrow \mathit{false}} \ [\text{E-Agent2}]$$

Figure 5.8: Operational semantics for guards, part 2

run. The rules for command evaluation are shown in Figure 5.15.

5.4 Evaluation Engine for Privacy APIs

The evaluation engine for Privacy Commands acts as the interpreter for the language, processing commands and constraints, deriving judgments, evaluating guards, and performing operations. In this section we describe its functionality with respect to the formal model of the language, not a particular implementation. We defer discussion of implementation details to Chapter 6 where we present our implementation of the evaluation engine using Promela in SPIN. Since the intended purpose of a Privacy API is the evaluation of the properties of policies, we use Privacy APIs as an interactive policy model, a formulation of

$$\frac{S \vdash \psi_1 \rightarrow \mathit{true} \quad S \vdash \psi_2 \rightarrow b}{S \vdash \psi_1 \textbf{ and } \psi_2 \rightarrow b} \text{[E-SEQ1]} \qquad \frac{S \vdash \psi_1 \rightarrow \mathit{false}}{S \vdash \psi_1 \textbf{ and } \psi_2 \rightarrow \mathit{false}} \text{[E-SEQ2]}$$

$$\frac{S \vdash \psi_1 \rightarrow \mathit{true} \quad \overline{\psi} = \psi_2 \textbf{ and } \overline{\psi_2} \quad S \vdash \psi_2 \textbf{ and } \overline{\psi_2} \rightarrow b}{S \vdash \psi_1 \textbf{ and } \overline{\psi} \rightarrow b} \text{[E-SEQ3]}$$

$$\frac{S \vdash \psi_1 \rightarrow \mathit{true} \quad \overline{\psi} = \psi_2 \quad S \vdash \psi_2 \rightarrow b}{S \vdash \psi_1 \textbf{ and } \overline{\psi} \rightarrow b} \text{[E-SEQ4]} \qquad \frac{S \vdash \psi_1 \rightarrow \mathit{false}}{S \vdash \psi_1 \textbf{ and } \overline{\psi} \rightarrow \mathit{false}} \text{[E-SEQ5]}$$

Figure 5.9: Operational semantics for guard sequences

$$\frac{S = (A, O, m, l) \quad o \notin O \quad S' = (A, O \cup \{o\}, m, l)}{S \vdash \textbf{create object } o \rightarrow S'} \text{[E-CREATE]}$$

$$\frac{S = (A, O, m, l) \quad o \in O \quad O' = O - \{o\} \quad S' = (A, O', m, l)}{S \vdash \textbf{delete object } o \rightarrow S'} \text{[E-DELO1]}$$

$$\frac{S = (A, O, m, l) \quad o \notin O \quad S' = (A, O, m, l)}{S \vdash \textbf{delete object } o \rightarrow S'} \text{[E-DELO2]}$$

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad o \in O \quad o' = o[t \mapsto b] \quad O' = \{O - \{o\}\} \cup o' \\ S' = (A, O', m, l) \end{array}}{S \vdash \textbf{set } o.t = b \rightarrow S'} \text{[E-SETTAG1]}$$

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad o \in A \quad o' = o[t \mapsto b] \quad O' = \{O - \{o\}\} \cup o' \\ A' = \{A - \{o\}\} \cup o' \quad S' = (A', O', m, l) \end{array}}{S \vdash \textbf{set } o.t = b \rightarrow S'} \text{[E-SETTAG2]}$$

Figure 5.10: Operational semantics for operations, part 1

the policy that permits experimentation, exploration of reachable states, and comparison of policies. The state space exploration program or mechanism used to perform the experiments, explorations, or comparisons is independent of the evaluation engine. Instead, the job of the evaluation engine is to enable scripts, state space explorers, or users to use the Privacy API as an interactive artifact by processing the requests and responses needed to run commands.

The jobs that the evaluation engine performs are as follows. We consider each of the jobs separately in the following subsections as noted.

1. Run commands and constraints, including process parameters as inputs and provide

$$\dfrac{S = (A, O, m, l) \qquad a \in A \quad o \in O \quad m' = m[(a,o) \mapsto \{m(a,o) \cup \{d\}] \\ S' = (A, O, m', l)}{S \vdash \textbf{insert } d \textbf{ in } (a,o) \rightarrow S'} \ [\text{E-InsertR}]$$

$$\dfrac{S = (A, O, m, l) \qquad a \in A \quad o \in O \quad m' = m[(a,o) \mapsto \{m(a,o) - \{d\}\}] \\ S' = (A, O, m', l)}{S \vdash \textbf{delete } d \textbf{ from } (a,o) \rightarrow S'} \ [\text{E-DeleteR}]$$

$$\dfrac{S = (A, O, m, l) \quad l' = l + s \quad S' = (A, O, m, l')}{S \vdash \textbf{insert } s \textbf{ in log } \rightarrow S'} \ [\text{E-InsertL}]$$

$$\dfrac{e \in \mathit{Command} \quad a, s, r \in A \quad f \in O \quad f' \notin O \\ \nexists c' \in \mathit{Constraint} \;.\; S \vdash c'(a, s, r, P, f, f', msg, e) \rightarrow \mathit{Forbid} \\ S \vdash e(a, s, r, P, f, f', msg) \rightarrow S'}{S \vdash \textbf{invoke } e(a, s, r, P, f, f', msg) \rightarrow S'} \ [\text{E-Invoke}]$$

Figure 5.11: Operational semantics for operations, part 2

$$\dfrac{}{S \vdash \bullet \rightarrow S} \ [\text{E-Seq6}] \qquad \dfrac{S \vdash \overline{\omega} \rightarrow S'}{S \vdash \overline{\omega};\ \textbf{return } b \rightarrow S', b} \ [\text{E-Seq7}]$$

$$\dfrac{S \vdash \omega \rightarrow S' \quad S' \vdash \overline{\omega} \rightarrow S''}{S \vdash \omega; \overline{\omega} \rightarrow S''} \ [\text{E-Seq8}]$$

Figure 5.12: Operational semantics for operations sequences

them to commands and constraints. (Section 5.4.1)

2. Derive judgments from constraints, both overloaded and non-overloaded. (Section 5.4.2)

3. Perform the constraint search required before commands are run. (Section 5.4.3)

The second and third jobs are more complex than the first and involve the management of references between commands and constraints. We discuss them in detail in the next two subsections, but first we give a short overview of the two types of references that the evaluation engine processes. Figure 5.16 illustrates the two types of references between constraints and commands. In the figure, Command2 contains a guard which references Constraint1's judgment. The reference tells the evaluation engine to run Constraint1,

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \\ e \in \{E \cup null\} \quad \overline{\psi^{st}} \rightarrow true \quad \overline{\psi^{r}} \rightarrow true \end{array}}{\begin{array}{c} S \vdash \textbf{CST}\ c(a, s, r, P, f, f', msg, e)\ \textbf{Scope}\ E\ \textbf{Such That}\ \overline{\psi^{st}} \\ \textbf{If}\ \overline{\psi^{r}}\ \textbf{then return}\ \boldsymbol{true}\textbf{Else return}\ \boldsymbol{false} \rightarrow \text{Allow} \end{array}} \text{[E-ALLOW]}$$

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \\ e \in \{E \cup null\} \quad \overline{\psi^{st}} \rightarrow true \quad \overline{\psi^{r}} \rightarrow false \end{array}}{\begin{array}{c} S \vdash \textbf{CST}\ c(a, s, r, P, f, f', msg, e)\ \textbf{Scope}\ E\ \textbf{Such That}\ \overline{\psi^{st}} \\ \textbf{If}\ \overline{\psi^{r}}\ \textbf{then return}\ \boldsymbol{true}\textbf{Else return}\ \boldsymbol{false} \rightarrow \text{Forbid} \end{array}} \text{[E-FORBID]}$$

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \\ e = null \quad \overline{\psi^{st}} \rightarrow false \quad \overline{\psi^{r}} \rightarrow true \end{array}}{\begin{array}{c} S \vdash \textbf{CST}\ c(a, s, r, P, f, f', msg, e)\ \textbf{Scope}\ E\ \textbf{Such That}\ \overline{\psi^{st}} \\ \textbf{If}\ \overline{\psi^{r}}\ \textbf{then return}\ \boldsymbol{true}\textbf{Else return}\ \boldsymbol{false} \rightarrow \text{Don't Care/Allow} \end{array}} \text{[E-DCA]}$$

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \\ e = null \quad \overline{\psi^{st}} \rightarrow false \quad \overline{\psi^{r}} \rightarrow false \end{array}}{\begin{array}{c} S \vdash \textbf{CST}\ c(a, s, r, P, f, f', msg, e)\ \textbf{Scope}\ E\ \textbf{Such That}\ \overline{\psi^{st}} \\ \textbf{If}\ \overline{\psi^{r}}\ \textbf{then return}\ \boldsymbol{true}\textbf{Else return}\ \boldsymbol{false} \rightarrow \text{Don't Care/Forbid} \end{array}} \text{[E-DCF]}$$

$$\frac{S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \quad e \neq null \quad \overline{\psi^{st}} \rightarrow false}{\begin{array}{c} S \vdash \textbf{CST}\ c(a, s, r, P, f, f', msg, e)\ \textbf{Scope}\ E\ \textbf{Such That}\ \overline{\psi^{st}} \\ \textbf{If}\ \overline{\psi^{r}}\ \textbf{then return}\ \boldsymbol{true}\textbf{Else return}\ \boldsymbol{false} \rightarrow \text{Ignore} \end{array}} \text{[E-IGNORE1]}$$

$$\frac{S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \quad e \neq null \quad e \notin E}{\begin{array}{c} S \vdash \textbf{CST}\ c(a, s, r, P, f, f', msg, e)\ \textbf{Scope}\ E\ \textbf{Such That}\ \overline{\psi^{st}} \\ \textbf{If}\ \overline{\psi^{r}}\ \textbf{then return}\ \boldsymbol{true}\textbf{Else return}\ \boldsymbol{false} \rightarrow \text{Ignore} \end{array}} \text{[E-IGNORE2]}$$

Figure 5.13: Operational semantics for constraints

derive its judgment, and return it to Command2 for use in the guard. When the evaluation engine runs Constraint1 we say that it is *run by reference*. Command1 does not contain a reference to Constraint1 as a guard, but it is included in the scope for Constraint1. When the evaluation engine is instructed to run Command1 (*e.g.,* when a user attempts to run Command1, a state space explorer tests whether Command1 leads to some state, etc.), it finds Constraint1 and runs it since Command1 is in its scope. The evaluation engine then uses the judgment from Constraint1 to determine whether Command1 will be permitted to run.

$$\frac{c_1 \in C_c \quad S \vdash c_1(a,s,r,P,f,f',msg,e)(E,\overline{\psi_1^{st}},\overline{\psi^r}_1) \rightarrow \text{Allow}}{S \vdash c(a,s,r,P,f,f',msg,e) \rightarrow \text{Allow}} \;[\text{E-RefO1}]$$

$$\frac{\begin{array}{c} c_1 \in C_c \quad S \vdash c_1(a,s,r,P,f,f',msg,e)(E,\overline{\psi_1^{st}},\overline{\psi^r}_1) \rightarrow \text{Forbid} \\ \nexists c_2 \in C_c \;.\; S \vdash c_2(a,s,r,P,f,f',msg,e)(E,\overline{\psi_2^{st}},\overline{\psi_2^r}) \rightarrow \text{Allow} \end{array}}{S \vdash c(a,s,r,P,f,f',msg,e) \rightarrow \text{Forbid}} \;[\text{E-RefO2}]$$

$$\frac{\begin{array}{c} c_1 \in C_c \quad S \vdash c_1(a,s,r,P,f,f',msg,e)(E,\overline{\psi_1^{st}},\overline{\psi_1^r}) \rightarrow \text{Don't Care/Allow} \\ \nexists c_2 \in C_c \;.\; S \vdash c_2(a,s,r,P,f,f',msg,e)(E,\overline{\psi_2^{st}},\overline{\psi_2^r}) \rightarrow \{\text{Allow}, \text{Forbid}\} \end{array}}{S \vdash c(a,s,r,P,f,f',msg,e) \rightarrow \text{Don't Care/Allow}} \;[\text{E-RefO3}]$$

$$\frac{\begin{array}{c} c_1 \in C_c \quad S \vdash c_1(a,s,r,P,f,f',msg,e)(E,\overline{\psi_1^{st}},\overline{\psi_1^r}) \rightarrow \text{Don't Care/Allow} \\ \nexists c_2 \in C_c \;.\; S \vdash c_2(a,s,r,P,f,f',msg,e)(E,\overline{\psi_2^{st}},\overline{\psi_2^r}) \rightarrow \{\text{Allow}, \text{Forbid}, \\ \text{Don't Care/Allow}\} \end{array}}{S \vdash c(a,s,r,P,f,f',msg,e) \rightarrow \text{Don't Care/Forbid}} \;[\text{E-RefO4}]$$

Figure 5.14: Operational semantics for overloading

Table 5.7: Judgment combination for constraint search

Constraint 1	Constraint 2	Combined
Allow	Allow	Allow
Allow	Forbid	Forbid
Allow	Ignore (Allow)	Allow
Forbid	Ignore (Allow)	Forbid
Forbid	Forbid	Forbid
Ignore (Allow)	Ignore (Allow)	Ignore (Allow)

5.4.1 Running Commands and Constraints

The first job, running commands and constraints and processing parameters for them is the process of interpreting a Privacy API as an executable program. First, parameter passing is performed by renaming members of the knowledge state to match the parameter names listed. The renaming is akin to call-by-reference in that if an object o is passed as parameter f to a function and f is modified, o changes as well. The evaluation engine then runs commands and constraints as follows.

Running Commands The evaluation engine runs a command either because a user (or automated script, state space explorer, etc.) requests that the command be run or

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \\ c \in \mathit{Constraint} \quad S \vdash c(a, s, r, P, f, f', msg, e) \rightarrow \text{Forbid} \quad S \vdash \overline{\omega^f} \rightarrow S' \end{array}}{S \vdash \textbf{Cmd } e(a, s, r, P, f, f', msg) \textbf{ If } \overline{\psi} \textbf{ Then } \overline{\omega^t} \textbf{ Else } \overline{\omega^f} \rightarrow S', \mathit{false}} \text{[E-CMD1]}$$

$$\frac{S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \quad S \vdash \overline{\psi} \rightarrow \mathit{false} \quad S \vdash \overline{\omega^f} \rightarrow S'}{S \vdash \textbf{Cmd } e(a, s, r, P, f, f', msg) \textbf{ If } \overline{\psi} \textbf{ Then } \overline{\omega^t} \textbf{ Else } \overline{\omega^f} \rightarrow S', \mathit{false}} \text{[E-CMD2]}$$

$$\frac{\begin{array}{c} S = (A, O, m, l) \quad a, s, r \in A \quad f \in O \quad f' \notin O \quad S \vdash \overline{\psi} \rightarrow \mathit{true} \\ \nexists c \in \mathit{Constraint} \; . \; S \vdash c(a, s, r, P, f, f', msg, e) \rightarrow \text{Forbid} \quad S \vdash \overline{\omega^t} \rightarrow S' \end{array}}{S \vdash \textbf{Cmd } e(a, s, r, P, f, f', msg) \textbf{ If } \overline{\psi} \textbf{ Then } \overline{\omega^t} \textbf{ Else } \overline{\omega^f} \rightarrow S', \mathit{true}} \text{[E-CMD3]}$$

Figure 5.15: Operational semantics for commands

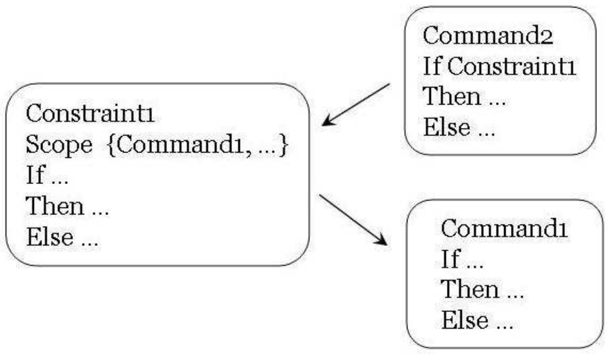


Figure 5.16: Reference types

because another already running command includes a reference to it in an operation (*i.e.*, **invoke** e). In either cases, the evaluation engine first performs a constraint search to find all applicable constraints for the command as defined below in Section 5.4.3. If the judgment from the constraint search is Allow or Ignore (Allow), the evaluation engine runs the guards in $\overline{\psi}$ for the command. If the result from all of the guards is true, the evaluation engine runs the operations in $\overline{\omega^t}$. If the judgment from the constraint search is Forbid or the result from all of the guards if false, the evaluation engine runs the operations in $\overline{\omega^f}$ and the return value of the command is false.

Running Constraints The evaluation engine runs constraints in two modes: constraint search and by a constraint reference. In either case, the evaluation engine performs the same steps to derive three boolean results: (1) whether the calling command is a member of the constraint's scope set, (2) whether all of the *such that* guards are satisfied, and (3) whether all of the regular guards are satisfied. We denote the three results using the following boolean variables: b_{scp} for the scope membership, b_{st} for the result of the *such that* guards, and b_r for the result of the regular guards. Commands accept a parameter set a, s, r, P, f, f', msg which provides input information. The evaluation engine also provides an implicit parameter $cmd \in Command$ which is defined as follows:

1. If the constraint is run by the evaluation engine for a constraint search before running command e, $cmd = e$. In Figure 5.17, $cmd =$ Command1 since Constraint1 is run during a constraint search for Command1.

2. If the constraint is run because a command e includes a reference to it, $cmd = e$. In Figure 5.18, $cmd =$ Command1 since Constraint1 is run because of a reference to it in Command1.

3. If the constraint is run because a constraint c' which was run by cmd' includes a reference to it, $cmd = cmd'$. In Figure 5.19, $cmd =$ Command1 since Constraint2 is run because of a reference to it in Constraint1 and Constraint1 is run because of a reference to it in Command1.

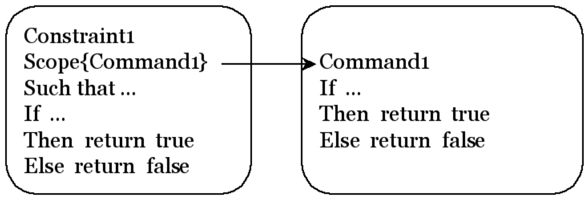


Figure 5.17: Scenario for running a constraint: Search

Intuitively, cmd is the command which causes the constraint to be run. The boolean results for the constraint are derived as follows. Let $c.E$ denote the scope set E for constraint c as per Definition 5.2.10:

- $b_{scp} = true$ iff $cmd \in c.E$.

- $b_{st} = \bigwedge_{\psi \in \overline{\psi^{st}}} \psi$.

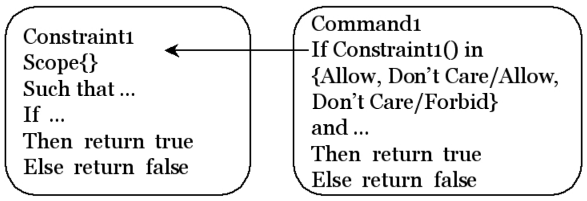


Figure 5.18: Scenario for running a constraint: Reference

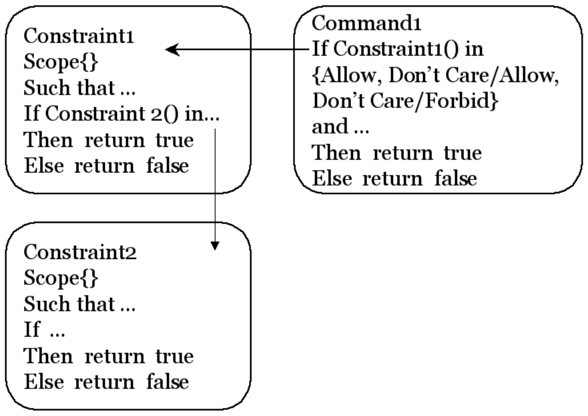


Figure 5.19: Scenario for running a constraint: Chain

- $b_r = \bigwedge\limits_{\psi \in \overline{\psi^r}} \psi.$

The evaluation engine uses the boolean values b_{scp}, b_{st}, and b_r to derive judgments from the constraints based on whether they are run because of a reference (Section 5.4.2) or for the constraint search (Section 5.4.3) and whether they are overloaded.

5.4.2 Judgment Derivation

Based on the different combinations of results, the resulting judgment from a constraint may be one of five judgments as we shall describe:

Allow Explicitly allow the execution of the intended command

Forbid Explicitly forbid the execution of the intended command

Ignore (Allow) Do not issue a ruling on the command. Since by default commands may execute, ignore essentially allows the execution. It differs semantically from Allow (as shown in Table 5.9) in that it represents a don't care judgment from a non-applicable constraint and therefore may be overridden by other, more applicable judgments.

Don't Care/Allow The constraint is not directly applicable to the query. If certain properties of it were different (*i.e.,* the *such that* guards were satisfied), the constraint would have allowed the execution of the intended command.

Don't Care/Forbid The constraint is not directly applicable to the query. If certain properties of it were different (*i.e.,* the *such that* guards were satisfied), the constraint would have forbidden the execution of the intended command.

The third job of the evaluation engine is to run constraints as appropriate and derive the correct judgments from them. The derived judgments may be returned to a calling command or constraint for use in a guard or as part of a constraint search. In this subsection we discuss the judgment derivation process for when a constraint is run due to a reference in a guard in a command or constraint. The judgment derivation algorithm is slightly different in the case of the constraint search, so we postpone its description to Section 5.4.3 where we address constraint search in detail.

As noted in Section 5.2.4, constraints perform three groups of checks when run: scope, *such that* guards, and regular guards. Scope determines whether the command which referenced the constraint is subject to the constraint as an invariant. The *such that* guards examine whether the case is relevant. The regular guards examine whether the action should be permitted. Since each can be assigned a boolean value, there are in theory eight possible outcomes from the three groups which must be considered. When running a constraint from a reference, however, the scope check is not relevant since the constraint is being run from a reference, not during a search for invariants, so are only concerned with the four potential outcomes from the *such that* and regular guards. We return to the scope check in Section 5.4.3 during our discussion of the constraint search algorithm.

Single Constraint For a single constraint, the judgment derivation algorithm is summarized in Table 5.8. The table shows all eight possible boolean combinations between the scope (b_{scp}), *such that* (b_{st}), and regular guards (b_r) and the resulting judgment (j). As noted, the scope result may be ignored in the derivation of the judgment and so the cases where scope is true and false parallel each other. When both the *such that* and regular guards sets result in true (*i.e.,* both are satisfied), the resulting judgment is Allow. This corresponds to the constraint being both relevant to the situation and permitting of the action. When the *such that* guards are true but the regular guards are not, the resulting judgment is Forbid. This corresponds to the constraint being relevant to the situation, but forbidding of the action. When the *such that* guards are false (*i.e.,* not satisfied) and the regular guards are true, the resulting judgment is Don't Care/Allow. This corresponds to the constraint not being relevant to the situation, but permitting if it were. Finally, when both the *such that* and regular guards are false, the resulting judgment is Don't Care/False. This corresponds to the constraint not being relevant to the situation, but forbidding if it were.

Table 5.8: Judgment from a single constraint run by reference

In Scope (b_{scp})	Such That (b_{st})	Regular (b_r)	Judgment (j)
True	True	True	Allow
True	True	False	Forbid
True	False	True	Don't Care/Allow
True	False	False	Don't Care/Forbid
False	True	True	Allow
False	True	False	Forbid
False	False	True	Don't Care/Allow
False	False	False	Don't Care/Forbid

The design of the judgment derivation algorithm is based on the legal policy idiom of applicability and reference. Paragraphs in legal documents give rulings for what is permissible or required in a given class of situations. The ruling of the paragraph can then be applied to cases which match the situations cited in the paragraph. For situations which are not included in the situations mentioned in the paragraph, the paragraph neither permits nor forbids the action. It simply is not relevant. In order to capture this idiom, we place the situation definition in the *such that* guards and the ruling of the paragraph in the regular guards. By separating out the two types of guards we enable a constraint to issue three judgments: Allow, Forbid, and *don't care* .

We further specialize the *don't care* judgments into Don't Care/Allow and Don't Care/Forbid to better support references in the legal text. As we show in Example 5.2.2, a paragraph may reference the permissions given in another paragraph with respect to a different case. The reference idiom is of the type "Paragraph A: You may do action a in situations that paragraph B would allow". If paragraph B is not relevant to all of the cases that the A refers to, B may yield a *don't care* ruling since its *such that* guards may not be satisfied. Still, the permission in A is based on cases that B would have allowed and so it is interested in the result of its regular guards, ignoring its *such that* guards. We call such a decision Don't Care/Allow, indicating that B is not relevant to the case, but if it were, it would have permitted the action. Don't Care/Forbid is similar, used for cases where the paragraph is not relevant but would have forbidden the action if it were.

Overloaded Constraints The evaluation engine derives judgments for overloaded constraints similarly to how it derives judgments for single constraints with the addition of an extra judgment combination step. As noted above in Section 5.2.4, overloaded constraints are combined using an OR to support cases where one paragraph issues rulings for multiple situations. The judgment combination algorithm is therefore uses a *most lenient* combination method. The algorithm for combining judgments from an overloaded constraint run from a reference is straightforward and is shown in Figure 5.20. The code begins on line 1 with some temporary variables. Line 2 gets the first instance of the overloaded constraint and stores it in a temporary variable. The do/while loop on lines 3–6 gets the judgment from each constraint instance using the single constraint judgment derivation algorithm discussed above, stores it in the temporary variable `j` and combines it with the final judgment (`final`) using the `most-lenient` algorithm. We perform the judgment combination iteratively since the combination of judgment is commutative.

The `most-lenient` combination algorithm uses the judgment combination rules in Table 5.9. Since the combination is commutative, the table elides redundant cases (*i.e.,* Allow/Forbid and Forbid/Allow). The algorithm uses the following two rules:

1. The most lenient judgment overrules.

```
Judgment j, final; int i = 0;
Constraint c = overloaded-constraint-set[i];
do {
    j = Run c;
    final = most-lenient(final, j);
} while (c = overloaded-constraint-set[++i])
```

Figure 5.20: Pseudocode for overloaded constraint derivation by reference

2. A *don't care* judgment is overruled by a Allow or Forbid.

The use of these rules leads to a total order on the judgments: (1) Allow, (2) Forbid, (3) Don't Care/Allow, (4) Don't Care/Forbid. The intuition to the combination is that *don't care* judgments are from constraint instances that are not applicable and therefore should be overridden by ones that are more applicable, even if the more applicable one is stricter. As shown in the table therefore, for each combination of judgments, the one higher on the total order of judgments is selected.

Table 5.9: Judgment from an overloaded constraint run by reference

Constraint 1	Constraint 2	Combined
Allow	Allow	Allow
Allow	Forbid	Allow
Allow	Don't Care/Allow	Allow
Allow	Don't Care/Forbid	Allow
Forbid	Forbid	Forbid
Forbid	Don't Care/Allow	Forbid
Forbid	Don't Care/Forbid	Forbid
Don't Care/Allow	Don't Care/Allow	Don't Care/Allow
Don't Care/Allow	Don't Care/Forbid	Don't Care/Allow
Don't Care/Forbid	Don't Care/Forbid	Don't Care/Forbid

Figure 5.21 illustrates a simple constraint overloading scenario. Command2 refers to Constraint2 using a guard. The first instance of Constraint2 issues an Allow judgment if A is true. The second instance issues Allow if B is true. The evaluation engine runs both instances and combined their judgments using Table 5.9. As a result, if either A or B are true, the resulting judgment is Allow and the guard in Command2 is satisfied.

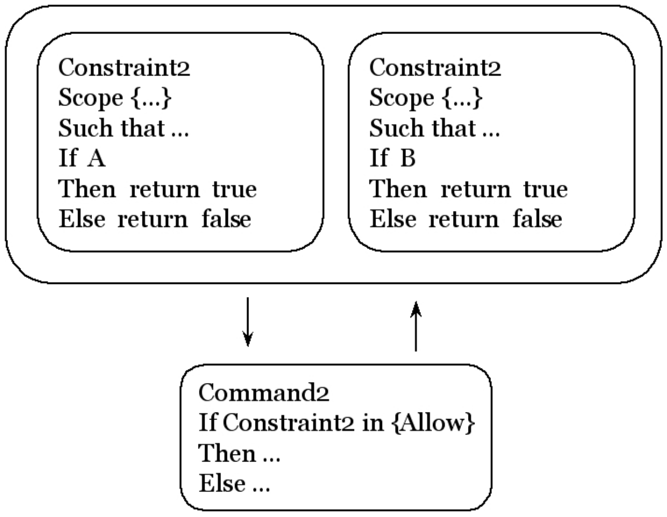


Figure 5.21: Overloaded constraint scenario

5.4.3 Constraint Search

The fourth job that the evaluation engine performs the constraint search performed before a command is run. The purpose of the constraint search is to discover any constraints which are applicable to the command as invariants. The intuition is that a paragraph may be subject to a limitation from another paragraph which is not explicitly mentioned in it. Such limitations are typically in the form of invariants which give rules with broad applicability, but are not mentioned explicitly in the commands which are subject to them. Figure 4.1 on page 73 shows a schematic for invariant references. In the scenario shown, the job of the evaluation engine is to discover that the three constraints have the command in their scopes, run them, and use their judgments to decide if the command should be allowed to run. If result of the constraint search is Forbid, it is the equivalent of the false branch of the command is executed.

The algorithm for finding applicable constraints is straightforward and its pseudocode is shown in Figure 5.22. On lines 2–7, the evaluation engine loops through each constraint to determine if `cmd` is in scope. For each constraint that contain `cmd` in scope, the evaluation engine derives its judgment (using the single and overloaded constraint derivation algorithms) and combines the result using a most-strict algorithm (lines 4–5). If the final result from the constraint search is Forbid (lines 8–9), the search returns false. Otherwise, the search returns true.

```
1   Command cmd; Judgment j, final;
2   foreach (Constraint cst in Constraints) {
3       if (cmd in cst.scope) {
4           j = Run cst;
5           final = most-strict(j, final);
6       }
7   }
8   if (final == Forbid)
9       return false;
10  else
11      return true;
```

Figure 5.22: Constraint search pseudocode

The `most-strict` combination algorithm on line 5 combines the judgments derived from the constraints using a most strict algorithm, shown in Table 5.7. As in Section 5.4.2 with respect to overloaded constraints there is a total order on the judgments: (1) Forbid, (2) Allow, (3) Ignore (Allow). Since Ignore and Allow are effectively the same, the algorithm may be summarized as "Allow unless a constraint Forbids."

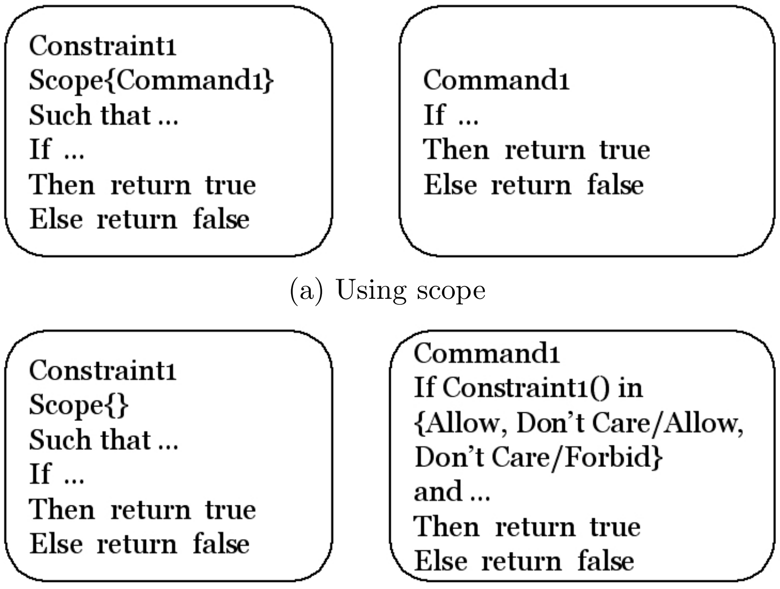
Logically, the return value from the constraint search is the equivalent of an extra guard added to the beginning of each command. For example, for a command with the form:

CMD Command1(a, s, r, P, f, f', msg)
if a = s
then ...
else ...

Could be logically rewritten as:

CMD Command1(a, s, r, P, f, f', msg)
if Constraint-Search(Command1) in {Allow, Don't Care/Allow, Don't Care/Forbid}
and a = s
then ...
else ...

From the model's perspective, the two cases in Figure 5.23 are equivalent. In case (a), Constraint1 includes Command1 in its scope. If Constraint1 yields Forbid, the false branch of Command1 will be run. Similarly, in case (b), Command1 includes a reference to Constraint1. If its resulting judgment is Forbid, the guard is not satisfied and the false branch of Command1 is run. The interchangeability of the two reference types is important since it allows policies to have redundant references without changing the semantics of the policy. For instance, in Example 5.2.1 above, the constraint Permitted506c1 included TreatmentUse506c1 in its scope and TreatmentUse506c1 included a reference to Permitted506c1 which accepts only Allow. The inclusion of TreatmentUse506c1 in Permitted506c1's scope since the guard is more restrictive than the constraint check, but it helps make the Privacy API more readable since it guides readers to the commands that are subject to the restrictions in the constraint paragraph.



(b) Using a reference

Figure 5.23: Equivalence of scope and references

The judgment derivation for constraints during the constraint check differs slightly between the processing of single constraints and overloaded constraints as with references in Section 5.4.2. For the constraint check we collapse the two *don't care* judgments into a single Ignore (Allow) judgment. The intuition is that if the constraint is not applicable to the command either because the command is not in scope or its *such that* guards are not satisfied, it may be ignored. We place a higher bar for applicability for the constraint search since the goal is to discover constraints which limit the command that is about to be run but are not mentioned explicitly with a reference in the command. We are therefore interested only in constraints which are applicable to the command and the situation.

Single Constraints Table 5.10 shows how the evaluation engine determines judgments from constraints during the constraint search. As noted, if either the command is not in scope or the *such that* guards are not satisfied, the default judgment is Ignore. If the command is in scope and the *such that* guards are satisfied, the judgment is determined based on the results of the regular guards.

Table 5.10: Judgment from a single constraint during constraint search

In Scope (b_{scp})	Such That (b_{st})	Regular (b_r)	Judgment (j)
True	True	True	Allow
True	True	False	Forbid
True	False	True	Ignore (Allow)
True	False	False	Ignore (Allow)
False	True	True	Ignore (Allow)
False	True	False	Ignore (Allow)
False	False	True	Ignore (Allow)
False	False	False	Ignore (Allow)

Overloaded Constraints For overloaded constraints, each instance of the constraint is run using the same algorithm shown above in Figure 5.20 except that the judgment combination algorithm is based on Table 5.11. As in Table 5.9, there is a total order on the judgments: (1) Allow, (2) Forbid, (3) Ignore (Allow). The order reflects the rules that *don't care* judgments are overruled by Allow or Forbid and that we take the most lenient judgment.

5.4.4 Chained References

Note that whenever a constraint c_1 is run using a reference $c_1(a, s, r, P, f, f', msg) \in J$ from another constraint c_2, the judgment derivation procedure is followed, regardless of

Table 5.11: Judgment from an overloaded constraint during constraint search

Constraint 1	Constraint 2	Combined
Allow	Allow	Allow
Allow	Forbid	Allow
Allow	Ignore (Allow)	Allow
Forbid	Forbid	Forbid
Forbid	Ignore (Allow)	Forbid
Ignore (Allow)	Ignore (Allow)	Ignore (Allow)

whether c_2 was run as part of a constraint search. That is, if c_2 is invoked as part of a constraint search and in order to evaluate one of its guards, c_2 must be evaluated, the judgment derivation procedure in this subsection is followed for deriving the judgment of c_2 while the constraint search judgment procedure is followed for deriving the judgment for c_1.

For instance, in Figure 5.19, Constraint1 is run because of a reference in Command1. Constraint2 in turn is run because Constraint1 contains a reference to it. Conversely, in Figure 5.24, Constraint1 is run because of a reference to it in Command1. Constraint2 in turn is run because of a reference to it in Constraint1. In the first case, the judgment from Constraint1 is derived using Table 5.8 while in the second case the judgment for Constraint1 is derived using Table 5.10. However, in both cases the judgment for Constraint2 is derived using the judgment algorithm in Table 5.8.

The most important reason for the difference is that it makes the semantics of constraint references uniform. That is, a constraint can include a reference to other constraints (*i.e.,* a guard of the form **if** $e(a, s, r, P, f, f', msg) \in \{J\}$) and receive judgments uniformly, regardless of whether it was run as part of a constraint search or from a reference. Intuitively, it also is logical that when a constraint is run inside of a reference, the evaluation engine should return a detailed judgment, including Don't Care/* judgments instead of the more generic Ignore judgment so that the calling constraint may have the most information. Conversely, when a constraint is run as part of a constraint search, the command never receives the judgment from the search and therefore more generic judgments can be derived.

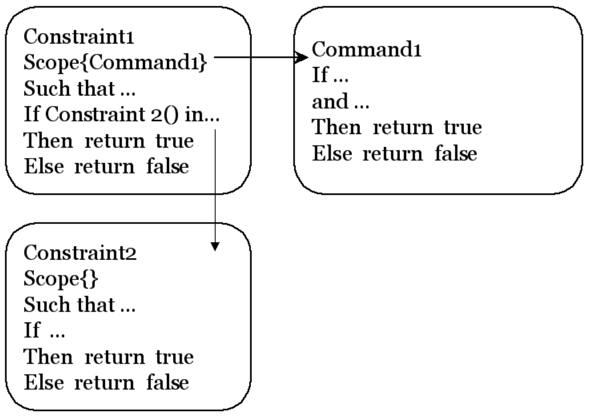


Figure 5.24: Constraint chained from a search

5.4.5 Termination

We now consider the termination properties of the evaluation engine. Intuitively, we may posit that since constraints may reference other constraints in a circular manner (*i.e.,* Constraint1 has a reference to Constraint2 in its guards and Constraint2 has Constraint1 in its guards), we must impose circularity restrictions on references included in constraints. Similarly, we must impose circularity restrictions on how commands reference each other (*i.e.,* Command1 runs Command2 if satisfied and Command2 runs Command1).

Let us concretize the intuition above using the following lemmas. First, recall that *Agent*, *Object*, *Purpose*, *Tag*, and *Role* are restricted to being finite (in Section 5.1) so may ensure that guards always eventually terminate.

Let G_c be a directed graph with a node n_c for each $c \in \mathit{Constraint}$. For each c, let ψ^{st} be c's *such that* guards and ψ^r be c's regular guards. Let us draw a directed edge from n_c to $n_{c'}$ if $\exists \psi \in \{\psi^{st} \cup \psi^r\}$. $\psi \equiv c'(a, s, r, P, f, f', msg) \in J$.

Lemma 5.4.1 *(Constraint Circularity)*

Constraint search and constraint references always terminate if Constraint and Command are finite and G_c is acyclic.

Proof: Let us consider constraint search first. Since *Constraint* is finite, the evaluation engine only needs to consider a finite number of constraints in its search. We therefore must show that the evaluation of each constraint eventually terminates. Let us consider the evaluation of a constraint c performed by the evaluation engine as part of a constraint search for command e.

First, the engine checks the scope of c to see if e is present. Since *Command* is finite, the scope is also finite and the engine eventually discovers in e is present in the set.

Second, the engine evaluates the *such that* guards of the constraint to see if it is applicable. Let us consider each guard individually to discuss its termination properties, considering the guard $c(args) \in J$ last since it is more complicated.

- d in (a, o). The evaluation terminates since *Object* is finite and therefore the matrix is of finite size.
- $o.t = b$. The evaluation terminates since o is identified and it has only a finite number of tags.
- k in $Roles(a)$. The evaluation terminates since k and a are identified and *Role* is finite.
- p in$_a$ P, p in$_f$ P. The evaluation terminates since *Purpose* is finite and $P \subseteq Purpose$.
- $a_1 = a_2$. The evaluation terminates since a_1 and a_2 are identified and *Agent* is finite.
- $!\psi$. The evaluation terminates since each ψ terminates.

For $c'(args) \in J$, the evaluation requires that c' be run as well. Let us observe that by the construction of G_c, running c' is equivalent to traversing the edge from n_c to $n_{c'}$ in G_c. Similarly, any references in the guards of c' are represented by edges from c' to the referenced guards. We may therefore consider the evaluation of c' as a walk from c to c' and then from c' to all constraints referenced by c'. Since G_c is finite and acyclic, we are guaranteed that the walk from c' will terminate and therefore the evaluation of $c'(args) \in J$ will terminate as well.

Third, if the *such that* guards all yield true, the evaluation engine runs the regular guards for the constraint. The argument for the termination of the regular guards is identical to the argument for the termination of the *such that* guards.

Fourth, the engine combines the judgments derived from the applicable constraints into a final judgment for the running of e using a logical AND. Since *Constraint* is finite, the combination is finite and terminates.

The argument for constraint reference is similar since the evaluation engine performs the second, third, and fourth steps as above with the exception that the third step is performed regardless of whether the second step yields true. $\square$

For commands, we offer a similar construction and proof. Let G_e be a directed graph with a node n_e for each $e \in \mathit{Command}$. For each e, let $\overline{\omega^t}$ be e's true branch operations. Let us draw a directed edge from n_e to $n_{e'}$ if $\exists \omega \in \overline{\omega^t} \; . \; \omega \equiv \text{invoke } e'(a, s, r, P, f, f', msg)$.

Lemma 5.4.2 *(Command Circularity)*

Commands and command references always terminate if Constraint and Command are finite and G_e and G_c are acyclic.

Proof: Let us consider the steps taken by the evaluation engine when running a command e, whether by reference or not.

First, the evaluation engine performs a constraint search for e. We have shown in Lemma 5.4.1 that the constraint search always terminates if *Command* and *Constraint* are finite and G_c is acyclic.

Second, the evaluation engine evaluates the guards of e. Lemma 5.4.1 shows that all guard evaluation terminates if *Command* and *Constraint* are finite and G_c is acyclic.

Third, if the evaluation finds that the guards are true and the constraint search permits e to run, it runs the operations in $\overline{\omega^t}$. Let us consider each operation separately with respect to its termination properties, considering invoke $e(args)$ last since it is more complicated.

- create object o, delete object o. The operations perform a finite number of steps in creating or deleting an object since *Object* is finite and so both terminate.
- set $o.t = b'$. The operation terminates since o is identified and *Tag* is finite.
- insert d in (a, o), delete d from (a, o). The operations terminate since d, a, and o are identified and *Agent* and *Object* are finite.

- insert s in log. The operation terminates since the string s must be finite.
- inform a of msg. The operation terminates since a and msg are identified and $Agent$ is finite.
- return b. The operation terminates since it does not perform any state updates.

For invoke $e'(args)$, the evaluation requires that e' be run as well. Let us observe that by the construction of G_e, running e' is equivalent to traversing the edge from n_e to $n_{e'}$ in G_e. Similarly, any commands run by e' are represented by edges from e' to the referenced commands. We may therefore consider the evaluation of e' as a walk from e to e' and then from e' to all commands referenced by e'. Since G_e is finite and acyclic, we are guaranteed that the walk from e' will terminate and therefore the evaluation of invoke $e'(args)$ will terminate as well.

Fourth, if the evaluation engine finds that the guards are false, the engine runs the operations in $\overline{\omega^f}$. They are a subset of $Operation$ and therefore all terminate by the argument presented for $\overline{\omega^t}$. $\square$

Using the above lemmas, we arrive at a theorem regarding the termination of the evaluation engine. Let G_c and G_e be constructed as above.

Theorem 1 *(Evaluation Engine Termination)*

For a Privacy API $\phi = (Command, Constraint)$, the evaluation engine terminates if Command and Constraint are finite and G_c and G_e are acyclic.

Proof: By Lemma 5.4.2 we have the for running any command $e \in Command$, the evaluation engine terminates if *Command* and *Constraint* are finite and G_c and G_e are acyclic. A straightforward application of the result to a collection of commands yields the result for ϕ.

$\square$

5.5 Policies and Licensing

In the previous sections we presented the syntax and semantics of the Privacy Commands language and how we construct commands and constraints in it from legal policies. In this section we consider the properties of sets of commands and constraints. That is, we shall examine the kinds of properties that we can derive from Privacy APIs and use them to denote equivalence and stricter than relations. We are interested in particular relations because of their versatility. Specifically we explore two action level relations, *strong licensing* and *weak licensing* which not only induce policy equivalence but let us quantify policy permissiveness. The relations in this section relate to the Privacy API at the policy level and so their evaluation is defined at the level of the formal language. The specifics of how we evaluate the relations are discussed later in Chapter 6.

The rest of this section is organized as follows. In Section 5.5.2 we examine the types of relations that we can evaluate over policies and motivate our close examination of two relating to licensing of actions. In Section 5.5.3 we develop a policy equivalence relation with respect to strong licensing. In Section 5.5.4 we develop several short examples to show how we use the relations to compare policies and commands.

5.5.1 Notation and Definitions

In the following discussion, we denote Privacy APIs with the symbol ϕ. We summarize Privacy APIs as collections of operations ω and guards ψ as defined above in Section 5.3. We use the variable e to range over *Command* and c to range over *Constraint*. For convenience, we define the following functions.

We define the following functions to summarize the operations and guards in a command $e \in$ *Command* or constraint $c \in$ *Constraint*:

Definition 5.5.1 (Operations of e) $\mathsf{operations}(e) = \overline{\omega^t} \cup \overline{\omega^f}$ for e as in Definition 5.2.8. □

Definition 5.5.2 (Guards of e and c)

- $\mathsf{guards}(e) = \overline{\psi}$ for e as in Definition 5.2.8.
- $\mathsf{guards}(e) = \overline{\psi^{st}} \cup \overline{\psi^r}$ for c as in Definition 5.2.10.

□

For convenience we denote an ordered *command series* with the bar notation. For instance, $\overline{e} = e_1, e_2, \ldots$ is a command series. Let $s_1, s_2, \ldots \in$ *State* be knowledge states. Let arguments $g_1, g_2 \in \mathit{ParametersE}$, $g_1 = (a_1, s_1, r_1, P_1, f_1, f_1', msg_1), g_2 = (a_2, s_2, r_2, P_2, f_2, f_2', msg_2), \ldots$ summarized in an argument series $\overline{g}$. Then, unless otherwise noted, the following definition holds:

Definition 5.5.3 (Command Series Execution) $s_1 \xrightarrow{\overline{e}(\overline{g})} s_{n+1}$ iff $s_1 \xrightarrow{e_1(g_1)} s_2 \xrightarrow{e_2(g_2)} s_2 \xrightarrow{\ldots} s_{n+1}$. $\square$

In the special case where $g_1 = g_2 = \ldots = g_n$, we write:

Definition 5.5.4 $s_1 \xrightarrow{\overline{e}(g)} s_{n+1}$ iff $s_1 \xrightarrow{e_1(g)} s_2 \xrightarrow{e_2(g)} s_2 \xrightarrow{\ldots} s_{n+1}$. $\square$

Since a Privacy API ϕ contains both commands and constraints we use the superscript notation ϕ^e to refer to the commands in ϕ and ϕ^c to refer to the constraints in ϕ. As in sets, we use the * notation to refer to the (infinite) set of series of members from a set with possible repetitions. For instance ϕ^{e^*} is the set of series of commands derivable from a Privacy API.

We define the following functions to denote the differences between two states $s_1 = (A_1, O_1, m_1, l_{)}, s_2 = (A_2, O_2, m_2, l_2)$. Since we use the relations to quantify the changes necessary to derive one state from another (*i.e.,* to find $\overline{\omega}$ such that $s_1 \xrightarrow{\overline{\omega}} s_2$) we consider just the one way relation between s_1 and s_2:

Definition 5.5.5 (Objects Deleted) $\mathsf{deletedo}(s_1, s_2) = \{o | o \in O_1, o \notin O_2\}$ $\square$

Definition 5.5.6 (Objects Created) $\mathsf{created}(s_1, s_2) = \{o | o \notin O_1, o \in O_2\}$ $\square$

Definition 5.5.7 (Tags Modified) $\mathsf{tagsm}(s_1, s_2) = \{o.t | o \in O_1, o \in O_2, s_1 \vdash o.t \neq s_2 \vdash o.t\}$ $\square$

Definition 5.5.8 (Rights Inserted) $\mathsf{inserted}(s_1, s_2) = \{(a, o, d) | \exists d \in s_1 \vdash m(a, o) \ . \ d \notin s_2 \vdash m(a, o)\}$ $\square$

Definition 5.5.9 (Rights Deleted) $\mathsf{deletedr}(s_1, s_2) = \{(a, o, d) | \exists d \in s_2 \vdash m(a, o) \ . \ d \notin s_1 \vdash m(a, o)\}$ $\square$

We also define the following equivalence relation for logs in order to quantify differences and similarities between logs without respect to the order of the entries. Since logs are built by concatenating string entries, we need a way to delimit where one note ends and another begins. We require that the delimiting character not appear in any log entry, a requirements which varies by the character set used for the policy, so we generalize the delimiting character as δ where δ is not part of the character set used for entries in the log. The function $\mathsf{entries} : \textsc{Log} \rightarrow \textsc{String}^*$ takes a log and divides it into a set of strings by breaking it up by δ. Using the $\mathsf{entries}$ function we define a prefix function for logs as follows:

Definition 5.5.10 (Log Prefix) $\mathsf{prefix}(l_1, l_2)$ iff $\mathsf{entries}(l_1) \subseteq \mathsf{entries}(l_2)$. □

Since we introduce many relations and functions in the section, we begin with a summary of the relations defined in Table 5.12. The different columns indicate the properties of the relations and hint towards their usage.

5.5.2 Policy Relations

A Privacy API is a policy for the permitted and forbidden actions that agents may perform based on the source document. There are many relations and properties of interest that we may explore with relation to them. We are interested in the evaluation of properties, however, that are both decidable and versatile. That is, while some properties and relations are interesting, if we can not construct an evaluation algorithm for them that will eventually terminate it is not as useful as one that will.

The types of relations we define relate to natural questions individuals and stakeholders often ask about legal policies such as "What does this legal policy allow?", "How does it differ from an older version of the policy?", and "Does this policy allow an agent A to perform action B?"

With that restriction in mind we are interested in relations that will let us resolve policy comparison and permissiveness. At the action level we seek to answer the questions "Does a Privacy API ϕ allow the performance of a transition $s \longrightarrow s'$?" and more loosely "Is there a way that ϕ at least performs $s \longrightarrow s''$?" The former question relates to

Table 5.12: Relations between commands, command series, and policies

Relation	Relates	Reflexive	Parameter Lists
$\phi \models_{(s,g)} e$	$(Policy, Command)$	No	1
$\phi \models_{(s)} e$	$(Policy, Command)$	No	All
$\phi \models_{(s,g)} \overline{e}$	$(Policy, Command^*)$	No	1
$\phi \models_{(s)} \overline{e}$	$(Policy, Command^*)$	No	All
$\mathsf{noconflict}_{(s,g)}(e_2, e_1)$	$(Command, Command)$	No	1
$\mathsf{noconflict}_{(s)}(e_2, e_1)$	$(Command, Command)$	No	All
$\mathsf{noconflict}_{(s,g)}(\overline{e}, e_1)$	$Command^*, Command)$	No	1
$\mathsf{noconflict}_{(s)}(\overline{e}, e_1)$	$(Command^*, Command)$	No	All
$\mathsf{noconflict}_{(s,\overline{g})}(\overline{e_2}, \overline{e_1})$	$(Command^*, Command^*)$	No	1
$\mathsf{noconflict}_{(s)}(\overline{e_2}, \overline{e_1})$	$(Command^*, Command^*)$	No	All
$\phi \models^*_{(s,g)} e$	$(Policy, Command)$	No	1
$\phi \models^*_{(s)} e$	$(Policy, Command)$	No	All
$\phi \models^*_{(s,g)} \overline{e}$	$(Policy, Command^*)$	No	1
$\phi \models^*_{(s)} \overline{e}$	$(Policy, Command^*)$	No	All
$\phi_1 \dot{\sim}_{(s,g)} \phi_2$	$(Policy, Policy)$	Yes	1
$\phi_1 \dot{\sim}_{(s)} \phi_2$	$(Policy, Policy)$	Yes	All
$\phi_1 \dot{\sim} \phi_2$	$(Policy,\ Policy)$	Yes	All
$\phi_1 \sim^{e}_{(s,g)} \phi_2$	$(Policy, Policy, Command)$	Yes	1
$\phi_1 \sim^{e}_{(s)} \phi_2$	$(Policy, Policy, Command)$	Yes	All
$\phi_1 \sim^{\overline{e}}_{(s,g)} \phi_2$	$(Policy, Policy, Command^*)$	Yes	1
$\phi_1 \sim^{\overline{e}}_{(s)} \phi_2$	$(Policy, Policy, Command^*)$	Yes	All
$\phi_1 \prec_{(s,a)} \phi_2$	$(Policy, Policy)$	No	1
$\phi_1 \prec_{(s)} \phi_2$	$(Policy, Policy)$	No	All

strict performance: is there a combination of commands that can be executed to perform precisely the desired transition from s to s'. The latter question is more relaxed: is there a combination of commands that will lead to the effects of the transition being performed, even if s'' includes extra operations not included in the transition. Note that we are performing our comparison based on the outcome of a command, not the precise format of the command which led to the transition in question. Such relations are interesting since they are essential to common understanding of legal policies as they are applied without respect to their phraseology or structure.

At the primary level, our goal is to be able to answer the questions "Are Privacy

APIs ϕ_1 and ϕ_2 equivalent?" and "Is ϕ_1 stricter than ϕ_2?" Relations that answer these questions are useful in examining the properties of particular legal policies and evaluating whether one fulfilling one policy is sufficient to fulfill another. At the secondary level we are interested in evaluating the more generic property of "Does compliance with a policy ϕ_1 imply compliance with a policy ϕ_2?"

For two policies ϕ_1 and ϕ_2, since each command may be executed on its own, policy merging is a pairwise union of the two command sets provided that there are no name collisions. In case of name clashes, alpha-renaming one policy will resolve the conflict. The resulting policy allows anything that was permitted under either input policy.

In order to concretize what is permitted by a policy we define two metrics for policies, strong and weak licensing. A transition strong licensed by ϕ is permitted by it directly. A transition weakly licensed by ϕ is permitted by it, so long as some other actions are performed as well. We develop the relations as an effort at translating the notions of strong and weak bisimulation from process calculi. The process calculi definitions are useful as building blocks for comparing running and interactive systems, a useful paradigm to apply to applying policy requirements.

Strong Licensing

If a policy ϕ has a command which can precisely perform the effects of another command e at state $s \in \mathit{State}$ with parameters $g \in \mathit{ParametersE}, g = (a, s, r, P, f, f', msg)$ as above in Section 5.2.3, we say it *strongly licenses* it at s with g, denoted $\phi \models_{(s,g)} e$. Let s_1, s_2 be states such that $s_1 \xrightarrow{e(g)} s_2$.

First, let us consider a limited definition of strong licensing: when a policy can precisely perform the operations of e from just a given initial state s_1 and argument set g. Then we write:

$$\phi \models_{(s_1,g)} e \text{ iff } s_1 \xrightarrow{e(g)} s_2 \implies \exists e' \in \phi \;.\; s_1 \xrightarrow{e'(g)} s_2$$

If we generalize for all arguments at a given state we have a more flexible definition for strong licensing:

Definition 5.5.11 (Strong Licensing) $\phi \models_{(s)} e$ iff $\forall g \in \mathit{ParametersE}, s_1 \xrightarrow{e(g)} s_2 \implies \exists e' \in \phi \;.\; s_1 \xrightarrow{e'(g)} s_2$ $\square$

Generalizing from individual commands to series of commands, if ϕ can precisely perform the effects of all commands in a series $\overline{e}$ from an initial state $s_1 = (A, O, m, l)$ with parameters $\overline{g} \in \mathit{ParametersE}^*$, we say that ϕ *strongly licenses* the series at s_1 with parameter list $\overline{g}$ ($\phi \models_{(s_1,\overline{g})} \overline{e}$). Let us denote the resulting state from the execution of $\overline{e}$ as $s_1 \xrightarrow{\overline{e}(\overline{g})} s_2$. From a single state we have the following:

$$\phi \models_{(s_1,\overline{g})} \overline{e} \text{ iff } s_1 \xrightarrow{\overline{e}(\overline{g})} s_2 \implies \exists \overline{e_2} \in \phi^{e^*} \;.\; s_1 \xrightarrow{\overline{e_2}(\overline{g})} s_2.$$

with the side condition that $|\overline{e}| = |\overline{e_2}| = |\overline{g}|$. Generalizing for all arguments lists combinations at a given state:

Definition 5.5.12 (Strong Licensing Series) $\phi \models_{(s_1)} \overline{e}$ iff $\forall \overline{g} \in \mathit{ParametersE}^*, s_1 \xrightarrow{\overline{e}(\overline{g})} s_2 \implies \exists \overline{e_2} \in \phi^* \;.\; s_1 \xrightarrow{\overline{e_2}(\overline{g})} s_2$, requiring that $|\overline{e}| = |\overline{g}| = |\overline{e_2}|$ and finite. $\square$

Since $\models_{(s)}$ parameterizes over ϕ^{e^*} it is an infinite relation and may not be decidable in general. We may be certain, however, that the relation is decidable for finite series of commands. The argument term $\overline{g}$ also ranges over the infinite set $\mathit{ParametersE}^*$, but since its length is fixed by the $\overline{e}$, for finite series of commands the quantification $\forall \overline{g}$ will also be finite and decidable provided that *Agent*, *Object* and *Purpose* are finite.

Strong licensing corresponds to a policy precisely performing the actions of a command. It also restricts the relationship between the policy and the command(s) to be "in lockstep" meaning that for e or each $e \in \overline{e}$, the policy has one command which can precisely perform e's behavior with the same arguments. It is also useful to define a weaker relation for policies which perform the actions of a command with some slight modifications. In order to define that relation, however, we must first define what types of "slight modifications" we mean.

Non-conflicting Commands

In order to generalize our relations, let us define non-conflicting commands. We define our relations of non-conflict in terms of commands and command series rather than states in order to let us develop relations between policies rather than outcomes, but since they are defined in terms of how commands change states based on input, it is straightforward to adapt them to purely state focussed relations. Doing so would enable analysis and comparison of knowledge states independent of the policies that operate over them.

Let $e_1, e_2 \in \mathit{Command}$, $g \in \mathit{ParametersE}$. $g = (a, s, r, P, f, f', msg)$, and $s \in \mathit{State}$. Let us denote the state reached by invoking e_1 on $s = (A, O, m, l)$ with parameters g as $s \xrightarrow{e_1(g)} s_1$ where $s_1 = (A_1, O_1, m_1, l_1)$ and the state reached by invoking e_2 on s with arguments g as $s \xrightarrow{e_2(g)} s_2$ where $s_2 = (A_2, O_2, m_2, l_2)$.

First let us give a limited definition for non-conflicting commands at a given state and argument set: when e_2 does not conflict with e_1 at a given state s with arguments g (denoted $\mathsf{noconflict}_{(s,g)}(e_2, e_1)$).

Definition 5.5.13 (Non-conflicting Commands) $\mathsf{noconflict}_{(s,g)}(e_2, e_1)$ iff

(a) $O_2 \cap \mathsf{deletedo}(s, s_1) = \emptyset \wedge$

(b) $\mathsf{created}(s, s_1) \subseteq O_2 \wedge$

(c) $\forall o.t \in \mathsf{tagsm}(s, s_1), s_2 \vdash o.t = s_1 \vdash o.t \wedge$

(d) $\forall (a, o, d) \in \mathsf{deletedr}(s, s_1), d \notin s_2 \vdash m(a, o) \wedge$

(e) $\forall (a, o, d) \in \mathsf{inserted}(s, s_1), d \in s_2 \vdash m(a, o) \wedge$

(f) $\mathsf{prefix}(l_2, l_1)$.

$\square$

The intuition for the definition is that the state reached by e_2 does not conflict with the state reached by e_1 if (a) the objects deleted by e_1 are also deleted by e_2, (b) the objects created by e_1 are also created by e_2, (c) the tags modified by e_1 have the same values as modified by e_2 , (d) the rights deleted by e_1 are also deleted by e_2, (e) the rights added by e_1 are also added by e_2, and (f) the log entries added by e_1 are also added by e_2 without respect to their order added. Note that $\mathsf{noconflict}_{(s,g)}(e_2, e_1) \not\Rightarrow \mathsf{noconflict}_{(s,g)}(e_2, e_2)$.

Generalizing for all arguments at an initial state, we have the following definition of non-conflicting commands. To denote that e_2 does not conflict with e_1, we write:

Definition 5.5.14 (Non-conflicting Commands) $\mathsf{noconflict}_{(s)}(e_2, e_1) = \mathit{true}$ iff $\forall g \in \mathit{ParametersE}$. $\mathsf{noconflict}_{(s,g)}(e_2, e_1)$. $\square$

The intuition for the definition is that the state reached by e_2 does not conflict with the state reached by e_1 if the tags modified by e_1 have the same values in as modified by e_2, the objects deleted by e_1 are also deleted by e_2, the rights deleted by e_1 are also deleted by e_2, the rights added by e_1 are also added by e_2, and the log entries added by e_1 are also added by e_2 without respect to their order added. Note that the relation is not necessarily reflexive, so $\mathsf{noconflict}(e_2, e_1) \not\Rightarrow \mathsf{noconflict}(e_2, e_2)$.

We may also generalize the definition of non-conflicting commands to comparing one command to a series of commands. Let $e_1, e_2, \ldots, e_n$ be commands. Let us denote the series of commands $\overline{e} = e_2, e_3, \ldots, e_n$. Let $g \in \mathit{ParametersE}$ and $s = (A, O, m, l)$ be as above. Let us denote the state reached by invoking e_1 on $s = (A, O, m, l)$ with arguments g as $s \xrightarrow{e_1(g)} s_1$ where $s_1 = (A_1, O_1, m_1, l_1)$ and the state reached by invoking $e_2, e_3, \ldots, e_n$ on s with arguments g as $s \xrightarrow{\overline{e}(g)} s_2$ where $s_2 = (A_2, O_2, m_2, l_2)$. We restrict the relation to a single argument list for the entire series $\overline{e}$ to restrict the comparison to similar inputs. Otherwise, for instance, if e_1 performs an action for purpose p_1 which may only be performed by e_2 for purpose $p_2 \neq p_1$, it is not possible in general to tell whether the two should be equivalent (*e.g.,* e_1 permits disclosure of information for marketing while e_2 permits the same disclosure only for emergency services).

Since we have defined nonconflicting commands in terms of their outcome on the initial state, our limited definition for a single state s and arguments g is identical to Definition 5.5.13: $\overline{e}$ does not conflict with e_1 at s with g (denoted $\mathsf{noconflict}_{(s_1,g)}(\overline{e}, e_1)$) when:

$$\mathsf{noconflict}_{(s,g)}(\overline{e}, e_1) = \mathit{true} \text{ iff } O_2 \cap \mathsf{deletedo}(s, s_1) = \emptyset,\ \mathsf{created}(s, s_1) \subseteq O_2,$$
$$\forall o.t \in \mathsf{tagsm}(s, s_1), s_2 \vdash o.t = s_1 \vdash o.t,\ \forall (a, o, d) \in \mathsf{inserted}(s, s_1)\ .\ d \in s_2 \vdash m(a, o),$$
$$\forall (a, o, d) \in \mathsf{deletedr}(s, s_1)\ .\ d \notin s_2 \vdash m(a, o),\ \mathsf{prefix}(l_1, l_2).$$

Generalizing for all arguments lists at a given initial state, we have the following definition of non-conflicting command series similar to Definition 5.5.14. $\overline{e}$ does not conflict with e_1 if the following relation holds:

Definition 5.5.15 (Non-conflicting Commands Series) $\mathsf{noconflict}_{(s)}(\overline{e}, e_1) = \textit{true}$ iff $\forall g \in \textit{ParametersE}$. $\mathsf{noconflict}_{(s,g)}(\overline{e}, e_1)$. □

We generalize the definitions to comparison of the results of series to other series. Let $\overline{e_1}, \overline{e_2}$ be commands series. Let us denote the series of commands $\overline{e_1} = e_{1_1}, e_{1_2}, \ldots$, $\overline{e_2} = e_{2_1}, e_{2_2}, \ldots$. Let $\overline{g} \in \textit{ParametersE}^*$ and $s = (A, O, m, l)$ be as above. Let us denote the state reached by invoking $\overline{e_1}$ on $s = (A, O, m, l)$ with arguments $\overline{g}$ as $s \xrightarrow{\overline{e_1}(\overline{g})} s_1$ where $s_1 = (A_1, O_1, m_1, l_1)$ and the state reached by invoking $\overline{e_2}$ on s with arguments $\overline{g}$ as $s \xrightarrow{\overline{e_2}(\overline{a})} s_2$ where $s_2 = (A_2, O_2, m_2, l_2)$.

Our limited definition for a single state s and arguments g is then similar again to Definition 5.5.13. $\overline{e_2}$ does not conflict with $\overline{e_1}$ at s with g (denoted $\mathsf{noconflict}_{(s_1,a)}(\overline{e_2}, \overline{e_1})$) when:

$$\mathsf{noconflict}_{(s,g)}(\overline{e_2}, \overline{e_1}) = \textit{true} \text{ iff } O_2 \cap \mathsf{deletedo}(s, s_1) = \emptyset,\ \mathsf{created}(s, s_1) \subseteq O_2,$$
$$\forall o.t \in \mathsf{tagsm}(s, s_1), s_2 \vdash o.t = s_1 \vdash o.t,\ \forall (a, o, d) \in \mathsf{inserted}(s, s_1)\ .\ d \in s_2 \vdash m(a, o),$$
$$\forall (a, o, d) \in \mathsf{deletedr}(s, s_1)\ .\ d \notin s_2 \vdash m(a, o),\ \mathsf{prefix}(l_1, l_2).$$

Generalizing for all initial arguments at a given initial state, we have the following definition about $\overline{e_2}$ not conflicting with $\overline{e_1}$:

Definition 5.5.16 (Series Non-conflicting with Series) $\mathsf{noconflict}_{(s)}(\overline{e_2}, e_1) = \textit{true}$ iff $\forall \overline{g} \in \textit{ParametersE}^*$. $\mathsf{noconflict}_{(s,\overline{g})}(\overline{e_2}, \overline{e_1})$. □

In summary, we have defined relations for describing whether commands and commands series are non-conflicting. We use the definitions for concretizing what we meant above by "slight modifications": that if the resulting states from two commands or command series are non-conflicting, we may consider one close enough to the other to be equivalent, which leads us to our definition of weak licensing.

Weak Licensing

Using the above definition for non-conflicting commands and commands series, let us define a more flexible relation between a command and a policy. We define weak licensing in terms of a policy weakly licensing a command rather than one command weakly licensing another since we are interested in resolving questions such as whether a policy permits the actions performed by a single command from one or many states. The definitions can be applied to the case of one command weakly licensing another by considering cases of $|\phi^e| = 1$.

If a policy ϕ can perform the effects of a command e at state s with arguments g with a command series which is not conflicting with e, we say ϕ *weakly licenses* it at s with g, denoted $\phi \models^*_{(s,g)} e$. For a single state we would like to write:

$$\phi \models^*_{(s_1,g)} e \text{ iff } \exists \overline{e} \in \phi^{e^*} \text{ . } \mathsf{noconflict}_{(s_1,g)}(\overline{e}, e_1).$$

The problem with the above definition is that it is undecidable since ϕ^{e^*} is unbounded. We may use a model checker to explore the space and check if the policy reaches a fixed point. Otherwise, for decidability we focus on the more limited power set of commands:

$$\phi \models^*_{(s_1,g)} e \text{ iff } \exists \overline{e} \in \mathsf{pwr}(\phi^e) \text{ . } \mathsf{noconflict}_{(s_1,g)}(\overline{e}, e_1).$$

The intuition for the limitation is that in deciding whether an action is permitted it is sufficient to try all possible permissions once. This imposes the (reasonable) assumption on the source policy that permissions are not enabled by repeated performance of the same action.

Generalizing for all arguments at a given state, we write the following if a policy weakly licenses the actions of a command:

Definition 5.5.17 (Weak Licensing) $\phi \models^*_{(s)} e$ iff $\forall g \in \mathit{ParametersE}$, $\exists \overline{e} \in \mathsf{pwr}(\phi^e)$. $\mathsf{noconflict}_{(s,g)}(\overline{e}, e)$. □

Note that $\models \subseteq \models^*$.

If ϕ can perform all of the effects of a series $\overline{e}$ from an initial state $s = (A, O, m, l)$ with parameters $g = (a, s, r, P, f, f', msg)$ with non-conflicting effects as well, we say that

ϕ *weakly licenses* the series at s_1 with $\overline{g}$ ($\phi \models^*_{(s_1,\overline{g})} \overline{e}$). Let us denote the resulting state from the execution of $\overline{e}$ as $s_1 \xrightarrow{\overline{e}(\overline{g})} s_2$. For a single state we have:

$$\phi \models^*_{(s_1,\overline{g})} \overline{e} \text{ iff } \exists \overline{e_2} \in \mathsf{pwr}(\phi^e) \ . \ \mathsf{noconflict}_{(s_1,\overline{g})}(\overline{e_2}, \overline{e}).$$

Generalizing for all argument lists, we have the following:

Definition 5.5.18 (Weak Licensing Series) $\phi \models^*_{(s)} \overline{e}$ iff $\forall \overline{g} \in \mathit{ParametersE}^*$, $\exists \overline{e_2} \in \mathsf{pwr}(\phi^e)$. $\mathsf{noconflict}_{(s,\overline{g})}(\overline{e_2}, \overline{e})$. □

Intuitively, weak licensing for series of commands $\overline{e}$ means that the policy ϕ can execute a series of one or more commands such that the final result is a state which is non-conflicting with the result from $\overline{e}$. We do not look at the intermediate states reached by $\overline{e_2}$, only restricting that they use the same argument list in $\models^*_{(s_1,\overline{g})}$ and Definition 5.5.18. By forcing the two lists to be the same length, we ensure that both relations are decidable since for any $\overline{e}$ chosen, we know precisely that we only have to search for an $\overline{e_2}$ such that $|\overline{e_2}| = |\overline{e}|$.

We use weak licensing to model constraints of the form: "You may do A if it is permitted by ϕ" where ϕ is an external policy or set of commands. We use $\models^*$ to quantify what it means for a policy to permit an action as follows. Let $s, s' \in \mathit{State}$ such that $e \in \mathit{Command}$ with arguments $g \in \mathit{ParametersE}$ yields $s \xrightarrow{e(g)} s'$.

Definition 5.5.19 (Permitted By) $s \xrightarrow{e(g)} s'$ is *permitted by* ϕ *at* s *with* g if $\phi \models^*_{(s,g)} e$. □

The intuition is that ϕ permits $e(g)$ if it allows $e(g)$'s side effects, potentially augmented with other actions. In Chapter 6 we develop an automated technique for resolving $\models^*$ and thereby a mechanism for determining whether actions are permitted by a policy.

A logical extension to the Privacy Commands guards would be a guard of the form $\mathit{Permitted}(\overline{e}(g))$ for $\overline{e} \subset \mathit{Command}$ and $g \in \mathit{ParametersE}$. The guard would then be satisfied if at the current knowledge state s for a policy ϕ with commands $\overline{e}$, $\phi \models^*_{(s,g)} e$. We leave this extension for future work because of the additional complexity for policy resolution that it imposes, making it difficult to translate for automated verification using the technique we have developed.

5.5.3 Policy Comparison

Using strong and weak licensing we make the following observations and prove the following theorems about policies permitting transitions or command series. If $\phi \models s \xrightarrow{e(g)} s'$, then we say ϕ *generates* $e(g)$ *at* s. If $\phi \models \overline{e}$ for a series $\overline{e}$, then we say ϕ *generates* $\overline{e}$. If $\phi \models^* s \xrightarrow{e(g)} s'$, then ϕ *permits* $e(g)$ *at* s, but does not generate it since other actions may be performed as well. If $\phi \models^* \overline{e}$ for a series $\overline{e}$, then there exists an execution of ϕ that *permits* $\overline{e}$, but may perform other actions as well.

Observational Equivalence

In relating policies it is useful to consider when two policies are observationally equivalent. That is, when the behavior of one policy is indistinguishable from the behavior of another. Let ϕ_1, ϕ_2 be policies. Let s be an initial state and e be a command.

Definition 5.5.20 ϕ_1 and ϕ_2 are *observationally equivalent for* s *with* $\overline{g}$ ($\phi_1 \dot{\sim}_{(s,\overline{g})} \phi_2$) for a state s and arguments lists $\overline{g}$ if $\forall \overline{e_1} \in \phi_1^{e^*}, \exists \overline{e_2} \in \phi_2^{e^*} \,.\, s \xrightarrow{\overline{e_1(g)}} s_1 \Leftrightarrow s \xrightarrow{\overline{e_2(\overline{g})}} s_1$ and $\forall \overline{e_2} \in \phi_2^{e^*}, \exists \overline{e_1} \in \phi_1^{e^*} \,.\, s \xrightarrow{\overline{e_2(\overline{g})}} s_2 \Leftrightarrow s \xrightarrow{\overline{e_1(\overline{g})}} s_2$. □

The relation refers to the equivalence of execution from a given initial state and given argument list. The intuition for it is that at a given snapshot and for a desired combination of arguments, any command series in ϕ_1 can be mirrored by some command series in ϕ_2 and vice versa. We relate $\dot{\sim}_{(s,\overline{g})}$ to $\models_{(s,g)}$ by observing that it is equivalent to showing strong licensing for any command series that begins from s and uses the argument series $\overline{g}$:

$$\phi_1 \dot{\sim}_{(s,\overline{g})} \phi_2 \Leftrightarrow (\forall \overline{e_1} \in \phi_1^{e^*}, \phi_2 \models_{(s,\overline{g})} \overline{e_1} \wedge \forall \overline{e_2} \in \phi_2^{e^*}, \models_{(s,\overline{g})} \overline{e_2})$$

The intuition of the equivalence is that in order to evaluate whether two policies are observationally equivalent from a given state s and arguments lists, we must evaluate whether any command series from s is strongly licensed by the other policy. Of course, since the command series may be of infinite length, if the set of states is finite it may be shorter to evaluate observational equivalence by enumerating all of the states reachable from s using either policy. We then can show $\dot{\sim}_{(s,\overline{g})}$ by showing strong licensing for all reachable states from s.

Returning to our definitions for observation equivalence, generalizing for all arguments for observational equivalence we have:

Definition 5.5.21 (State Equivalence) ϕ_1 and ϕ_2 are *observationally equivalent for s* ($\phi_1 \dot{\sim}_{(s)} \phi_2$) for a state s if $\forall g \in \mathit{ParametersE}, \forall \overline{e_1} \in \phi_1^{e^*} \exists \overline{e_2} \in \phi_2^{e^*} \ . \ s \xrightarrow{\overline{e_1}(g)} s_1 \Leftrightarrow s \xrightarrow{\overline{e_2}(g)} s_1$ and $\forall \overline{e_2} \in \phi_2^{e^*} \exists \overline{e_1} \in \phi_1^{e^*} \ . \ s \xrightarrow{\overline{e_2}(g)} s_2 \Leftrightarrow s \xrightarrow{\overline{e_1}(g)} s_2$. □

The State Equivalence relation of Definition 5.5.21 means that at a given state, any command series that ϕ_1 can run can be mirrored by a command series in ϕ_2 and vice versa. Essentially, from the state s, the two policies are equivalent. Generalizing for all states we have: ϕ_1 and ϕ_2 are *observationally equivalent* ($\phi_1 \dot{\sim} \phi_2$) under the following definitions

Definition 5.5.22 (Policy Equivalence) $\phi_1 \dot{\sim} \phi_2$ iff $\forall s \in \mathit{State}, \forall g \in \mathit{ParametersE}, \forall \overline{e_1} \in \phi_1^{e^*} \exists \overline{e_2} \in \phi_2^{e^*} \ . \ s \xrightarrow{\overline{e_1}(g)} s_1 \Leftrightarrow s \xrightarrow{\overline{e_2}(g)} s_1$ and $\forall \overline{e_2} \in \phi_2^{e^*} \exists \overline{e_1} \in \phi_1^{e^*} \ . \ s \xrightarrow{\overline{e_2}(g)} s_2 \Leftrightarrow s \xrightarrow{\overline{e_1}(g)} s_2$. □

The last relation, Policy Equivalence in Definition 5.5.22, generalizes the relation for all states. Policies which are equivalent under Policy Equivalence may be interchanged since any state reachable by one is reachable by the other as well. The relation $\dot{\sim}$ is stronger than our definitions of $\models$ above, but it follows directly from our argument above that $\dot{\sim}$ is equivalent to $(\forall s \in \mathit{State}, \forall \overline{e_1} \in \phi_1^{e^*}, \phi_2 \models_{(s)} \overline{e_1}) \bigwedge (\forall s \in \mathit{State}, \forall \overline{e_2} \in \phi_2^{e^*}, \phi_1 \models_{(s)} \overline{e_2})$.

The relations $\dot{\sim}_{(s,g)}, \dot{\sim}_{(s)}, \dot{\sim}$ are policy comparisons, relating two policies as shown above in Table 5.12. They are applications of strong licensing to policies, giving us metrics for relating what actions both policies permit.

Relative Permissiveness

The relations in the $\dot{\sim}$ family relate policies to each other but it is often useful to discuss relations between policies as they relate to individual commands and commands series. We label the relations in this family with $\sim^e$, showing the that the relationship is based on the behavior of a given command e. The intuition for defining these relations is tied to questions such as "Do two policies agree on this behavior?" In such queries, we are interested in finding out whether the two policies agree on the behavior, either both permitting or both forbidding. Relaxing the relations also leads us to comparative relations such as *at least as strict as* ($\prec$) which lets us compare whether one policy is stricter than another.

We begin by defining the *equal permissiveness* relations, variations of the relation $\sim$. For the following definitions, let e be a command and $\overline{e} = e_1, e_2, \ldots$ a command series. Let ϕ_1, ϕ_2 be policies. Let $g \in \mathit{ParametersE}\ .\ g = (a, s, r, P, f, f', msg)$ be arguments. For a single state s and argument list g we denote the agreement of ϕ_1 and ϕ_2 on command e as follows:

$$\phi_1 \sim^e_{(s,g)} \phi_2 \text{ iff } (\phi_1 \models_{(s,g)} e \text{ iff } \phi_2 \models_{(s,g)} e).$$

The intuition for the relation is that for a given state and argument list, the two policies ϕ_1 and ϕ_2 either both strongly license e or neither does. It implies an equivalence for the two policies with respect to e at s with g. We can generalize the relation for all for all argument lists and a given command at an initial state:

Definition 5.5.23 (Command Equivalence) $\phi_1 \sim^e_{(s)} \phi_2$ iff ($\phi_1 \models_{(s)} e$ iff $\phi_2 \models_{(s)} e$). $\square$

The Command Equivalence relation means that for every argument list at a given state, the two policies either both strongly license the command e or neither does. It is a stronger measure of equivalence for the given command. We can further generalize the relation to command series $\overline{e}$ for any argument set at a given state:

Definition 5.5.24 (Series Equivalence) $\phi_1 \sim^{\overline{e}}_{(s)} \phi_2$ iff ($\phi_1 \models_{(s)} \overline{e}$ iff $\phi_2 \models_{(s)} \overline{e}$). $\square$

Finally, generalizing for all commands, we have a metric for general policy permissiveness:

Definition 5.5.25 (Permissiveness Equality) $\phi_1 \sim_{(s)} \phi_2$ iff $\forall e \in \mathit{Command}\ .\ (\phi_1 \models_{(s)} e$ iff $\phi_2 \models_{(s)} e)$. $\square$

Generalizing Definition 5.5.25 to command series is straightforward and therefore elided. The intuition for the relation is that two policies are equally permissive at a given state s for any command and any argument list. Note that when e or $\overline{e}$ are in one of the policies, the $\dot{\sim}$ and $\sim^e$ relations coincide. That is, for $e \in \phi_1$, $\dot{\sim}_{(s,g)} \Leftrightarrow \sim^e_{(s,g)}$ and $\dot{\sim}_{(s)} \Leftrightarrow \sim^e_{(s)}$ and similarly if $e \in \phi_2$. Also for $\overline{e} \in \phi_1^{e^*}$, $\dot{\sim}_{(s)} \Leftrightarrow \sim_{(s)}^{\overline{e}}$ and similarly for $\overline{e} \in \phi_2^{e^*}$. Since one policy automatically licenses the command(s), the other must as well for $\sim^e$.

The *at least as strict as* relations $\prec$ are a relaxation of $\sim$. Let g, e, $\overline{e}$, ϕ_1, and ϕ_2 be as above. We then define the following relations between two policies about a given command e. The relation $\prec_{(s)}$ describes a relation where the left hand side is possibly more strict than the right hand side. That is, we write $\phi_1 \prec \phi_2$ to indicate that ϕ_1 is potentially less permissive than ϕ_2. We express the relationship in terms of strong licensing, similar to $\sim^e$. That is, we use $\prec$ to indicate if ϕ_1 strongly licensing a command implies that ϕ_2 does too, but not necessarily the opposite. For a given initial state s, a single argument list $g \in \mathit{ParametersE}$, and any given command, we then write:

$$\phi_1 \prec_{(s,g)} \phi_2 \text{ iff } \forall e \in \mathit{Command} \ . \ \phi_1 \models_{(s,g)} e \implies \phi_2 \models_{(s,g)} e.$$

Generalizing for any argument list at the single initial state s, we have:

$$\phi_1 \prec_{(s)} \phi_2 \text{ iff } \forall e \in \mathit{Command} \ . \ \forall g \in \mathit{ParametersE} \ . \ \phi_1 \models_{(s,g)} e \implies \phi_2 \models_{(s,g)} e.$$

Generalizing for series of commands, we have:

Definition 5.5.26 (At least as strict as) $\phi_1 \prec_{(s)} \phi_2$ iff $\forall \overline{e} \in \mathit{Command}^*$. $\forall \overline{g} \in \mathit{ParametersE}^*$. $\phi_1 \models_{(s)} \overline{e} \implies \phi_2 \models_{(s)} \overline{e}$. □

Based on the definition note that, $\phi_1 \prec_{(s)} \phi_2 \not\Rightarrow \phi_2 \prec_{(s)} \phi_1$ but that $\phi_1 \prec_{(s)} \phi_2 \bigwedge \phi_2 \prec_{(s)} \phi_1 \Rightarrow \phi_1 \sim_{(s)} \phi_2$.

The relation $\prec$ is useful since it lets us discover when one policy subsumes another. That is, if everything that ϕ_1 permits can be performed under ϕ_2 as well, ϕ_2 subsumes ϕ_1. Alternatively, we may say that ϕ_1 is at least as strict as ϕ_2 since everything that ϕ_1 permits is permitted by ϕ_2 as well. That which ϕ_1 prohibits (*i.e.,* the operations and transitions that it does not allow to occur) may be allowed by ϕ_2, however.

5.5.4 Applications of Licensing

We use the relations defined above to better analyze policies. The five families of relations are summarized above in Table 5.12 and categorized by whether they relate commands, series, or policies as well as whether they are for a single argument list or parameterized for any argument list. Of the five families, four are directly useful for the following applications:

$\models$ Determining whether an action can be performed by an agent under the jurisdiction of a particular policy.

$\models^*$ Determining whether an action is permitted by a policy, but only as part of a larger operation.

$\dot{\sim}$ Determining whether one policy is equivalent to another in a particular situation.

$\prec$ Determining whether one policy subsumes another.

In Chapter 6 we discuss how we automate the search for $\models$ relationship between commands and policies using SPIN. In Chapter 7 we present case studies for relating policies using the relations as well, showing their flexibility and descriptiveness. As in Section 4.4, before we get to the technical details of automation and the applied case studies, we first develop some simple examples for the use of the relations to explore what kinds of properties can be expressed using the relations.

For the following examples, let us consider the policy snippet extracted from Example 5.2.1 in Section 5.2.6:

> (c) Implementation specifications: Treatment, payment, or health care operations.
>
> (1) A covered entity may use or disclose protected health information for its own treatment, payment, or health care operations.

In Example 5.2.1 we show the constraint for the paragraph along with the commands for treatment use and disclosure. For the following examples, let us consider the three commands for use of information for treatment, payment, health care operations. The full command set is shown in Appendix C.2.2.

```
         CST  Permitted506c1 (a, s, r, P, f, f', msg)
       Scope  {TreatmentUse506c1, PaymentUse506c1,
              HealthCareOperationsUse506c1, TreatmentDisclose506c1,
              PaymentDisclose506c1, HealthCareOperationsDisclose506c1}
   Such That  f.protected-health-information = true
         and  coveredEntity in Roles(a)
         and  individual in Roles(s)
          if  own $\textbf{in}_a$ P
        then  return true
        else  return false
```

```
 CMD  TreatmentUse506c1 (a, s, r, P, f, f', msg)
  if  Permitted506c1 (a, s, r, P, f, f', msg) $\in$ {Allow}
 and  local in (a, f)
 and  use $\textbf{in}_a$ P
 and  treatment $\textbf{in}_a$ P
then  insert treatment in (a, s)
 and  return true
else  return false
```

```
 CMD  PaymentUse506c1 (a, s, r, P, f, f', msg)
  if  Permitted506c1 (a, s, r, P, f, f', msg)
 and  local in (a, f)
 and  use $\textbf{in}_a$ P
 and  payment $\textbf{in}_a$ P
then  insert payment in (a, s)
 and  return true
else  return false
```

```
CMD   HealthCareOperationsUse506c1 (a, s, r, P, f, f', msg)
 if   Permitted506c1 (a, s, r, P, f, f', msg)
and   local in (a, f)
and   use $\mathbf{in}_a$ P
and   healthCareOperations $\mathbf{in}_a$ P
then  insert healthCareOperations in (a, s)
and   return true
else  return false
```

The three commands TreatmentUse506c1, PaymentUse506c1, and HealthCareOperationsUse506c1 share a common structure, beginning with a reference to the constraint Permitted506c1 and followed by checks that the actor has the right local on the object and that the purpose of the action is use. The last guard for each command is tailored to the purpose of the command: treatment, payment, and health care operations respectively. The inclusion uses permitted semantics since any child purpose of treatment, payment, or health care operations will suffice as well. The true branch for each command stores new the right for the agent on the subject and then returns true. The false branch simply returns false without any other operations.

Let us now define a policy ϕ_1 which contains just the above constraint and commands. Let C = {Permitted506c1}, E = {Treatment506c1, Payment506c1, HealthCareOperations506c1}, R = {coveredEntity}, T = {protected-health-information}, P = {use, treatment, payment, healthCareOperations, own}, and *Right* = {local, treatment, payment, healthCareOperations}. Let $\phi_1 = (C, E, R, T, P)$.

Example 5.5.1 (Strong Licensing)

Strong licensing gives a mechanism for comparing a command or command series against a given policy. The policy from HIPAA [§164.506(c)(1), v.2003] permits covered entities to use protected health information for treatment, payment, or health care operations. Let us consider the policy of a US state which decides to require covered entities to acquire consent from patients before using protected health information for payment purposes:

A covered entity may use or disclose protected health information for its own payment purposes with patient consent.

In order to compare the state's policy the HIPAA policy, we translate the policy statement into a command Strong1:

```
 1   CMD   Strong1(a, s, r, P, f, f', msg)
 2    if   f.protected-health-information = true
 3   and   coveredEntity in Roles(a)
 4   and   individual in Roles(s)
 5   and   own $\mathbf{in}_a$ P
 6   and   local in (a, f)
 7   and   use $\mathbf{in}_a$ P
 8   and   payment $\mathbf{in}_a$ P
 9   and   consent in (a, s)
10  then   insert payment in (a, s)
11   and   return true
12  else   return false
```

Since we are considering only Strong1 and not any other constraints on it, Strong1 contains no guard references, only direct examinations of state. The guards for Strong1 are identical to those of Permitted506c1 and PaymentUse506c1 with the addition of a guard on line 9. Line 9 checks that the actor has the right "consent" on the subject which indicates that the subject has granted consent to the actor.

Strong1 uses the same sets as ϕ_1 for roles (R), tags (T), and purposes (P), but has an additional right for consent. Thus *Right* = {treatment, payment, healthCareOperations, consent}. We consider several states to explore how ϕ_1 relates to Strong1. Let Cy work for a covered entity and Ike be an individual. Let there be an object f which is a file which contains information about Ike. Let us define the following invariant state: s_i = ({Cy, Ike}, {f}, $\emptyset$, ϵ) such that Roles(Cy) = {coveredEntity}, Roles(Ike) = {individual}, f.protected-health-information = true.

With Consent In a case Ike has granted consent to Cy for access to protected health information, the knowledge state is as follows: $s_1 = s_i$ except that m(Cy, Ike) =

{consent} and $m(\text{Cy}, f) = \{\text{local}\}$.

Without Consent In a case where Ike has not granted consent to Cy, the knowledge state is as follows: $s_2 = s_i$ except that $m(\text{Cy}, f) = \{\text{local}\}$.

Not Local In a case where the file is not local to Cy, but Ike has granted consent for Cy to see his records, the knowledge state is as follows: $s_3 = s_i$ except that $m(\text{Cy}, \text{Ike}) = \{\text{consent}\}$.

For each of the states s_1, s_2, and s_3 let us consider the following argument list: $g_1 =$ (Cy, Ike, Cy, {use, payment, own}, f, $\emptyset$, ϵ). For brevity, let us denote Strong1 as e_1 and Strong1(g_1) as $e_1(g_1)$. Let us denote Strong1(g_1) as $e(g_1)$. The following relations then hold for each state with g_1:

1. $\phi_1 \models_{(s_1,g_1)} e_1$. Running $e_1(g_1)$ at s_1 yields $s_1' = (A_1', O_1', m_1', l_1')$ as shown in the following table:

s_1	$\xrightarrow{e_1(g_1)}$	s_1'
$A_1 = \{\text{Cy, Ike}\}$		$A_1' = \{\text{Cy, Ike}\}$
$O_1 = \{f\}$		$O_1' = \{f\}$
$m_1(\text{Cy}, \text{Ike}) =$ $\{\text{consent}\}$ $m_1(\text{Cy}, f) =$ $\{\text{local}\}$		$m_1'(\text{Cy}, \text{Ike}) =$ $\{\text{consent}, \text{payment}\}$ $m_1'(\text{Cy}, f) =$ $\{\text{local}\}$
$l_1 = \epsilon$		$l_1' = \epsilon$

Since ϕ_1 can perform the same transition by running $s_1 \xrightarrow{e'(g_1)} s_1'$ for e' =PaymentUse50c1, it strongly licenses Strong1 at s_1 with g_1.

2. $\phi \models_{(s_1)} e$. Running e_1 at s_1 with any $g \in \mathit{ParametersE}$ yields the same result as running PaymentUse506c1. This is proven directly since for $g = (a, s, r, P, f, f', msg)$, if a =Cy and s =Ike, then line 8 of Strong1 is satisfied. Since the rest of the guards coincide with Payment506c1 and Permitted506c1, their results are identical. For any

other values of g and s, if $a \neq$Cy, line 3 of Strong1 and line 6 of Permitted506c1 will not be satisfied and if $s \neq$Ike, line 4 of Strong1 and line 7 of Permitted506c1 will not be satisfied. Since the guards otherwise coincide, the results of the two are the same and therefore ϕ_1 strongly licenses Strong1 at s_1 for any arguments.

3. $\phi_1 \models_{(s_2,g_1)} e_1, \phi_1 \not\models_{(s_2)} e_1$. Running $e_1(g_1)$ at s_2 yields $s_2' = (A_2', O_2', m_2', l_2')$ as shown in the following table:

s_2	$\xrightarrow{e_1(g_1)}$	s_2'
$A_2 = \{$ Cy, Ike$\}$		$A_2' = \{$Cy, Ike$\}$
$O_2 = \{f\}$		$O_2' = \{f\}$
$m_2(\text{Cy}, f) =$ $\{$local$\}$		$m_2'(\text{Cy}, f) =$ $\{$local$\}$
$l_2 = \epsilon$		$l_2' = \epsilon$

Since e_1 does not cause any changes to s_2 with g_1 or any $g \in \mathit{ParametersE}$, it is trivially strongly licensed by any non-satisfied command which has no operations in its false branch. In ϕ_1, Treatment506c1 also is not satisfied at s_2 with g_1, so $\phi_1 \models_{(s_2,g_1)} e_1$. The trivial licensing does not hold for all arguments, however, since for $g_2 = (\text{Cy}, \text{Ike}, \text{Cy}, \{\text{treatment, payment, healthCareOperations, use, own}\}, f, \emptyset, \epsilon\}$, every command in ϕ_1 is satisfied so there is no command which does not cause changes to s_2. Therefore, $\phi_1 \not\models_{(s_2)} e_1$.

4. $\phi \models_{(s_3,g_1)} e_1, \phi_1 \models_{(s_3)} e_1$. Running $e_1(g_1)$ at s_3 $s_3' = (A_3', O_3', m_3', l_3')$ as shown in the following table:

s_3	$\xrightarrow{e_1(g_1)}$	s_3'
$A_3 = \{$Cy, Ike$\}$		$A_3' = \{$Cy, Ike$\}$
$O_3 = \{f\}$		$O_3' = \{f\}$
$m_3(\text{Cy}, \text{Ike}) =$ $\{$consent$\}$		$m_3'(\text{Cy}, \text{Ike}) =$ $\{$consent$\}$
$l_3 = \epsilon$		$l_3' = \epsilon$

As in 3, since Strong1 does not cause any updates to the state at s_3 with g_1 or any $g \in \mathit{ParametersE}$, it is trivially licensed by any non-satisfied command which has no operations in its false branch. Treatment506c1 also is not satisfied for s_3 with g_1 and so it strongly licenses it. Since local $\not\in (\mathrm{Cy}, f)$, no command in ϕ_1 is satisfied for any $g \in \mathit{ParametersE}$ and therefore $\phi_1 \models_{(s_3)} e_1$.

□

The above example shows how strong licensing allows the evaluation of whether a command can be modeled precisely by another policy. Note that as shown with s_2, $\models$ may be trivially true for a command which is not satisfied and has no side effects on its false branch. When a command has no side effects, any other command with no side effects will strongly license it.

Example 5.5.2 (Weak Licensing)

Weak licensing gives a mechanism for comparing a command or command series against a given policy with some leeway given for operations which do not conflict. As in Example 5.5.1, let us consider the policy of a US state whose policy differs slightly from HIPAA's. The state decides that requiring consent for payment is too restrictive for covered entities and therefore decides that instead, whenever a covered entity gains the right to use protected health information for payment, the individual is given the right to audit how the information was used:

> A covered entity may use or disclose protected health information for its own payment purposes and the individual may audit its use.

The modified command for the policy is:

```
1   CMD   Weak1(a, s, r, P, f, f', msg)
2    if   f.protected-health-information = true
3   and   coveredEntity in Roles(a)
4   and   individual in Roles(s)
5   and   own in_a P
6   and   local in (a, f)
7   and   use in_a P
8   and   payment in_a P
9   then  insert payment in (a, s)
10  and   insert audit in (s, a)
11  and   return true
12  else  return false
```

As in Example 5.5.1, since we are considering only Weak1 and not any other constraints on it, Weak1 contains no guard references. The guards for Weak1 are identical to those of Permitted506c1 and PaymentUse506c1. Weak1 differs from Payment506c1 on line 10 with the addition of the right "audit" to the subject.

As in Strong1, Weak1 uses the same sets as ϕ_1 for roles (R), tags (T), and purposes (P), but has an additional right for auditing. Thus *Right* = {treatment, payment, healthCare-Operations, audit}. Let us denote Weak1 as e_2, and let us consider two states to explore how ϕ_1 relates to e_2. As in Example 5.5.1, let Cy work for a covered entity and Ike be an individual, and f be an object which contains information about Ike. We define the following invariant state: $s_i = (\{\text{Cy, Ike}\}, \{\text{f}\}, \emptyset, \epsilon)$ such that Roles(Cy) = {coveredEntity}, Roles(Ike) = {individual}, f.protected-health-information = true.

Payment In a case Cy seeks to access local protected health information for payment, the knowledge state is as follows: $s_1 = s_i$ except that $m(\text{Cy}, f) = \{\text{local}\}$.

Non-Local In a case where Cy seeks to access non-local protected health information for payment, the knowledge state is as follows: $s_2 = s_i$.

For each of the states s_1 and s_2 let us consider the following argument list: $g_1 = $ (Cy, Ike, Cy, {use, payment, own}, f, $\emptyset$, ϵ). Let us denote Weak1 as e_2 and therefore Weak1(g_1)

as $e_2(g_1)$. The following relations then hold for each state with g_1: The following relations then hold for each state with g_1:

1. $\mathsf{noconflict}_{(s_1,g_1)}(\text{Payment506c1}, e_2)$,$\mathsf{noconflict}_{(s_1)}(\text{Payment506c1}, e_2)$. Since the guards for Payment506c1 and Weak1 are identical, they are both satisfied at s_1 with g_1. Running $e_2(g_1)$ at s_1 yields $s_1' = (A_1', O_1', m_1', l_1')$ as shown in the following table:

s_1	$\xrightarrow{e_2(g_1)}$	s_1'
$A_1 = \{\text{Cy, Ike}\}$		$A_1' = \{\text{Cy, Ike}\}$
$O_1 = \{f\}$		$O_1' = \{f\}$
$m_1(\text{Cy}, f) = \{\text{local}\}$		$m_1'(\text{Cy}, f)) = \{\text{local}\}$ $m_1'(\text{Cy}, \text{Ike}) = \{\text{payment}\}$ $m_1'(\text{Ike}, \text{Cy}) = \{\text{audit}\}$
$l_1 = \epsilon$		$l_1' = \epsilon$

Running Payment506c1(g_1) at s_1 yields $s_1'' = (A_1'', O_1'', m_1'', l_1'')$:

s_1	$\xrightarrow{\text{Payment506c1}(g_1)}$	s_1''
$A_1 = \{\text{Cy, Ike}\}$		$A_1'' = \{\text{Cy, Ike}\}$
$O_1 = \{f\}$		$O_1'' = \{f\}$
$m_1 = (\text{Cy}, f) = \{\text{local}\}$		$m_1''(\text{Cy}, f)) = \{\text{local}\}$ $m_1''(\text{Cy}, \text{Ike}) = \{\text{payment}\}$
$l_1 = \epsilon$		$l_1'' = \epsilon$

The conditions for $\mathsf{noconflict}$ are satisfied: as defined in Definition 5.5.13:

(a) $\mathsf{deletedo}(s_1, s_1'') = \emptyset \cap O_1' = \emptyset$

(b) $\mathsf{created}(s_1, s_1'') = \emptyset \subseteq O_1'$

(c) $\mathsf{tagsm}(s_1, s_1'') = \emptyset$, so nothing needs to be compared

(d) $\mathsf{deletedr}(s_1, s_1'') = \emptyset$ so nothing needs to be compared

(e) $\mathsf{inserted}(s_1, s_1'') = \{(\text{Cy, Ike, }\{\text{payment}\}\}$ and $m_1'(\text{Cy, Ike}) = \{\text{payment}\}$

(f) $l_1' \equiv l_1''$ so $\mathsf{prefix}(l_1'', l_1')$

A similar argument shows that for any argument set $g \in \mathit{ParametersE}$, either both Payment506c1 and Weak1 are satisfied in which case the resulting states are as described or neither are satisfied in which case $\mathsf{noconflict}$ is trivially true.

2. $\phi_1 \models^*_{(s_1,g_1)} e_2, \phi_1 \models^*_{(s_1)} e_2$. Since we have shown in 1 that $\mathsf{noconflict}_{(s_1)} e_2$, we have by Definition 5.5.17 that ϕ_1 weakly licenses e_2 if we choose $\overline{e}$ = Payment506c1. Since $\mathsf{noconflict}$ is true for any $g \in \mathit{ParametersE}$ for s_1, we have that ϕ_1 weakly licenses Weak1 for any arguments at s_1.

3. $\mathsf{noconflict}_{(s_2,g_1)}(\text{Payment506c1}, e_2)$,$\mathsf{noconflict}_{(s_2)}(\text{Payment506c1}, e_2)$. Since the guards for Payment506c1 and Weak1 are not satisfied at s_2 for g_1, they do not perform any updates to s_2 and therefore $\mathsf{noconflict}_{(s_2,g_1)}$ holds trivially. A similar argument shows that for any $g \in \mathit{ParametersE}$, $\mathsf{noconflict}_{(s_2)}$ is also trivially true.

4. $\phi_1 \models^*_{(s_2,g_1)} e_2, \phi_1 \models^*_{(s_2)} e_2$. Since we have shown in 3 that $\mathsf{noconflict}_{(s_1)} e_2$, weak licensing is true in an argument similar to 2.

Since $\phi_1 \models^*_{(s_1)} e_2$ and $\phi_1 \models^*_{(s_2)} e_2$, the policy quoted above is licensed by HIPAA [§164.5069(c)(1), v.2003]. As in Definition 5.5.19, HIPAA therefore permits the above policy. The additional requirement of Weak1 to give individuals the right to audit the covered entity makes the policy in fact stricter since it places more requirements on the covered entity than HIPAA does. We could, however, easily come upon an example where the additional action would make the policy more lenient.

□

Example 5.5.2 shows how we can use weak licensing to show that a command is permitted by policy. We also see that weak licensing, like strong licensing, may be true in trivial cases where the command in question is not satisfied. Since we define $\models^*$ in terms of noconflict, when checking for weak licensing it is logical to first check whether there exists any commands which do not conflict. If non-conflicting commands are in the policy, we can easily derive $\models^*$ relationships.

Example 5.5.3 (Observational Equivalence)

Examples 5.5.1 and 5.5.2 showed relations between individual commands and a policy. Let us now consider one of the policy-to-policy relations: $\dot{\sim}$.

Let us return to the command in Example 5.5.1 and let us embed it in a policy ϕ_2 such that $\phi_2 = (\emptyset, \{\text{Strong1}\})$. Let us reuse the state s_1 above and examine the relationship between ϕ_1 and ϕ_2 with respect to $\dot{\sim}$.

1. $\phi \dot{\nsim}_{(s_1,g_1)} \phi_2$. To show $\dot{\sim}$, as per Definition 5.5.20, we must show that any transition which can be taken by ϕ_2 can be taken by ϕ_1 and vice versa. First, we have shown in Example 5.5.1 that $\phi \models_{(s_1,g_1)}$ Strong1, so we have that the transition by ϕ_2 can be performed by ϕ_1. Second, for s_1 at g_1, TreatmentUse506c1 and HealthCareOperationsUse506c1 do not perform any updates and therefore $s_1 \xrightarrow{\text{TreatmentUse506c1}(g_1)} s_1$ and similarly for HealthCareOperationsUse506c1. However, since Strong1(a_1) causes a state update at s_1 with g_1 and there is no other command in ϕ_2, there is no command that it can perform which does not cause an update. Therefore, the two are not observationally equivalent.

The result for ϕ_2 is disappointing since we can intuitively see that ϕ_1 and ϕ_2 are very similar, but $\dot{\sim}$ fails. If we augment ϕ_2 with a trivial and unsatisfiable command Empty(a, s, r, P, f, f', msg), however, we can get the intuitive results:

```
 CMD  Empty(a, s, r, P, f, f', msg)
   if  false
 then  return true
 else  return false
```

Let $\phi_2 = (\emptyset, \{\text{Strong1}, \text{Empty}\})$ and let us reconsider the state s_1 from Example 5.5.1:

1. $\phi \dot{\sim}_{(s_1,g_1)} \phi_2$. Returning to the case above, we have shown that $\phi \models_{(s_1,g_1)}$ Strong1. For Empty(g_1), since it is not satisfied for g_1, it does not produce an update. Since

TreatmentUse506c1(g_1) in ϕ_1 also does not produce an update at s_1 with g_1, we have that $\phi \models_{(s_1,g_1)}$ Empty and therefore we have that $\forall \overline{e_2} \in \mathsf{pwr}(\phi_2), s_1 \xrightarrow{\overline{e}(g_1)} s' \implies \exists \overline{e_1} \in \phi \ . \ s_1 \xrightarrow{\overline{e_1}(g_1)} s'$. For the other direction, since TreatmentUse506c1 and HealthCareOperationsUse506c1 are not satisfied for s_1 and produce no updates, they can be modeled in ϕ_2 by Empty, so $\phi_2 \models_{(s_1,g_1)}$ TreatmentUse506c1 and $\phi_2 \models_{(s_1,g_1)}$ HealthCareOperationsUse506c1. For PaymentUse506c1, as we have shown above in Example 5.5.1, it produces updates if and only if Strong1 does. When it does produce updates, they are identical as well since we have shown that $\phi \models_{(s_1,g_2)}$ Strong1 via PaymentUse506c1. Therefore, $\forall \overline{e_1} \in \mathsf{pwr}(\phi) \ . \ s_1 \xrightarrow{overlinee_1(g_1)} s' \exists \overline{e_2} \in \mathsf{pwr}(\phi_2) \ . \ s_1 \xrightarrow{\overline{e_2}(g_1)} s'$.

2. $\phi \dot{\nsim}_{(s_1)} \phi_2$. Although $\dot{\sim}$ holds for the argument values g_1, as we have shown in Example 5.5.1, for $g_2 = (\mathrm{Cy}, \mathrm{Ike}, \mathrm{Cy}, \{\text{treatment, use, own}\}, f, \emptyset, \epsilon)$, $\phi \not\models_{(s_1,g_2)}$ Strong1 and therefore observational equivalence is not true either.

Having shown that ϕ_2 and ϕ_1 are observationally equivalent for s_1 with g_1, let us consider another augmentation to ϕ_2:

```
 CMD   Marketing(a, s, r, P, f, f', msg)
  if   consent in (a, s)
 and   local in (a, f)
 and   coveredEntity in Roles(a)
 and   individual in Roles(s)
 and   f.protected-health-information = true
then   insert marketing in (a, s)
 and   return true
else   return false
```

The command Marketing permits a covered entity to use protected health information about the individual if the individual has previously given consent. Adding Marketing to ϕ_2, we have $\phi_2 = (\emptyset, \{\text{Strong1}, \text{Empty}, \text{Marketing}\})$. With the new command in ϕ_2, let us return to relating it to ϕ_1 at s_1 with g_1. Note that we must augment the set *Right* = {treatment, payment, healthCareOperations, local, marketing} to accommodate the new right "marketing" in Marketing(a, s, r, P, f, f', msg).

1. $\phi \dot{\not\approx}_{(s_1,g_1)} \phi_2$. With the addition of Marketing(a, s, r, P, f, f', msg), for s_1 with g_1, since Ike has given consent already, $s_1 \overset{\text{Marketing}(g_1)}{\longrightarrow} (\{\text{Cy}, \text{Ike}\}, \{f\}, \{m(\text{Cy}, f) = \{\text{local}\}, m(\text{Cy}, \text{Ike}) = \{\text{conset, marketing}\}\}, \emptyset)$. There is no command in ϕ_1, however, which can grant the right "marketing" to the actor and therefore $\phi \not\models_{(s_1,g_1)}$ Marketing and therefore the two policies are not observationally equivalent.

□

Example 5.5.3 shows that observational equivalence is easily derived from strong licensing. The addition of a trivial command which is never satisfied lets ϕ_2 strongly model ϕ_1 and thereby permit the derivation of observational equivalence. The inclusion of a null command is important since it is necessary for the modeling of commands which are not satisfied.

Example 5.5.4 (At Least as Strict)

We have shown in Example 5.5.3 that $\phi_2 = (\emptyset, \{\text{Strong1, Empty, Marketing}\})$, let us consider how they compare using the less strict relation of $\prec$, the at least as strict as relation.

Let us again reuse s_1 from Example 5.5.1 and argument list g_1. Let us also consider the argument list $g_2 = (\text{Cy}, \text{Ike}, \text{Cy}, \{\text{treatment, use, own}\}, f, \emptyset, \epsilon)$:

1. $\phi_1 \prec_{(s_1,g_1)} \phi_2$. As we have noted above in Example 5.5.3, for g_1, any transition that ϕ_1 can perform is matched by ϕ_2. Therefore, ϕ_1 is at least as strict as ϕ_2 at s_1.

2. $\phi_2 \not\prec_{(s_1,g_1)} \phi_1$. As in Example 5.5.3, since at s_1 with g_1, Marketing can perform an update which ϕ_1 can not, ϕ_2 is not at least as strict as ϕ_1.

3. $\phi_1 \not\prec_{(s_1)} \phi_2$. Since as shown in Example 5.5.3, for g_2, ϕ_1 can perform a transition which ϕ_1 can not using TreatmentUse506c1, it is not at least as strict as ϕ_2.

4. $\phi_2 \prec_{(s_1,g_2)} \phi_1$. Since for g_2 none of the commands in ϕ_2 are satisfied, the only transition it can take leaves s_1 unchanged. Using PaymentUse506c1, ϕ_1 can model a command which causes no updates since Payment506c1 is also not satisfied. Therefore, ϕ_2 is at least as strict as ϕ_1 at s_1 with g_2.

Even though ϕ_2 has a transition which ϕ_1 can not perform, we can relate the two policies with $\prec$ by focusing on the parts where they agree. As shown, for the initial state s_1 with g_1, ϕ_2 subsumes the permissions of ϕ_1. Therefore, ϕ_1 is stricter than ϕ_2 since it offers less permissions to perform. As shown in 3, the relation does not hold for all argument values since g_2 breaks the relation. However, note that for g_2 the relation is reversed, that ϕ_2 is at least as strict as ϕ_1. The relation is trivial since ϕ_2 is not satisfied for g_2 and therefore ϕ_1 can model it with any unsatisfied command as well.

$\square$

Example 5.5.4 shows that we can use $\prec$ to relate policies that subsume each other, even if they are not exactly the same. That is, if they offer different permissions but agree on particular states and argument lists. As shown in the example, two policies may be at least as strict at one another interchangeably based on the initial state and argument list.

5.6 Conclusion

In this chapter we have presented the formal language for privacy commands based on the foundational framework in Chapter 3. Privacy APIs are built using the two executable rules of privacy commands: commands and constraints. Commands encode actions that may be performed by agents based on the permissions given in a legal text. They may reference each other and check guards about the knowledge state. Constraints encode limitations on commands from legal language that limits when actions may be performed. They may claim a scope of commands that they limit and check *such that* conditions that limit their applicability. We also considered several relations between commands and policies that let us quantify the permissiveness of policies and commands. In the next chapter we show how to automate the evaluation of Privacy APIs by using SPIN and Promela as the evaluation engine and a relation evaluator.

Chapter 6

Translating Privacy APIs to Promela

In the previous chapter we presented the formal Privacy Commands language and how we use commands and constraints in the language to form Privacy APIs which represent policies. In particular, in section 5.4 we discussed how the evaluation engine evaluates commands and constraints to update the knowledge state. Strong and weak licensing are relations which let us compare commands and policies and quantify how permissive they are. In this chapter we show how to translate Privacy APIs into Promela, the input language for the SPIN automated state space exploration tool which lets us evaluate the static properties of a given policy. Such properties include evaluations of what actions are permitted by the policy and under what circumstances. The translation to Promela includes the functionality of the commands and constraints as well as the functionality of the evaluation engine which processes them. Our goal in static exploration (using SPIN) is to detect properties of the Promela model that give insights into the actions that are reachable or unreachable using a given formal Privacy API, in particular those related to the relations discussed in Section 5.5. Since the Privacy API is derived from the legal text and maintains its structure, we may then map the discovered properties from the Privacy API back to the source text.

At a high level, in order to translate a Privacy API into an executable model, we must translate the elements from the knowledge state universe as discussed in Section 5.1: *Role*, *Purpose*, *Tag*, *Right*. With the sets, we build commands and constraints based on the structure of the legal text using the translation methodology discussed in Section 4.2. The resulting commands and constraints are a static model of the legal document framed in the above terms.

We assert properties in terms of the values contained in a knowledge state $s \in State$ that are reachable from a given initial state $s - (A, O, m, l)$. With the static model and initial state, we explore what states are reachable by executing series of commands in the policy. By observing the evolution of the knowledge state from the commands, we determine the properties that are true of the static model and thus of the policy. Using SPIN lets us automate the process by letting it explore the reachable states. For input, we develop invariants which describe properties of the knowledge state that we want SPIN to test.

As discussed in Section 3.2, the aspects of the privacy law that are of interest to us are ones that policy writers and evaluators would normally examine by hand. They commonly wish to compare policies, combine policies, examine whether one policy complies with another. Such aspects are amenable for modeling using Privacy Commands since the language enables us to examine what permissions a legal privacy policy provides. We use the policy relations defined in Section 5.5 to examine the properties of policy and derive conclusions. Note that the properties and conclusions that we are able to extract are subject to the language and style of the source document. Thus we will have more interesting results with policies that are detailed and less interesting results with vague ones.

In this chapter we develop the techniques needed for making the transition between the formal Privacy APIs representation from Chapter 5 and the modeling language Promela. We first present an overview of how the translation process works in Section 6.1. We then present the step by step mapping from the Privacy Commands language to Promela processes in Section 6.2. The Promela processes are framed by auxiliary code to properly perform evaluation. We present the auxiliary code in Section 6.3. We then show the correctness of our mapping from Privacy Commands to Promela with an equivalence theorem in Section 6.4. We conclude in Section 6.5 by outlining the uses of the Promela model for property exploration as we demonstrate in Chapter 7.

6.1 Translation Overview

The process of translation from a Privacy API to a Promela model involves several steps. We develop the precise methodology for the translation into Promela in Section 6.2, but we outline the steps here in order to give the reader a clear picture of the process. We first present an example command and constraint to illustrate the process, shown in Figure 6.1. The commands are samples from a case study in Section 7.1.2 and are derived from HIPAA [§164.506(a)(1), v.2000].

```
       CST   Permitted506a1(a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506a1, PaymentUse506a1,
              HealthCareOperationsUse506a1, TreatmentDisclose506a1,
              PaymentDisclose506a1, HealthCareOperationsDisclose506a1}
 Such That   individual in Roles(s)
       and   f.protected-health-information = true
        if   Permitted506a2(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false

  CMD   TreatmentUse506a1 (a, s, r, P, f, f', msg)
   if   Permitted506a1(a, s, r, P, f, f', msg) $\in$ {Allow}
  and   individual in Roles(s)
  and   healthCareProvider in Roles(a)
  and   local in (a, f)
  and   treatment $in_f$ P
  and   use $in_a$ P
 then   insert treatment in (a, s)
  and   return true
 else   return false
```

Figure 6.1: Sample commands for translation

Let us consider the steps required to translate the above commands and constraints into an executable model, independent our choice of Promela. Let us presume for this discussion that we are given only the Privacy API and the purpose hierarchy *Purpose*. To illustrate the process, let us consider Permitted506a1 and TreatmentUse506a1 from

Figure 6.1.

Step 1 Extraction of sets. We must extract an executable representation of the sets from the Privacy API: *Role*, *Purpose*, *Right*, *Tag*. For example, for line 5 of Permitted506a1, we must have a mechanism for establishing when an agent holds a role and so on.

Step 2 Purposes. Given a set of purposes, *Purpose*, we construct an executable representation of the purpose hierarchy. For example, on lines 6–7 of TreatmentUse506a1, we must check whether the purposes “use” and “treatment” are included in the purpose set.

Step 3 Commands. For each command in the Privacy API, we need to create an executable representation of each command. Each command has a generic structure as per the definition, so we devise a generic structure for each command. We then fill in the guards and operations which make up the functionality of the command.

Step 4 Constraints. Similar to commands, we devise a generic executable representation which we fill in with executable versions of the guards and operations. For overloaded constraints we need some manner of combining their judgments.

Step 5 References. In order to allow the commands and constraints to reference each other and properly manage scopes, we need to create a mechanism to enable them to communicate.

6.2 Translation to Promela

After translating the section of legal text into a Privacy API, we convert the commands and constraints into Promela in a format suitable for input to the SPIN model checker. Using SPIN we define invariants and evaluate whether the policy respects them. We use the SPIN model checker (`www.spinroot.com`) since it provides a good interface for model creation and invariant checking, although other similar tools could have been used instead.

At the high level the translation between Privacy Commands and Promela requires the creation of run-time entities that have the same semantics as commands and constraints.

The run time entities (Promela *processes*) are managed by generic management code designed to handle the functionality of the evaluation engine (Section 5.4). We first discuss in detail the methodology for translating commands and constraints into Promela in Section 6.2.2 and then discuss the management code in Section 6.3. For brevity, full source and supplementary code is placed in Appendix A.

6.2.1 Promela Fundamentals

During execution, SPIN performs variable inspections and updates using Promela comparators and operators. The syntax and semantics of those inspections and updates are as follows. The operators we use in Promela are shown in Table 6.1. As in standard imperative programming languages, users can compose expressions which are combinations of operators and variables that are evaluated to a resulting value but produce no side effects. Statements are similar to expressions except that they produce side effects. We list the expressions and statements used in our Promela models, but since many of them are standard to imperative programming languages such as C and Java, we offer only limited elaboration.

Table 6.1: Promela operators

Operator	Usage
`==`	Equality comparison
`!=`	Inequality comparison
`<, <=`	Less than, less than or equal
`>, >=`	Greater than, greater than or equal
`=`	Assignment
`&&`	Boolean AND
`||`	Boolean OR
`!`	Channel send
`?`	Channel receive
`[]`	Array index
`++ --`	Increment and decrement
`run p()`	Run a process `p`
`atomic`	Restrict process interruption
`->, ;`	End of statement or expression

The expressions we use in our Promela models are as follows. For this discussion, generic variables or base types (excluding channels) we use the names `var1`, `var2`, ... We do not refer to a particular base type for them unless noted, so by default the following explanations apply equally to bits, integers, user defined types, and booleans. We use `bool1`, `bool2`, ... for boolean variables. We use `chan1`, `chan2`, ... for channel variables. We denote a generic expression `Exp1`, `Exp2`, ... We denote a generic statement `Stmt1`, `Stmt2`, ... We denote integer variables `int1`, `int2`, ...

- `var1 == var2; a[int1] != var3; var4 < var5; var6 > var7`. Variable and array index comparison. Result is a boolean value.
- `bool1 || bool2; bool3 && bool4; Exp1 && Exp2`. Boolean comparison. Result is a boolean value.
- `atomic{Stmt1, Stmt2}`. Atomic group of statements which will not be interrupted.

The statements we use in our Promela models are as follows. We use the same variable notation as for the expressions listing.

- `var1 = var2; var3 = (Exp1), a[int1] = var4`. Variable and array assignment. Outcomes are that `var1` gets the value of `var2`, `var3` gets the value to which `Exp1` evaluates, and `a` in index `int1` gets the value `var4`.
- `chan1!var1; chan2!var2, var3;` Sending on a channel. Outcome is that `var1` is sent over `chan1` and `var2, var3` are sent over `chan2`. For non-buffered channels, the send blocks until a corresponding receive is executed on the channel.
- `chan1?var4; chan2?var5, var6;` Receiving on a channel. Assuming messages are sent as in the previous item, the outcome is that `var4` gets the value of `var1`, `var5` gets the value of `var2`, and `var6` gets the value of `var3`.
- `int1++; int2--;`. Integer increment and decrement.
- `skip`. Nil statement.
- `break`. Exits the innermost iteration structure.

We use the following selection and repetition constructs in our models.

- `if ::Exp1 -> Stmt1; Stmt2; ::Exp2 -> Stmt3; ::Stmt4 -> Stmt5; ::else -> skip; fi`. The `if/fi` statement first evaluates all of the expressions and statements that immediately follow a double colon (the *guards*). It then randomly selects one to execute from all of the choices whose guards are executable (*i.e.,* expressions that are true or statements that are executable). If none of the guards are executable, the else choice is selected.

- `do :: Exp1 -> Stmt1; Stmt2; :: Exp2 -> Stmt3; :: else -> skip; :: Stmt4 -> Stmt5; break; od`. The `do/od` statement behaves similarly to an `if/fi` statement except that it iterates indefinitely until a `break` is executed.

We use the following special statements and functions:

- `atomic{Stmt1; Exp1;}` Executes the statements and expressions in the clause without preemption by other processes. The enclosed `Stmt1` and `Exp1` therefore execute as one atomic action.

- `assert(Exp1);` Evaluates the expression `Exp1` to a boolean result, similar to a C assertion. If the result is true, the program continues execution. If the result is false, program execution terminates immediately.

We use `atomic` and `assert` statements to enforce invariants on shared, fixed size arrays whose contents must be protected be properly used by many processes in a safe manner.

Promela includes several basic variable types that we will use for storage: `int` for integers, `bool` for booleans, and `bit` for bit flags. Table 6.1 includes the operators used in Promela which we discuss below

There is one enumerated type `mtype` which is used for enumerating message types. The message types used in our models are:

```
mtype = {command_request, command_response, constraint_request,
constraint_response, search_request, search_response, purpose_request,
purpose_response, hier_request, hier_response};
```

The use of all the values of `mtype` is shown in Table 6.5. Complex record types can be declared in terms of the basic types using `typedef`. Arrays can be made of any simple of complex type using a C-like array syntax:

```
int a[20];
bool b[5];
```

The above code creates a zero-indexed array of integers of size 20 called `a` and a zero-indexed array of booleans of size 5 called `b`. Like C, Promela requires array sizes to be constant integers available at compile time. Promela, however, does not have any features for dynamic allocation or deallocation of memory, so all arrays are of fixed size.

Selection is performed using a non-deterministic `if/fi` construct. Options for the selection statement are denoted with a double colon `::`. The first expression for each option is interpreted as a guard. The set of true or executable guards is collected at run time and one is selected at random to be executed. An `else` guard is selected only if all other options are not executable. If none of the guards are executable and there is no `else` statement, the `if/fi` blocks.

```
if
:: x < 5 -> y = 1;
:: x < 4 -> y = 2;
:: else -> y = 3;
fi
```

Here if `x` is 4, then only the first option is executable and `y` will get 1. If `x < 4` then there is an equal chance that `y` will get 1 or 2. If `x > 5` then `y` will get 3.

Iteration is performed using a non-deterministic `do/od` construct which behaves similar to the `if/fi` construct. At each iteration, the construct randomly selects one of the true or executable options for execution. After completion, the construct starts the process again. If no options are executable, the `do/od` blocks.

```
do
:: x < 5 -> x++;
:: x < 5 -> x--;
:: x > 5 -> y++;
od
```

Here if execution begins with x less than 5, there is an equal chance that x will be incremented or decremented. If x ever reaches 5, the construct will block since none of the statements will be executed. If x begins execution greater than 5, then y will be incremented indefinitely.

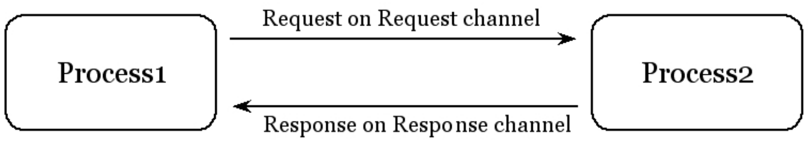


Figure 6.2: Sample communication

The execution atoms in Promela are processes, encapsulations of code which may accept and return information via shared variables and communication channels. We model each command and constraint as a separate process which takes an input query, performs some processing, and returns a result to it caller. As shown in Figure 6.2, processes communicate by sending messages across different channels. In the figure, Process1 sends a request to Process2 along the request channel and then waits for a response from Process2 along the response channel.

Some example channels are:

```
1   chan CMD1_chan = [0] of {mtype, bool, bool};
2   chan CMD2_chan = [0] of {mtype, bool};
3   chan A_chan = [2] of {int, bit};
```

Here, `CMD1_chan` and `CMD2_chan` (lines 1–2) are non-buffered channels (`[0]`), meaning that if a process sends a message over them, it blocks until another process receives it. `CMD1_chan` has a width of 3 and requires that all messages have the signature `mtype, bool, bool`. `CMD2_chan` is similar except that it is 2 wide and requires that all messages have the signature `mtype, bool`. `A_chan` (line 3) is a buffered channel (`[2]`), so senders do not block on it. It is 2 wide and requires that all messages have the signature `int, bit`. In our models we do not use buffered channels, so they all are typed `[0] of (...)`, but they may have different widths and message signatures. Two communicating processes `CMD1` and `CMD2` which use the above channels are shown below.

```
1   active proctype CMD1() {
2      bool result; CMD command = Cmd1;
3      CMD2_chan!request(true);
4      CMD2_chan?response(result);
5      CMD1_chan!command_response(result, result);
6   }
7
8   active proctype CMD2() {
9      bool result; CMD command = Cmd2;
10     int i = 2, j = 3;
11     do
12     :: CMD2_chan?command_request(_) ->
13        if
14        :: i < j ->
15           CMD_chan!command_response(true);
16        :: else ->
17           CMD_chan!command_response(false);
18        fi
19     od
20  }
```

Here process CMD1 sends (`!`) a request to CMD2 to execute (line 3) over its channel `CMD2_chan`. The message is a tuple, but is written `request(true)` since SPIN allows messages to be denoted using a shorthand `mtype(var1, var2, ...)` which is equivalent to `(mtype, var1, var2, ...)`. The sent message has a payload `true` which is included to match the width and signature of `CMD2_chan`. CMD1 then listens (`?`) for a message on CMD2's channel on Line 4. Since the mtype `response` is included, CMD1 will only accept messages with that mtype. The boolean payload is stored in the variable `result`. On line 5, CMD1 sends out its response in a `command_response` message with two copies of the result variable to match the type and width of `CMD1_chan`.

Process CMD2 has an infinitely repeating `do/od` loop on lines 10–19 which means that it will loop indefinitely, accepting one message at a time. Line 12 declares the only option (`::`) the loop may select from: listening for a `command_request` on `CMD2_chan`. When a `command_request` arrives, its payload value is discarded using the `_` symbol. The `if/fi` statement then executes (line 13) and if `i < j` (line 14), CMD2 returns true over its channel (line 15). Otherwise, the else option is chosen (line 16) and the command returns false (line 17). After one execution is completed, the loop returns to Line 12 to wait for more requests.

Most commands in our models behave like CMD2 above. They are active processes which contain infinite listen/action loops. Due to the constraints of Promela, `active` or always running processes can accept parameters only at start up, so the knowledge state is stored in a set of global variables that can be written and read by any process. Since we do not allow concurrency (*i.e.,* every process must wait for the completion of any request it sends before it may proceed), we do not need to worry about race conditions or read/write conflicts. We elaborate on the translation on commands in Section 6.2.6 and constraints in Section 6.2.4 and Section 6.2.5.

6.2.2 From Formal Model to Promela Model

Adapting a Privacy API to Promela requires the creation of structures parallel to the sets, matrices, and logs used in the Privacy Commands language. The correspondence between the knowledge representation in the formal model and the Promela code depends on the correspondence in the storage representations which we discuss next. In order to show correspondence between Privacy Commands and the Promela processes in Section 6.4, we first discuss the Promela types and language constructs that we use to parallel the formal language and evaluation engine. In Section 6.4.1 we discuss the semantics of the constructs discussed below to show that they correspond to the operational semantics in Section 5.3.

In the Privacy Commands model, knowledge state is stored in a series of sets and tables, as delineated in Table 6.2. The representations differ slightly due to the operational behavior of Promela and so we now detail them and their usage.

Table 6.2: Knowledge state representations

Formal Representation	Promela Representation
Agent	`AGENT` variable instantiations
Role	`Role` bit record type
Roles(a)	`mroles[]` array
Object	`OBJECT` variable instantiations and `object` array
Tag	`Tag` bit record type
tags(o)	`mtags[]` array
Right	`Rights` bit record type
Matrix	`Matrix m` composite matrix type
msg strings	`Note` bit record type
Log of strings	`log` array of `Note` type
Log of informs	`inform` array of `Note` type
Purpose	`PURPOSE` variable instantiations
Purpose partial order	`parent` array

We first consider at a high level what the corresponding Promela translation for each structure and relevant set are. The basis for the discussion is the section Section 5.1 regarding the fundamental sets and types. We then detail the translation in the rest of this subsection.

Agents and Objects As discussed in Section 5.1, there is a static set *Agent* with $n > 0$ members $a_1, \ldots, a_n$, all with unique names, and a dynamic set *Object* with $m > 0$ members $o_1, \ldots, o_m$. Correspondingly, in the Promela model we instantiate data types `AGENT` and `OBJECT` which use integer values to uniquely identify agents and objects, similar to their names in *Agent* and *Object*.

Roles As discussed in Section 5.1, roles are static attributes of agents, members of *Role*. Any member of *Role* may be true for any given agent. The corresponding Promela construct is the `mroles` array which stores a bit flag for each agent and role using a

`Role` record type. Roles are turned on for an agent setting the bit flag to true for the corresponding agent and role.

Rights As discussed in Section 5.1, rights are members of a set *Right* that are set as relationships between agents and objects. The corresponding Promela construct is the matrix `m` which contains records with bit flags for each right and is indexed using the agent and object names. A right is set between an agent and an object by setting the corresponding bit flag for the agent and object entry in a `Rights` record type.

Tags As discussed in Section 5.1, tags are boolean flags from the set *Tag* which are set to indicate meta-information. For an object o, we say the tag t is set iff $o.t = true$. The corresponding Promela construct is the `mtags` array which stores a bit flag for each object and tag in a `Tags` record type. A tag is set on a given object by assigning a boolean value to the corresponding object and tag.

Purpose As discussed in Section 5.1, purpose is modeled in a partial ordering between purpose names. In the Promela model, we simulate the hierarchy by assigning variables of type `PURPOSE` for each purpose and indexing them into an array `parent` which encodes the hierarchy. Specialiazed functions operate over the array to interpret it correctly.

Judgments In Section 5.1 we enumerate the judgments which the evaluation engine derives from constraints. We represent each judgment in Promela with a custom type `JUDGMENT` and use variables of the type to represent the results of constraints.

Messages As discussed in Section 5.1, free text strings may be entered in the log or sent to agents. Since Promela does not support a string type, we modify the contents of messages to a snapshot of the state when the log or inform action is taken which is appended to the log array.

Commands and Constraints Commands and constraints as defined in Definitions 5.2.8 and 5.2.10 respectively are combinations of guards and operations. The evaluation engine as described in Section 5.4 evaluates the guards and performs the operations as described. In Promela we represent each command and constraint as a Promela process. Each process listens on a specified channel for requests. When a request arrives, the process evaluates the guards and performs the operations for the command or constraint and upon completion sends it return value over its specified response channel. We elaborate on the translation for commands in Section 6.2.6 and constraints in Section 6.2.4 and Section 6.2.5.

Agents and Objects

The knowledge state as defined in Section 5.1 is stored in global variables which can be examined by all constraints and commands. Variable state such as agents, objects, purposes, tags, and messages are stored using enumerations with names paired up with unique integer numbers which are then hard coded into the model. Since agents are a subset of objects, their numbering must not overlap. For example, the agent set $A = \{\text{alice, bob, claire}\}$ and object set $O = \{\text{file1, file2}\}$ would be represented as:

```
1   #define AGENT int
2   #define OBJECT int
3   #define MAXAGENT 3
4   #define MAXOBJECT 20
5
6   AGENT alice = 0;
7   AGENT bob = 1;
8   AGENT claire = 2;
9   OBJECT file1 = 3;
10  OBJECT file2 = 4;
11
12  OBJECT objects[MAXOBJECT] = 0;
13  objects[alice] = 1; objects[bob] = 1; objects[claire] = 1;
14  objects[file1] = 1; objects[file2] = 1;
```

In order to make the types more obvious we use `#define` to create custom types for agents and objects (lines 1–2). They both map to type `int`, but using the custom types makes the code a bit easier to check for bugs since the intended typing is shown.

The agent set does not evolve over time, so there is no need to create new agents during execution. The number of agents is defined as `MAXAGENT` (line 3). The object set may grow and shrink, however, using the **create object** and **delete object** primitive operations, so we must handle the growth and shrinkage of the object set and consequently, the rights matrix as well (see next heading below). As noted above, Promela does not support dynamic allocation of arrays, so we must fix some size for the object set and the rights matrix, called `MAXOBJECT` (line 4). This places a limitation on the exploration space of SPIN and limits us to exploring properties that do not involve the creation of an unbounded or unknown number of objects at once. Also, for simplicity the slots for deleted objects are not reused. For an actual implementation of the policy engine over an operating database, both of these limitations would limit the usability of the database. However, since we are only interested in the exploration of policy properties they are not severe restrictions.

We instantiate a variable of type `AGENT` for each $a_i \in Agent$. The integer value assigned to it is `AGENT ai = i` where `ai` is the name assigned to the agent and `i` is an integer which is equal to a_i's index in *Agent*. The numbering is done consecutively, so $\forall x, y, x \neq y \implies x_i \neq y_i$ where x_i and y_i are the indexes of the agents (lines 6–8).

Similar to agents, we instantiate a variable of type `OBJECT` for each $o_i \in Object$. The integer value assigned to its is `OBJECT oi = i` where `oi` is the name assigned to the object and `i` is an integer which is equal to o_i's index in *Object*. The numbering is done consecutively, so $\forall x, y, x \neq y \implies x_i \neq y_i$ where x_i and y_i are the indexes of the objects. Since all entries `AGENT` are also members of `OBJECT`, the two types share a common storage type (*i.e.,* integers) and their indexes are established such that the agents are all assigned consecutive indexes at the bottom of the `OBJECT` indexing. That is, if `MAXOBJECT > MAXAGENT`, then $\forall o_i \in Object - Agent$, the index of $o_i >$ `MAXAGENT` (lines 9–10).

In order to allow the creation and deletion of objects we create an array `objects[]` of type `OBJECT` which maintains the state of whether an object with a given index is instantiated or not (line 12). In the example, the entries for the instantiated agents and objects are all set to 1 to indicate that an object or agent with the given index exists (*i.e.,* has been defined or created) (lines 13–14) . If an object is deleted, its entry is set to 0.

Since agents are also objects, the entries in `objects` corresponding to them are also set to 1. In order to check whether an object index O is an agent and therefore whether is may be deleted, the code checks if $o <$ `MAXAGENT`.

Roles

Roles are represented with a bit record of type `Role`. For each $k_1, \ldots \in \mathit{Role}$ we create a bit entry which can be set independently. The array `Role mroles[MAXAGENT]` stores `MAXAGENT` records for the number of agents that exist (`MAXAGENT`). For each agent `a`, we say that `a` has role `k` iff `roles[a].k == 1`. As an example, consider a role record with three roles is (based on Example 5.2.2):

```
typedef Role
{
    bit covered_entity;
    bit inmate;
    bit correctional_institution;
}
Role mroles[MAXAGENT];
mroles[alice].covered_entity = 1;
mroles[bob].inmate = 1;
```

Here there are three roles that may be occupied and the roles array `mroles` is declared of size `MAXAGENT`. Alice is given the role of a covered entity and Bob is given the role of an inmate. The roles array is static during the execution of the policy since no operations to modify it.

Rights Matrix

Rights are stored in a rights record, a bit vector which is set to represent the rights that are held by an agent over an object. The bit flags are named for the rights that they represent. For example, for the rights set $\mathit{Right} = \{\text{read, write}\}$, the rights record would be represented as:

```
typedef Rights
{
    bit read;
    bit write;
}
```

With this representation, a rights record instance can be indexed by the name of its fields. For example, to create a rights record for Alice and set the “read” permission, we would write:

```
Rights alices_rights;
alices_rights.read = 1;
```

The rights matrix is represented using a two dimensional matrix of agents and objects with a rights record as the entry. Since Promela does not directly support multidimensional arrays, we first use a typedef to create a type called vector which contains just an array of rights. We then create a second typedef which is an array of vectors. The second typedef therefore has an array of vectors, the equivalent of a two dimensional array:

```
typedef Vector
{
    Rights objects[MAXOBJECT];
}
typedef Matrix
{
    Vector mat[MAXAGENT];
}
```

The resulting matrix is indexed using the record names. In a model, we create one matrix `m` which stores the rights for all agents and objects. For example, to store Alice’s read right on file1:

```
Matrix m;
m.mat[alice].objects[file1].read = 1;
```

Tags

Tags are boolean flags that are associated with objects. We instantiate a record type `Tags` which consists of boolean flags for each tag. The boolean flags in each record may be set independently. Tags are stored in a special tags vector `mtags` which is indexed by object name, similar to the rights matrix. For example, to create two tags, we first create a tags record:

```
typedef Tags
{
    bool protected_health_information;
    bool psychotherapy_notes;
}
```

The first tag is a boolean flag for indicating whether an object is protected health information. The second is a boolean flag for indicating whether a file contains psychotherapy notes. Commands may erase tags by setting the boolean flag to false. Erasure is not done automatically, so maintaining valid combinations of mutually exclusive tags is enforced by the commands that use them. The tags are stored in a tags vector indexed by object name. The tag t on an object o is set iff `tags[o].t == true`. For example:

```
Tags mtags[MAXOBJECT];
mtags[file1].protected_health_information = true;
mtags[file1].psychotheray_notes = false;
```

The above example creates a tags matrix the size of the number of objects. It stores tags indicating that File1 is protected health information and is not psychotherapy notes.

Purposes

We represent purposes in Promela similar to objects and agents, with a set of names that map to unique, consecutive integer constants. The first purpose indexed is numbered 0, the second 1, and so on. Like agents and objects, we use `#define` to create a custom type `PURPOSE`. The total number of purposes is stored in a constant `MAXPURPOSE` and is used for creating the arrays that store information about the purpose hierarchy (Section 5.1) and for representing purpose sets passed as arguments to commands and constraints. Both the purpose hierarchy (stored in `PURPOSE parent[MAXPURPOSE]`) and purpose set parameters (stored as `PURPOSE P[MAXPURPOSE]`) are arrays of size `MAXPURPOSE`.

A purpose set `PURPOSE P[MAXPURPOSE]` is an array of `PURPOSE`s which represents a set of purposes in the formal model $P \subset \mathit{Purpose}$. We include a $p \in \mathit{Purpose}$ in the purpose set `P` by setting its corresponding integer value in `P` to 1. All purpose sets therefore have the same size and ordering, allowing for quick lookup, set union, intersection, and difference operations.

Since Promela does not offer any support for hierarchies, we simulate one using an array `PURPOSE parent[MAXPURPOSE]` which stores the purpose hierarchy. The array `parent` has with the property that the value of each array index is the corresponding purpose's parent (*i.e.,* `parent[C] = P` for purposes C and P using the unique integer numbers assigned to them). Root purposes have a parent value of -1 which is stored in a special purpose called `ROOT`. Using the parent array we can traverse up the purpose hierarchy in linear time, but down traversals have a worst case $O(n^2)$ complexity.

```
#define PURPOSE int
#define MAXPURPOSE 2
PURPOSE ROOT = -1;
PURPOSE p1 = 0;
PURPOSE p2 = 1;
PURPOSE parent[MAXPURPOSE];

parent[p1] = p2;
parent[p2] = ROOT;
```

In the above example we define two purposes `p1` and `p2`. The array `parent` is of size `MAXPURPOSE`. It indicates `p2` is the parent of `p1` and that `p2` is a root purpose.

We create two purpose set membership checking algorithms, one for allowed semantics and one for forbid semantics as defined in Section 5.1. Both algorithms use the parent array using only upward traversals for efficiency.

To illustrate how purposes are represented and used in our Promela models, let us return to the purpose example discussed above in Example 5.1.1.

Example 6.2.1 (Using Purposes in Promela)

In Example 5.1.1 above we presented an example hierarchy of purposes based on different types of surgeries that a policy considers: Treatment is the parent of Surgery, Surgery is the parent of Oral Surgery and Eye Surgery. For convenience, we reproduce the hierarchy shown above in Figure 5.1 here in Figure 6.3. Let us assign the purposes indexes in

$$\begin{tikzcd}
 & \text{Treatment (0)} \arrow[d] & \\
 & \text{Surgery (1)} \arrow[dl] \arrow[dr] & \\
\text{Oral Surgery (2)} & & \text{Eye Surgery (3)}
\end{tikzcd}$$

Figure 6.3: Hierarchy for surgery example (reprise)

order. In Promela:

```
#define PURPOSE int
#define MAXPURPOSE 4
PURPOSE ROOT = -1;
PURPOSE Treatment = 0;
PURPOSE Surgery = 1;
PURPOSE Oral_Surgery = 2;
PURPOSE Eye_Surgery = 3;
```

The parent array is of size `MAXPURPOSE`:

Index	0	1	2	3
Parent	-1	0	1	1

In Promela:

```
PURPOSE parent[MAXPURPOSE];
```

```
parent[Treatment] = ROOT;
parent[Surgery] = Treatment;
parent[Oral_Surgery] = Surgery;
parent[Eye_Surgery] = Surgery;
```

An action which is to be performed for Oral Surgery would provide a purpose set:

Index	0	1	2	3
Value	0	0	1	0

Note that the bits for Treatment and Surgery are set to 0 since Oral Surgery is the most specific purpose. If Surgery, for example, had been set to 1 as well, it would indicate that the action is for all types of surgery, including Eye Surgery. □

Using the purpose indexing and parent table we implement lookup functions such as isParentOf, isChildOf, isAncestorOf, isDescendantOf using algorithms based on the definitions in Section 5.1. The code for the functions are in Appendix A.1.

The algorithms for set inclusion with permit and forbid semantics are layered on top of the hierarchical algorithms. For checking p in$_a$ P for allowed semantics we check $\exists p' \in P$: isAncestorOf(p', p) and for p in$_f$ P for forbid semantics we check $\exists p' \in P$: isAncestorOf(p', p) || isDescendantOf(p', p). The function isPermittedByA(p, P) implements the check for p in$_a$ P and isForbiddenByA implements the check for p in$_f$ P. The implementations for all of the algorithms are provided in Appendix A.1.

Judgments

As defined in Section 5.1 and Section 5.4.2, judgments are derived from the constraint return values. The evaluation engine derives the judgments based on the values of the boolean results, whether the constraint is overloaded, and whether is run as part of a constraint search. We now detail how judgments and their derivations are performed in the Promela model.

We enumerate the five different judgments as constants, and define a type `JUDGMENT` which we `#define` to `int` for storage of judgment results:

```
#define JUDGMENT int
```

```
#define ALLOW 0
#define FORBID 1
#define IGNORE 2
#define DONT_CARE_ALLOW 3
#define DONT_CARE_FORBID 4
```

We defer the discussion of judgment derivation in Promela to Section 6.2.7 during our discussion of translating commands and constraints.

Log and Inform

Since Promela does not support characters or strings, we use records to simulate the storage of log notes and the sending of messages to agents. The knowledge state therefore includes an array of log notes and inform messages sent. To perform a log or an inform operation, a new log or inform record is placed in the next open space in the array. As with the objects array, this places a limit on the number of log entries and inform messages sent. The types used for the record fields are as discussed above. The records for log and inform records are as follows:

```
typedef Note
{
    CMD command;
    AGENT a;
    AGENT s;
    AGENT r;
    int P[MAXPURPOSE];
    OBJECT f;
    OBJECT few;
}
Note log[MAXLOG];
Note inform[MAXINFORM];
```

Global Variables

Each command and constraint is converted into a process which listens and responds along a named public channel. The constants described in this section are summarized in Table 6.3. Globally readable arrays contains bit flags for tags, roles, log notes, and inform messages as described above. Global variables hold the values of the parameters for commands and constraints: a, s, r, P, f, f', msg. The names and type declarations the global variables and arrays are shown in Table 6.4. Table 6.5 shows the different message types and how they are used.

Table 6.3: Declared Constants for Promela models

Name	Use
MAXAGENT	Number of agents
MAXOBJECT	Maximum objects
MAXPURPOSE	Number of purposes
MAXLOG	Maximum log slots
MAXINFORM	Maximum inform slots
MAXCMD	Number of commands

6.2.3 Translating Guards and Operations

In Section 6.2.4 and Section 6.2.6 we discuss the framework for translating constraints and commands, but first we discuss the translation of the operations and guards. We first discuss the particulars of each translation based on the notation presented in the Privacy Commands grammar in Section 5.2.5. We then provide and explain the full translation tables in Tables 6.7 and 6.8.

Guards and operations are translated line by line using the Promela model structure described in Section 6.2.2. The following translation tables show how we translate each guard and operation to Promela. Table 6.7 contains the translation for guards and Table 6.8 contains the translation for operations. Since we will present a detailed explanation for the Promela translations in Section 6.4.2, we postpone a detailed discussion of Tables 6.7 and 6.8. The proof of Lemma 6.4.4 on Page 226 details the Promela code in Table 6.7. The proof of Lemma 6.4.5 on Page 230 details the Promela code in Table 6.8.

Table 6.4: Global variables for Promela models

Name	Declaration	Use
a	`AGENT a;`	Current active agent
s	`AGENT s;`	Subject of current action
r	`AGENT r;`	Recipient of current action
f	`OBJECT f;`	Object for current action
few	`OBJECT few;`	Slot for creating new object
topObj	`OBJECT topObj;`	Next open object slot
objects	`OBJECT objects[MAXOBJECT];`	Available objects
mroles	`Role mroles[MAXAGENT];`	Active roles
m	`Matrix m;`	Rights matrix
mtags	`Tags mtags[MAXOBJECT];`	Tags for the objects
P	`int P[MAXPURPOSE];`	Purposes for current action
parent	`PURPOSE parent[MAXPURPOSE];`	Purpose hierarchy
topLog	`int topLog;`	Next open log slot
log	`Note log[MAXLOG];`	Log
topInform	`int topInform;`	Next open inform slot
inform	`Note inform[MAXINFORM];`	Messages sent to inform

Table 6.8 does not include a translation for the operation **return** *b* since the placement of **return true** and **return false** is restricted to the true and false branches of commands and constraints and the logic that they implement is distributed throughout the command and constraint processes in Figures 6.6 and 6.4. We show in Lemmas 6.4.6 and 6.4.7 that the commands and constraints structures correctly implement the logic in the **return** operations.

As discussed in Section 6.2.2, arrays in Promela are of fixed size. Therefore, the operations which add entries to matrices with fixed sizes (*i.e.,* object creation, logging, inform messages) are framed in `assert` statements. The assertions perform bounds checks on the arrays before adding new records. If an assertion fails, the model's execution ceases and the error is caught by SPIN.

6.2.4 Translating Constraints

We use the Promela foundation in Section 6.2.2 in writing the Promela translations of constraints and commands. We discuss the translation of constraints in this section, the

Table 6.5: Message Types

`mtype`	Type	Use
`command_request`	Request	Sent to a command process to start.
`command_response`	Response	From a command which completed.
`constraint_request`	Request	Sent to a constraint process to start.
`constraint_response`	Response	From a constraint which completed.
`search_request`	Request	Sent to start a constraint search.
`search_response`	Response	Response from a constraint search.
`purpose_request`	Request	To the purpose algorithms.
`purpose_response`	Response	From the purpose algorithms.
`hier_request`	Request	Initiates a traversal of the purpose hierarchy.
`hier_response`	Response	Completion of a purpose hierarchy traversal.

translation of scopes in Section 6.2.5, and the translation of commands in Section 6.2.6. Where needed we write special functions that perform auxiliary tasks in Promela and provide their code in Appendix A.

As per Definition 5.2.10, constraints have the following structure:

CST	$c(a, s, r, P, f, f', msg)$
Scope	$\{e_1, \ldots\}$
Such That	ψ_1^{st}
and	$\ldots$
if	ψ_1^{r}
and	$\ldots$
then	**return true**
else	**return false**

As discussed in Section 5.4.2, there are then three boolean results derived from a constraint by the evaluation engine: b_{scp} indicating whether the calling command is in scope, b_{st} indicating whether the *such that* guards are satisfied, and b_r indicating whether the regular guards are satisfied. Judgment derivation is performed by the evaluation engine depending on whether the constraint is run due to a reference or during a constraint search. Tables 5.8 and 5.10 show how the evaluation engine determines judgments from the boolean results.

Constraint processes accept requests with one `CMD` parameter using an `mtype` `constraint_request` (as shown in Table 6.5) over a channel type which accepts messages of type `[0] of {mtype, CMD}`. For a constraint named CST1, the request channel type is: `CST1_request_chan = [0] of {mtype, CMD};` Constraint processes return three boolean values using an `mtype` called `constraint_result` and over a channel of type `[0] of {mtype, bool, bool, bool}`. Judgments are derived from the message by the recipient process or the constraint search process. For CST1, the response channel type is: `CST1_response_chan = [0] of {mtype, bool, bool, bool};`

Using the notation above for the different guards sets, the generalized Promela structure for a constraint is listed in Figure 6.4. Line 1 declares the process name, `CST1` and that it takes no parameters. Lines 2–7 declare the local variables that will be used during the constraint's execution. The first three, `scope`, `such_that`, and `regular` store the boolean results from the scope check, such that guards, and regular guards respectively. The next two, `result` and `temp` are used as temporary boolean variables during evaluation. The next local variable, `command` stores the `CMD` value for the command that sent the message to run the constraint. The last local variable, `j` stores the judgment results from constraints invoked. Lines 8–22 are the main `do/od` loop of the constraint. Line 9 declares the only loop option: listening for a message of type `constraint_request` on the constraint's request channel `CST1_request_chan`. The request stores the value for the `CMD` parameter in `command`. Lines 10–11 reset the values of the boolean variables from any stale values from previous loop iterations.

Lines 13–16 process the constraint's scope by checking the value of `command` against the list of `CMD` values in the constraint's scope. Label L1 on line 14 is the location where the scope checks are inserted. Their format is discussed below in Section 6.2.5. Line 15's `else` option sets `scope` to `false` in case `command` is not in scope.

The *such that* and regular guards are inserted in lines 18 and 20 at labels L2 and L3 respectively. The format of individual guard translation is in Section 6.2.3. The temporary variable `result` is used to accumulate boolean results from the guards that are executed. After the *such that* guards have completed, line 19 stores `result`'s value in `such_that` and then resets `result` for use in the regular guards. Similarly, line 21 stores `result`'s value in `regular` at the completion of the regular guards' evaluation.

Lines 22–23 returns the results from the constraint's execution over `CST1`'s response channel `CST1_response_chan`. Line 24–25 closes the `do/od` loop.

```
1   active proctype CST1() {
2      bool scope = true;
3      bool such_that = true;
4      bool regular = true;
5      bool result = true;
6      bool temp = true;
7      CMD command; JUDGMENT j;
8      do
9      ::  CST1_request_chan?constraint_request(command) ->
10         scope = true; such_that = true; regular = true;
11         result = true; temp = true;
12
13         if
14         L1: (scope checks go here)
15         :: else -> scope = false;
16         fi;
17
18         L2: (such that guards here)
19         such_that = result; result = true;
20         L3: (regular guards here)
21         regular = result;
22         CST1_response_chan!constraint_response(scope, such_that,
23            regular);
24     od;
25  }
```

Figure 6.4: Promela framework for constraints

Overloaded Constraints

As discussed in Section 5.2.4, constraints may be overloaded with multiple instantiations sharing a single name and scope. As discussed in Section 5.4, the evaluation engine uses a most lenient in deriving judgments from the overloaded instantiations. In the formal model there is no way to specifically reference one instantiation of an overloaded constraint, so our model translation uses a slightly different representation of overloaded constraints than the formal model does. Lemma A.2.4 in Section A.2 shows that the modified translation yields the same results as the evaluation engine. The combination algorithm is derived from

Tables 5.11 and 5.9 above. The derived combination truth table is shown in Table 6.6 and is shown in terms of the resulting values in the local variables shown in Figure 6.4 with the exclusion of `scope` since it never needs to be combined.

Table 6.6: Result combination for overloaded constraints in Promela

Constraint 1		Constraint 2		Result	
`such_that`	`regular`	`such_that`	`regular`	`such_that`	`regular`
True	True	True	True	True	True
True	True	True	False	True	True
True	True	False	True	True	True
True	True	False	False	True	True
True	False	True	True	True	True
True	False	True	False	True	False
True	False	False	True	True	False
True	False	False	False	True	False
False	True	True	True	True	True
False	True	True	False	True	False
False	True	False	True	False	True
False	True	False	False	False	True
False	False	True	True	True	True
False	False	True	False	True	False
False	False	False	True	False	True
False	False	False	False	False	False

We combine the initial values of the variables with new ones using a most lenient algorithm. Let us denote the results from a constraint as b_1/b_2 where b_1 is the result from the *such that* guards and b_2 is the result from the regular guards. As shown in Table 6.6 lines 1–5, 9, and 13, if either the initial or the new values are True/True, then the resulting value is the most lenient result of True/True. As reflected in Tables 5.11 and 5.9, a `such_that` result equal to true overrides a `such_that` result equals to false. The intuition is that a result with `such_that` equal to false is not as applicable as one with `such_that` equal to true, even if its regular guards yield a more lenient result. Thus, in cases such as on lines 7 and 9 which have False/True + True/False, the result is True/False. We provide a complete proof for the correctness of Table 6.6 in Lemma A.2.3 in Section A.2.

Denoting the initial variables `such_that_r` and `regular_r` the new variables such_that_n and regular_n, and storing the result values back in such_that_r and regular_r, the logic for truth table may be collapsed to the Promela statements shown in Figure 6.5. By straightforward enumeration of the truth table resulting from the code we arrive at the following lemma:

Lemma 6.2.1 *The combination of the results from two overloaded constraints using Table 6.6 yields true for* `such_that` *in the Result column if and only if* `such_that_r` *is equal to* `true` *after executing the statements in Figure 6.5. Similarly, Table 6.6 yields true for* `regular` *in the Result column if and only if* `regular_r` *is equal to* `true` *after executing the statements in Figure 6.5.*

```
if
:: such_that_r==such_that_n ->  regular_r=(regular_r || regular_n);
:: such_that_r==true  && such_that_n==false -> regular_r=regular_r;
:: such_that_r==false && such_that_n==true  -> regular_r=regular_n;
fi;
such_that_r = such_that_r || such_that_n;
```

Figure 6.5: Promela code for the combination of overloaded constraint results

Since the above result combination method differs from non-overloaded constraints, the structure for an overloaded constraint differs slightly from Figure 6.4 and due to its length is located in Appendix A.2. As a summary, let CST2 be an overloaded constraint process with constraints $c_1, c_2, c_3, \ldots, c_n$. CST2 first executes both c_1 and c_2 and combines their results using the Promela code above. For each subsequent constraint $e_i : i > 2$, it executes c_i and combines its results with the stored results so far. After executing c_n and combining its results with the stored results, the process sends back the final results along its response channel CST2_response_chan.

6.2.5 Translating Scope

A constraint scope is a list of commands that a constraint applies to. As noted above in Section 6.2.2, commands are given unique indexes of type CMD. A scope is implemented as a list of type CMD which is compared against the local variable command. The general format for a scope check for a scope = $\{e_1, e_2, \ldots, e_n\}$ is as follows. The format is equivalent logically to checking an OR of all of the command comparisons and could just as easily be implemented in that manner. The format presented here makes it slightly easier for automated translation.

```
:: e1 == command -> scope = true;
:: e2 == command -> scope = true;
...
:: en == command -> scope = true;
```

The code above is enclosed in the `if/fi` selection structures at label L1 in both the non-overloaded and overloaded constraint structures discussed above.

6.2.6 Translating Commands

As per Definition 5.2.8, commands have the following structure:

CMD	e(a, s, r, P, f, f', msg)
if	ψ_1
and	...
then	ω_1^t
and	...
and	**return true**
else	ω_1^f
and	...
and	**return false**

Command processes accept requests with no parameter other than an `mtype` called `command_request` (as per Table 6.5 over a channel of type `[0] of {mtype}`. For a command named `CMD1`, the request channel type is: `CMD1_request_chan = [0] of {mtype};` Command processes return one boolean value using an `mtype` called `command_result` and over a channel of type `[0] of {mtype, bool}`. For `CMD1`, the response channel type is: `CMD1_response_chan = [0] of {mtype, bool};`

We assign each command process a unique value of type CMD, defined to be `int`:

```
#define CMD int
```

```
2   #define MAXCMD 2
3   CMD Cmd1 = 0;
4   CMD Cmd2 = 1;
```

Like agent and object numbers, command numbers are unique, consecutive beginning from 0, and are hard coded into the model. In the above example, there are two `CMD` variables defined: `Cmd1` (line 3) for the first command and `Cmd2` for the second command. The total number of commands is denoted MAXCMD and defined in line 2. A constraint scope `scope` is list of CMD values that it accepts. The list may be searched during execution using an `if/fi` selection structure. If the CMD value of the invoking command matches one of the list, the invoking command is in scope. See Section 6.2.5 for more details. We also use `CMD` variables to fill in log and inform notes. As such, each process stores a local instance of a `CMD` variable called `command` which is set to the index of the executing command.

Commands are translated similarly to constraints, using the same general structure for the processing of parameters, guards, and return values. Commands differ in structure from constraints in including operations which update the knowledge state. The general structure for a command in Promela is shown in Figure 6.6.

```
active proctype CMD1() {
   bool result = true; CMD command = Cmd1; bool scope; bool such_that;
   bool temp = true;  bool regular; JUDGMENT j;
   do
   :: CMD1_request_chan?command_request ->
      result = true; temp = true;
      L1: (Guards go here)

      if
      :: result == true -> L2: (True operations go here)
      :: else -> L3: (False operations go here)
      fi;

      CMD1_response_chan!command_response(result);
  od;
}
```

Figure 6.6: Promela framework for commands

Lines 1–3 are as above, declaring the process name and temporary variables. Lines 4–15 are the main `do/od` loop for the process. Line 5 is the only choice, listening for messages on the process' request channel `CMD1_request_chan`. The line listens for messages with `mtype` of is `command_request` that have no other payload. Line 6 re-initiates the temporary variables for a new iteration, wiping them from any stale values. Location L1 on Line 7 is where the translations of all guards $\psi_1, \ldots$ are placed based on the guard translation Table 6.7. Line 9 begins an `if/fi` construct which selects an option based on the value of `result`. Line 10 is selected if `result` is true and the operations $\omega_1^t, \ldots$ are inserted at label L2 there based on the operations translations discussed in Section 6.2.3. Line 11 is selected if `result` is false and the operations $\omega_1^f, \ldots$ are inserted at label L3. Line 14 returns the command's result over the command's response channel `CMD1_response_chan`. Lines 15–16 close out the process.

To give the reader a feeling for how a full command appears in Promela, we now show a complete sample command.

Example 6.2.2 (Sample Command Translation)

The rules for use of protected health information for treatment under [§164.506(a)(1), v.2000] are presented in Example 7.1.1. We show here the translation to Promela of one sample command from the example, TreatmentUse506a1.

CMD TreatmentUse506a1 (a, s, r, P, f, f', msg)
if Permitted506a1(a, s, r, P, f, f', msg) $\in$ {Allow}
and individual **in** Roles(s)
and healthCareProvider **in** Roles(a)
and local **in** (a, f)
and treatment **in**$_f$ P
and use **in**$_a$ P
then **insert** treatment **in** (a, s)
and **return true**
else **return false**

The result from translating the above command into Promela is:

```
1   active proctype pTreatmentUse506a1() {
```

```
bool result = true; CMD command = TreatmentUse506a1;
bool scope; bool such_that;
bool temp = true;  bool regular; JUDGMENT j;

do
:: TreatmentUse506a1_request_chan?command_request ->
   result = true; temp = true;

/* Guards go here */
/* Permitted506a1(a, s, r, P, f, f', msg) in {Allow} */
Permitted506a1_request_chan!constraint_request;
Permitted506a1_response_chan?constraint_response(scope,
  such_that, regular);
reference_judgment(scope, such_that, regular, j);
if
:: j == ALLOW -> temp = true;
:: else -> temp = false;
fi;
result = result && temp;
/* individual in Roles(s) */
temp = (a < MAXAGENT && a >= 0 && mroles[s].individual == 1);
result = result && temp;
/* healthCareProvider in Roles(a) */
temp = (a < MAXAGENT && a >= 0 && mroles[s].healthCareProvider == 1);
result = result && temp;
/* local in (a, f) */
temp = (f < MAXOBJECT && f >= 0 && objects[f] == 1 && a < MAXAGENT
  && a >= 0 && m.mat[a].objects[f].blocal == 1);
result = result && temp;
 /* treatment in_f P */
```

```
    run isPermittedByA();
    isForbiddenByA_request_chan!purpose_request(treatment);
    isForbiddenByA_response_chan?purpose_response(temp);
    result = result && temp;
    /* use in_a P */
    run isPermittedByA();
    isPermittedByA_request_chan!purpose_request(use);
    isPermittedByA_response_chan?purpose_response(temp);
    result = result && temp;

    if
    :: result == true ->
      /* True operations go here */
      /* insert treatment in (a, s) */
      if
      :: (f >= MAXOBJECT || f < 0 || objects[s] == 0 || a < 0 ||
         a >= MAXAGENT) -> result = false;
      :: else -> m.mat[a].objects[s].treatment = 1;
      fi;
    :: else -> skip; /* False operations go here */
    fi;

    TreatmentUse506a1_response_chan!command_response(result);
    od;
}
```

The translation places the functionality for the command inside the Promela structure for commands shown above in Figure 6.6. The guards and operations are translated as per Table 6.8 and 6.7. □

6.2.7 Judgment Derivation

As noted in Section 6.2.4, constraints return three boolean results: `scope`, `such_that`, and `regular`. The Promela logic for deriving judgments from the boolean results is as follows. For constraint search, we use the algorithm in Figure 6.7 which implements the logic in Table 5.10. For constraint references, we use the algorithm in Figure 6.8 which implements the logic in Table 5.8.

```
inline cst_search_judgment(scope, such_that, regular, j)
{
   if
   :: scope  && such_that  && regular          -> j = ALLOW;
   :: scope  && such_that  && regular == false -> j = FORBID;
   :: else -> j = IGNORE;
   fi;
}
```

Figure 6.7: Code for deriving a constraint search judgment

```
inline reference_judgment(scope, such_that, regular, j)
{
   if
   :: such_that          && regular          -> j = ALLOW;
   :: such_that          && regular == false -> j = FORBID;
   :: such_that == false && regular          -> j = DONT_CARE_ALLOW;
   :: such_that          && regular == false -> j = DONT_CARE_FORBID;
   fi;
}
```

Figure 6.8: Code for deriving a constraint reference judgment

The algorithms in both figures are enclosed in `inline` functions which are similar to C preprocessor macros. They accept four parameters, the three boolean results to examine and a result `j` of type `JUDGMENT`. As defined, in pre-command invocation the judgment is Ignore (Allow) if either scope or the *such that* guards are false and scope is ignored during regular invocation.

For convenience we show two technical lemmas to formalize the purpose of the above inline functions. By straightforward enumeration of the truth table derived from the `if/fi` structure in `cst_search_judgment` and compared with Table 5.10 we arrive at the following lemma:

Lemma 6.2.2 *The inline function `cst_search_judgment` stores `ALLOW` (`FORBID`, `IGNORE`) in `j` if and only if the judgment Allow (Forbid, Ignore) is derived from Table 5.10 with `scope` storing the boolean value b_{scp}, `such_that` storing b_{st}, and `regular` storing b_r.*

By straightforward enumeration of the truth table derived from the `if/fi` structure in `reference_judgment` and compared with Table 5.8 we arrive at the following lemma:

Lemma 6.2.3 *The inline function `reference_judgment` stores `ALLOW` (`FORBID`, `DONT_CARE_ALLOW`, `DONT_CARE_FORBID`) in `j` if and only if the judgment Allow (Forbid, Don't Care/Allow, Don't Care/Forbid) is derived from Table 5.8 with `scope` storing the boolean value b_{scp}, `such_that` storing b_{st}, and `regular` storing b_r.*

6.3 Building Models

Using the translation methodology presented in Section 6.2, we create processes and properties out of the commands and constraints which make up a Privacy API. In addition to the processes, however, in order to make the Promela model work we need some structural code to support queries, constraint search, and transaction monitoring. The structural code is mostly generic for all models, but requires tailoring in some specific locations to fit the precise number and names used in the model. We first discuss the code for constraint search and then the code for transaction processing.

6.3.1 Constraint Search

In Section 5.4.3 we discuss how the evaluation engine in the formal model performs a constraint search. Figure 5.22 shows a pseudocode representation of the process which we adapt for use in Promela models. As discussed in Section 6.2.2, each command is assigned a value of type `CMD` which is used for checking the scope of constraints (Section 6.2.5) and recording log notes and inform messages (Section 6.2.3). The constraint search algorithm also uses the CMD value for commands in combination with the rest of the global variables shown in Table 6.4.

The constraint search is performed by a process called Cst_Search shown in Figure 6.9 which listens on the channel Cst_Search_request_chan. Before a command is run at the top level of the model, Cst_Search is sent a message of mtype constraint_request (see Table 6.5) with a parameter e indicating the CMD to be checked. The constraint search code then runs each constraint process available, providing e as the running command. As noted in Section 6.2.4, each constraint process evaluates the command name and global variables and returns three boolean results (scope, such_that, regular). Cst_Search uses the return values to derive judgments as per Table 5.10. If any judgment does yields Forbid, the constraint search returns false and the command CMD will not be run. The code for investigating a single constraint CST1 is in Figure 6.9.

```
active proctype Cst_Search()
{
   JUDGMENT j; JUDGMENT final; CMD e;
   bool scope; bool such_that; bool regular;
   do
   :: Cst_Search_request_chan?constraint_request(e);
      j = final = ALLOW;
      CST1_request_chan!constraint_request(e);
      CST1_response_chan?constraint_response(scope, such_that,
      regular);
      cst_search_judgment(scope, such_that, regular, j);
      if
      :: j == FORBID -> final = FORBID;
      fi;
      L1: (More constraints)
      if
      :: final == FORBID ->
         Cst_Search_response_chan!command_response(false);
      :: else ->
         Cst_Search_response_chan!command_response(true);
      fi;
   od
}
```

Figure 6.9: Promela code for single constraint search

The code in Figure 6.9 first declares several local variables on lines 3–4. The variables `j` and `final` are used to store judgments from the various constraints as execution proceeds. The other variables are as in processes in this section. Lines 5–21 are the main loop for the process. Line 7 resets the judgment variables for use. The others will be overwritten as constraints are queried. Lines 8–10 query the constraint CST1 and wait for its responses. Line 11 uses the inline function `cst_search_judgment` shown in Figure 6.7 to derive the judgment from the constraint's results. Lines 12–14 check the resulting judgment and set the final judgment to be `FORBID` if the result is `FORBID`. Lines 16–21 return a boolean value (using the `command_response` message type since it takes just one boolean) indicating whether the command is to be allowed (*i.e.,* `true`) or forbidden (*i.e.,* `false`).

The code in Figure 6.9 is structured for just a single constraint. For policies with more than one constraint, lines 8–14 would be repeated for each constraint and inserted on line 15 at the label L1. The final judgment accumulates in the variable `final` and is used at the end to derive the final boolean message to send. Figure 6.12 shows the general structure.

6.3.2 Transaction Processing

As discussed in Section 6.2.6, command processes return boolean results indicating success or failure (*i.e.,* an operation got stuck). When a command successfully completes its operations, its updates to the knowledge state need to be made permanent so subsequent commands and constraints see them. Conversely, when a command gets stuck in any of its operations because its preconditions were not satisfied, we choose a failure mode wherein any changes effected by them are not committed to the global variables. The exception to this rule is array overflows which are caught by asserts as noted in Section 6.2.6 since they are not recoverable and are not policy based errors (*i.e.,* increasing the maximum array size would solve the problem).

We implement the transaction enforcement functionality at the top level of the model, in the code that handles the running of commands. Before each command is run, a snapshot of the knowledge state (*i.e.,* global variables in Table 6.4) is stored in temporary variables using the inline function `snapshot` shown in Appendix A.3.1. If the command returns true, the next command may be run and the snapshot is forgotten. If the command returns false, the snapshot is restored to the working copy of the variables, undoing any changes performed by the command that had just run. The snapshot restoration is performed by the inline function `restore` shown in Appendix A.3.2. Code to perform checking and restoration is shown in Figure 6.10. The example shown has one command CMD1 which is executed and checked for success.

```
snapshot();
Constraint_search_request_chan!search(CMD1);
Constraint_search_response_chan?command_response(temp);
if
:: temp -> CMD1_request_chan!command_request;
           CMD1_response_chan?command_response(temp);
           if
           :: temp == false -> restore();
           :: else -> skip;
           fi;
:: else -> skip;
fi;
```

Figure 6.10: Promela code for pre- and post-command state management

In order to support multiple commands, we copy the code in Figure 6.10 for each command to be included in the model. Space state exploration is then performed by letting the SPIN model checker explore execution paths by executing series of commands.

6.4 Equivalence of Promela Model and Privacy Commands

We now have two representations for the operational behavior of a Privacy API: the Privacy Commands representation detailed in Chapter 5 and the Promela model representation detailed in this chapter. In Section 6.2 we discussed the structural translation between Privacy Commands and Promela, but have not proved its correctness. We do so in this section, proving that the Promela model behaves in a semantically equivalent manner to the Privacy Commands from which it is derived. We first present a description of the semantics of the subset of Promela which our models use. We then present the correspondences between the formal model and the Promela model, detailing the definitions and invariants used in the code. Using the correspondences and definitions, we then show an equivalence theorem for the two representations by proceeding upwards in structural complexity, considering first operations and guards and then commands and constraints.

Our general approach is illustrated in Figure 6.11. The formal model's knowledge state (A, O, m, l) evolves via Privacy Commands to a new state (A, O', m', l'). Let us denote the equivalent Promela variable state for a given knowledge state $\mathsf{Pr}(A, O, m, l)$. In order to show equivalence for commands, therefore we show equivalence mappings between the initial knowledge state (A, O, m, l) and a corresponding SPIN state in Promela $\mathsf{Pr}(A, O, m, l)$ and the final knowledge state (A, O', m', l') and the resulting SPIN state in Promela $\mathsf{Pr}(A, O', m', l')$. By showing that the initial and final states correspond for every command, we show that the Privacy Commands and SPIN versions have the same effects.

$$\begin{tikzcd}
(A, O, m, l) \arrow[r, "\texttt{Commands}"] \arrow[d, "\cong"'] & (A, O', m', l') \arrow[d, "\cong"'] \\
\mathsf{Pr}(A, O, m, l) \arrow[r, "\texttt{Processes}"] & \mathsf{Pr}(A, O', m', l')
\end{tikzcd}$$

Figure 6.11: Equivalence mappings

Notation As a convention in this section, we use the following notation. Variables (`a`), expressions (`true && false`), and statements written in `teletype font` represent code fragments in Promela. Variables written in mathematical italics (*e.g.,* a, o, P) represent variables, sets, or functions defined in the formal model. Operators written in mathematical italics font (*e.g.,* $=$, $<$, $\leq$) have their usual mathematical definitions. Operators written in `teletype font (e.g., =, ==, <, <=)` have the semantics of Promela. Thus, the mathematical expressions $a = b$ and `a` $=$ `b` are a comparison between the values stored in variables a and b while the code fragment `a = b` in Promela stores the value of variable `b` in variable `a` (see Table 6.1).

The correspondence between the knowledge representation in the formal model and the Promela code depends on the correspondence in the storage representations which we discuss in Section 6.2.2. In order to show correspondence between Privacy Commands and the Promela processes in Section 6.4, we must first show that the transitions induced by the two lead to corresponding states in the respective representations. In order to do so, we first present the operational semantics for the code in the Promela model and its effects on the state it stores in terms of variables and arrays. We then show the correspondence between the semantic effects of the Promela processes and the commands and constraints that they implement.

6.4.1 Semantics of the Promela Model

We present the semantics of the Promela model in terms of the knowledge state in Promela code. The universe we consider in the Promela code consists of the following variables and sets:

Definitions

`AGENT` The variables assigned type `AGENT`. The number of variables with the type `AGENT` corresponds to the constant `MAXAGENT` while the members are indexed consecutively on the interval $[0, \texttt{MAXAGENT})$. An integer `n` represents an agent therefore if $\texttt{0} \leq \texttt{n} < \texttt{MAXAGENT}$.

`OBJECT` The variables assigned the type `OBJECT` combined with the variables assigned the type `AGENT`. The number of variables corresponds to the constant `MAXOBJECT` while the members are indexed consecutively on the interval $[0, \texttt{MAXOBJECT})$. The variables solely of type `OBJECT` are indexed consecutively on the interval $[\texttt{MAXAGENT}, \texttt{MAXOBJECT})$. An integer `n` therefore represents an object if $\texttt{0} \leq \texttt{n} < \texttt{MAXOBJECT}$ and an object which is not an agent if $\texttt{MAXAGENT} \leq \texttt{n} < \texttt{MAXOBJECT}$.

`mroles` The roles array which is of size `MAXAGENT`. $\forall$ `n` such that $\texttt{0} \leq \texttt{n} < \texttt{MAXAGENT}$, the array is defined and its contents are valid.

`Roles` The record for storing roles. A Roles record contains a bit flag for each right $k \in \mathit{Role}$ which has the same name (*i.e.,* `k`) as the role in the formal model. For a Roles record `R`, if $\texttt{R.k} = 1$ then the role `k` (and consequently k) is true for the associated agent. If $\texttt{R.k} = 0$ then the right `k` (and consequently k) is not true for the associated agent.

`objects` The array of objects. The array is of size `MAXOBJECT`. The array is defined for and its contents are valid for any index `n` such that `0` $\leq$ `n` $<$ `topObj`. $\forall$ `n` . `objects[n]` $=$ `0`, `n` does not exist. $\forall$ `n` . `objects[n]` $=$ `1`, `n` exists. The variable `topObj` points to the next open slot in `objects`, so its value is on the interval [MAXAGENT, MAXOBJECT) and $\forall j \in$ [topObj, MAXOBJECT), `objects`$[j] = 0$.

`mtags` The tags array. The array is of size `MAXOBJECT`. The array is defined for any index `n` . `0` $\leq$ `n` $<$ `MAXOBJECT`. $\forall$ `n` . `0` $\leq$ `n < topObj` and `objects[n]` $=$ `1`, the contents of `mtags[n]` are valid.

`m` The rights matrix. The array is of size `MAXAGENT` $\times$ `MAXOBJECT`. Index `m(a,o)` is written `m.mat[a].objects[o]`. The matrix is defined $\forall$ `m(a,o)` . `a` $<$ `MAXAGENT` $\wedge$ `o` $<$ `MAXOBJECT`. Its contents are valid if `objects[o]` $=$ `1`. Each valid entry `m(a,o)` contains a Rights record `R`.

`Rights` The record for storing rights. A Rights record contains a bit flag for each right $d \in$ *Right* which has the same name (*i.e.,* `d`) as the right in the formal model. For a Rights record `R`, if `R.d` $= 1$ then the right `d` (and consequently d) is present. If `R.d` $= 0$ then the right `d` (and consequently d) is not present.

`PURPOSE` The variables of type `PURPOSE`. The number of variables with the type `PURPOSE` corresponds to the constant `MAXPURPOSE` while the members are indexed consecutively on the interval [0, `MAXPURPOSE`). An integer `n` represents a purpose if `0` $\leq$ `n` $<$ `MAXPURPOSE`. The array `parent` describes the partial order over the members.

`parent` The purpose partial order array. The array is of size `MAXPURPOSE`. $\forall$ `n` . `0` $\leq$ `n` $<$ `MAXPURPOSE`, `n` is defined and its contents are valid. For an index `c` . `parent[c]` $=$ `p`, `p` is `c`'s parent and `c` is `p`'s child.

`log` The log array. The array is of size `MAXLOG`. $\forall$ `n` . `0` $\leq$ `n` $<$ `MAXLOG`, the array is defined. $\forall$ `n` . `0` $\leq$ `n` $<$ `topLog`, the contents are valid. The variable `topLog` points to the next open slot in `log`, so its value is on the interval [0, MAXLOG), $\forall i \in$ [0, topLog).

`inform` The record of messages sent. The array is of size `MAXINFORM`. $\forall$ `n` . `0` $\leq$ `n` $<$ `MAXINFORM`, the array is defined. $\forall$ `n` . `0` $\leq$ `n` $<$ `topInform`, the contents are valid. The variable `topInform` points to the next open slot in `inform`, so its value is on the interval [0, MAXINFORM).

`CMD` The currently executing command type. The instance variable `CMD command` stores the index of the currently executing command.

Invariants

Models enforce the following invariants:

1. `topObj` $\leq$ `MAXOBJECT`. The next available slot for objects may not exceed the number of object slots available.

2. `topObj` points to the next open slot in `objects`: $\forall i \in$[topObj, MAXOBJECT), `objects`$[i] = 0$.

3. `topLog` $\leq$ `MAXLOG`. The next available slot for logs may not exceed the number of object slots available.

4. `topLog` points to the next open slot in `log` and $0 \leq$ topLog $<$ MAXLOG.

5. `topInform` $\leq$ `MAXINFORM`. The next available slot for inform entries may not exceed the number of object slots available.

6. `topInform` points to the top open slot in `inform` and $0 \leq$ topLog $<$ MAXLOG.

7. `MAXOBJECT` $\geq$ `MAXAGENT`. There must be at least `MAXAGENT` object slots.

8. $\nexists\, \mathtt{p1}, \mathtt{p2} \in [0, \mathtt{MAXPURPOSE})\ .\ \mathtt{p1} \neq \mathtt{p2} \wedge \mathtt{p1} \in \mathsf{ancestors}(\mathtt{p2}) \wedge \mathtt{p2} \in \mathsf{ancestors}(\mathtt{p1})$. There are no loops (cycles) in the purpose partial order.

9. `command` contains the index of the currently executing command.

6.4.2 Translation Correspondence

Based on the above semantics for Promela and the semantics for Privacy Command in Section 5.3, we argue the following lemmas showing that the Promela model derived from the above translation methodology yields a model with corresponding semantics to the formal model from which it is derived. We reach the final theorem showing the desired correspondence by first showing correspondence lemmas for the constraint search code (Lemma 6.4.3), guards (Lemma 6.4.4), and operations (Lemma 6.4.5). We then use then use those lemmas to show correspondence lemmas for commands (Lemma 6.4.6) and constraints (Lemma 6.4.7). With those lemmas we prove the final correspondence theorem, Theorem 2.

Before proceeding to the main correspondence theorem, we first prove a few technical lemmas for objects and agents.

Lemma 6.4.1 *(Object Existence) For a knowledge state $s = (A, O, m, l)$ and a corresponding Promela representation as per Section 6.2.2, an object $o \in Object$ if and only if the Promela expression* `o < MAXOBJECT && o >= 0 && objects[o] == 1` *evaluates to true.*

Proof: By the definition of `OBJECT` above, an integer `o` represents an object if $0 \leq$ `o` $<$ `MAXOBJECT` and it exists if `objects[o]` $= 1$, which is checked by the expression. If `o` is not in the range `0` $\leq$ `o` $<$ `MAXOBJECT`, then it will fall outsize of the bounds for the array `objects` and the array lookup will fail. The expression therefore first checks that `o` falls in the correct range. If $0 \leq$ `o` $<$ `MAXOBJECT` and `objects[o]` $\neq 1$ then the object does not exist and correspondingly, the result of the Promela check is false. □

Lemma 6.4.2 *(Agent Existence) For a knowledge state $s = (A, O, m, l)$ and a corresponding Promela representation as per Section 6.2.2, an object $a \in Agent$ if and only if the Promela expression* `a < MAXAGENT && a >= 0` *evaluates to true.*

Proof: By the definition of `AGENT` above, an integer `a` represents an agent if $0 \leq$ `a` $<$ `MAXAGENT`. The expression therefore first checks that `a` falls in the correct range. If $0 >$ `o` or `o` $\geq$ `MAXAGENT` then the agent does not exist and correspondingly, the result of the Promela check is false. □

Lemma 6.4.3 *(Constraint Search) For a knowledge state s and a command to be executed e, the final result of performing a constraint search as per Section 5.4.3 and derived from Table 5.10 forbids e to execute if and only if the constraint search code in Figure 6.9 as adapted for $|Constraint|$ number constraints sends false over its response channel.*

Proof: We argue the correctness of the code by induction on the size of the constraint set $|Constraint|$. There are two cases to consider: the base case when $|Constraint| = 1$ and the step case when $|Constraint| = n$ for $n > 1$. In the first case, there is only one constraint to query and so the code to combine the judgments is not exercised. In the second case, the combination code is exercised to combine the judgments from multiple constraints. The proof below relies on two invariants. Let `CSTn` be the process which implements constraint $c_n \in Constraint$ such that $1 \leq n \leq |Constraint|$. After the execution of `CSTn`, processing by the inline function `cst_search_judgment`, and the `if/fi` structure on lines 12–14, the following two properties hold:

Invariant 1: `j` holds the judgment from c_n.

Invariant 2: `final` holds `FORBID` if and only if $\exists i \leq n$ such that the judgment of c_i is Forbid. Otherwise, `final` holds `ALLOW`.

Let use refer to lines 8–14 for a constraint `CSTn` as *the evaluation of* c_n. Our induction hypothesis is thus the following:

Induction Hypothesis: After evaluation of c_n, Invariants 1 and 2 are true.

Case $|Constraint| = 1$ When there is only one constraint, the code for the constraint search algorithm is as in Figure 6.9 with the label L1 empty (*i.e.,* a `skip;` statement). Let the constraint $c \in Constraint$ be `CST1` as per the figure. Lines 3–4 of the code declare the temporary variables used during evaluation. Judgment variables `j` and `final` store the temporary and final judgments of the process respectively. Command variable `e` stores the command to be executed (*i.e.,* the command for which the constraint search is being performed). Boolean variables `scope`, `such_that`, and `regular` store the results from constraints as returned over their response channels. Lines 5–21 are the main loop for the constraint searching algorithm and loop to perform the search process for each request received. Line 6 listens for a constraint search request on the request channel. The channel's type is `chan Cst_Search__request_chan = [0] of {mtype, CMD};`. The listen request accepts messages of type `constraint_request` with an executing command `e`.

When a request arrives, the loop begins execution. Line 7 resets the values of `j` and `final` to `Allow` for the iteration. The judgment from the process is thus `Allow` by default unless it's changed by a response from a constraint as required by Invariant 2. Lines 8–10 query the constraint `CST1`. As in Lemmas 6.4.7 and A.2.4, querying the process `CST1` on its request channel `CST1_request_chan` with the command `e` returns the values `scope`, `such_that`, and `regular` on the response channel `CST1_response_chan`. As shown in the lemmas, regardless of whether CST1 is overloaded, `scope`$= b_{scp}$, `such_that`$= b_{st}$, and `regular`$= b_r$ for command e. Line 11 uses the inline function `cst_search_judgment` to derive the judgment from the boolean values. As shown in Lemma 6.2.2, the function stores the judgment in `j` as per Table 5.10. The result is that the judgment from CST1 is stored in `j`. Thus, `j` is `FORBID` if and only if the judgment from CST1 is Forbid. This maintains Invariant 1 above.

Lines 12–14 check if the judgment from CST1 is Forbid. If it is, `final` gets the judgment value `FORBID`. Otherwise, it retains its previous value of `ALLOW`. Thus, `final` is valued `FORBID` if and only if `j` is `FORBID`, which as shown is only if the judgment from CST1 is Forbid. Otherwise, `final` is Allow. If `j` is `IGNORE` or `ALLOW`, note that `final` is still valued at `ALLOW`. This establishes Invariant 2.

Since $|Constraint| = 1$, Line 15 is ignored. Lines 16–21 return true or false over the constraint response channel. The code returns `false` if and only if the value of `final` is `FORBID` which is equivalent to CST1 yielding the judgment Forbid. Otherwise, if `final` has the value `ALLOW`, line 20 returns `true`. As noted above in Section 5.4.1, e may execute if the judgment from the constraint search is either Ignore (Allow) or Allow and as shown, `true` is sent on the channel if and only if CST1's judgment is Ignore (Allow) or Allow.

Step: $|Constraint| = n$ When there is more than one constraint, the code for the constraint search algorithm is longer by repetition of lines 8–14 for each constraint. For example, Figure 6.12 shows the constraint search algorithm for n constraints, skipping the code for constraints $2 \ldots n-1$.

The proof for $|Constraint| = 1$ proves the correctness of lines 1–14 and shows that both invariants are true at the end of line 14. For $n > 1$, from the induction hypothesis we may assume Invariants 1 and 2 are true, so `j` holds the value from the constraint c_{n-1} and `final` is `FORBID` if and only if $\exists i < n$ such that the judgment of c_i is Forbid.

Let `CSTn` be the process that implements constraint c_n. Let us refer to the code in Figure 6.12. From the argument above in the base case we have that lines 15–17 store the boolean results of `CSTn`. The inline function `cst_search_judgment` stores therefore stores the judgment for CSTn (and thus c_n) in `j` on line 18, establishing Invariant 1.

Lines 19–21 store `FORBID` in `final` if and only if `j` stores `FORBID`. There are three cases to consider:

1. If $\exists i < n$ such that the judgment of c_i is Forbid, `final` already stores `FORBID` and regardless of c_n's judgment, `final` remains `FORBID`.

2. If $\nexists i < n$ such that the judgment of c_i is Forbid, then if c_n yields Forbid, we have that $\exists i \leq n$ such that the judgment of c_i is Forbid for the case $i = n$ and correspondingly `final` is set to `FORBID`.

3. If c_n yields Ignore (Allow) or Allow, then we still have $\nexists i \leq n$ such that the judgment of c_i is Forbid and correspondingly, `final` remains unchanged. By the induction hypothesis, we have that `final` is therefore `ALLOW`.

We have therefore established Invariant 2.

We have now shown that both Invariants 1 and 2 are true after the evaluation of c_n and thus that `final` always contains `FORBID` if and only if $\exists c_i \in Constraint$ such that c_i's judgment is Forbid. Otherwise, `final` contains `ALLOW`. As shown in the base case, the code in lines 23–28 sends `false` over the response channel `Cst_search_response_chan` if and only if `final` contains `FORBID`. Otherwise, if `final` contains `Allow`, the lines send `true`. Since as defined in Section 5.4.3, e is forbidden to execute if and only if the judgment of any constraint is Forbid, we have the desired correspondence. $\square$

Using the above lemma, we now prove an equivalence lemma for guards:

Lemma 6.4.4 *(Guard Correspondence) For a knowledge state $s = (A, O, m, l)$, an argument set $a = (a, s, r, P, f, f', msg)$, and a corresponding Promela representation for each as per Section 6.2.2, the result of any guard ψ at s with a is true if and only if the corresponding guard translation as per Section 6.2.3 yields true.*

Proof: We proceed in the proof structurally, considering each guard independently. We consider the if and only if directions simultaneously, showing that the results of the ψ guards coincide with the result of the Promela code. We use the code translations as per Table 6.7. Let us assume that the representations have been initialized correctly and are in a corresponding state. Let `a`, `s`, `r` be integers of type `AGENT`, `o` be an integer of type `OBJECT`, `p` be an integer of type `PURPOSE`, `P` be of type `PURPOSE[MAXPURPOSE]`, `b` be a boolean. Let $a, s, r \in Agent$ and a, s, r's indices in *Agent* be stored in `a`, `s`, `r` respectively. Let $o \in Object$ and o's index in *Object* be stored in `o`.

d **in** (a, o) The guard is true iff $a \in Agent$, $o \in Object$, and d is in $m(a, o)$. The corresponding Promela code examines the logical AND of six expressions. The first three expressions (1) check that `o` is an object (`o < MAXOBJECT && o >= 0 && objects[o] == 1`), the next two expressions (2) check that `a` is an agent (`a < MAXAGENT && a >= 0`), and the last expression (3) checks for the presence of the right (`m.mat[a].objects[o].d == 1`).

The first expressions (1) checks the existence of `o` as an object as per Lemma 6.4.1. The second expressions (2) check the existence of `a` as an agent as per Lemma 6.4.2.

The third expression (3) checks the status of the rights matrix and only is of relevance if `o` is an object which exists and `a` is an agent since if `o` is not an object or doesn't exist, (1) will be false, and if `a` is not an agent then (2) will be false. By the definition of `AGENT` above, an integer `a` is an agent if $0 \leq$ `a` $<$ `MAXAGENT`. If `a` is not in that range, then the guard is false since a doesn't correspond to an agent and correspondingly the lookup `m.mat[a]` will fail since `a` will fall out of bounds. If `a` is in the range, then the array index `m.mat[a].objects[o]` will refer to the `Rights` record that a has over o as per the definition of the matrix `m` in Section 6.4.1. As defined in Section 6.2.2, the rights record `m.mat[a].objects[o]` has bit flags for each of members of *Right*. For a right $d \in Right$, the corresponding bit flag `d` is set to 1 iff the agent has the right d on the object as is checked by (2) above.

$o.t = b$ The guard is true iff $o \in Object$ and tag $t = b$ for it. The corresponding Promela code examines the logical AND of four expressions. The first three expressions (1) check that `o` is an object (`o < MAXOBJECT && o >= 0 && objects[o] == 1`) and the last expression (2) checks that the tag is set to `b` (2) `mtags[o].t == b`.

The first expressions (1) checks the existence of `o` as per Lemma 6.4.1.

The last expression (2) checks the status of the tags arrays and only is of relevance if `o` is an object which exists since if `o` is not an object or doesn't exist, (1) will be false. By the definition of `mtags` in Section 6.4.1, `mtags` is defined for any index n such that `0` $\leq$ `n` $<$ `MAXOBJECT` and its contents are valid if `objects[o]` $= 1$. If `o` is not in the valid range or `objects[o]` $\neq$ `1` then (1) will be false and we may ignore (2). If `o` is in the valid range and `objects[o]` $= 1$ then the contents of `mtags[o]` will be a valid `Tags` record as per the definition in Section 6.4.1. As per the definition in Section 6.2.2, the tags record `mtags[o]` has boolean flags for each of members of Tag. If the tag t for o is set true then `mtags[o].t` will be true and if it is set to false then `mtags[o].t` will be false. Therefore, expression (2) examines whether `mtags[o].t` is equal to `b` which is the equivalent of checking whether $o.t = b$.

k **in** $Roles(a)$ The guard is true iff $a \in Agent$ and a has role k activated. The corresponding Promela code examines three expressions. The first two (1) check that `a` is an agent (`a < MAXAGENT && a >= 0`) and the last one (2) checks the role (`mroles[a].k == 1`).

The first expressions (1) check the existence of `a` as an agent as per Lemma 6.4.2.

The last expression (2) checks the present of the role `r` for the agent `a`. By the definition of `mroles` above, $\forall$ `n` . $0 \leq$ `n` $<$ `MAXAGENT` the array is defined and its contents are valid. If `a` is not in the range $0 \leq$ `a` $<$ `MAXAGENT` then by the definition of `AGENT` in Section 6.4.1, `a` does not represent an agent and so the expression and guard will both be false. If `a` is in the range then `a` represents an agent and `mroles[a]` contains a `Roles` record. As defined in Section 6.2.2, a `Roles` record contains a bit flag for each role that may be activated, so if `mroles[a].k` $= 1$ then the role `k` is activated for `a`. By the definition of `Roles`, therefore k is activated for a as well. If `mroles[a].k` $\neq 1$ then the role `k` is not activated for `a` and similarly k is not activated for a either.

p **in**$_a$ P The guard is true iff $\exists p' \in P \ . \ p' \in \mathsf{descendants}(p)$. The corresponding Promela code (1) starts the `isPermittedByA` process (`run isPermittedByA()`), (2) queries it with the `PURPOSE` variable `p` (`isPermittedByA_request_chan!purpose_request(p)`), and (3) waits for the response (`isPermittedByA_response_chan?purpose_response(temp)`).

The first statement (1) begins by running the process `isPermittedByA`. If the process count for SPIN is exceeded, this statement will fail, ending the verification run. The statement does not however, have any effect on the logic of the evaluation.

The second statement (2) sends a message to the process `isPermittedByA` with the variable parameter `p`. The third statement (3) listens for the response from the process on its response channel. The code for the process is shown in Appendix A.1. Lemma A.1.7 on Page 321 shows that the process returns true if and only if `p` in$_a$ `P` and the partial order is acyclic (as per invariant 8).

p **in**$_f$ P The guard is true iff $\exists p' \in P \ . \ p \in \{\mathsf{ancestors}(p') \cup \mathsf{descendants}(p')\}$. The corresponding Promela code (1) starts the `isForbiddenByA` process (`run isForbiddenByA()`), (2) queries it with the `PURPOSE` variable `p` (`isForbiddenByA_request_chan!purpose_request(p)`), and (3) waits for the response (`isForbiddenByA_response_chan?purpose_response(temp)`). The proof is similar to the proof for p in$_a$ P relying on Lemma A.1.8 on Page 323 instead.

$c(a, s, r, P, f, f', msg) \in J$ The guard is true if and only if running c with parameters a, s, r, P, f, f', msg by the evaluation engine yields boolean results b_{scp}, b_{st}, b_r that yield a judgment $j \in J = \{j_1, \ldots, j_n\}$ as per Table 5.8. The Promela code (1) sends a request to the constraint process `c` over its request channel (`c_request_chan!constraint_request`) and (2) listens for its response on its response channel (`c_response_chan?constraint_response(scope, such_that, regular)`). It then (3) examines the boolean responses using an inline function (`reference_judgment(scope, such_that, regular, j)`) which stores the judgment in `j`. The code then examines `j` in an `if/fi` structure (4) for each `j_1, ..., j_n` to see if any match. If not, the result is stored as false (`if ::j == j_1 -> temp=true; ::...::else -> temp=false`).

The first two statements (1)–(2) send and receive messages from the appropriate constraint `c`. If `c` is not overloaded then it properly models the behavior of c as per Lemma 6.4.7. If `c` is overloaded then it properly models the behavior of referencing and combining all constraints $c_1, \ldots, c_n$ with the name c as shown in Lemma A.2.4. Statement (3) uses the inline function `reference_judgment` to derive the correct judgment from Table 5.8 as is shown in Lemma 6.2.3. Statement (4) compares `j` against all of the members of J by using a equivalence test as the guard for each `::` option. The `if/fi` statement evaluates all of the guards and then randomly selects one for execution. $\forall j_i \in J \;.\; j_i =$ `j`, the option `::j == j_i` will evaluate to true and therefore may be selected. The result is that `temp` will be set to true. If $\nexists j_i \in J \;.\; j_i =$ `j`, then none of the guards will be satisfied and the `else` option will be selected, setting the result to false. This corresponds to the semantics of the formal guard since if the resulting judgment from the constraint is present in J, its value is true. Otherwise, its value is false.

$a_1 = a_2$ The guard is true iff $a_1, a_2 \in Agent$ and they refer to the same agent name. The corresponding Promela code checks the logical AND of five conditions. First it (1) checks the interval of `a1` (`a1 >= 0 && a1 < MAXAGENT`). Then it (2) checks the interval of `a2` (`a2 >= 0 && a2 < MAXAGENT`). Finally it (3) checks the equality of the `AGENT` variables (`a1 == a2`). As per the definition of the variable type `AGENT`, if both integers `a1` and `a2` are in the interval [0, `MAXAGENT`) then they both represent agents. Expression (1) checks the interval of `a1` to check that it represents an agent. Expression (2) checks the interval of `a2` to check that it represents an agent. If they are both on the interval then both (1) and (2) are true and comparing their integer values using the `==` operator is equivalent to examining their names in the formal model. If they coincide then the result is true, otherwise the result is false. If either `a1` or `a2` do not fall on the interval then (1) or (2) will be false respectively and therefor the entire expression will evaluate to false. This coincides with a_1 or a_2 not being agents which would make the guard evaluate to false. $\square$

Having now shown the correspondence between the members of *Guard* and their Promela translations, we next show a parallel lemma for showing the correspondence of operations and their Promela translations.

Lemma 6.4.5 *(Operation Correspondence) For a knowledge state $s = (A, O, m, l)$, a corresponding Promela representation as per Section 6.2.2, and an argument list $a = a, s, r, P, f, f', msg$, the result of any operation ω at s with a such that $s \xrightarrow{\omega} s'$ is a knowledge state $s' = (A, O', m', l')$ with a corresponding Promela representation as derived from the translation of ω in Table 6.8.*

Proof: We proceed in the proof structurally, considering each operation independently and showing that the results of the ω operations coincide with the outcome of the Promela code. We use the code translations as per Table 6.8. Our inductive assumption is that the representations have been initialized correctly and are in a corresponding state. We show that for each possible update using an operations ω, the resulting state from the Promela code corresponds to the result from executing ω. Let `a`, `s`, `r` be integers of type `AGENT`, `o` be an integer of type `OBJECT`, `p` be an integer of type `PURPOSE`, `P` be of type `PURPOSE[MAXPURPOSE]`, `b` be a boolean. Let $a, s, r \in$ *Agent* and a, s, r's indices in *Agent* be stored in `a`, `s`, `r` respectively. Let $o \in$ *Object* and o's index in *Object* be stored in `o`.

create object o The operation creates a new object o in O if no object o previously exists with that name. The resulting state is that $O' = O \cup \{o\}$. If $o \in$ *Object*, the outcome of the operation is undefined. The corresponding Promela code performs its operations under an `atomic` statement to ensure that it is not preempted by any other processes. It first (1) asserts that there is more space for objects to be created (`assert(topObj < MAXOBJECT)`). It then (2) stores the value of the next open space (`o = topObj`) and (3) creates the object entry in the objects vector (`objects[topObj] = 1`). Finally (4) it updates the top object slot (`topObj++`).

Placing the Promela code in an `atomic` block ensures that the variable `topObj` is not updated between the time that it is read and when the objects matrix is updated. The first statement (1) asserts that there is still space in the objects matrix to create another object by checking that `topObj` is less than `MAXOBJECT`. If the check fails, the model execution terminates. Since the size of the objects matrix is not part of the logic code, the error is not caught by the Promela model, but is handed back to the SPIN system for processing. If the check succeeds, the code continues unhindered.

The second statement (2) stores the value of the next available slot in the variable name `o` which corresponds to the new name assigned in the operation. If the name o already exists in *Object*, Section 5.3 indicates that the outcome is undefined. In the Promela model, we create a new object with the index of `topObj` and proceed. Statement (3) then sets the `objects` vector to indicate the new slot is occupied by an object as per the definition of `objects` above that a vector entry set to 1 corresponds to object creation. The setting of `objects[o]` $= 1$ corresponds to the addition of o to *Object*. Statement (4) then updates `topObj` to maintain invariant 2 above.

In order to handle two **create object** operations in a single command, the naming of the new variable in the Promela code is kept identical to the naming of the variable in the formal operation. Therefore subsequent operations may use the name `o` for any operations. For instance, if the operation **create object** o_1 is followed by **create object** o_2, the code maintains the names of the objects as per the formal model in case o_1 and o_2 are reused during the rest of the command.

delete object o The operation delete an object o that already exists in O. If o already exists then the result is that $O' = O - \{o\}$. If o does not exist then the operation is ignored. The corresponding Promela code predicates its actions on the existence of `o` using an `if/fi` structure. The first option checks two conditions (1) that the name `o` corresponds to an object and not an agent (`o < MAXOBJECT && o >= MAXAGENT`). If both conditions are satisfied then (2) the object is deleted (`objects[o] = 0`). If the conditions are not satisfied, the `else` just continues with the next operation (3) (`else -> skip`).

The first expression (1) checks two conditions. As defined above, an integer `o` represents an object if it falls in the range $0 \leq$ `o` $<$ `MAXOBJECT` and it represents an agent if it falls in the range $0 \leq$ `o` $<$ `MAXAGENT`. Since only objects can be deleted, we must check that `o` is an object but not an agent, in other words that $\texttt{MAXAGENT} \leq \texttt{o} < \texttt{MAXOBJECT}$. The two expressions `o < MAXOBJECT` and `o >= MAXAGENT` check that condition. If both are satisfied then `o` is an object and may be deleted with the operation. Otherwise, the operation is ignored (3).

If (1) is satisfied then the option is selected by the `if/fi` structure and statement (2) is performed. Statement (2) performs deletion by setting $\texttt{objects[o]} = 0$ as per the definition of `objects` above. The result is that `o` is deleted from the existing objects, paralleling the deletion of o from O'.

set $o.t = b$ The operation sets the value of the tag t on o to the boolean value b. The operation succeeds if o is an object and t is a tag on o. The result is that $O' = O$ except for o where $o.t = b$. The corresponding Promela code predicates its actions on the existence of `o`. It first checks that `o` is an object (1) (`o >= MAXOBJECT || o < 0 || objects[o] == 0`). If `o` is not an object then the operation is stuck. We indicate that by settings its result to to false (2) (`result = false`) which the model uses to indicate that the command got stuck. Otherwise, the tag `t` is set on `o` to the new boolean value (3) (`mtags[o].t = b`).

The first expression (1) checks that `o` is an existing object as per Lemma 6.4.1. The second statement (2) is run if `o` is not an object, leading to a stuck result if o does not exist.

The last statement (3) executes only if `o` is an existing object. As defined in Section 6.4.1, the array `mtags` is defined for and has valid contents for all existing objects (*i.e.,* $\forall n \ . \ 0 \leq n < \texttt{MAXOBJECT} \wedge \texttt{objects}[n] = 1$). Since (3) only runs if (1) is false (*i.e.,* if `o` is an existing object) we are sure that `mtags[o]` has valid contents of a `Tags` record. As defined in Section 6.2.2, the `Tags` record stores a boolean flag for each tag `t` that is set to true if and only the tag $o.t$ is true in the formal representation. Statement (3) stores the new boolean value for `mtags[o].t` which corresponds to the updating of $o.t = b$ in the new object set O'.

insert d **in** (a, o) The operation sets the right d for a on o. It succeeds if $a \in Agent$ and $o \in Object$. The result is that $m' = m$ except for $m'(a, o) = m(a, o) \cup \{d\}$. The corresponding Promela code inserts the new right for `a` on `o` in `m`. It first checks if `o` is an object (1) (`o >= MAXOBJECT || o < 0 || objects[o] == 0`) and if (2) `a` is an agent (`a < 0 || a >= MAXAGENT`). If `o` is not an object or `a` is not an agent then the operation is stuck which we indicate by setting the command's result to false (3) (`result = false`). Otherwise, (4) the right `r` is set (`m.mat[a].objects[o].d = 1`).

The first expression (1) checks that `o` is an existing object as per Lemma 6.4.1. The second expression (2) checks that `a` is an agent as per Lemma 6.4.2. Statement (3) executes if `o` is not an object or `a` is not an agent, leading to a stuck result if o or a do not exist.

Statement (4) executes only if `o` is an object and `a` is an agent. As defined in Section 6.4.1, the matrix `m` is defined for and contains valid contents for `m.mat[a].objects[o]` if `a` is an agent and `o` is an object. As defined in Section 6.2.2, its contents is a `Rights` record with bit flags for each right that an agent may hold over an object. The bit flags have the same name as the corresponding right in the formal model. Statement (4) sets the bit flag `m.mat[a].objects[o].r` $= 1$, corresponding to the update of $m'(a, o) = m(a, o) \cup \{r\}$ as per the definition of `m` and `Rights`.

delete d **from** (a, o) The operation removes the right d for a on o. It succeeds if $a \in$ *Agent* and $o \in$ *Object*. If $d \in m(a, o)$, the result is that $m' = m$ except for $m'(a, o) = m(a, o) - \{d\}$. If $d \notin m(a, o)$, the operation has no effect so $m' = m$. The corresponding Promela code is similar to the code for **insert** d **in** $m(a, o)$ except that statement (4) remove the right (`m.mat[a].objects[o].d = 0`) instead of inserting it.

The correctness proof is identical to the proof for **insert** d **in** $m(a, o)$ for (1)–(3). For statement (4) it is sufficient to note that if `a` is an agent and `o` is an object then the `Rights` record in `m.mat[a].objects[o]` has a bit flag `d` for the corresponding right d. As per the definition of `m` in Section 6.4.1 and `Rights` in Section 6.2.2, setting `m.mat[a].objects[o].d = 0` corresponds to the update of $m'(a, o) = m(a, o) - \{d\}$ if `a` previously had the right `d` on `o`. If `a` did not have the right `d` on `o` then `m.mat[a].objects[o].d = 0` has no effect, corresponding to the formal operation which also has no effect.

insert s **in log, inform** a **of** msg The log operation inserts a new log entry. It always succeeds and its effect is that the log l is appended with the string s, denoted $l' = l + s$ with $+$ the string concatenation operator. The corresponding Promela code creates a new entry in the log array and fills it with the values for the current query. As in **create object**, it frames the operations in an `atomic` statement to prevent preemption. It begins by (1) checking the bounds for the log array and updating the `topLog` variable (`atomic{assert(topLog < MAXLOG); l = topLog; topLog++;})`. If then (2) enters in the values for the new log entry (`log[l].command = command; log[l].a = a; log[l].s = s; log[l].r = r; arrayCopy(P, log[l].P, MAXPURPOSE); log[l].f = f; log[l].fnew = few;`).

As in **create object**, the first statements (1) assert that there is an empty slot in the `log` array by examining `topLog < MAXLOG`. If the assertion fails, the execution of the model is terminated. If it succeeds, execution continues unhindered, the variable `l` stores the new slot's index, and the variable `topLog` is incremented to maintain invariants 3 and 4.

The next statements (2) store the current global variable values in `log[l]` along with the value of the current command (in the local variable `command` as per invariant 9) in `log[l].command`. The inline function `arrayCopy` is defined in Appendix A. It copies the purpose array `P` by copying `MAXPURPOSE` elements from `P` to `log[l].P` as explained Appendix A on page 314.

The inform operation is similar to the log operation, but uses the `inform` array and `topInform` and `MAXINFORM` variables.

The correctness proof for log insertion and inform differs from proofs above. Since Promela does not support strings, we can not implement directly the semantics of the formal model. Instead of strings we therefore store the full query information for the command being executed, copying all global variables to a new spot in the array. The resulting semantics are such that `log` and `inform` are updated with new entries if there is space in their respective arrays. Since no members of ψ inspect l or i, the change does not affect the outcome or behavior of the model.

invoke $e(args)$ The invoke operation first performs a constraint search for a command e and runs it if the constraints allow. The operation leaves the states s unchanged except for the changes performed by e while it is run.

The corresponding Promela code first performs the constraint search by (1) sending a message to the search process with the name of the command to be executed and listening for its response (`Cst_request_chan!search_request(e); Cst_response_chan?search_response(temp);`). Using an `if/fi` structure, the code checks (2) if the response is true (*i.e.,* Allow) (`if ::temp == true`). If it is, (3) the command's process `e` is sent a message to begin its execution and the operation listens for its response (`e_request_chan!command_request; e_response_chan?command_response(temp);`). If the response if false (*i.e.,* Forbid), the `else` option (4) is chosen, setting the command's result to false (`result = false;`) and indicating that the operation was stuck.

The first statements (1) invoke the constraint search process `Cst_Search` defined above in Section 6.3.1 and listens for its response. As shown in Lemma 6.4.3, the constraint search returns true if and only if the constraint search in the formal model would return Allow or Ignore (Allow). Otherwise it returns false.

If the result is true, statement (2) is chosen in the `if/fi` structure and the next statements (3) run the process `e` by sending it a message on its request channel and then listening for its response. This corresponds to running a command if the constraint search permits it. Running the command using its Promela process corresponds to executing the formal command e as shown in Lemma 6.4.6.

If the result is false, the `else` option is chosen and statement (4) is chosen and the operation is stuck. This corresponds to not running the formal command if the constraint search forbids it or any operation in it get stuck.

return b The return operation returns the boolean value b from the executing command or constraint. As noted in Section 6.2.3 the logic for returning the correct boolean result is distributed throughout the command and constraint processes. Therefore there is no explicit code generated for a **return** operation. The correctness of the return value is shown in Lemmas 6.4.6 and 6.4.7. □

Having shown correspondence lemmas for the members of *Guard* and *Operation* and their Promela translations in Tables 6.7 and 6.8, we now use the proofs for two important lemmas which show the correctness of the translation of commands and constraints into Promela.

Lemma 6.4.6 *(Commands) A command e in the Privacy Commands formal language which is run by the evaluation engine with arguments a, s, r, P, f, f', msg on a knowledge state (A, O, m, l) leads to a resulting state (A, O', m', l') if and only if the corresponding Promela process $\mathsf{Pr}(e)$ leads from $\mathsf{Pr}(A, O, m, l)$ to $\mathsf{Pr}(A, O', m', l')$ for the same parameters as global variables as per Table 6.4 when predicated by the transaction code in Figure 6.10.*

Proof: The structure for a command in Promela is shown in Figure 6.6 and is explained in Section 6.2.6. As defined in Section 5.4.1, when running a command, the evaluation engine first checks the boolean values of its guards $\overline{\psi}$ and based on their outcome either runs the operations in the true branch $\overline{\omega_t}$ or the operations in the false branch $\overline{\omega_f}$. If the true branch is selected and any $\omega \in \overline{\omega_t}$ is stuck, we introduce in the translation a guarantee of atomicity that the knowledge state will remain unchanged with the exception of the log (*i.e.,* (A, O, m, l') is the result). Since the operational semantics do not define a behavior for stuck operations and commands, we choose this as the failure mode.

Let us denote $\mathsf{Pr}(e)$ as the process `e`. It listens on the channel `e_request_chan` for a request. Receiving a message on the channel is the equivalent of a command invocation in the formal model. The process then proceeds through a translation of each guard $\psi \in \overline{\psi}$, collecting the result of each translated guard in the variable `result`. As shown in Lemma 6.4.4, each guard stores its result in `temp` which is then logically AND-ed with `result`. After line 7, `result` is true if and only if $\bigwedge_{i=0}^{n} \psi_i = true$.

If the variable `result` is true after the guard evaluation, the first option of the `if/fi` structure is chosen. It executes the Promela translations of the operations in $\overline{\omega_t}$ which yield the Promela translation $\mathsf{Pr}(A, O', m', l')$ as per Lemma 6.4.5. If any of them are stuck, the `result` variable is set to false, but the rest of the operations are executed.

If the variable `result` is false after the guard evaluation, the second option of the `if/fi` structure is chosen. It executes the Promela translations of the operations in $\overline{\omega_f}$ which yield the Promela translation $\mathsf{Pr}(A, O', m', l')$ as per Lemma 6.4.5.

The final value of `result` is sent back on the response channel. The transaction handling code in Figure 6.10 performs the final commit for the operations and if `result` is true ($\mathsf{Pr}(A, O', m', l')$), otherwise it only commits the log ($\mathsf{Pr}(A, O, m, l')$). $\square$

Lemma 6.4.7 *(Constraints) A constraint c in the Privacy Commands formal language which is run by the evaluation engine with arguments a, s, r, P, f, f', msg on a knowledge state (A, O, m, l) yields three boolean results b_{scp}, b_{st}, b_r if and only if the corresponding Promela process $\mathsf{Pr}(c)$ on state $\mathsf{Pr}(A, O, m, l)$ and arguments a, s, r, P, f, f', msgin global variables as per Table 6.4 sends boolean values `scope`, `such_that`, and `regular` on its response channel where b_{scp} =`scope`, b_{st} =`such_that`, and b_r =`regular`.*

Proof: The structure for a constraint in Promela is shown in Figure 6.4 and is explained in Section 6.2.4. As defined, constraints return three boolean values over their response channels. Those boolean values correspond to the checks for scope, *such that* guards, and regular guards.

After receiving a message on its request channel, a constraint process `c` evaluates whether the `command` variable passed with the message is equal to any of `CMD` values in its scope. The code for the search a straightforward `if/fi` selection as shown in Section 6.2.5. If `command` matches any of the options, `scope` is set to true. Otherwise it set to false. This corresponds to the scope checking for constraints and `scope` is true iff the command represented by `command` is in *scp*, the scope of *c*.

After checking scope, `c` checks the *such that* guards which store true in `result` iff $\bigwedge_{i=0}^{n} \psi_{st}^{i} = true$ for all $\psi_{st}^{i} \in \overline{\psi_{st}}$ as per Lemma 6.4.4. The value in `result` is stored in `such_that` and `result` is reset. A similar procedure is then performed for the regular guards and their result is stored in `regular` which is true iff $\bigwedge_{i=0}^{n} \psi_{r}^{i}$ is true as per Lemma 6.4.4. $\square$

Using the above lemmas, we prove the following theorem regarding the correspondence between the formal representation and the Promela model.

Theorem 2 *For a Privacy API (C, E, R, T, P) consisting of a command set $E \in$ Command and a constraint set $C \in$ Constraint and the Promela translation of both sets* *denoted* $\mathsf{Pr}(E)$ *and* $\mathsf{Pr}(C)$ *respectively, for any knowledge state* (A, O, m, l) *which transitions to a state* (A', O', m', l') *via a command* $e \in E$*, the corresponding Promela state* $\mathsf{Pr}(A, O, m, l)$ *transitions to* $\mathsf{Pr}(A', O', m', l')$ *via* $\mathsf{Pr}(e)$*.*

Proof: As shown in Lemmas 6.4.6, the translation of any command e yields a Promela process denoted $\mathsf{Pr}(e)$ which has equivalent behavior. Constraint search for e in the formal model allows commands to execute if and only if the corresponding Promela constraint search allows $\mathsf{Pr}(e)$ to execute as per Lemma 6.4.3. Therefore, command $\mathsf{Pr}(e)$ may execute if and only if e may execute. If it does execute, its results are equivalent to e. □

Theorem 2 shows the correspondence between the knowledge state transformations performed by commands and the Promela state evolution performed by the Promela processes. An important corollary to this result is the following.

Corollary 1 *(Properties) For a Privacy API* (C, E, R, T, P) *and an initial knowledge state* (A, O, m, l)*, the knowledge state* (A, O', m', l') *is reachable via a series of commands* $e_1(args_1), e_2(args_2), \ldots, e_n(args_n)$ *where* $args_1, args_2, \ldots, args_n$ *are the respective parameters for the commands if and only if the Promela translations* $\mathsf{Pr}(E)$ *and* $\mathsf{Pr}(C)$ *from the initial state* $\mathsf{Pr}(A, O, m, l)$ *can reach* $\mathsf{Pr}(A', O', m', l')$ *by sending request messages to processes* $\mathsf{Pr}(e_1), \mathsf{Pr}(e_2), \ldots, \mathsf{Pr}(e_n)$ *with global variables* $\mathsf{Pr}(args_1), \mathsf{Pr}(args_2), \ldots, \mathsf{Pr}(args_n)$ *at each respective request.*

Proof: The argument is a straightforward application of Theorem 2 from single step transitions to a chain of n transitions. □

6.5 Conclusion

In this chapter we have presented a methodology for the translation of Privacy APIs to Promela models which can be evaluated using the SPIN model checker. Our mapping includes direct translations from the guard and operation sets in the Privacy Commands language to checks and changes on a Promela model built on a set of global variables. We build Promela processes based on the commands and constraints in the formal model and show some of the extra code necessary for the processing of command invocations.

An important result from this chapter is that we may use Corollary 1 to use a Promela model for the exploration of reachable properties in of a Privacy API. It tells us that any knowledge state which is reachable in the formal model is also reachable in the Promela model, so we are guaranteed that the model is complete. Importantly, it also tells us that any state which is reached in the Promela model is also reachable in the formal model via the commands which correspond to the Promela processes. Therefore, any properties that we discover via SPIN space exploration in the Promela model correspond to knowledge states reachable in the formal model and we may use the trace of the SPIN exploration to see which commands led to it. We use this observation in Chapter 7 where we use SPIN to explore the properties of the Promela models and map them back to the source text.

```
active proctype Cst_Search()
{
   JUDGMENT j; JUDGMENT final; CMD e;
   bool scope; bool such_that; bool regular;
   do
   :: Cst_Search_request_chan?constraint_request(e);
      j = final = ALLOW;
      CST1_request_chan!constraint_request(e);
      CST1_response_chan?constraint_response(scope, such_that,
      regular);
      cst_search_judgment(scope, such_that, regular, j);
      if
      :: j == FORBID -> final = FORBID;
      fi;  ...
      CSTn_request_chan!constraint_request(e);
      CSTn_response_chan?constraint_response(scope, such_that,
      regular);
      cst_search_judgment(scope, such_that, regular, j);
      if
      :: j == FORBID -> final = FORBID;
      fi;

      if
      :: final == FORBID ->
         Cst_Search_response_chan!command_response(false);
      :: else ->
         Cst_Search_response_chan!command_response(true);
      fi;
   od
}
```

Figure 6.12: Promela code an n constraint search

Table 6.7: Promela translation for members of *Guard*

d **in** (a, o)	`temp = (o < MAXOBJECT && o >= 0 &&`
	`objects[o] == 1 && a < MAXAGENT && a >= 0`
	`&& m.mat[a].objects[o].d == 1);`
	`result = result && temp;`

$o.t = b$	`temp = (o < MAXOBJECT && o >= 0 &&`
	`objects[o] == 1 && mtags[o].t == b);`
	`result = result && temp;`

k **in** $Roles(a)$	`temp = (a < MAXAGENT && a >= 0 && mroles[a].k == 1);`
	`result = result && temp;`

p $\textbf{in}_a$ P	`run isPermittedByA();`
	`isPermittedByA_request_chan!purpose_request(p);`
	`isPermittedByA_response_chan?purpose_response(temp);`
	`result = result && temp;`

p $\textbf{in}_f$ P	`run isForbiddenByA();`
	`isForbiddenByA_request_chan!purpose_request(p);`
	`isForbiddenByA_response_chan?purpose_response(temp);`
	`result = result && temp;`

$c(a, s, r, P, f, f', msg)$
$\in \{j_1, \ldots\}$
	`c_request_chan!constraint_request;`
	`c_response_chan?constraint_response(scope,`
	`such_that, regular);`
	`reference_judgment(scope, such_that, regular, j);`
	`if`
	`::  j == j_1 -> temp = true;`
	`::  ...`
	`::  else -> temp = false;`
	`fi;`
	`result = result && temp;`

$a_1 = a_2$	`temp = (a1 >= 0 && a1 < MAXAGENT && a2 >= 0`
	`&& a2 < MAXAGENT && a1 == a2);`
	`result = result && temp;`

Table 6.8: Promela translation for members of *Operation*

create object o	`atomic{assert(topObj < MAXOBJECT); o = topObj;`
	`objects[topObj] = 1; topObj++;}`

delete object o	`if`
	`::  o < MAXOBJECT && o >= MAXAGENT -> objects[o] = 0;`
	`::  else -> skip;`
	`fi;`

set $o.t = b$	`if`
	`::  o >= MAXOBJECT || o < 0 || objects[o] == 0 ->`
	`result = false;`
	`::  else -> mtags[o].t = b;`
	`fi;`

insert d **in** (a, o)	`if`
	`::  (o >= MAXOBJECT || o < 0 || objects[o] == 0 ||`
	`a < 0 || a >= MAXAGENT)-> result = false;`
	`::  else -> m.mat[a].objects[o].d = 1;`
	`fi;`

delete d **from** (a, o)	`if`
	`::  (o >= MAXOBJECT || o < 0 || objects[o] == 0 ||`
	`a < 0 || a >= MAXAGENT)-> result = false;`
	`::  else -> m.mat[a].objects[o].d = 1;`
	`fi;`

insert s **in log**	`atomic{assert(topLog < MAXLOG); l = topLog; topLog++;}`
	`log[l].command = command; log[l].a = a; log[l].s = s;`
	`log[l].r = r; arrayCopy(P, log[l].P, MAXPURPOSE);`
	`log[l].f = f; log[l].fnew = few;`

inform a **of** s	`atomic{assert(topInform < MAXINFORM); i = topInform;`
	`topInform++;} inform[i].command = command;`
	`inform[i].a = a; inform[i].s = s; inform[i].r = r;`
	`arrayCopy(P, inform[i].P, MAXPURPOSE);`
	`inform[i].f = f; inform[i].fnew = few;`

invoke $e(args)$	`Cst_request_chan!search_request(e);`
	`Cst_response_chan?search_response(temp);`
	`if`
	`::  temp == true -> e_request_chan!command_request;`
	`e_response_chan?command_response(temp);`
	`result = result && temp;`
	`::  else -> result = false;`
	`fi;`

return b	No additional code.

Chapter 7

Case Studies

The techniques that we have delineated in previous chapters give us the machinery to do two types of analysis. First, we can use the mapping from Privacy Commands to Promela to let us explore the permissions granted by a legal text. We do this by translating the legal document to a Privacy API, designing an initial knowledge state and invariants, mapping them to a Promela model, and using SPIN to explore the model's reachable states to see if the invariants are true. Second, we can use the formal model itself to evaluate conformance metrics such as strong and weak licensing between Privacy APIs. The metrics let us compare the permissiveness of legal policies and determine whether one document is at least as strict as the another document.

Our goal in this chapter is to exercise our techniques to show their effectiveness at addressing a variety of real world legal documents. We present here three case studies in the evaluation and comparison of legal privacy policies. We first demonstrate the ability to compare different versions of legal policies using HIPAA as an example in Section 7.1. To exercise policy comparison we also show how to compare parts of HIPAA to the Insurance Council of Australia's Privacy Code [72] in Section 7.2. As an exercise in conformance determination, we develop a case study comparison between the US Cable TV Privacy Act of 1984 [48] (CTPA) and the TiVo corporation's (`tivo.com`) privacy policy [62] that is subject to it in Section 7.3. We conclude in Section 7.4.

7.1 HIPAA Consent

As noted in Section 2.1.3, the HIPAA Privacy Rules was issued by the HHS's Office for Civil Rights to regulate the usage and disclosure of health information. As a document, the Privacy Rule presents an interesting case study since it has gone through several major revisions during its lifetime. The initial version of HIPAA's Privacy Rule was published in the Federal Register in 2000 [42]. It was published by the Department of Health and Human Services (HHS) with the intent of gathering public comments and criticisms from the entities affected by the Rule and so there was a comment period before it was to go into effect. After the comment period, the HHS published list of major comments along with its responses to them [43]. In 2003 it released a new version of the Privacy Rule with significant changes from the initial version [46]. Since 2003, the Privacy Rule has had some minor revisions as well which are reflected in the currently available version on the HHS's web site.

Since we have access to both versions of the HIPAA Privacy Rule along with a complete listing of relevant comments, it is of interest to ask whether the new version of the Rule changed to meet the public comments. Conveniently, we can use Privacy APIs and their corresponding Promela models to help answer the question in an semi-automated fashion. We do so by translating the 2000 and 2003 versions of HIPAA into Privacy APIs, converting them to Promela model, interpreting the comments as properties of the models, and using SPIN to determine whether the comment properties are true in them.

For this case study we limit our translation to one section of the Privacy Rule which deals with consent for treatment, payment, and health care operations. We show the document level differences between the two versions, translate the sections to Privacy APIs and Promela models, and show how we use SPIN to explore the differences between the models with respect to the comments issued. We have discussed the consent rules in §164.506 above in Example 3.2.3.

7.1.1 Model Development

As noted above, the consent rules were considerably rewritten between 2000 and 2003. The 2000 version was about one page long in the Federal Register. The 2003 version was shortened to one third of a column. The content and style of the document changed as well, with the 2003 version using many more references to other parts of the document than the 2000 version. The result of the modifications is that direct comparison between the documents at the text level is somewhat complicated. We can, however, use Privacy APIs to make the job easier.

Our goal in this case study is to translate the relevant sections of HIPAA from text to Privacy APIs, attempt to discover the three properties relevant to the comments mentioned above in Example 3.2.3 in the 2000 version, and explore if they are still true in the 2003 version. Finding the properties in the 2000 version shows the correctness of the model in extracting known properties of the text. Exploring the properties in the 2003 version shows the usefulness of the model since it shows how to find differences between policies.

In order to discover the desired properties, we follow the steps enumerated in Figure 1.1. First, we translate the text into the Privacy Commands language, enumerating the roles, tags, rights, and purposes relevant to the consent section. Second, we devise commands and constraints which parallel the paragraphs and sentences of the consent rules, capturing the contents and intent of the phrases as shown in the previous chapters. Next, we devise initial knowledge states for the comments that we are looking for based on the comments documents and translate everything into three Promela models. We then derive the LTL properties implied by the comments and encode them in terms of the Promela model. Finally, we use SPIN to explore the properties of the models, exploring whether the LTL properties are maintained.

In designing and analyzing the models that we are developing, it is important to keep in mind the types of properties that we are examining. The Privacy Rule contains rules that relate to the processing of private information, a policy that entities covered by the law must follow. Since the Privacy API is a derivative of the Privacy Rule document, the model derived will be at the same level of abstraction. The models we derive therefore are at the policy level, not the implementation level. Thus, the model consider aspects such as the granting, revocation, and creation of rights over objects. It does not consider aspects such as the management of identities, the consumption of rights, or the creation of state that the policy uses to make decisions. For example, for a policy statement which permits health care providers to use information about an individual for the purpose of treatment, the model will check the roles, properties, and purposes for the access request and grant the right "treatment" on the object if satisfied. The model will not check, however, how actors are designated as "health care providers", how the object is marked protected health information, and how a computer database will interpret the right for treatment in terms of the data that it sends back from a database query. Such implementation level models are describable using Privacy Commands, but since the Privacy Rule does not discuss them, they are not included in the case study.

7.1.2 Consent from 2000

The rules regarding the need for consent for treatment, payment, and health care operations from the 2000 version of the Privacy Rule have six major divisions (paragraphs) as described in Section 2.1.3. The paragraphs have different purposes and therefore different levels of specificity. In creating the Privacy API based on the model, we must carefully analyze the text to establish the different roles, purposes, actions, and agents which are to be included. For reference we include the full text for [§164.506, v.2000] in Appendix B.1.

The regulatory text in paragraph (e) deals with the resolution of conflicting consent forms and the contacting of individuals to verify their intent. The resolution and document level discussions that are necessary to properly model its rules are beyond the scope of the Privacy APIs language since the policy steps required deal too deeply with the semantics of a consent form and not with the permission that it yields. If we are given two consent forms with differing permissions we could translate them to Privacy APIs and then use the strong and weak licensing relations to analyze their relative permissiveness, but since this case study is concerned with analysis at the regulatory level, such implementation level analysis is not possible. The ability to express guards at the meta-policy level (*e.g.,* If A is more restrictive than B then you may do C) is an exciting extension that merits further study. In particular, the need to merge document and instance level requirements into a single evaluation engine makes the task of modeling far more complex.

Roles

The text in §164.506 mentions the following roles. They are the members of *Role*. The definitions below are as per [§164.501, v.2000] and the list is in alphabetical order.

Covered Entity An individual or organization that is either a health care provider, a health insurance company, or a health information clearinghouse. Entities designated as covered entities are subject to the Privacy Rule while others are not.

Health Care Provider An individual or organization which provides health care services.

Health Plan An organization which provides health insurance coverage.

Individual A person with protected health information.

Purposes

The text mentions the following purposes for actions. They are the members of *Purpose*. The definitions below are as per [§164.501, v.2000] and the list is in alphabetical order.

Disclose The transfer (disclosure) of information held by one entity to another.

Emergency Health service provided in an emergency.

Enrollment Enrolling an individual in a health care plan.

Grant Consent An individual wishes to grant consent to an entity for protected health information.

Health Care Operations Any one of a list of actions performed by an entity that relate to health care.

Indirect Treatment A relationship of treatment where the doctor and patient are not in direct contact.

Joint Consent An individual wishes to grant consent to more than one entity at a time for protected health information.

Payment Any one of a list of actions performed by an entity to perform that relates to billing and payment processing.

Research Actions related to systematic research to lead to generalizable knowledge.

Revoke An individual wishes to retract consent from an entity.

Transfer Consent An entity wishes to act using a consent granted to another entity.

Treatment Any one of a list of actions performed by an entity that relate to the provision, management, and coordination of health care services.

Use Accessing or processing some protected health information in order to use it for a particular purpose.

Voluntary Consent Obtaining consent from an individual when not strictly required to by the law or regulation.

Rights

The text mentions the following relationships between agents and objects. They represented as rights and therefore are the members of *Right*. The rights represent one way relationships between an agent and object and are not reflexive. For instance, if d in (a, b), then we say that a has right d on b, but not the reverse. As needed the descriptions will refer to a and b for clarity.

Attempted Consent Agent a has attempted to receive consent from agent b.

Begin Treat Agent a begins a treatment relationship with agent b.

Consent Agent a has received consent from agent b.

Health Care Operations Agent a has received permission to use objects about agent b for health care operations.

Indirect Agent a has an indirect treatment relationship with agent b.

Local Agent a has object b locally and may access it if allowed by the Privacy API.

Organized Health Care Arrangement Agent a participates in an organized health care arrangement with agent b.

Payment Agent a has received permission to use objects about agent b for payment purposes.

Required To Treat Agent a is required by law to treat agent b.

Research Agent a has received consent from b to use files about b for research.

Treatment Agent a has received permission to use objects about agent b for treatment purposes.

Tags

The text mentions a large number of properties and checks that we include in tags. Since there are many tags they are listed in Table C.1. Individual tags are elaborated on as necessary during the rest of the discussion.

Command and Constraint Examples

We translate the legal text into commands and constraints as described in Chapter 5. The commands and constraints include references as necessary. Constraint scopes are applied as per the explicit and implicit references included in the text. To give a good feel for the depth and breadth of the model, we next present a few sample commands and constraints from the full Privacy API which is included in Appendix C.1.

Example 7.1.1 ([§164.506(a)(1), v.2000]) As an example of constraint overloading and basic commands, we consider the following text from [§164.506(a)(1), v.2000]:

> Except as provided in paragraph (a)(2) or (a)(3) of this section, a covered health care provider must obtain the individual's consent, in accordance with this section, prior to using or disclosing protected health information to carry out treatment, payment, or health care operations.

The text provides three options for a covered health care provider to use or disclose protected health information: if (a)(2) permits it, if (a)(3) permits it, or if the health care provider has the individual's consent. Since options are joined with a logical OR, if any of them are true, the paragraph permits the use or disclosure. The three options are therefore mapped to three separate overloaded constraints. This is an example of constraint overloading since we may take the most lenient result from all three constraints as described in Section 5.2.4. The three constraints are as follows:

CST	Permitted506a1(a, s, r, P, f, f', msg)
Scope	{TreatmentUse506a1, PaymentUse506a1,
	HealthCareOperationsUse506a1, TreatmentDisclose506a1,
	PaymentDisclose506a1, HealthCareOperationsDisclose506a1}
Such That	individual **in** Roles(s)
and	f.protected-health-information = true
if	Permitted506a2(a, s, r, P, f, f', msg) $\in$ {Allow}
then	**return true**
else	**return false**

CST	Permitted506a1(a, s, r, P, f, f', msg)
Scope	{TreatmentUse506a1, PaymentUse506a1,
	HealthCareOperationsUse506a1, TreatmentDisclose506a1,
	PaymentDisclose506a1, HealthCareOperationsDisclose506a1}
Such That	individual **in** Roles(s)
and	f.protected-health-information = true
if	Permitted506a3(a, s, r, P, f, f', msg) $\in$ {Allow}
then	**return true**
else	**return false**

```
       CST   Permitted506a1(a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506a1, PaymentUse506a1,
              HealthCareOperationsUse506a1, TreatmentDisclose506a1,
              PaymentDisclose506a1, HealthCareOperationsDisclose506a1}
 Such That   individual in Roles(s)
       and   f.protected-health-information = true
        if   consent in (a, s)
      then   return true
      else   return false
```

The first constraint's scope includes the commands that implement actions under paragraph (a)(1). It checks if the constraint for (a)(2) allows the action to be performed, that subject of the information is an individual, and that the object is protected health information. The *such that* guards check if the subject of the information is an individual and the object is protected health information, since if they are not, the paragraph is not applicable. The Permitted506a2 constraint reference in the regular guard checks that the judgment received is Allow.

The second and third constraints are similar with the second one checking the constraint for (a)(3) and the third checking if the individual has given consent.

Each of the above constraints implement the logic of the paragraph by checking one of the options. As noted in Section 5.2.4 and explained in Section 5.4.2, when a command references the constraint Permitted506a1 or a command in its scope is about to be executed, the evaluation engine derives judgments from the constraints are combined using a most lenient algorithm (see Tables 5.11 and 5.9). The result is that if any of the three constraints yields Allow, the resulting judgment is Allow, permitting the command to be run.

The commands that we derive from the above quote are divided by purpose - use and disclosure for treatment, payment, and health care operations. The first one we present relates to the use of information for treatment.

```
CMD   TreatmentUse506a1 (a, s, r, P, f, f', msg)
 if   Permitted506a1(a, s, r, P, f, f', msg) $\in$ {Allow}
 and  individual in Roles(s)
 and  healthCareProvider in Roles(a)
 and  local in (a, f)
 and  treatment $\mathbf{in}_f$ P
 and  use $\mathbf{in}_a$ P
then  insert treatment in (a, s)
 and  return true
else  return false
```

The command TreatmentUse506a1 is invoked when an agent wishes to acquire permission to use an object for treatment. It first checks if the constraint for (a)(1) allows the intended action. If it does, it checks that the subject is an individual and that the actor is a health care provider. It then checks that the object is locally visible to the actor and that the purpose of the action is to use the object for treatment. The purpose is checked using forbidden semantics since if treatment or any of its ancestors are present, the requirements apply. If all of the guards are satisfied, the command inserts the treatment right for the actor on the object and returns true. Otherwise it returns false. The commands for use for payment and health care operations are similar and shown in Section C.1.2 on page 350.

Note that the permission to use an object for treatment is indicated by the right "treatment" in the matrix at entry (actor, subject). Therefore, once an agent has the treatment right on the subject, it can be used to permit accesses in the future to any object about the subject until the right is removed. This parallels the real world notion in the HIPAA rules that a consent gives permission for an agent to use information until the consent is revoked and that consent is a contract between people, not related to a specific data object.

While the constraint Permitted506a1 already checked that the subject is an individual, the command repeats the check in its guard list. This done as a safety measure for commands which depend on the status of a particular guard to operate. Since the constraint invoked may be overloaded and therefore be satisfied by multiple combinations of guards, it is a good practice for commands to include all guards which must be true for their correct operation.

In addition to the use action, the paragraph permits disclosure of protected health information for treatment, payment, and health care operations. We model the disclosure of information by the granting of the local right to the recipient. The recipient actor can then operate of the object since it is locally visible.

```
 CMD  TreatmentDisclose506a1 (a, s, r, P, f, f', msg)
  if  Permitted506a1(a, s, r, P, f, f', msg) $\in$ {Allow}
 and  individual in Roles(s)
 and  healthCareProvider in Roles(a)
 and  local in (a, f)
 and  treatment $\mathbf{in}_f$ P
 and  disclose $\mathbf{in}_a$ P
then  insert local in (r, f)
 and  return true
else  return false
```

The command TreatmentDisclose506a1 implements the action of disclosing protected health information for the purpose of treatment. As in TreatmentUse506a1, it checks that the relevant constraint allows the action, that the subject is an individual, that the actor is a health care provider, that the object is locally visible to the actor, and that purpose of the action is for disclosing information for treatment purposes. If all of the guards are satisfied, the command grants local visibility on the file to the recipient. The commands for disclosure for payment and health care operations are similar and shown in Section C.1.2 on page 350. □

The above example demonstrates the use of commands and constraints reflecting a single paragraph. The commands and constraints derived exercised a common set of Privacy Command features: rights, roles, and objects tags. We now show an example which exercises the *don't care* judgments to allow for restrictions.

Example 7.1.2 ([§164.506(b), v.2000])

Paragraph (b) of the consent rules discusses general requirements for the acquisition and maintenance of consent documents. Each subparagraph relates to a different aspect of consent. In this example we consider one subparagraph of (b):

> *(b) Implementation specifications: General requirements.*
> (1) A covered health care provider may condition treatment on the provision by the individual of a consent under this section.

Subparagraph (b)(1) permits health care providers to refuse treatment to individuals who do not grant consent. Since health care providers are not required to refuse care, the constraint in (b)(1) may issue a judgment which is ignored. The constraint for (b)(1) is as follows:

CST	Condition506b1(a, s, r, P, f, f', msg)
Scope	{BeginTreatment506b1}
Such That	a.refuse-without-consent = true
and	individual **in** Roles(s)
if	consent **in** (a, s)
then	**return true**
else	**return false**

The scope is the command BeginTreatment506b1 which we discuss next. Its *such that* guards check that the actor intends to refuse treatment without consent and that the subject is an individual. If the actor does not intend to refuse treatment without consent or the subject is not an individual then the constraint is not applicable. The regular guard checks that the subject has given consent. If the subject has given consent then the constraint permits the treatment. Otherwise it does not.

Since the text gives permission to health care providers to condition treatment based on the receipt of consent but does not require it, we place the a.refuse-without-consent check in a *such that* guard. By doing that we enable commands to use the Don't Care/Allow and Don't Care/Forbid judgments appropriately. For instance, if the health care provider does not wish to condition treatment on the receipt of consent (*i.e.,* a.refuse-without-consent = false) and the individual has not granted consent (*i.e.,* consent $\notin$ (a, s)), the constraint's returned judgment will be Don't Care/Forbid. However, if the health care provider does intend to condition treatment on the receipt of consent (*i.e.,* a.refuse-without-consent = true), the returned judgment will be Forbid. Commands can thereby use the different judgments to take appropriate action as we show next.

The command for the paragraph applies the notion of beginning treatment using the beginTreat right. It uses the constraint Condition506b1 above described above.

```
 CMD   BeginTreatment506b1 (a, s, r, P, f, f', msg)
   if  Condition506b1(a, s, r, P, f, f', msg) ∈ {Allow, Don't Care/Allow,
       Don't Care/Forbid}
  and  healthCareProvider in Roles(a)
  and  treatment in_a P
 then  insert beginTreat in (a, s)
  and  return true
 else  return false
```

Command BeginTreatment506b1 first checks constraint Condition506b1's judgment. The constraint check is true if the judgment returns is either Allow, Don't Care/Allow, or Don't Care/Forbid. As we noted above, we use the Don't Care/Allow and Don't Care/Forbid judgments to separate cases where the actor doesn't intend to condition treatment on the receipt of consent. The next guards check that the actor is a health care provider and that the purpose of the action is treatment. If all of the guards are satisfied, the command inserts the right beginTreat in the matrix to indicate that the health care provider will begin treatment for the subject. If the guards are not satisfied, the command returns false. □

Having shown two examples involving rights, roles, tags, and judgments, we now present a more involved example showing the complexity of interactions between commands and constraints as illustrated in the rules regarding the granting of consent for treatment, payment, and health care operations. The complexity in the example is not due to the Privacy APIs formulation, but rather due to the inherent complexity of the Privacy Rule document. Privacy Commands make the references and complexity explicit, enabling users to envision the relationships between different parts of the regulation.

Example 7.1.3 (Granting Consent)

The granting of consent by an individual to an health care provider is the subject of many rules in §164.506. The regulations require that consent must be granted in the majority of cases as quoted above [§164.506(a)(1), v.2000]:

> (1) Except as provided . . . a covered health care provider must obtain the individual's consent, in accordance with this section, prior to using or disclosing protected health information. . .

In order to properly model the granting of consent as per (a)(1), we include a command which grants consent to an agent from a subject:

```
 CMD   GrantConsent506a(a, s, r, P, f, f', msg)
   if   individual in Roles(s)
  and   healthCareProvider in Roles(a)
 then   insert consent in (a, s)
  and   RecordConsent506b6 (a, s, r, P, f, f', msg)
  and   return true
 else   return false
```

The command checks that the subject is an individual and that the recipient actor is a health care provider. If both are satisfied, the command inserts the right "consent" into the actor's rights over the subject. The command does not mention any constraints explicitly, but it is subject to the constraints of paragraph (c) regarding the content of the consent document that created the right as we describe below.

Note that there are other circumstances described in the section for the granting of consent. For brevity in this example we include only GrantConsent506a. The other cases are included in the full Privacy API in Appendix C.1. There are many rules in the section that limit the manner in which consent is granted. Let us consider two of them: §164.506(b)(6) which requires the retention of the records of granted consent and §164.506(c) which places requirements on the content of the consent document.

First let us consider subparagraph (b)(6) [§164.506(b)(6), v.2000]:

> *(b) Implementation specifications: General requirements.*
>
> (6) A covered entity must document and retain any signed consent under this section as required by Sec. 164.530(j).

It requires covered entities to retain any signed consent forms as required by §164.530(j). The text in §164.530(j) requires entities to maintain electronic or written records of communications and actions that occur in the context of health care activities. The relevance to (b)(6) is that the communication of consent or revocation must be recorded electronically or in writing [§164.530(j), v.2000]:

> *(j)(1) Standard: Documentation.* A covered entity must:
> (ii) If a communication is required by this subpart to be in writing, maintain such writing, or an electronic copy, as documentation; and

The full text of §164.530(j) is included in Section B.1.2 on page 339 for convenience.

The constraint for (b)(6) is as follows:

CST	ReqRecordConsent506b6 (a, s, r, P, f, f', msg)
Scope	{GrantConsent506a, OptionalConsent506a4, CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
Such That	coveredEntity **in** Roles(a)
if	grant-consent **in**$_a$ P
and	s.consent-in-writing = true
and	a.will-record = true
and	Maintain530j(a, s, r, P, f, f', msg) $\in$ {Allow, Don't Care/Allow, Don't Care/Forbid}
then	**return true**
else	**return false**

The scope for the constraint is all of the commands that involve the granting of consent from a subject. The such guard checks that the actor is a covered entity since otherwise the subparagraph does not apply. The regular guards check that the purpose of the action is granting consent, that the consent is in writing, and that the actor will record the consent once granted. The last guard checks that the constraint Maintain530j returns an Allow, Don't Care/Allow, or Don't Care/Forbid judgment. The *don't care* judgments permit cases where retention is not required as per §164.530 as we show below.

The constraint uses the tag will-record to check whether the actor will record the consent that is granted. The tag implies an obligation on behalf of the actor to record the consent in the future. Privacy Commands do not have any way to impose and fulfill obligations since they are not designed as models of a particular implementation. Specific implementations will impose and record the fulfillment of obligations to be in compliance with the regulation, but unless the particulars of the obligation system are included in the regulatory text they are not included in the Privacy API.

The constraint for §164.530(j) is as follows:

```
  CST  Maintain530j(a, s, r, P, f, f', msg)
Scope  {RevokeConsent506b5, RevokeJointConsent506f2ii, GrantConsent506a,
       OptionalConsent506a4, CombineConsent506b4i, CombinedConsent506b4ii,
       GrantJointConsent506f1}
   if  Policies530ji(a, s, r, P, f, f', msg) ∈ {Allow, Don't Care/Allow,
       Don't Care/Forbid}
  and  MaintainWritten530jii(a, s, r, P, f, f', msg) ∈ {Allow, Don't Care/Allow,
       Don't Care/Forbid}
 then  return true
 else  return false
```

The constraint's scope is all commands that are involved in the creation of communications. For the model we present, we include all such commands, but a wildcard symbol could be used as well if it were added to the syntax of Privacy Commands scopes. The constraint checks that both of its subparagraphs are fulfilled. The first subparagraph (j)(i) checks that policies mentioned in §164.530(i) are maintained. The second checks that communications that are required to be in writing are maintained in written or electronic format. For brevity, we elide the details of the constraints from this example, but they can be found in full in Section C.1.2 beginning on page 379.

Commands fulfill their obligation to record the granting of consent as per §164.506(b)(6) using the following command:

```
CMD   RecordConsent506b6 (a, s, r, P, f, f', msg)
 if   consent $\mathbf{in}_a$ P
then  insert "Consent granted" in log
 and  invoke Record530j(a, s, r, P, f, f', msg)
 and  return true
else  return false
```

The command RecordConsent506b checks that the intended action is the granting of consent. If it is, it records the granting using the log and then invokes the command Record530j to comply with the rules in section §164.530(j) as per the requirement in the constraint ReqRecordConsent506b6. Invoking the command is a fulfillment of the tag will-record that is checked by ReqRecordConsent506b6 and Maintain530j.

The command Record530j is as follows:

```
CMD   Record530j(a, s, r, P, f, f', msg)
 if   true
then  insert msg in log
 and  return true
else  return false
```

Its guard is nil (only the trivial value true) since there are no preconditions for its recording in the log. It records the message (in msg) in the log, corresponding to the recording of the communication that has taken place.

As noted above, RecordConsent506b6 is invoked by GrantConsent506a in the course of noting the granting of consent. All other commands related to the granting of consent invoke RecordConsent506b6 as well. Other command which depend on communications of agents invoke Record530j directly as part of their execution. See for example RevokeConsent506b5 on page 371.

Now let us consider the rules for the content of consent documents are listed in §164.506(c). The text in the paragraph begins as follows [§164.506(c), v.2000]:

> *(c) Implementation specifications: Content requirements.* A consent under this section must be in plain language and:
>
> (1) Inform the individual that protected health information may be used and disclosed to carry out treatment, payment, or health care operations;

The paragraph contains rules regarding the contents of the consent document which are submitted to subjects for approval. The contents of the consent document are detailed in (c) with respect to the kinds of information which must be included. Paragraph (c)(1) is an example of the requirements from the paragraph. It requires that the document inform the individual that the information covered in the consent may be used for treatment, payment, and health care operations. The 6 subparagraphs under (c) are combined with a logical AND and therefore must all be satisfied. The full text of the paragraph and its subparagraphs is in Section B.1.1 on page 337.

The requirements in paragraph (c) are constraints which limit the granting of consent by individuals to health care providers. Therefore, the constraints refer to all consents received in §164.506 and all commands which grant consent in the section are in their scope. The top level constraint for (c) reflects this by including all of its subparagraph constraints in its guards:

```
  CST  ConsentContent506c (a, s, r, P, f, f', msg)
Scope  {GrantConsent506a, OptionalConsent506a4,
        CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if  a.consent-plain-language = true
  and  ConsentContent506c1(a, s, r, P, f, f', msg) ∈ {Allow}
  and  ConsentContent506c2(a, s, r, P, f, f', msg) ∈ {Allow}
  and  ConsentContent506c3(a, s, r, P, f, f', msg) ∈ {Allow}
  and  ConsentContent506c4(a, s, r, P, f, f', msg) ∈ {Allow}
  and  ConsentContent506c5(a, s, r, P, f, f', msg) ∈ {Allow}
  and  ConsentContent506c6(a, s, r, P, f, f', msg) ∈ {Allow}
 then  return true
 else  return false
```

The constraint begins enumerating its scope to include all commands which grant consent. Included in the scope is GrantConsent506a discussed above in Example 7.1.1. The other commands are included in Appendix C.1 for reference. The constraint first checks that the actor has given a consent document which is plain language, the limitation mentioned in the first sentence of (c). The rest of the guards check that all six subparagraph constraints of (c) allow the consent document. If all of the guards are true, the constraint returns true. Otherwise, it returns false.

Note that the check for plain language in the consent document is performed by examining a tag attached to the actor. In our model, all attributes of documents issued by a specific agent are recorded in tags associated with the agent. That is, we do not model the consent document as an object by itself, but as attributes of the agent who issues it.

The constraint for (c)(1) checks tags on the actor as per the requirements of the text:

```
  CST   ConsentContent506c1 (a, s, r, P, f, f', msg)
Scope   {GrantConsent506a, OptionalConsent506a4,
        CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if   a.informs-may-be-used-for-treatment = true
  and   a.informs-may-be-used-for-payment = true
  and   a.informs-may-be-used-for-health-care-operations = true
 then   return true
 else   return false
```

The constraint declares the same scope as ConsentContent506c since it also discusses a requirement for all consent documents. The guards in the constraint are checks for tags indicating that the actor has included clauses informing the subject that the information may be used for treatment, payment, or health care operations. The rest of the constraints for subparagraphs (c)(2)–(6) are in Section C.1.2 on pages 373–375.

As noted above, before the any command is run by the evaluation engine, a constraint search is performed to find all applicable constraints and check their judgments. For example, before the execution of command GrantConsent506a above, the constraints ConsentContent506c and ConsentContent5061 would be executed to check that they allow its execution. □

The previous examples have exercised almost all of the features of Privacy Commands with the exception of the **inform** operation. We now present an example from the Privacy Rule which depends on it.

Example 7.1.4 ([§154.506(f)(2)(ii), v.2000])

The early sections of §164.506 discuss the details of when consent is required and how is it obtained by health care providers. Paragraph (f), the last paragraph in §164.506 details the rules for consent given to health care providers which behave as part of a organized health care arrangement. The rule that we consider in this example concerns the revocation of joint consent [§164.506(f)(2)(ii), v.2000]

> *(f)(1) Standard: Joint consents.* Covered entities that participate in an organized health care arrangement and that have a joint notice under Sec. 164.520(d) may comply with this section by a joint consent.
>
> (2) Implementation specifications: Requirements for joint consents.
>
> (ii) If an individual revokes a joint consent, the covered entity that receives the revocation must inform the other entities covered by the joint consent of the revocation as soon as practicable.

Subparagraph (f)(2)(ii) requires that if a joint consent is revoked, then the covered entity which receives the revocation must inform all members of the organization. The command which implements this is:

```
 CMD   RevokeJointConsent506f2ii (a, s, r, P, f, f', msg)
  if   revoke $\mathbf{in}_a$ P
 and   jointConsent $\mathbf{in}_a$ P
then   invoke RevokeConsent506b5 (a, s, r, P, f, f', msg)
 and   inform r of "Consent revoked"
 and   invoke Record530j(a, s, r, P, f, f', msg)
 and   return true
else   return false
```

The command begins checking that the purpose of the action is the revocation of a consent and that the consent is a joint consent. If both are true, it deletes the consent from the actor using the command RevokeConsent506b5 (see page 371). It then informs the recipient agent that the consent has been revoked. Since revocations are performed by communication, the recording command Record530j is invoked as well (see above in Example 7.1.3).

The inform operation here sends a notice only the agent referred to in the recipient parameter. This is because the model does not have any way of determining which agents are part of the organized health care arrangement relevant to the consent that has been revoked. Implementations would use a database to store the other agents who must be informed about the revocation of any particular consent, but the above mechanism suffices for our model. □

The above examples are an excerpt of the 80 commands and constraints in the Privacy API as shown in Section C.1. The commands and constraints are mostly restricted to the text in §164.506 with a few commands and constraints from §164.530 as noted in Example 7.1.3.

We next discuss the 2003 version of the Privacy Rule, how it differs from the 2000 version, and how those differences manifest in its corresponding Privacy API.

7.1.3 Consent from 2003

In Section 2.1.3 we summarize the paragraphs from §164.506 and §164.508 which relate to use and disclosure of protected health information for treatment, payment, and health care operations. The full text for [§164.506, v.2003] is in Section B.2.1. Since only a portion of [§164.508, v.2003] is applicable to the consent rules, we include only the relevant portion of the rules in Section B.2.3, paragraph §164.508(a). The full text is available online and from the Government Printing Office. As for the 2000 Privacy API in Section 7.1.2, we list the roles, rights, purposes, and tags used in the 2003 Privacy API. The full commands and constraints for it are in Section C.2.

Roles

The text in §164.506 and §164.508(a) mentions the following roles. They are the members of *Role* : Covered Entity, Health Care Provider, Individual. Since the roles are a subset of the roles from the 2000 model, we refer the reader to the previous section Section 7.1.2 on page 247 for their definitions.

Purposes

The text mentions the following purposes for actions. They are the members of *Purpose*. The definitions are as per [§164.501 and §164.508, v.2003] and the list is in alphabetical order. Purposes which overlap with those of the 2000 model retain their definitions for the 2003 model. We refer the read to the previous section Section 7.1.2 on page 247 for their definitions.

Communication Facilitate communication between two entities.

Compliance Aid in the evaluation of legal compliance.

Defend From Individual Aid in an entity's legal defense from the claims of the individual.

Disclose As above.

Face To Face A communication to an individual performed in his presence.

Health Care Fraud Abuse Detection Aid in the detection of health care fraud and abuse.

Health Care Operations As above.

Improve Counseling Aid in training programs "in which students, trainees, or practitioners in mental health learn under supervision to practice or improve their skills in group, joint, family, or individual counseling"

Legal Action Aid in a legal action.

Marketing Used for the marketing of goods or services.

Nominal Value Use for the granting of an item of nominal value.

Own Purposes stated are relevant to the actor.

Paragraph1, Paragraph2 Shorthand for the purposes listed in paragraphs (1) and (2) of the definition of health care operations as per [§164.501, v.2003]. For convenience, the definitions are listed in Section B.2.2 and the hierarchy shown in Section C.2.1.

Payment As above.

Proceedings Use in a review or legal proceedings.

Promotional Gift Use for the granting of a promotional gift.

Recipient Purposes stated are relevant to the recipient of the action.

Related Relationship Action relates to a relationship currently or previously held by an agent on another.

Treatment As above.

Use As above.

Rights

The text mentions the following relationships between agents and objects. They are represented as rights and are the members of *Right*. As in Section 7.1.2, the rights represent one way relationships between an agent and object and are not reflexive. As needed, the descriptions refer to the agent a and object b for clarity. Rights which overlap with those of the 2000 model retain their definitions for the 2003 model. We refer the reader to the previous section Section 7.1.2 on page 248 for their definitions.

Authorization The agent b has granted a written authorization to a to use or disclose information about b.

Consent As above.

Health Care Operations As above.

Local As above.

Organized Health Care Arrangement The agents a and b participate in an organized health care arrangement together.

Originator The agent a created or originated object b.

Payment As above.

Relationship The agent a has a relationship with the agent b.

Treatment As above.

Tags

The 2003 regulatory text mentions fewer tags than the 2000 version. As in Section 7.1.2, they are intuitively named and are listed in Table 7.1. Specific tags are elaborated on as necessary during the rest of the discussion.

Table 7.1: Tags for the HIPAA 2003 Privacy API

authorization-states-remuneration	direct-remuneration
from-third-party	past-relationship
protected-health-information	psychotherapy-notes

Constraint Example

The Privacy API for the 2003 text is similar to the 2000 model in its roles, rights, and purposes. The tag sets have less of an overlap due to textual differences. For instance, the 2000 version has a significant discussion about the required contents of consent documents which is missing in the 2003 version. Conversely, the 2003 version refers to the rules regarding the use and disclosure of psychotherapy notes and information for the purpose of marketing. The rules for them are present in the 2000 version, but are not referenced explicitly in [§164.506, v.2000]. We have presented extensive examples from the 2000 model in Section 7.1.2 and so present only a short example from the 2003 Privacy API which illustrates the flexibility necessary for handling purposes in the model.

Example 7.1.5 ([§164.506(c)(4), v.2003])

The definitions for the purposes in the Privacy API come from the definitions in §164.501. The general purposes treatment, payment, and health care operations include many different types and categories of actions which are enumerated in the definitions. In our discussions of the 2000 model we did not enumerate the child purposes of treatment, payment, or health care operations since they are all subsumed under the general headings without exception. The 2003 text differs, however, in §164.506(c)(4) where it specifies permitted actions based on a subset of the purposes included in health care operations:

> (4) A covered entity may disclose protected health information to another covered entity for health care operations activities of the entity that receives the information, if each entity either has or had a relationship with the individual who is the subject of the protected health information being requested, the protected health information pertains to such relationship, and the disclosure is:
>
> (i) For a purpose listed in paragraph (1) or (2) of the definition of health care operations; or
>
> (ii) For the purpose of health care fraud and abuse detection or compliance.

The purposes listed in paragraphs 1 and 2 are a subset of the purposes included in the definition of health care operations. The full definition of health care operations is in Section B.2.2 on page 340. The purpose hierarchy for the definition in §164.501 is shown in Section C.2.1. By including only the purposes in paragraphs 1 and 2, §164.506(c)(4) limits the purposes for which the action is permitted. In the model we enforce the separate child purposes by denoting the purposes mentioned in paragraphs 1 and 2 as children of the purposes "Paragraph 1" and "Paragraph 2" respectively.

The constraints which implement the purpose checks for §164.506(c)(4) are as follows:

```
  CST   Permitted506c4i (a, s, r, P, f, f', msg)
Scope   {}
   if   Paragraph1 $\mathbf{in}_a$ P
 then   return true
 else   return false

  CST   Permitted506c4i (a, s, r, P, f, f', msg)
Scope   {}
   if   Paragraph2 $\mathbf{in}_a$ P
 then   return true
 else   return false

  CST   Permitted506c4ii (a, s, r, P, f, f', msg)
Scope   {}
   if   healthCareFraudAbuseDetection $\mathbf{in}_a$ P
 then   return true
 else   return false
```

```
   CST   Permitted506c4ii (a, s, r, P, f, f', msg)
 Scope   {}
    if   compliance $\mathbf{in}_a$ P
  then   return true
  else   return false
```

The constraints themselves are similar to the constraints discussed in the 2000 model, but their usage of the purpose hierarchy in an unusual way shows the flexibility of the Privacy Commands. Other guards which check "healthCareOperations $\mathbf{in}_a$ P" would be true if any child purpose of health care operation were true. □

The above example is an excerpt of the 77 commands and constraints in the Privacy API as shown in Section C.2. The commands and constraints include text from §164.506 and §164.508. There are many external references (*e.g.,* see §164.508(a)(2)(ii)) due to the textual style of the 2003 Privacy Rule text. We do not include the translations of all of the external references or commands for §164.508 since they greatly increase the size of the model. Since they refer specifically to psychotherapy notes and marketing, they do not affect the queries we are interested in.

7.1.4 Queries

We translate the Privacy APIs for the 2000 and 2003 rules into Promela models using the methodology of Chapter 6. Due to the length of the Promela models (each thousands of lines long and repetitive) we elide the full models from this document. To give the reader a sample of the models, we provide a sample command from the 2000 Privacy API in Example 6.2.2. The queries that we perform on the models are derived from the three comments mentioned above on Page 52. We reproduce them here for convenience:

> [(1)] Emergency medical providers were also concerned that the requirement that they attempt to obtain consent as soon as reasonably practicable after an emergency would have required significant efforts and administrative burden which might have been viewed as harassing by individuals, because these providers typically do not have ongoing relationships with individuals.
>
> [(2)] The transition provisions would have resulted in significant operational problems, and the inability to access health records would have had an adverse

> effect on quality activities, because many providers currently are not required to obtain consent for treatment, payment, or health care operations.
>
> [(3)] Providers that are required by law to treat were concerned about the mixed messages to patients and interference with the physician-patient relationship that would have resulted because they would have had to ask for consent to use or disclose protected health information for treatment, payment, or health care operations, but could have used or disclosed the information for such purposes even if the patient said "no."

Let us denote the Privacy API for the 2000 rules ϕ_{2000} and the Privacy API for the 2003 rules ϕ_{2003}. The queries that we derive are in terms of the relations defined in Section 5.5. The queries for the comments above are as follows.

Ambulance Workers

For the first comment regarding ambulance workers in an emergency situation, we must examine whether the two policies permit them to use or disclose protected health information for treatment in an emergency situation without prior consent and without requirements to gain consent after the fact. Although the comment addressed emergency workers in general, we concretize the query to ambulance workers, a class of emergency workers, to make the roles more specific and the policy slightly more readable. The resulting query and policies, though, are not specific to ambulance workers and our conclusions are applicable to any emergency medical providers. For brevity let us consider the case for use of protected health information since the disclosure case is similar.

Initial State The initial state for the query is as follows. Let the agent set A_1 = {ambulance, patient} where *Roles*(ambulance) = {healthCareProvider, ambulanceWorker} and *Roles*(patient) = {individual}. All tags for ambulance and patient are set to false. Let the object set $O_1 = \{o_1\}$ where o_1.protected-health-information = true. Let the rights matrix m_1 be the following:

$m_1 =$

	ambulance	patient	o_1
ambulance			local
patient			local

Let l_1 be the log. Let us denote the initial state $s_1 = (A_1, O_1, m_1, l_1)$.

Resulting State The transition that we are interested in can be described in a command:

```
CMD   AmbulanceUse(ambulance, patient, ambulance, $P$, $o_1$, null, $\epsilon$)
 if   healthCareProvider in Roles(a)
and   ambulanceWorker in Roles(a)
and   individual in Roles(s)
and   emergency $\textbf{in}_a$ P
and   treatment $\textbf{in}_a$ P
and   use $\textbf{in}_a$ P
then  insert treatment in (a, s)
and   return true
else  return false
```

Since the ambulance workers may be interested in using information that is not classified as protected health information, AmbulanceUse does not check that f.protected-health-information = true. Their intent is to be able to carry out their jobs for treatment in emergency situations with all the information necessary. Also note that the second guard checks for a role ambulanceWorker which is not present in the definitions for the two Privacy APIs. The check is included because the intent of the comment was to allow ambulance workers to perform their duties without hindrance, not to permit all covered entities to use information without acquiring consent.

The desired result from the transition is a state $s_2 = (A_2, O_2, m_2, l_2)$ with the following properties. The agent and object sets as well as the log should remain unchanged $A_2 = A_1$, $O_2 = O_1$ maintaining all tag states and $l_2 = l_1$. Importantly, it should be the case that ambulance.will-obtain-consent-asap = false. The rights matrix m_2 should be the following:

$m_2 =$

	ambulance	patient	o_1
ambulance		treatment	local
patient			local

Let the purposes set $P = \{$emergency, treatment, use$\}$. Let us denote the argument list $g =$ (ambulance, patient, ambulance, P, o_1, null, ϵ). Our query is then whether the 2000 policy ϕ_{2000} and the 2003 policy ϕ_{2003} permit the transition $s_1 \xrightarrow{\text{AmbulanceUse}(g)} s_2$.

We are interested in proving therefore two relations - $\phi_{2000} \models^*_{(s_1,g)}$ AmbulanceUse(g) and $\phi_{2003} \models^*_{(s_1,g)}$ AmbulanceUse(g). In order to prove the two relations we use SPIN to evaluate the Promela models. We translate s into a Promela initial state and queried invariant as in Figure 7.1. Since the role ambulanceWorker doesn't appear in the 2000 or 2003 models, it is elided from the query. The query for the SPIN checker is then whether `result` is eventually true. Since we have restricted the purposes in the query, we do not allow the running of actions for other purposes which might affect the reachability of `result`.

Results Spin found the Ambulance query to be true in the 2000 version as expected. Since there the commands in 2003 do not use an after the fact consent requirement, the invariant was never true, a trivial false.

Transition Provisions

The second comment refers to the transitional rules in the 2000 Privacy Rule which required covered entities to acquire consent for the use and disclosure of protected health information collected previously. The aim of the comment was to request that the rules requiring consent be removed entirely since it would change the way that health care providers deal with their records. For this query let us consider just one aspect of their request, that health care providers be permitted to use existing protected health information without acquiring consent. This lets us focus on perhaps the strongest aspect of their argument: that the new rules would make it impossible to access already existing records without first acquiring consent. To evaluate the query, we must examine whether ϕ_{2000} and ϕ_{2003} permit the use and disclosure of existing protected health information without consent. As with the previous query let us consider just the use case since the disclose case is similar.

Initial State The initial state for the query is as follows. Let the agent set $A_1 = \{$ce, patient$\}$ where *Roles*(ce)= $\{$ healthCareProvider$\}$ and *Roles*(patient) = $\{$individual$\}$. All tags for ce and patient are set to false. Let the object set be $O_1 = \{o\}$ where the tag o.protected-health-information=true. Let the rights matrix m_1 be:

$$m_1 = \begin{array}{|c||c|c|c|} \hline & \text{ce} & \text{patient} & o \\ \hline \text{ce} & & & \text{local} \\ \hline \text{patient} & & & \text{local} \\ \hline \end{array}$$

Let $l_1 = \epsilon$ be the (empty) log.

Resulting State The transition that we are interested in for the query is represented in the following commands:

```
 CMD   TransitionTreatmentUse(a, s, r, P, f, f', msg)
   if  healthCareProvider in Roles(a)
  and  individual in Roles(s)
  and  f.protected-health-information = true
  and  f.before-transition = true
  and  local in (a, f)
  and  use $\mathbf{in}_a$ P
  and  treatment $\mathbf{in}_a$ P
 then  insert treatment in (a, s)
  and  return true
 else  return false

 CMD   TransitionPaymentUse(a, s, r, P, f, f', msg)
   if  healthCareProvider in Roles(a)
  and  individual in Roles(s)
  and  f.protected-health-information = true
  and  f.before-transition = true
  and  local in (a, f)
  and  use $\mathbf{in}_a$ P
  and  payment $\mathbf{in}_a$ P
 then  insert payment in (a, s)
  and  return true
 else  return false
```

```
CMD   TransitionHealthCareOperationsUse(a, s, r, P, f, f', msg)
 if   healthCareProvider in Roles(a)
and   individual in Roles(s)
and   f.protected-health-information = true
and   f.before-transition = true
and   local in (a, f)
and   use $\mathbf{in}_a$ P
and   healthCareOperations $\mathbf{in}_a$ P
then  insert healthCareOperations in (a, s)
and   return true
else  return false
```

The commands TransitionTreatmentUse, TransitionPaymentUse, and Transition-HealthCareOperationsUse are identical except for the purposes for which permission is sought (*i.e.,* treatment, payment, or health care operations). The commands include a reference to a tag which is not present in either *Tag* of ϕ_{2000} or ϕ_{2003}, "before-transition." Objects tagged with "before-transition" are those which were created before the transition provisions came into effect. Since the tag is not present in either Privacy API, we add it to the tag set for both ϕ_{2000} and ϕ_{2003}. Since no commands in either policy reference the tag, however, it may be ignored by the query engine.

The desired result is a state where the covered entity has rights to treatment, payment, or health care operations even though the patient has not yet given consent. Any of the configurations of the rights matrix below meet that result:

$$m_t = \begin{array}{|c||c|c|c|}\hline & \text{ce} & \text{patient} & o \\ \hline \text{ce} & & \text{treatment} & \text{local} \\ \hline \text{patient} & & & \text{local} \\ \hline \end{array}$$

$$m_p = \begin{array}{|c||c|c|c|}\hline & \text{ce} & \text{patient} & o_1 \\ \hline \text{ce} & & \text{payment} & \text{local} \\ \hline \text{patient} & & & \text{local} \\ \hline \end{array}$$

$m_h =$

	ce	patient	o_1
ce		healthCareOperations	local
patient			local

The rights matrices m_t, m_p, and m_h are the outcomes of successfully running TransitionTreatmentUse, TransitionPaymentUse, and TransitionHealthCareOperationsUse respectively. Our query for ϕ_{2000} and ϕ_{2003} is therefore if there is any command which will lead from m_1 to m_t, m_p, or m_h. Since the additional right in (ce, patient) is different for each matrix above, we are in fact checking three queries, one for each right. We may therefore denote the desired resulting states as follows for treatment, payment, and health care operations respectively. Let A_1, O_1, and l_1 be as defined above for the initial state.

$$s_t = (A_1, O_1, m_t, l_1) \quad s_p = (A_1, O_1, m_p, l_1) \quad s_h = (A_1, O_1, m_h, l_1)$$

Let the purpose set P = {treatment, payment, heathCareOperations, use}. Let us denote the argument list g = (ce, patient, ce, P, o, $\emptyset$, ϵ). We include treatment, payment, and healthCareOperations in the purpose list P for simplicity. Let us denote e_t = TransitionTreatmentUse, e_p = TransitionPaymentUse, and e_h = TransitionHealthCareOperations. Note then that:

$$s_1 \xrightarrow{e_t(g)} s_t \quad s_1 \xrightarrow{e_p(g)} s_p \quad s_1 \xrightarrow{e_h(g)} s_h$$

Our queries then amount to evaluating the following relations:

1. $\phi_{2000} \models^*_{s_1,g} e_t$ and $\phi_{2003} \models^*_{s_1,g} e_t$.

2. $\phi_{2000} \models^*_{s_1,g} e_p$ and $\phi_{2003} \models^*_{s_1,g} e_p$.

3. $\phi_{2000} \models^*_{s_1,g} e_h$ and $\phi_{2003} \models^*_{s_1,g} e_h$.

We translate the commands above to Promela as described in Chapter 6 and evaluate the following invariant called `result`:

```
result = (m.mat[ce].objects[patient].treatment == 1 ||
m.mat[ce].objects[patient].payment == 1 ||
m.mat[ce].objects[patient].healthCareOperations == 1);
```

We then create a SPIN model similar to the one shown for Ambulance query in Figure 7.1 use SPIN to evaluate whether `result` is eventually true.

Result The transition rules query diverged for the 2000 model, indicative that SPIN could not find a command which lead to the satisfaction of the invariant in the 2000 model. A manual evaluation of $\models^*$ shows that no command in the 2000 rules will grant treatment, payment, or health care operations without consent from state s_1. The query for the 2003 version returns true as expected.

Treatment Required By Law

The third comment refers to situations where a health care provider is required by law to treat an individual. In such cases, the health care provider is permitted to use or disclose protected health information for treatment, payment, or health care operations without consent, but still must ask for it anyway. The law would put the health care provider in a difficult situation where the individual must be asked for consent, but an answer in the negative could be ignored.

We are interested in evaluating for both Privacy Rules, then, whether there are situations where a health care provider asks for consent, but may gain the right to treatment, payment, or health care operations even if the consent is refused. The query as phrased is slightly more general than the intent of the content which was concerned that the health care provider was *required* to request consent. This relaxation is motivated partially by a technical concern in it is difficult to express required actions in Privacy APIs. It is also motivated by a general interest in broadening the scope of the stakeholder's comment since they undoubtedly would be concerned about any situation where health care providers may ignore a negative response from an individual.

Initial State The initial state for the query is as follows. Let the agent set be $A_1 = \{\text{ce, patient}\}$ where $\textit{Roles}(\text{ce}) = \{\text{healthCareProvider}\}$ and $\textit{Roles}(\text{patient}) = \{\text{individual}\}$. All tags for ce and patient are false. Let the object set be $O = \{o\}$ where $o.\text{protected-health-information} = \text{true}$. Let the rights matrix m_1 be:

$$m_1 = \begin{array}{|c||c|c|c|} \hline & \text{ce} & \text{patient} & o \\ \hline \text{ce} & & \text{requiredToTreat} & \text{local} \\ \hline \text{patient} & & & \text{local} \\ \hline \end{array}$$

Let $l_1 = \epsilon$ be the (empty) log.

Resulting State The transition that we are interested in is described in the following commands. As in the previous queries we present just the commands related to use since disclose is similar.

```
 CMD  DenyConsent(a, s, r, P, f, f’, msg)
  if  individual in Roles(a)
 and  healthCareProvider in Roles(r)
 and  a = s
 and  denyConsent in_a P
 and  use in_a P
then  delete consent from (r, a)
 and  insert deniedConsent in (r, a)
 and  insert attemptedConsent in (r, a)
 and  return true
else  return false
```

The command DenyConsent is run when an individual denies consent to a health care provider to use protected health information. It indicates the denial using a right “deniedConsent” which is inserted in the rights matrix after removing the consent right. We also use to following commands for the insertion of treatment, payment, and health care operations rights:

```
CMD   IgnoreConsentTreatmentUse(a, s, r, P, f, f’, msg)
 if   healthCareProvider in Roles(a)
and   individual in Roles(s)
and   f.protected-health-information = true
and   local in (a, f)
and   use $\mathbf{in}_a$ P
and   treatment $\mathbf{in}_a$ P
and   requiredToTreat in (a, s)
then  insert treatment in (a, s)
and   return true
else  return false
```

```
CMD   IgnoreConsentPaymentUse(a, s, r, P, f, f’, msg)
 if   healthCareProvider in Roles(a)
and   individual in Roles(s)
and   f.protected-health-information = true
and   local in (a, f)
and   use $\mathbf{in}_a$ P
and   payment $\mathbf{in}_a$ P
and   requiredToTreat in (a, s)
then  insert payment in (a, s)
and   return true
else  return false
```

```
 CMD  IgnoreConsentHealthCareOperationsUse(a, s, r, P, f, f', msg)
  if  healthCareProvider in Roles(a)
 and  individual in Roles(s)
 and  f.protected-health-information = true
 and  local in (a, f)
 and  use $\mathbf{in}_a$ P
 and  healthCareOperations $\mathbf{in}_a$ P
 and  requiredToTreat in (a, s)
then  insert healthCareOperations in (a, s)
 and  return true
else  return false
```

The commands IgnoreConsentTreatmentUse, IgnoreConsentPaymentUse, and Ignore-ConsentHealthCareOperationsUse permit health care providers who are required to treat individuals to obtain rights for treatment, payment, and health care operations. The rights are given ignoring whether consent has been granted or denied to the health care provider.

We are interested then in discovering whether ϕ_{2000} and ϕ_{2003} permit running Deny-Consent followed by IgnoreConsentTreatmentUse, IgnoreConsentPaymentUse, IgnoreCon-sentHealthCareOpertionsUse. The command DenyConsent uses the right "deniedCon-sent" and the purpose "denyConsent" which are not present in either ϕ_{2000} and ϕ_{2003}, so both must be added to both models. Since no commands in either policy use "de-niedConsent" or "denyConsent", we must also add the command DenyConsent to both models. For the query we therefore modify the policies in the following manner. Let us denote the rights and purpose sets for the 2000 and 2003 models as $Right_{2000}$, $Purpose_{200}$, $Right_{2003}$, and $Purpose_{2003}$ respectively. Let us denote the rights sets as $Right'_{2000} = Right_{2000} \cup \{\text{deniedConsent}\}$ and $Right'_{2003} = Right_{2003} \cup \{\text{deniedConsent}\}$ respectively and the purpose sets are $Purpose'_{2000} = Purpose_{2000} \cup \{\text{denyConsent}\}$ and $Purpose'_{2003} = Purpose_{2003} \cup \{\text{denyConsent}\}$ respectively. Let us denote the tag set for the 2000 policy as shown in Table C.1 as Tag_{2000} and the tag set for the 2003 policy as shown in Table 7.1 as Tag_{2003}. Let us denote the roles for the 2000 policy as shown in Section 7.1.2 as $Role_{2000}$ and the roles for the 2003 policy as shown in Section 7.1.3 as $Role_{2003}$. The modified policies are then: $\phi_{2000} = (C_{2000}, E_{2000} \cup \{DenyConsent\}, Role_{2000}, Tag_{2000}, Purpose'_{2000})$ and $\phi_{2003} = (C_{2003}, E_{2003} \cup \{DenyConsent\}, Role_{2003}, Tag_{2003}, Purpose'_{2003})$.

Our desired result is the transition of the rights matrix in two stages. First m_1 should transition to the following configuration which we label m_2:

$m_2 =$

	ce	patient	o
ce		denyConsent	local
patient			local

This corresponds to the patient first denying consent to the health care provider. The matrix should then transition to one of the following configurations:

$m_t =$

	ce	patient	o
ce		denyConsent, treatment	local
patient			local

$m_p =$

	ce	patient	o
ce		denyConsent, payment	local
patient			local

$m_h =$

	ce	patient	o
ce		denyConsent, healthCareOperations	local
patient			local

As in the query for the transition rules, if the matrix reaches any one of configurations m_t, m_p, or m_h, the desired state has been reached since the health care provider then has the right to perform treatment, payment, or health care operations. Our query for ϕ_{2000} and ϕ_{2003} is therefore if there is any command series which will lead from m_1 to m_2 and then from m_2 to m_t, m_p, or m_h. Since the additional right in (ce, patient) is different for each matrix above, we are in fact checking three queries, one for each right. We may therefore denote the desired resulting states as follows for treatment, payment, and health care operations respectively. Let the two purpose sets be as follows: $P_1 = \{$denyConsent, use$\}$ and $P_2 = \{$treatment, payment, healthCareOperations, use$\}$. Let the two argument lists then be: $g_1 = (\text{patient}, \text{patient}, \text{ce}, P_1, o, \emptyset, \epsilon)$ and $g_2 = (\text{ce}, \text{patient}, \text{ce}, P_2, o, \emptyset, \epsilon)$. Let us denote the following resulting states, where A_1, O_1, l_1 are as defined in the initial state.

$$s_2 = (A_1, O_1, m_2, l_1) \quad s_t = (A_1, O_1, m_t, l_1) \quad s_p = (A_1, O_1, m_p, l_1) \quad s_h = (A_1, O_1, m_h, l_1)$$

Let us denote e_t = IgnoreConsentTreatmentUse, e_p = IgnoreConsentPaymentUse, and e_h = IgnoreConsentHealthCareOperationsUse. Note then that:

$$s_1 \xrightarrow{\text{DenyConsent}(g_1)} s_2 \xrightarrow{e_t} s_t \quad s_1 \xrightarrow{\text{DenyConsent}(g_1)} s_2 \xrightarrow{e_p} s_p$$
$$s_1 \xrightarrow{\text{DenyConsent}(g_1)} s_2 \xrightarrow{e_h} s_h$$

Our queries then amount to evaluating the following relations. Let $\overline{e_1} = \{\text{DenyConsent}, e_t\}$, $\overline{e_2} = \{\text{DenyConsent}, e_p\}$, and $\overline{e_3} = \{\text{DenyConsent}, e_h\}$. Let $\overline{g_1} = \{g_1, g_2\}$.

1. $\phi'_{2000} \models^*_{(s_1, \overline{g_1})} \overline{e_1} \wedge \phi'_{2003} \models^*_{(s_1, \overline{g_1})} \overline{e_1}$
2. $\phi'_{2000} \models^*_{(s_1, \overline{g_1})} \overline{e_2} \wedge \phi'_{2003} \models^*_{(s_1, \overline{g_1})} \overline{e_2}$
3. $\phi'_{2000} \models^*_{(s_1, \overline{g_1})} \overline{e_3} \wedge \phi'_{2003} \models^*_{(s_1, \overline{g_1})} \overline{e_3}$

As in the previous queries we translate the above states into Promela and evaluate an LTL property based on the following two invariants, `deny` and `result`:

```
deny = (m.mat[ce].objects[patient].denyConsent == 1);
result = (m.mat[ce].objects[patient].treatment == 1 ||
m.mat[ce].objects[patient].payment == 1 ||
m.mat[ce].objects[patient].healthCareOperations == 1);
```

We are interested in evaluating whether the provider can gain rights over the patient's information even after the patient has denied consent and so we wish to restrict that `result` should be false until `deny` is true. After `deny` is true, `result` should be false. The intuition is that we wish to exclude a case where the provider gains rights on the information before the refusal of the patient. In LTL we would write:

$$(\neg\texttt{result}\ \mathcal{U}\ \texttt{deny}\ )\ \mathcal{U}\ \texttt{result}$$

In order to speed up the search we may help the search by first instructing SPIN to run DenyConsent before performing the search so that we may reduce the search to a simple invariant check whether `result` is eventually true.

Results Spin found the required by law query to be true for the 2000 model, but surprisingly returned true for the 2003 model as well. Upon inspection we found that there was a provision in the (current) 2003 rules stating that even though consent is not required for treatment, payment, or health care operations, health care entities optionally may request consent anyway [164.506(b)(1), v.2003]. No paragraph in the section declares that an optional consent is binding, however. To find out what this omission meant, we consulted with Lauren Steinfeld, Privacy Officer of the University of Pennsylvania. Ms. Steinfeld remarked that a situation of denial of an optional consent request for treatment, payment, and health care operations is legally complex because it has a conflict of patient expectations of privacy and potential medical necessity. In practice, this case may be affected by state laws which preempt the federal guidelines. In short, the section of HIPAA studied here is not sufficient to resolve to case discussed here since it is affected by other laws. To resolve a particular situation of this type of conflict would require examination of the details of the case including the jurisdiction in which it occurs and the manner in which the denial of consent occurred.

7.1.5 Discussion

We learned a number of lessons from our experiments.

First, following the structure and style of the law let us discover a somewhat subtle policy property in required by law treatment query. We had expected that the result for the 2003 version would be negative since the paragraph about entities required by law to provide health care was removed. However, since the 2003 version is silent on the need to respect optional consent, our query found an ambiguity in the legal text that requires deeper legal analysis to resolve.

Second, we can discover properties of the system based on the presence or absence of permissions or environmental flags in the model. For some concerns, an assertion that no command includes a particular obligation is sufficient as in our first query.

Third, we have indication that our current model may suffer from an undecidability property noted by HRU. The original HRU paper [56] proved that the general question of safety, whether granting a general right to one principal can eventually lead to it being leaked to an unauthorized principal, for systems described in their syntax is undecidable. They note that certain systems may not be subject to their conclusion, but we have not yet explored that question for our system. We will consider ways of modifying our model or queries to further explore this problem. The undecidability issue does not preclude manual inspection of the model, however.

Finally, we have an indication as to the limits of automated evaluation of our models. SPIN returns speedy positive results, but may not converge for negative results. In cases where the model checker does not converge, it is a hint that the property may be false, a conclusion which may be verified by manual inspection.

7.2 ICA Privacy Code and HIPAA

In Section 2.1.3, we quoted a section of the Insurance Council of Australia's (ICA) Privacy Code with respect to the disclosures of patient information. For a hypothetical Australian health insurer interested in expanding to the US market, the insurer would need to evaluate whether its policies need to be changed to conform to the US' HIPAA Privacy Rule which applies to health insurers.

The ICA policy includes a rule for "Disclosure by a health service provider" which permits the disclosure of health information by an organization for the purposes of treatment when the individual (*i.e.,* patient) is incapable of consenting. The surrounding text for the quote below is included in Appendix B.3.

> 2.4 Despite the rules on use and disclosure of personal information in Privacy Principle 2.1, an organisation that provides a health service to an individual may disclose health information about the individual to a person who is responsible for the individual if:
>
> (a) the individual:
>
> (i) is physically or legally incapable of consenting to the disclosure; or

(ii) physically cannot communicate consent to the disclosure; and

(b) a natural person (the carer) providing the health service for the organisation is satisfied that either:

(i) the disclosure is necessary to provide appropriate care or treatment of the individual; or

(ii) the disclosure is made for compassionate reasons; and

(c) the disclosure is not contrary to any wish:

(i) expressed by the individual before the individual became unable to give or communicate consent; and

(ii) of which the carer is aware, or of which the carer could reasonably be expected to be aware; and

(d) the disclosure is limited to the extent reasonable and necessary for a purpose mentioned in paragraph (b).

7.2.1 Broadening Constraints

Sections 2.5 and 2.6 list individuals who are responsible for an individual as required in 2.4. The full Privacy API for sections 2.4, 2.5 and 2.6 is in Appendix C.3. The listing of responsible individuals in 2.5 is essentially a list of roles. Section 2.6 is unusual in that it includes definitions which widen the permissions of 2.5 (*i.e.,* include more types of people). We model the extra permissions of 2.6 by including overloaded constraints in 2.5 which reference the permissions in 2.6 as shown in the following example.

Example 7.2.1 (Broadening Permissions)

Section 2.5 of the Insurance Council of Australia's Privacy Code lists people who are responsible for an individual. People who meet the requirements of 2.5 are eligible to receive health information about individuals as provided in 2.4. The permissions in 2.5 include a list of roles which a person must satisfy in order to be responsible for an individual. For instance, the first role listed is as follows:

2.5 For the purposes of 2.4, a person is responsible for an individual if the person is:
(a) a parent of the individual; or

Since "parent" is a relationship between two agents, not a generic role (*i.e.,* it is not correct to label an agent as having the role "parent" since the agent can not be the parent of everyone), it is implemented as right entered in the matrix. The constraint which implements 2.5(a) checks that the recipient of the information has the right “parent” as per the text:

```
       CST   Responsible2.5a(a, s, r, P, f, f’, msg)
     Scope   {}
 Such That   individual in Roles(s)
        if   parent in (r, s)
      then   return true
      else   return false
```

Taken independently, the natural interpretation of 2.5(a) is the natural father or mother of an individual, which is the implication of agents who hold the right “parent”. Section 2.6, however, broadens the definition to include step-parents, adoptive parents, and foster-parents:

> 2.6 For the purposes of 2.5: ...
>
> *parent* of an individual includes a step-parent, adoptive parent and a foster-parent, of the individual.

While we could perhaps ignore definition of 2.6 by simply defining that agents who are step-parents, adoptive parents, and foster-parents may also hold the right “parent”, doing so would make the model less expressive accurate to the text. Additionally, other locations in the document where “parent” may have the natural definition would create unnecessary confusion of semantics. We therefore define specific rights for step-parents, adoptive parents, and foster-parents which we check in the following constraints:

```
       CST   Parent2.6(a, s, r, P, f, f’, msg)
 Such That   individual in Roles(s)
        if   stepParent in (r,s)
      then   return true
      else   return false
```

```
       CST   Parent2.6(a, s, r, P, f, f', msg)
 Such That   individual in Roles(s)
        if   adoptiveParent in (r,s)
      then   return true
      else   return false

       CST   Parent2.6(a, s, r, P, f, f', msg)
 Such That   individual in Roles(s)
        if   fosterParent in (r,s)
      then   return true
      else   return false
```

In order to broaden Responsible2.5a, we must include an option to follow the broadened definition of 2.6 for parents and so we create an overloaded constraint Responsible2.5a:

```
       CST   Responsible2.5a(a, s, r, P, f, f', msg)
     Scope   {}
 Such That   individual in Roles(s)
        if   Parent2.6(a, s, r, P, f, f', msg) ∈ {Allow}
      then   return true
      else   return false
```

The effect of including the above constraint is that a reference Responsible2.5a (for instance Responsible2.5 on page 419) may be satisfied by fulfilling either one of the two copies of Responsible2.5a.

□

In Example 7.2.1 we would rather have made the broad definition of 2.6 able to affect 2.5(a) without needing to include a reference to 2.6 via an overloaded constraint. First, since 2.6 references 2.5 but not vice versa, doing so would preserve the reference structure of the text better. Second, if 2.6 is ever changed or modified, 2.5(a) would need to be modified as well, breaking some of the encapsulation of paragraphs. We can not do so, however, for two reasons which illustrate some of the limitations of the Privacy Commands language:

1. Section 2.5 is translated as constraints only and constraints can not have other constraints in their scope. This limitation is placed to prevent circular references and scope for the evaluation engine.

2. Constraints can only place limitations on other constraints or commands, not override them with more permissive judgments. In order to allow constraints to grant permissions which override prohibitions we need to include some notion of precedence for constraints' judgments.

The first limitation could be solved by relaxing the definition of scope in Definition 5.2.10 to permit constraints in scope lists. Doing so would, however, require extra restrictions to prevent circular references in scopes. While not overly burdensome to add to the model, permitting constraints to modify other constraints may lead to policy constructions that are harder for policy writers to understand. The second limitation is inherent to the semantics of the language. Since there is no notion of priority or precedence between constraints, there is no way for one to override another without using overloading, as shown in Example 7.2.1.

7.2.2 Comparison of Disclosure Rules

Let us return to our hypothetical case of an insurance company in Australia interested in discovering whether compliance with the ICA's code is sufficient for compliance with HIPAA. For this case study let us focus on the quote from 2.4 above which grants permission to health providers to disclose health information about individuals when they are an incapable of providing consent. We consider only a selection of the constraints from 2.4 here and provide the full Privacy API in Appendix C.3.

```
       CST   Permitted2.4(a, s, r, P, f, f', msg)
     Scope   {Disclose2.4}
Such That    healthProvider in Roles(a)
       and   individual in Roles(s)
       and   f.health-information = true
        if   Permitted2.4a(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Permitted2.4b(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Permitted2.4c(a, s, r, P, f, f', msg) ∈ {Allow}
       and   responsibleFor in (r, s)
       and   disclose $\mathbf{in}_a$ P
      then   return true
      else   return false
```

The constraint Permitted2.4 permits the disclosure of health information about an individual by a health care provider when all of the conditions in 2.4(a), (b), and (c) are fulfilled. For brevity, the referenced constraints with the exception of one are elided here, but are provided in full in Appendix C.3.

Section 2.4(b) requires that the disclosure be for either treatment or compassionate purposes. Section 2.4(b)(i) permits disclosure if "the disclosure is necessary to provide appropriate care or treatment of the individual;" and it is implemented with the following overloaded constraints:

```
  CST   Permitted2.4bi(a, s, r, P, f, f', msg)
Scope   {}
   if   treatment $\mathbf{in}_a$ P
  and   a.satisfied-disclosure-necessary-to-provide-appropriate-care = true
 then   return true
 else   return false
```

```
1    CST    Permitted2.4bi(a, s, r, P, f, f', msg)
2  Scope    {}
3     if    treatment $\mathbf{in}_a$ P
4    and    a.satisfied-disclosure-necessary-to-provide-appropriate-treatment = true
5   then    return true
6   else    return false
```

Both constraints check that the purpose includes treatment, but differ with respect to a tag check on line 4. If either constraint is satisfied, the requirements of 2.4(b)(i) are satisfied.

The disclosure command for 2.4 is as follows:

```
 1   CMD    Disclose2.4(a, s, r, P, f, f', msg)
 2    if    Permitted2.4(a, s, r, P, f, f', msg)
 3   and    individual in Roles(s)
 4   and    healthProvider in Roles(a)
 5   and    responsible in (r, s)
 6   and    disclose $\mathbf{in}_a$ P
 7   and    f.health-information = true
 8   and    local in (a, f)
 9  then    insert local in (r, f)
10   and    return true
11  else    return false
```

The command Disclose2.4 performs the disclosure of health information to a recipient person when Permitted2.4 is satisfied and other side conditions are satisfied. If all of the guards are satisfied, the command performs the disclosure on line 9.

For comparison, let us consider parallel permissions in 2000 Privacy Rule [§164.506(a)(3)(i)(C), v.2000] regarding disclosure of health information for treatment purposes when an individual is unable to communicate consent:

> (3)(i) A covered health care provider may, without prior consent, use or disclose protected health information created or received under paragraph (a)(3)(i)(A)–(C) of this section to carry out treatment, payment, or health care operations:

> . . . or (C) If a covered health care provider attempts to obtain such consent from the individual but is unable to obtain such consent due to substantial barriers to communicating with the individual, and the covered health care provider determines, in the exercise of professional judgment, that the individual's consent to receive treatment is clearly inferred from the circumstances.
>
> (ii) A covered health care provider that fails to obtain such consent in accordance with paragraph (a)(3)(i) of this section must document its attempt to obtain consent and the reason why consent was not obtained.

Paragraph §164.506(a)(3)(i)(C) permits the disclosure of health information for treatment purposes, regardless of the recipient. When any disclosure is performed under the paragraph, (a)(3)(ii) requires that the health care provider document its attempt. Let us consider a subset of the commands and constraints which implement the above quote. The full Privacy API for the 2000 rules is in Section B.1.1.

```
1         CST   Permitted506a3(a, s, r, P, f, f', msg)
2       Scope   {TreatmentUse506a3, PaymentUse506a3,
3               HealthCareOperationsUse506a3, TreatmentDisclose506a3,
4               PaymentDisclose506a3, HealthCareOperationsDisclose506a3}
5   Such That   individual in Roles(s)
6         and   f.protected-health-information = true
7          if   Permitted508a3i(a, s, r, P, f, f', msg) ∈ {Allow}
8        then   return true
9        else   return false
```

The constraint for (a)(3), Permitted506a3 checks that the subject is an individual and the object is protected health information. The sole regular guard on line 7, a reference to Permitted506a3i checks that the conditions in the subparagraph are satisfied as well.

```
1          CST   Permitted506a3i(a, s, r, P, f, f', msg)
2        Scope   {TreatmentUse506a3, PaymentUse506a3,
3                HealthCareOperationsUse506a3, TreatmentDisclose506a3,
4                PaymentDisclose506a3, HealthCareOperationsDisclose506a3}
5    Such That   individual in Roles(s)
6          and   f.protected-health-information = true
7           if   Permitted506a3iC (a, s, r, P, f, f', msg) ∈ {Allow}
8         then   return true
9         else   return false
```

The overloaded constraints for (a)(3)(i), Permitted506a3i, check the same *such that* guards as (a)(3) and include a single regular guard on line 7 which is a reference to the subparagraph (a)(3)(i)(A), (a)(3)(i)(B), and (a)(3)(i)(C). For brevity we show only the one which refers to (a)(3)(i)(C). The other two are shown in Appendix C.1.2 beginning on page 362.

```
1          CST   Permitted506a3iC (a, s, r, P, f, f', msg)
2    Such That   s.barriers-to-communication = true
3          and   s.professional-judgment-indicates-consent = true
4          and   individual in Roles(s)
5          and   f.protected-health-information = true
6          and   healthCareProvider in Roles(a)
7           if   a.attemptedConsent = true
8         then   return true
9         else   return false
```

The constraint for (a)(3)(i)(C), Permitted506a3iC, checks tags on the subject to see that there are barriers to communication from the individual (line 2) and that the subject indicates consent which is discernable by professional judgment (line 3). It also checks that the subject is an individual (line 4), that the object is protected health information (line 5), that the actor is a health care provider (line 6), and that the actor has attempted to obtain consent from the individual (line 7).

```
 1   CMD   TreatmentDisclose506a3 (a, s, r, P, f, f', msg)
 2    if   Permitted506a3 (a, s, r, P, f, f', msg) $\in$ {Allow}
 3   and   individual in Roles(s)
 4   and   healthCareProvider in Roles(a)
 5   and   local in (a, f)
 6   and   treatment $\textbf{in}_a$ P
 7   and   disclose $\textbf{in}_a$ P
 8  then   insert local in (r, f)
 9   and   invoke noteAttempt506a3ii(a, s, r, P, f, f', msg)
10   and   return true
11  else   return false
```

The command TreatmentDisclose506a3 is one of several from (a)(3) as shown in Appendix C.1.2 beginning on page C.1.2. Since it is the most relevant to our comparison to the ICA Privacy Code, we include it. It checks that the constraint for the paragraph, Permitted506a3, is satisfied (line 2), and that the purposes and roles are correct (lines 3–7). If all of the guards are satisfied, line 8 performs the disclosure to the recipient and line 9 references noteAttempt506a3ii to record the reason for the disclosure despite the lack of consent.

```
1   CMD   noteAttempt506a3ii (a, s, r, P, f, f', msg)
2    if   a.attemptedConsent = true
3  then   insert attemptedConsent in (a, s)
4   and   insert "Attempted consent and failed" in log
5   and   return true
6  else   return false
```

The command noteAttempt506a3ii performs the recording requirement mentioned in (a)(3)(ii), checking that the actor attempted to receive consent (line 2). Line 3 then grants the right "attemptedConsent" for the actor on the recipient to indicate that the actor has attempted to receive consent from the individual. Line 4 enters a note in the log indicating the reason for the granting of the permission, that consent was attempted and failed.

Let us refer to the Privacy API for the ICA Privacy Code as ϕ_{ica} and the Privacy API for the 2000 Privacy Rule as ϕ_{2000}. The roles, purposes, rights, and tags for ϕ_{2000} are shown in Section 7.1.2. Let use denote them $Role_{2000}$, $Purpose_{2000}$, $Right_{2000}$, and Tag_{2000} respectively. The roles, purposes, rights, and tags for ϕ_{ica} are as follows. Let us denote them $Role_{ica}$, $Purpose_{ica}$, $Right_{ica}$, and Tag_{ica} respectively. Denoting the commands for the ICA privacy code at E_{ica} and the constraints C_{ica}, we define $\phi_{ica} = (C_{ica}, E_{ica}, Role_{ica}, Tag_{ica}, Purpose_{ica})$.

Roles

The roles mentioned in 2.4, 2.5, and 2.6 are as follows. They are the members of $Role_{ica}$. The definitions are derived based on the usage and intent of the text.

Cannot Physically Communicate Consent An individual who can not communicate consent due to physical incapacity.

Health Provider An individual of organization which provides health services.

Individual A natural person about whom health information is held by a health provider.

Legally Incapable An individual who is legally incapable of granting consent.

Natural Person A person (to the exclusion of an organization).

Physically Incapable An individual who is physically incapable of granting consent.

Providing Health Service A natural person who provides health services on behalf of a health provider organization.

Purposes

The purposes mentioned in 2.4, 2.5, and 2.6 are as follows. They are the members of $Purpose_{ica}$. The definitions are based on the usage and intent of the text.

Compassion An action performed for compassionate reasons.

Disclose Disclosure of information to another agent.

Necessary An action deemed necessary for another purpose.

Reasonable An action deemed reasonable for fulfillment of another purpose.

Treatment An action performed to aid the treatment of an individual.

Rights

The rights mentioned in 2.4, 2.5, and 2.6 are as follows. They are the members of $Right_{ica}$. The definitions are based on the usage and intent of the text and as defined in 2.6 For clarity, in the following definitions we use the variables a and b to represent members of $Agent$ and o to represent members of $Object - Agent$. Let us consider each right below as being held by a over b (*i.e.,* (a, b)) or held by a over o (*i.e.,* (a, o)).

Guardian Agent a is the legal guardian of b.

Household Member Agent a is a member of b's household.

Intimate Personal Relationship Agent a has an intimate personal relationship with b.

Local Agent a has local access to o.

Nominated Emergency Contact Agent a has been nominated as an emergency contact for b.

Power of Attorney Agent a holds power of attorney over b.

Responsible Agent a is responsible for b.

The text also includes the following familial relationships which are self explanatory: Parent, Step Parent Adoptive Parent Foster Parent, Child, Adopted Child, Step Child, Foster Child, Sibling, Half Brother, Half Sister, Adoptive Brother, Adoptive Sister, Step Brother, Step Sister, Foster Brother, Foster Sister, Spouse, Defacto Spouse, Relative, Grandparent, Grandchild, Uncle, Aunt, Nephew, and Niece. Each relationship is represented as a right in the matrix. The relation "Relative" is special in that it is defined only in terms of other relations (*i.e.,* grandparent, grandchild, uncle, aunt, nephew, and niece) and therefore can not be held independently.

Tags

The tags which we derive from conditions mentioned in 2.4, 2.5, and 2.6 are as follows. They are the members of Tag_{ica}. For clarity, in the following definitions we use the variable a to represent members of $Agent$ and o to represent members of $Object - Agent$.

At Least 18 Agent a is at least 18 years of age.

Aware Of Wish Agent a is aware of a wish from an agent.

Health Information Object o contains health information about an agent.

Nominated By Individual Agent a has been nominated by an individual for a purpose.

Power Exercisable To Health Agent a has been granted power which is exercisable with respect to health information.

Power Granted By Individual Agent a has been granted power of attorney by an individual.

Reasonably Expected To Be Aware Of Wish Agent a is aware of a wish of which it is reasonable that the agent should be aware.

Satisfied Disclosure Necessary To Provide Appropriate Care Agent a is satisfied that a proposed disclosure is necessary to provide appropriate care to an agent.

Satisfied Disclosure Necessary To Provide Appropriate Treatment Agent a is satisfied that a proposed disclosure is necessary to provide appropriate treatment to an agent.

Wish Contrary Agent a has expressed a wish contrary to an action.

Wish Expressed Before Unable To Consent Agent a has expressed a wish before becoming unable to grant consent.

We now compare the relative permissiveness of ϕ_{ica} and ϕ_{2000} in the following scenarios. Since ϕ_{ica} and ϕ_{2000} use different roles, purposes, rights, and tags, we must express scenarios in terms of the set-wise union of each set. Let us denote the combined sets as follows:

$Role = Role_{2000} \cup Role_{ica}$, $Purpose = Purpose_{2000} \cup Purpose_{ica}$, $Right = Right_{2000} \cup Right_{ica}$, and $Tag = Tag_{2000} \cup Tag_{ica}$.

Example 7.2.2 (Non-responsible recipient)

Let us consider the case where a care giver working on behalf of a health care provider needs to disclose health information about an individual for purposes of treatment. The care giver is satisfied that the disclosure is necessary for the provision of appropriate care, however the recipient is a second health care provider rather than a person who is responsible for the individual. The individual is incapacitated and thus unable to respond to the health care providers first attempt to gain consent to the disclosure, but has not previously expressed a wish contrary to the disclosure.

The agents for the scenario are then: $Agent = \{$hcp1, patient, hcp2$\}$ where $Roles$(patient) $= \{$individual, physicallyIncapable$\}$, $Roles$(hcp2) $= \{$healthProvider, healthCareProvider$\}$, and $Roles$(hcp1) $= \{$healthProvider, healthCareProvider, naturalPerson, providingHealthService$\}$. If we observe that the role healthProvider $\in Role_{ica}$ is identical to the role healthCareProvider $\in Role_{2000}$, we may collapse the two roles into a single role, but for this example we preserve the two roles. The following tags are set to true: hcp1.satisfied-disclosure-necessary-to-provide-appropriate-treatment, hcp1.attempted-consent, patient.barriers-to-communication, patient.profesional-judgment-indicates-consent. All other tags on hcp1, patient, and hcp2 are set to false.

We define one object for the scenario $Object = \{o\}$ where information about agent patient is included in o. The tags o.health-information and o.protected-health-information are set to true. All other tags on o are set to false.

The rights matrix before the disclosure is to occur is m_1 as shown:

$m_1 =$

	hcp1	patient	hcp2	o
hcp1				local
patient				
hcp2				

The log begins empty $l_1 = \epsilon$.

The purposes for the disclosure are $P = \{$disclose, treatment, reasonable, necessary$\}$.

The initial state for the scenario is then $s_1 = (A_1, O_1, m_1, l_1)$ where $A_1 = \{$hcp1, patient, hcp2$\}$ and $O_1 = \{o\}$. Let us consider the following argument list $g_1 =$ (hcp1, patient, hcp2, P, o, null, ϵ). In order for the first health care provider to perform the disclosure, we must reach a state where hcp2 has local access to o (*i.e.,* local **in** (hcp2, o)). The target matrix is therefore:

$$m_2 = \begin{array}{|c||c|c|c|c|}
\hline
 & \text{hcp1} & \text{patient} & \text{hcp2} & o \\
\hline
\text{hcp1} & & & & \text{local} \\
\hline
\text{patient} & & & & \\
\hline
\text{hcp2} & & & & \text{local} \\
\hline
\end{array}$$

We may characterize the transition then as $s_1 \xrightarrow{e(g_1)} s_2$ where $s_2 = (A_1, O_1, m_2, l_1)$ and e is some command which performs the desired update of m_1 with arguments g_1:

```
 CMD  e(a, s, r, P, f, f', msg)
   if  true
 then  insert local in (r, f)
  and  return true
 else  return false
```

Analyzing ϕ_{2000} we find that TreatmentDisclose506a3 weakly licenses e at s_1 with g_1 since it performs the transition $s_1 \xrightarrow{\text{TreatmentDisclose506a3}(g_1)} s_h$ where $s_h = (A_1, O_1, m_h, l_h)$, $l_h = l_1 +$ "Attempted consent and failed", and m_h is as follows:

$$m_h = \begin{array}{|c||c|c|c|c|}
\hline
 & \text{hcp1} & \text{patient} & \text{hcp2} & o \\
\hline
\text{hcp1} & & \text{attemptedConsent} & & \text{local} \\
\hline
\text{patient} & & & & \\
\hline
\text{hcp2} & & & & \text{local} \\
\hline
\end{array}$$

Since $\mathsf{noconflict}(s_2, s_1)$ by Definition 5.5.13, we have that $\phi_{2000} \models^*_{(s_1, g_1)} e$.

Analyzing ϕ_{ica} we find that there are no commands which can reach m_2 or any state which weakly licenses s_2. Since hcp2 is not responsible for patient (*i.e.,* !responsible **in** (hcp2, patient)), the constraints in 2.4 are not satisfied. As a result, the ICA rules are more strict than the HIPAA rules and therefore compliance with the ICA rules is sufficient for compliance with the HIPAA rules. Since ϕ_{ica} does not license any transitions for any arguments, we have trivially then that $\phi_{ica} \prec_s \phi_{2000}$ as per Definition 5.5.26. Conversely, since ϕ_{2000} does license TreatmentDisclose506a3 at s_1 while ϕ_{ica} does not, $\phi_{2000} \not\prec_s \phi_{ica}$.

$\square$

Let us now consider a different scenario where the recipient is responsible for the subject.

Example 7.2.3 (Responsible recipient)

Let us modify the case in Example 7.2.2 slightly to a case where a health care provider needs to disclose health information about an individual for purposes of treatment to an aunt who is responsible for the individual. As in Example 7.2.2, the health provider is satisfied that the disclosure is necessary for the provision of appropriate care. The individual is incapacitated and thus unable to respond to the health care providers first attempt to gain consent to the disclosure, but has not previously expressed a wish contrary to the disclosure.

The agents for the scenario are then: *Agent* = {hcp1, patient, aunt} where *Roles*(patient) = {individual, physicallyIncapable}, *Roles*(hcp1) = {healthProvider, healthCareProvider, naturalPerson, providingHealthService}, *Roles*(aunt) = {}, and If we observe that the role healthProvider $\in Role_{ica}$ is identical to the role healthCare-Provider $\in Role_{2000}$, we may collapse the two roles into a single role, but for this example we preserve the two roles. The following tags are set to true: hcp1.satisfied-disclosure-necessary-to-provide-appropriate-treatment, hcp1.attempted-consent, patient.barriers-to-communication, patient.profesional-judgment-indicates-consent, and aunt.at-least-18. All other tags on hcp1, patient, and hcp2 are set to false.

We define one object for the scenario $Object = o$ where information about agent patient is included in o. The tags o.health-information and o.protected-health-information are set to true. All other tags on o are set to false.

The rights matrix before the disclosure is to occur is m_1 as shown:

$m_1 =$

	hcp1	patient	aunt	o
hcp1				local
patient				
aunt		aunt, responsible, householdMember		

The log begins empty $l_1 = \epsilon$. The purposes for the disclosure are $P = \{$disclose, treatment, reasonable, necessary$\}$. The initial state for the scenario is then $s_1 = (A_1, O_1, m_1, l_1)$ where $A_1 = \{$hcp1, patient, aunt$\}$ and $O_1 = \{o\}$. Let us consider the following argument list $g_1 =$ (hcp1, patient, aunt, P, o, null, ϵ). In order for the health care provider to perform the disclosure, we must reach a state where aunt has local access to o (*i.e.,* local **in** (aunt, o)). The target matrix is therefore:

$m_2 =$

	hcp1	patient	aunt	o
hcp1				local
patient				
aunt		aunt, responsible, householdMember		local

We may characterize the transition then as $s_1 \xrightarrow{e(g_1)} s_2$ where $s_2 = (A_1, O_1, m_2, l_1)$ and e is some command which performs the desired update of m_1 with arguments g_1:

```
 CMD  e(a, s, r, P, f, f', msg)
   if  true
 then  insert local in (r, f)
  and  return true
 else  return false
```

Analyzing ϕ_{2000} we find that as in Example 7.2.2, TreatmentDisclose506a3 weakly licenses e at s_1 with g_1 since it performs the transition $s_1 \xrightarrow{\text{TreatmentDisclose506a3}(g_1)} s_h$ where $s_h = (A_1, O_1, m_h, l_h)$, $l_h = l_1+$ "Attempted consent and failed", and m_h is as follows:

$m_h =$

	hcp1	patient	aunt	o
hcp1		attemptedConsent		local
patient				
aunt		aunt, responsible, householdMember		local

Since $\mathsf{noconflict}(s_2, s_1)$ by Definition 5.5.13, we have that $\phi_{2000} \models^*_{(s_1, g_1)} e$.

Analyzing ϕ_{ica} we find that Disclose2.4 can reach m_2 from s_1 with g_1, so $\phi_{ica} \models_{(s_1,g_1)} e$. However, $\phi_{2000} \not\models_{(s_1,g_1)} e$ since TreatmentDisclose506a3 requires the addition of the right "attemptedConsent" and appending to the log. As a result, the HIPAA rules are more strict than the ICA rules and therefore compliance with the ICA rules is not sufficient for compliance with the HIPAA rules. Since at s_1 with g_1, ϕ_{ica} strongly licenses Disclose2.4 but not TreatmentDisclose506a3 while ϕ_{2000} strongly licenses Treatment506a3 but not Disclose2.4, neither is at least as strict as the other even under the more restrictive definition: $\phi_{2000} \not\prec_{(s_1,g_1)} \phi_{ica}$ and $\phi_{ica} \not\prec_{(s_1,g_1)} \phi_{2000}$.

□

Examples 7.2.2 and 7.2.3 illustrate that for complex, real world policies it is unlikely to find situations where one policy strictly subsumes another from all states and with any argument list. Example 7.2.2 shows a case where the ICA is trivially at least as strict as the 2000 Privacy Rule for all arguments at a given state since it does not strongly license any commands at s_1. In Example 7.2.3 we find that neither policy is at least as strict as the other for the given state even restricted to one argument list. Even though we can not easily find policy level comparisons, we can use strong and weak licensing to examine the permissiveness of policies on a command by command level, checking whether one policy strongly or weakly licenses the commands of another. This illustrates the usefulness of the flexibility of our definitions. By defining flexible relations which let us denote policy comparisons at smaller levels of granularity we can express relations applicable to real world legal policies and situations.

7.3 Cable TV Privacy Act and TiVo

In Section 2.1.3 we discuss the US Cable TV Privacy Act. As discussed, the Cable TV Privacy Act restricts the uses and disclosures that cable operators may perform. Since TiVo's DVR service runs over cable, it is interesting to explore whether the privacy policy as published by TiVo is compliant with the Cable TV Privacy Act. To do so, we translate sections of the policies into commands and constraints and evaluate whether CTPA strongly or weakly licenses TiVo's policy [62].

7.3.1 Cable TV Privacy Act

Section 551(a) of the CTPA discusses the situations when a cable operator may disclose personal information about its subscribers. The text of the CTPA is included for reference in Appendix B.4. Let us consider the following quote from 551(c)(1) regarding the rules for disclosure:

> (c) Disclosure of personally identifiable information.
>
> (1) Except as provided in paragraph (2), a cable operator shall not disclose personally identifiable information concerning any subscriber without the prior written or electronic consent of the subscriber concerned and shall take such actions as are necessary to prevent unauthorized access to such information by a person other than the subscriber or cable operator.
>
> (2) A cable operator may disclose such information if the disclosure is
>
> (A) necessary to render, or conduct a legitimate business activity related to, a cable service or other service provided by the cable operator to the subscriber;

Paragraph (c)(1) forbids all disclosures of personally identifiable information without prior consent with three exceptions listed in (c)(2)(A), (c)(2)(B), and (c)(2)(C). Let us consider the permission in (2)(A), the one quoted above. The corresponding commands and constraints for the above paragraphs are as follows.

Paragraph (c)(1) permits the disclosure of personally identifiable information in three situations: (1) in the cases permitted by (c)(2), (2) if the subscriber had previously granted written consent, or (3) if the subscriber has previously granted written consent. The constraints corresponding to the paragraph are the three overloaded constraints Permitted551c1:

```
 1         CST  Permitted551c1(a, s, r, P, f, f', msg)
 2       Scope  {Disclose551c1, Disclose551c2A, Disclose551c2B, Disclose551c2C }
 3   Such That  cableOperator in Roles(a)
 4         and  subscriber in Roles(s)
 5         and  f.personally-identifiable-information = true
 6         and  f.subject-subscriber = true
 7         and  disclose $\mathbf{in}_a$ P
 8          if  Permitted551c2(a, s, r, P, f, f', msg) $\in$ {Allow}
 9        then  return true
10        else  return false
```

The first constraint Permitted551c1 permits disclosure of personally identifiable information if the constraint Permitted551c2 (*i.e.,* 551(c)(2)) permits the disclosure (line 8). The *such that* guards check the roles of the actor and subject (lines 3–4), that the object is personally identifiable information (line 5) and about the subject (line 6), and that the purpose of the action is disclosure (line 7).

```
 1         CST  Permitted551c1(a, s, r, P, f, f', msg)
 2       Scope  {Disclose551c1, Disclose551c2A, Disclose551c2B, Disclose551c2C }
 3   Such That  cableOperator in Roles(a)
 4         and  subscriber in Roles(s)
 5         and  f.personally-identifiable-information = true
 6         and  f.subject-subscriber = true
 7         and  disclose $\mathbf{in}_a$ P
 8          if  consent in (a, s)
 9         and  written in (a, s)
10        then  return true
11        else  return false
```

```
 1         CST   Permitted551c1(a, s, r, P, f, f', msg)
 2       Scope   {Disclose551c1, Disclose551c2A, Disclose551c2B, Disclose551c2C }
 3   Such That   cableOperator in Roles(a)
 4         and   subscriber in Roles(s)
 5         and   f.personally-identifiable-information = true
 6         and   f.subject-subscriber = true
 7         and   disclose $\mathbf{in}_a$ P
 8          if   consent in (a, s)
 9         and   electronic in (a, s)
10        then   return true
11        else   return false
```

The second and third Permitted551c1 constraints are similar to the first with the exception of line 8. In the second constraint lines 8–9 checks that the subject has granted written consent to the cable operator. The third constraint checks that the subject has granted electronic consent to the cable operator.

```
1   CMD   Disclose551c1(a, s, r, P, f, f', msg)
2    if   Permitted551c1(a, s, r, P, f, f', msg) $\in$ {Allow}
3   and   f.personally-identifiable-information = true
4   and   cableOperator in Roles(a)
5   and   subscriber in Roles(s)
6   and   disclose $\mathbf{in}_a$ P
7  then   insert local in (r, f)
8   and   return true
9  else   return false
```

The command Dislcose551c1 allows cable operators (line 4) to disclose (line 6) personally identifiable information (line 3) about subscribers (line 5) provided that the paragraph permits the disclosure (line 2). Since Permitted551c1 is overloaded, if any of the three constraints shown above permit the action, the disclosure is performed (line 7).

```
       CST   Protect551c1(a, s, r, P, f, f', msg)
     Scope   { }
 Such That   cableOperator in Roles(a)
        if   a.takes-actions-prevents-unauthorized = true
      then   return true
      else   return false
```

The constraint Protected551c1 represents the requirement in 551(c)(1) that the cable operator "shall take such actions as are necessary to prevent unauthorized access to such information by a person other than the subscriber or cable operator." We implement the requirement using a constraint which checks a tag indicating that the cable operator has taken such actions as required. The constraint illustrates a limitation of the Privacy Commands language in that policies can not describe actions which must be performed unless they are preconditions for other actions. Since the legal text does not delineate what actions are to be taken to prevent access and does not proscribe any actions based on the non-fulfillment of the clause, the Privacy API formal model has no mechanism for enforcing the fulfillment of the requirement.

The roles, purposes, rights, and tags for the CTPA Privacy API are as follows. Let us denote the CTPA Privacy API as $\phi_{ctpa} = (C_{ctpa}, E_{ctpa}, Role_{ctpa}, Tag_{ctpa}, Purpose_{ctpa})$ where C_{ctpa} and E_{ctpa} are the constraints and commands for the Cable TV Privacy Act as shown in Appendix C.4 and $Role_{ctpa}$, Tag_{ctpa}, and $Purpose_{ctpa}$ are the roles, tags, and purposes for the policy as we define next.

Roles

The roles included in the CTPA text are as follows. They are the members of $Role_{ctpa}$ and are defined as per their usage in the policy.

Subscriber An agent subscribed to a service provided by a cable operator.

Cable Operator A provider of cable services, including cable television.

Cable Subscriber An agent who subscribes to cable services.

Other Service Subscriber An agent who subscribes to services from a cable operator other than cable service.

Government Entity An entity which is part of the government.

Purposes

The purposes included in the CTPA text are as follows. They are the members of $Purpose_{ctpa}$ and are defined as per their usage in the policy.

Business Activity For the purpose of conducting a business activity.

Cable Communications For the purpose of detecting access to cable communications.

Cable Service For the purpose of providing cable service.

Collect For the purpose of collecting information.

Collect PII For the purpose of collecting personally identifiable information.

Conduct For the purpose of conducting another purpose.

Detect Unauthorized For the purpose of detecting unauthorized usage of a service.

Disclose For the disclosure of information.

Legitimate For the purpose of conducting a legitimate activity.

Not Directly Reveal Action which does not directly reveal information.

Not Indirectly Reveal Action which does not indirectly reveal information.

Obtain Necessary Info For the purpose of obtaining necessary information for the completion of another purpose.

Order Authorizes Action performed which is authorized by a court order.

Other Service For the purpose of providing a service other than cable service.

Provide Prohibit Purpose of providing the opportunity for an agent to prohibit an action.

Table 7.2: Tags for the Cable TV Privacy Act

clear-conspicuous	clear-convincing-evidence
cable-service	entered-before-effective
entering-agreement	extent-of-viewing
extent-of-use	frequency-disclosure
identification-types-persons-disclosure	information-material-evidence
limitations-collection-disclosure	maintained-period
message-notifies	nature-disclosure
nature-of-transactions-over-system	nature-personally-identifiable-info
nature-use-information	opportunity-to-appear
opportunity-to-contest	other-service
personally-identifiable-information	purpose-disclosure
right-subscriber-f-h-enforce	separate-statement
subject-suspected-criminal-activity	subject-subscriber
takes-actions-prevents-unauthorized	times-place-subscriber-access
written-statement	

Provide Limit Purpose of providing the opportunity for an agent to limit an action.

Pursuant Court Order Action performed pursuant to court order.

Render For the purpose of rendering a service.

Yearly Notice For the purpose of delivering a yearly privacy policy notice.

Tags

The tags for the CTPA in the text as follows. They are the members of Tag_{ctpa} and are listed in Table 7.2. The tags are descriptively named and so are self-explanatory.

7.3.2 Comparing Disclosure Rules

The TiVo privacy policy includes a section of definitions (section 1) followed by eight more sections. Section 2 describes how TiVo uses subscriber information. Section 3 describes how TiVo discloses user information. Let us consider a quote from paragraph 3.4 regarding how TiVo discloses subscriber information to companies related to TiVo.

> 3.4 The "Corporate Family." TiVo may share some or all of your User Information with any parent company, subsidiaries, joint ventures, or other companies under a common control (collectively "Affiliates"). In such event, TiVo will

require its Affiliates to honor this Privacy Policy. If another company acquires TiVo, or acquires assets of TiVo that comprise or include your User Information, that company will possess the User Information collected by TiVo and it will assume the rights and obligations regarding your User Information as described in this Privacy Policy.

Paragraph 3.4 describes that TiVo may share some User Information with companies which are affiliated with TiVo, including those which are not directly related to TiVo's service. The disclosure of User Information in such cases is not predicated on the receipt of consent from the subscriber. A command which implements the disclosure action is as follows:

```
 CMD  DiscloseFamily3.4(a, s, r, P, f, f', msg)
  if  f.user-information = true
 and  corporateFamily in (r, a)
 and  local in (a, f)
 and  disclose $\textbf{in}_a$ P
then  insert local in (r, f)
 and  return true
else  return false
```

In order to evaluate how DiscloseFamily3.4 compares to the CTPA's rules on disclosure in 551(c) we must first compare the definitions of "personally identifiable information" in CTPA and "user information" in TiVo's policy. The definition in CTPA is in 551(a)(2)(A):

(2) For purposes of this section, other than subsection (h) of this section —

(A) the term "personally identifiable information" does not include any record of aggregate data which does not identify particular persons;

Since (a)(2)(A) defines "personally identifiable information" by indicating what is not included in it (*i.e.,* aggregate data), the implication is that any information which is personally identifiable is included. The definition in TiVo's policy is in 1.1. Due to its length, we do include a direct quote here, but refer the reader to Appendix B.5 on page 346. The policy defines "User Information" as a generic term for many forms of information, including account information, contact information, and service information. Account and contact information are personally identifiable since they include addresses, phone numbers, and TiVo device service number. Service information includes technical information about the TiVo device and its software status and so is likely non-personally identifiable. Since “User Information” may include personally identifiable information in the form of account or contact information, we may safely assume that when 3.4 mentions disclosure of user information it may include personally identifiable information under the CTPA definition.

Let us now consider two scenarios for the comparison of the CTPA and the TiVo command. For convenience let us denote the CTPA policy ϕ_{ctpa}.

Example 7.3.1 (No consent)

Let us consider the case of a subscriber to TiVo who has not granted consent for TiVo to send information to TiVo's corporate family. Let us compare whether the CTPA strongly or weakly licenses the TiVo policy.

The agent set for the scenario is Agent = {tivo, sub, fam} where Roles(tivo) = {cableOperator}, Roles(sub) = {subscriber}, and Roles(fam) = {}. The tags for the agents are all set to false.

The object set is $\mathit{Object} = \{o\}$. The following tags are set to true: o.user-information, o.personally-identifiable-information, o.subject-subscriber. All other tags for o are set to false.

The initial state for the matrix is as in m_1:

$$m_1 = \begin{array}{|c||c|c|c|c|}\hline & \text{tivo} & \text{sub} & \text{fam} & o \\ \hline\hline \text{tivo} & & & & \text{local} \\ \hline \text{sub} & & & & \\ \hline \text{fam} & \text{corporateFamily} & & & \\ \hline \end{array}$$

The log is initially empty $l_1 = \epsilon$.

Thee initial knowledge state is then $s_1 = (A_1, O_1, m_1, l_1)$ where A_1 = {tivo, sub, fam} and $O_1 = \{o\}$. The purpose set for the action is P = {disclose}. The argument list for the action is g_1 = (tivo, sub, fam, P, o, null, ϵ).

The transition performed by the TiVo command is then $s_1 \xrightarrow{\text{DiscloseFamily3.4}(g_1)} s_2$ where $s_2 = (A_1, O_1, m_2, l_1)$ where m_2 is as follows.

$m_2 =$

	tivo	sub	fam	o
tivo				local
sub				
fam	corporateFamily			local

The effect of the command is the granting of "local" for fam on o. Examining ϕ_{ctpa} we find that there is no command which can perform the transition $s_1 \longrightarrow s_2$ since tivo does not have consent from sub and no other circumstances are indicated. Therefore $\phi_{ctpa} \not\models^*_{(s_1,g_1)}$ DiscloseFamily3.4. A close examination shows that the property holds for all argument values and therefore $\phi_{ctpa} \not\models^*_{(s_1)}$ DiscloseFamily3.4. The meaning of the result is that the TiVo command does not comply with the CTPA policy for the state s_1. $\square$

Let us consider an alternative scenario for ϕ_{ctpa} and DiscloseFamily3.4.

Example 7.3.2 (Consent)

Let us consider the opposite case from Example 7.3.1 when a subscriber to TiVo has granted written consent for TiVo to send information to TiVo's corporate family. Let us compare whether the CTPA strongly or weakly licenses the TiVo policy.

The agent set *Agent* and object set *Object* are identical to the sets in Example 7.3.1. The tags for the agents are all set to false. The following tags for o are set to true: o.user-information, o.personally-identifiable-information, o.subject-subscriber. All other tags for o are set to false.

The initial state for the matrix is as in m_3:

$m_3 =$

	tivo	sub	fam	o
tivo		consent, written		local
sub				
fam	corporateFamily			

The log is initially empty $l_3 = \epsilon$.

Thee initial knowledge state is then $s_3 = (A_3, O_3, m_3, l_3)$ where $A_3 = \{\text{tivo, sub, fam}\}$ and $O_3 = \{o\}$. The purpose set for the action is $P = \{\text{disclose}\}$. The argument list for the action is $g_3 = (\text{tivo, sub, fam, } P\text{, } o\text{, null, } \epsilon)$.

The transition performed by the TiVo command is then $s_3 \xrightarrow{\text{DiscloseFamily3.4}(g_3)} s_4$ where $s_4 = (A_3, O_3, m_4, l_3)$ where m_4 is as follows.

$m_4 =$

	tivo	sub	fam	o
tivo		consent, written		local
sub				
fam	corporateFamily			local

The effect of the command is the granting of "local" for fam on o. Examining ϕ_{ctpa} we find that command Disclose551c1 strongly licenses the transition since tivo has written consent from sub. Therefore $\phi_{ctpa} \models_{(s_3,g_3)}$ DiscloseFamily3.4. A close examination shows that the property holds for all argument values and therefore $\phi_{ctpa} \models_{(s_1)}$ DiscloseFamily3.4. The meaning of the result is that the TiVo command complies with the CTPA policy for the state s_3. □

Examples 7.3.1 and 7.3.2 show two scenarios for comparing the CTPA policy and the TiVo policy. Example 7.3.1 shows an example where TiVo's policy is not compliant with the CTPA, however personal communication with several TiVo subscribers indicates that all subscribers are required to grant consent for disclosures before they can receive TiVo service. Since consent for disclosures is mandatory for all subscribers, the scenario in Example 7.3.1 is likely only theoretical.

7.4 Conclusion

The case studies in this chapter illustrate the flexibility of the Privacy Commands language in modeling five diverse privacy policies: the 2000 Privacy Rule, the 2003 Privacy Rule, the Insurance Council of Australia Privacy Code, the Cable TV Privacy Act, and TiVo Corporation's privacy policy. We have illustrated the methodology of translation from policy text to Privacy APIs described in Section 4.2 through several involved examples.

We have also illustrated the usefulness of the licensing relations developed in Section 5.5 by showing how they can be used to characterize the permissiveness of different policies. During the course of the examples we raise three difficulties in translation from text to Privacy Commands which we solve through various methods.

```
AGENT ambulance = 1;
AGENT patient = 2;
OBJECT o1 = 3;
#define MAXAGENT 2
#define MAXOBJECT 3
OBJECT objects[MAXOBJECT];
PURPOSE P[MAXPURPOSE];
PURPOSE parent[MAXPURPOSE];

(Commands and Constraints go here)

init{
mroles[ambulance].healthCareProvider = 1;
mroles[patient].individual = 1;
P[emergency] = 1;
P[treatment] = 1;
P[use] = 1;
m.mat[ambulance].objects[o1].local = 1;
m.mat[patient].objects[o1].local = 1;

a = ambulance;
s = patient;
r = ambulance;
f = o1;
topObj = 3;
f_new = topObj;

do
:: (Select Commands)
od;

result = (m.mat[ambulance].objects[patient].treatment == 1 &&
mtags[ambulance].will_obtain_consent_asap == 0 &&
m.mat[ambulance].objects[patient].consent == 0);
}
```

Figure 7.1: Initial state and invariant for ambulance query

Chapter 8

Conclusion

In summary, this dissertation has developed the formal language called Privacy Commands which we use to model legal privacy policies. We use the language to translate natural language legal privacy policies into commands and constraints which mimic the logic and actions of the text on a paragraph by paragraph basis. The Privacy Commands language is characterized by the use of commands and constraints as policy atoms and a robust and flexible means for modeling references between different policy paragraphs.

Our goal in developing Privacy Commands is to aid the understanding and comparison of legislative privacy regulations by leveraging the techniques of computer science access control and privacy policies. There are three aspects to our work: the Privacy Commands formal language for describing legal privacy requirements, a methodology for using automated model checker SPIN to examine properties of policies and a proof of the correctness of the translation, and policy oriented relations that allow the examination and comparison of different policies in various circumstances. Together these aspects yield a foundation for understanding legal policies and a beginning stage for tools to help in their analysis and enforcement.

The foundational models described in Chapter 3 motivates the requirements for analysis of privacy policies and how the Formal Privacy framework can address them. Chapter 4 develops Auditable Privacy Systems, a Formal Privacy model which is the basis for the Privacy Commands formal language. Chapter 5 develops the Privacy Commands language formally including its syntax and semantics. Section 5.5 develops policy relations based on Privacy Commands including strong licensing ($\models$) and weak licensing ($\models^*$). Chapter 6 develops a methodology for translating Privacy APIs into Promela for automated analysis and shows a correctness proof for the translation.

Chapter 7 uses the structures developed in the previous chapters and shows their usefulness in a series of in dept case studies examining five diverse privacy policies. The policies are interesting in their own right, but also serve as illustrative examples for how we exercise the functionality of commands and constraints and how we can use strong and weak licensing to compare and contrast complex policies under various scenarios.

Our contribution to the computer science discipline in general is the design and analysis of a formalism which is well tuned to the analysis of certain aspects of privacy legislation. There is considerable work left to be done in the vein, including proper ways for modeling the legal formulations found in many privacy laws which are beyond the scope of this work. Examples of such formulations include conditional obligations, temporal obligations and rights, contingencies, and interaction requirements. We hope that this work will be a foundation for future studies and research in using computational methods to better understand, model, and enforce privacy legislation.

Appendix A

Supplementary Promela Code

The translation between Privacy Commands and Promela in Chapter 6 uses a number of helper functions and supplemental code to produce a complete and runnable SPIN model. This appendix contains the source code and algorithms that are necessary along with a brief explanation for its purpose.

An important auxiliary function used throughout this appendix is the array copying inline function shown below.

```
inline arrayCopy(ac_from, ac_to, ac_size)
{
    ac_upto = 0;
    do
    :: ac_upto < ac_size -> to[ac_upto] = from[ac_upto]; ac_upto++;
    :: ac_upto == ac_size -> break;
    od;
}
```

The function copies the first `ac_size` elements of array `ac_from` to array `ac_to` and uses the temporary variable `ac_upto` for storage. Since inline functions do not define their own scopes, any function which uses the `arrayCopy` function must define a temporary variable called `ac_upto` or else a syntax error will result.

Since `typedef` objects can not be copied with a single assignment statement, we need to create specialized copying functions for the rights matrix and tag records which behave similarly to the above code.

```
inline objectsCopy(oc_from, oc_to, oc_size)
{
    oc_upto = 0;
    do
    :: oc_upto < oc_size ->
       to[oc_upto].right1 = from[oc_upto].right1;
       to[oc_upto].right2 = from[oc_upto].right2;
       to[oc_upto]....    = from[oc_upto]....;
       oc_upto++;
    :: oc_upto == oc_size -> break;
    od;
}
inline matrixCopy(mc_from, mc_to, mc_dim1, mc_dim2)
{
   mc_upto = 0;
   do
   :: mc_upto < mc_dim1 ->
      objectsCopy(mc_from.mat[mc_upto].objects,
         mc_to.mat[mc_upto].objects, mc_dim2);
      mc_upto++;
   :: mc_upto == mc_size -> break;
   od;
}
inline tagsCopy(tc_from, tc_to, tc_size)
{
   tc_upto = 0;
   do
   :: tc_upto < tc_size ->
```

```
        tc_to[tc_upto].tag1 = tc_from[tc_upto].tag1;
        tc_to[tc_upto].tag2 = tc_from[tc_upto].tag2;
        tc_to[tc_upto]....  = tc_from[tc_upto]....
        tc_upto++;
    :: tc_upto == tc_size -> break;
    od;
}
```

The sections in this appendix are as follows. Section A.1 contains auxiliary code used for the managing of purpose sets. Section A.2 contains the Promela structure for overloaded constraints. Section A.3 contains the Promela code for taking and restoring a state snapshot.

A.1 Purposes Code

The code in this section implements the purposes hierarchy described in Section 6.2.2. For efficiency purposes, a few of the simple functions are declared as `inline` functions rather than `proctype` processes. Inline functions behave similarly to C preprocessor macros in that their bodies are cut-and-pasted inside the statements that invoke them. Additionally, since recursion is used in some of the processes, they do not have the same `do/od` structure of the command and constraint processes. Instead, processes are started (using the keyword `run`) as needed by the execution and stop upon completion of their tasks. This makes the code a bit more efficient and also will aid in keeping below the 255 process limit for the SPIN model checker.

We describe the functionality of each function or process and present lemmas where appropriate relating the code to the formal representation of the purpose partial order.

The function `isParentOf` checks if `pa` is the parent of `pb`.

```
inline isParentOf(p_pa, p_pb, p_result)
{
    if
    :: parent[p_pb] == p_pa -> p_result = true;
5       :: else -> p_result = false;
6       fi;
7    }
```

Lemma A.1.1 *The inline function* `isParentOf` *stores true in* `p_result` *if and only if the purpose represented by* `p_pa` *is the parent of* `p_pb` *in the purpose partial order.*

Proof: By the definition of the array `parent` above in Section 6.4.1, an entry `parent[c] = p` if and only if the purpose represented by `p` is the parent of the purpose represented by `c`. By inspection, line 4 of the function stores true in `p_result` if `parent[p_pb]` = `p_pa` as per the definition. Otherwise the `else` option is selected and false is stored in `p_result`. □

The function `isChildOf` checks if `pa` is a child of `pb`.

```
inline isChildOf(c_pa, c_pb, c_result)
{
   isParentOf(c_pb, c_pa, c_result);
}
```

Lemma A.1.2 *The inline function* `isChildOf` *stores true in* `p_result` *if and only if the purpose represented by* `p_pa` *is the child of* `p_pb` *in the purpose partial order.*

Proof: By Lemma A.1.1, `isParentOf` stores true in its result if `c_pb` is the parent of `c_pa`. If it is, then by definition `c_pa` is a child of `c_pb`. Otherwise it is not. □

The process `isAncestorOf` checks if `pa` is an ancestor of `pb`.

```
proctype isAncestorOf()
{
   PURPOSE pa; PURPOSE pb; bool isParentResult;

   isAncestorOf_request_chan?hier_request(pa, pb) ->
   isParentOf(pa, pb, isParentResult);
7       if
8       :: pb == pa || isParentResult == true ->
9          isAncestorOf_response_chan!hier_response(true);
10      :: parent[pb] == -1 ->
11         isAncestorOf_response_chan!hier_response(false);
12      :: else -> run isAncestorOf();
13         isAncestorOf_request_chan!hier_request(pa, parent[pb]);
14      fi;
15   }
```

Lemma A.1.3 *If the process* `isAncestorOf` *receives a request of type* `hier_request` *with purpose parameters* `pa` *and* `pb`*, it returns true over its response channel if and only if the purpose represented by* `pa` *is an ancestor of the purpose represented by* `pb` *(i.e., $pb \in \mathsf{ancestors}(pa)$).*

Proof: As defined above, ancestorsis the transitive closure of the parent relation. The process `isAncestorOf` performs a recursive search maintaining the invariants `pa` $\neq -1$ and `pb` $\neq -1$. First, the inline function `isParentOf` determines if `pa` is `pb`'s parent (as per Lemma A.1.1 (line 6). The `if/fi` structure then examines if `pb == pa` or if `pa` is the parent of `pb`. If either is the case then `pa` is `pb`'s ancestor and the result sent back is true (line 9). If `pb` $\neq$ `pa` and `pb` is a root (`parent[pb] == -1` as per the definition of `parent`) then `pa` can not be `pb`'s ancestor and the result sent back is false (line 11). If neither option is satisfied then the `else` option runs a new instance of the process and sends it a request with `pa` and `parent[pb]`, querying if `pa` is the ancestor of `pb`'s parent (lines 13–13). The newly created process correctly determines if `pa` is `parent[pb]`'s ancestor by the inductive assumption and sends back the result over the response channel. Since `isAncestorOf` processes only respond on `isAncestorOf_response_chan` if they have a definite response (`pa` is or is not `pb`'s ancestor), the requesting process sees only the correct answer over it. □

The process `isDescedantOf` checks if `pa` is a descendant of `pb`.

```
1    proctype isDescendantOf()
```

```
2   {
3      int pa; int pb; bool result;
4
5      isDescendantOf_request_chan?hier_request(pa, pb) ->
6      run isAncestorOf();
7      isAncestorOf_request_chan!hier_request(pb, pa);
8      isAncestorOf_response_chan?hier_response(result);
9      isDescendantOf_response_chan!hier_response(result);
10  }
```

Lemma A.1.4 *If the process* `isDescendantOf` *receives a request of type* `hier_request` *with purpose parameters* `pa` *and* `pb`*, it returns true over its response channel if and only if the purpose represented by* `pa` *is a descendant of the purpose represented by* `pb` *(i.e.,* $pb \in \textsf{descendants}(pa)$*).*

Proof: As per the definition of `descendants`, in order to determine if `pa` is a descendant of `pb` it is sufficient to show that `pb` is an ancestor of `pa`. Lines 6–8 run an `isAncestorOf` process which determines that correctly as per Lemma A.1.3. The result is sent back over the response channel (line 9). □

The function `isPermittedBy` checks if pa in_a $\{pb\}$. It does so be checking if `pa` is an ancestor of `pb`.

```
inline isPermittedBy (pb_pa, pb_pb, pb_result)
{
   if
   :: pb_pa == pb_pb -> pb_result = true;
   :: else -> isAncestorOf_request_chan!hier_request(pb_pa, pb_pb);
      isAncestorOf_response_chan!hier_response(pb_result);
   fi;
}
```

Lemma A.1.5 *The inline function* `isPermittedBy` *stores true in* `pb_result` *if and only if the purpose represented by* `pb_pa` *includes the purpose represented by* `pb_pb` *using permitted semantics.*

Proof: By definition, pa in_a $\{pb\}$ is true if and only if $pa \in \mathsf{ancestors}(pb)$. As per Lemma A.1.3, the process `isAncestorOf` returns true (which is then stored in `pb_result` on line 6) if and only if `pb_pa` is an ancestor of `pb_pb`. □

The function `isForbiddenBy` checks if pa in_f $\{pb\}$. It does so be checking if `pa` is an ancestor or a descendant of `pb`.

```
1   inline isForbiddenBy (fb_pa, fb_pb, fb_result)
2   {
3      /* Check if pa is a descendant of pb*/
4      isDescendantOf_request_chan!hier_request(fb_pa, fb_pb);
5      isDescendantOf_response_chan?hier_response(fb_result);
6
7      /* If not, check if pa is an ancestor of pb*/
8      if
9      :: fb_result == false ->
10        isAncestorOf_request_chan!hier_request(fb_pa, fb_pb);
11        isAncestorOf_response_chan?hier_response(fb_result);
12     :: else -> skip;
13     fi;
14  }
```

Lemma A.1.6 *The inline function* `isForbiddenBy` *stores true in* `fb_result` *if and only if the purpose represented by* `fb_pa` *includes the purpose represented by* `fb_pb` *using forbidden semantics.*

Proof: By definition, pa in_f $\{pb\}$ is true if and only if $pa \in \{\mathsf{ancestors}(pb) \cup \mathsf{descendants}(pb)\}$. s per Lemma A.1.4, the process `isDescendantOf` returns true (which is then stored in `pb_result` on line 5) if and only if `fb_pa` is a descendant of `fb_pb`. Lines

8–12 are an `if/fi` structure which selects the first option (lines 9–11) if `isDescendantOf` sends back false. It sends a message to the process `isAncestorOf` which sends back true if and only if `fb_pa` is an ancestor of `fb_pb` as per Lemma A.1.3. The final result is stored in `fb_result` which is true if either process returns true and false if both processes return false. □

The process `isPermittedByA` checks if pa in$_a$ P where P is the for representation of `P`, global PURPOSE array defined above in Table 6.4. It does so be checking if pa in$_a$ $\{pb\}$ for any $pb \in P$.

```
1   proctype isPermittedByA()
2   {
3      PURPOSE upto = 0; bool result; PURPOSE pa;
4
5      isPermittedByA_request_chan?purpose_request(pa) ->
6      do
7      :: P[upto] == 1 && upto < MAXPURPOSE ->
8         isPermittedBy(pa, upto, result);
9         if
10        :: result == true -> break;
11        fi;
12     :: upto == MAXPURPOSE -> result = false; break;
13     :: else -> upto++;
14     od;
15     isPermittedByA_response_chan!purpose_response(result);
16  }
```

Lemma A.1.7 *(Permitted By) For a purpose p and purpose set P under a partial order properly defined in Purpose the process* `isPermittedByA` *sends true over its response channel if and only if* $\exists pb \in P \;.\; pa \; in_a \{pb\}$.

Proof: By definition, pa in$_a$ P is true if any member of P is included in pa with permitted semantics. The process `isPermittedByA` therefore iterates over all the members of `P` (which corresponds to P by definition) and uses `isPermittedBy` to determine if any member is permitted by `pa` (as per Lemma A.1.5).

The main loop of the process (lines 6–14) iterate over the array `P` using the index `upto`. At each iteration, the first option (lines 7–11) is selected if the purpose represented by `upto` is in `P` (`P[upto] == 1`) and the index of `upto` is still below the maximum value for `PURPOSE`. If the option's guards are satisfied (*i.e.,* the purpose `upto` is a valid purpose index and it is present in `P`), the inline function `isPermittedBy` checks if `pa` includes `upto`. As per Lemma A.1.5, the function stores true in `result` iff `pa` includes `upto` using permitted semantics. If the result is true, the loop quits with a `break` (lines 9–11).

If the first option's guards are not satisfied, the second option's (mutually exclusive) guard may be true if `upto` no longer refers to a valid purpose (`upto == MAXPURPOSE`) (line 12). If the second option is satisfied, the result is false and the loop quits with a `break`.

The `else` option (line 13) is selected if neither the first or second options are satisfied. It increments the `upto` variable to the next `PURPOSE`.

The loop terminates eventually since at every iteration either `upto` is incremented or a `break` statement is executed. The final result is sent back over the response channel. The first option will check $pa \text{ in}_a \{pb\}$ for all $pb \in P$ and return true if it is satisfied. The second option will catch when all $pb \in P$ have been examined and return false, indicating that $pa \text{ in}_a P$ is false. □

The process `isForbiddenByA` checks if $pa \text{ in}_f P$ where P is the for representation of `P`, global PURPOSE array defined above in Table 6.4. It does so be checking if $pa \text{ in}_f \{pb\}$ for any $pb \in P$.

```
1   proctype isForbiddenByA()
2   {
3     int upto = 0; bool result; PURPOSE pa;
4
5      isForbiddenByA_request_chan?purpose_request(pa) ->
6      do
7      :: P[upto] == 1 && upto < MAXPURPOSE->
```

```
        isForbiddenBy(pa, upto, result);
        if
        :: result == true -> break;
        fi
    :: upto == MAXPURPOSE -> result = false;
    :: else -> upto++;
    od;
    isForbiddenByA_response_chan!purpose_response(result);
}
```

Lemma A.1.8 *(Forbidden By) For a purpose p and purpose set P under a partial order properly defined in Purpose the process* `isForbiddenByA` *sends true over its response channel if and only if $\exists pb \in P$. pa $in_f\{pb\}$.*

Proof: The proof is similar to Lemma A.1.7 using the inline function `isForbiddenBy` and Lemma A.1.6. □

A.2 Overloaded Constraints Structure

Since the judgment derived from the boolean values differs depending on whether it is being invoked as part of a constraint search or an explicit invocation, the code does not operate on judgments. Above in Section 6.2.4 we argued that the truth table in Table 6.6 combined the results of overloaded constraints as per Tables 5.11 and 5.9 correctly. Table 6.6 ignored the scope because it is not needed for the combination, a decision that we justify in the following lemmas.

First we consider the constraint search case. We show the following lemma:

Lemma A.2.1 *(Combining for pre-command search) Let c_1 and c_2 be overloaded constraints. Derive judgments j_1 from c_1's boolean results and j_2 from c_2's boolean results using Table 5.10. Combine j_1 and j_2 using Table 5.11 and let j be the resulting judgment. Combine c_1 and c_2's boolean results using Table 6.6 and derive judgment j' from Table 5.10. The following property then holds: $j = j'$.*

Proof: We consider all possible combinations of the results from the two constraints c_1 and c_2. Let the boolean results be b_{scp}, b_{st}, and b_r as in Table 5.8 and the judgment combination as per Table 5.10. The derived judgments are shown in Table A.1. For brevity we use T for True, F for False, A for allow, I for Ignore (Allow), and F for Forbid. Note that since the scopes of overloaded constraints must be the same, the values for b_{scp} must be identical and we therefore only show cases where they agree.

□

Next let us consider the explicit command invocation case. We show the following lemma:

Lemma A.2.2 *(Combining for invocation) Let c_1 and c_2 be overloaded constraints. Derive judgments j_1 from c_1's boolean results and j_2 from c_2's boolean results using Table 5.8. Combine j_1 and j_2 using Table 5.9 and let j be the resulting judgment. Combine c_1 and c_2's boolean results using Table 6.6 and derive judgment j' from Table 5.8. The following property then holds: $j = j'$.*

Proof: We consider all possible combinations of the results from the two constraints c_1 and c_2. Let the boolean results be b_{scp}, b_{st}, and b_r as in Table 5.8. The derived judgments are shown in Table A.2. For brevity we use T for True, F for False, A for allow, F for Forbid, DCA for Don't Care/Allow, DCF for Don't Care/Forbid. As in Table A.1, since the scopes of overloaded constraints must be the same, the values for b_{scp} must be identical and we therefore only show cases where they agree.

□

By combining the results from Lemma A.2.1 for pre-command execution and Lemma A.2.2 for invocation, we directly arrive at the following general lemma:

Lemma A.2.3 *(Boolean result combination) Combining boolean results from c_1 and c_2 using Table 6.6 and then obtaining a judgment is equivalent to deriving judgments from the boolean results using Table 5.10 for pre-command search and Table 5.8 for invocation and then combining the judgments using Table 5.11 for pre-command search and Table 5.9 for invocation.*

The overloaded constraint code uses the following inline function combineOverload to combine the boolean results from the different constraints. The code is identical to Figure 6.5 substituting co_st_r for such_that_r, co_st_n for such_that_n, co_r_r for regular_r, and co_r_n for regular_n due to technical language restrictions of Promela. As shown in Lemma 6.2.1, the logic produces the same result as in Table 6.6, storing the results in co_st_r and co_r_r.

```
inline combineOverload (co_st_r, co_st_n, co_r_r, co_r_n)
{
       if
       :: co_st_r == co_st_n -> co_r_r = (co_r_r || co_r_n);
       :: co_st_r == true  && co_st_n == false ->
          co_r_r = co_r_r;
       :: co_st_r == false && co_st_n == true  ->
          co_r_r = co_r_n;
       fi;
       co_st_r = co_st_r || co_st_n;
}
```

As discussed in Section 6.2.4, the structure for overloaded constraints differs slightly from the structure of normal constraints. We present here the full formulation for an overloaded constraint due to its length and similarity to the structure in Figure 6.4.

```
active proctype CST2() {
   bool scope_r; bool scope; bool such_that_r; bool such_that_n;
   bool such_that; bool regular_r; bool regular_n; bool regular;
   bool result; bool temp; CMD command; JUDGMENT j;
   do
   ::  CST2_request_chan?constraint_request(command) ->
        scope = true; such_that_r = true; such_that_n = true;
        regular_r = true; regular_n = true; result = true; temp = true;
        if
10          L1: (scope checks go here)
11          :: else -> scope = false;
12          fi;
13          L2a: (first such that guards here)
14          such_that_r = result; result = true;
15          L3a: (first regular guards here)
16          regular_r = result;
17
18          L2b: (second such that guards here)
19          such_that_n = result; result = true;
20          L3a: (second regular guards here)
21          regular_n = result; result = true;
22          combineOverload(such_that_r, such_that_n, regular_r, regular_n);
23
24          CST2_response_chan!constraint_response(scope_r, such_that_r,
25               regular_r);
26      od;
27  }
```

The structure of the overloaded constraint CST2 differs from the non-overloaded constraint CST1 in Figure 6.4 in that it executes the logic contained in multiple constraints and combines their results using the truth table in Table 6.6. Lines 2–4 declare the variables for the execution including the extra variables needed for storing the results of the different individual constraints. Note that since overloaded constraints share the same scope there is no need for a duplicate scope variable, but since constraint invocations may overwrite the variable `scope`, we keep a result `scope_r` which is sent back at the completion of execution. The scope determination on lines 9–12 is made as usual with the scope code inserted at label L1. Lines 13–16 execute the first constraint in the overloaded set and store the resulting values in `such_that_r` and `regular_r`. Labels L2a and L3a indicate the location where the *such that* and regular guards code are inserted. Lines 18–21 then execute the second overloaded constraint and store the resulting values in `such_that_n` and `regular_n`. Line 22–36 invokes the inline function `combineOverload` shown above to combine the results from the two constraints as per Table 6.6. The final resulting scope, such that, and regular variable values are then sent back along the constraint's response channel on lines 24–25.

The above structure shows the format for an process implementing two overloaded constraints. For each constraint greater than 2, lines 18–22 would be inserted for the extra constraint between lines 22 and 24. The response on lines 24–25 would remain unchanged at the end of all of the guard logic.

We use the overloaded constraint code above to implement multiple constraints in the formal model as a single large Promela process. As part of our correctness argument, we show that the code properly imitates multiple overloaded constraints and combines their results as per Table 6.6.

Lemma A.2.4 *(Overloaded Constraints) For overloaded constraints $c_1, \ldots, c_n$, the corresponding process constructed as per process* `CST2` *above sends boolean results* `scope_r`*,* `such_that_r`*,* `regular_r` *over the response channel that coincide with the results of invoking $c_1, \ldots, c_n$ individually and combining their results as per Table 6.6.*

Proof: Let us denote the set of overloaded constraints C. We then argue the proof using induction on the number of overloaded constraints that are simulated by the process: $|C|$. Let us denote the Promela equivalent for a constraint c_n as `CSTn` as per Section 6.2.4. Let us denote the boolean value b_{st} for c_n as b^i_{st} and b_r for c_n as b^i_r. Let us denote the execution of lines 18–2 for c_n as *the execution of* c_n. We maintain the following invariants which comprise our induction hypothesis.

Invariant 1: After the execution of c_n, `scope_r` contains `true` if and only if command `e` is in scope for c_n.

Invariant 2: After the execution of c_n:

`such_that_r = true` and `regular_r = true`	iff	$\exists c_i \in C \; . \; (b^i_{st} = true, b^i_r = true)$
`such_that_r = true` and `regular_r = false`	iff	$\exists c_i \in C \; . \; (b^i_{st} = true, b^i_r = false)$ and $\nexists c_j \in C \; . \; (b^j_{st} = true,$ $b^i_r = true)$
`such_that_r = false` and `regular_r = true`	iff	$\exists c_i \in C \; . \; (b^i_{st} = false, b^i_r = true)$ and $\nexists c_j \in C \; . \; b^j_{st} = true$
`such_that_r = false` and `regular_r = false`	iff	$\forall c_i \in C (b^i_{st} = false, b^i_r = false)$

Note that the logic in Invariant 2 is equivalent to the repeated application of the truth table in Table 6.6 for $n > 2$ constraints which can be shown by direct enumeration of the cases. If the above invariants are maintained, at the end of execution the resulting values sent back over the response channel correspond to the results of Table 6.6 since the scope is calculated correctly for the constraints and the combination maintains the table's logic and the overloaded constraint logic from Section 5.4.

The induction hypothesis is thus as follows: **Induction Hypothesis:** After completing the execution for n, both Invariants 1 and 2 hold.

Base Case $|C| = 1$ The base case is when there is only one constraint. As shown in Lemma 6.4.7, lines 9–16 of the code in `CST2` above store the scope value in `scope_r`, the *such that* guards result in `such_that_r`, and the regular guards result in `regular_r`. Invariant 1 is established since the scope is stored in `scope_r`. Invariant 2 is established because for $|C| = 1$, the properties are true trivially since c_1 is the only constraint and its results are stored directly in `such_that_r` and `regular_r`.

Step $|C| = n$ The induction hypothesis for this case presumes that after $n-1$ constraints, both invariants are true. Since `scope_r` is not modified by any of the code for c_n, the invariant remains true trivially.

Lines 18–21 for c_n evaluate its *such that* and regular guards and store the results in `such_that_n` and `regular_n` are is shown in Lemma 6.4.7. Line 22 then combines the results for c_n with the tallied results for the other $c_i \; . \; i < n$. The boolean results are combined using the inline function `combineOverload`. There are four cases to consider:

Case 1: `such_that_n = true`, `regular_n = true` Lines 3–9 of the function examine the values of `such_that_r` and `such_that_n`. Since `such_that_n = true`, only the first and third options in the `if/fi` structure are possible.

If `such_that_r` is true, then the first option on line 4 is selected. Since `regular_n = true`, a logical OR with `regular_r` always yields `true`, so `regular_r` gets `true`.

If `such_that_r = false`, option 3 of the `if/fi` structure is selected, storing `true` in `regular_r`.

Line 10 stores the logical OR of `such_that_r` and `such_that_n` in `such_that_r` which is always `true` since `such_that_n = true`. The result is that after executing the function `such_that_r = regular_r = true` which satisfied Invariant 2 for n since $\exists c_i, i \leq n \;.\; b^i_{st} = true, b^i_r = true$ for $i = n$.

Case 2: `such_that_n = true`, `regular_n = false` As in the previous case, options 1 and 3 of the `if/fi` structure are possible.

If `such_that_r = true` then the first option is selected. It stores the OR of `regular_r` and `regular_n = false` in `regular_r`, leaving `regular_r` unchanged. As in the previous case, `such_that_r` is assigned `true` on line 10 since `such_that_n = true` and `such_that_r = true`. If the old value of `regular_r` was `true`, by the induction hypothesis $\exists c_i \in C, i < n \;.\; (b^i_{st} = true, b^i_r = true)$ and the new values are unchanged: `such_that_r = true`, `regular_r = true` as noted, maintaining Invariant 2 for n since c_i satisfies the first line of the invariant. If the old value of `regular_r` was `false`, the new values are `such_that_r = true`, `regular_r = false`. By the induction hypothesis $\nexists c_i \in C, i < n \;.\; (b^i_{st} = true, b^i_r = true)$. Now we have that $\exists c_i \in C, i \leq n \;.\; (b^i_{st} = true, b^i_r = false)$ for $i = n$ which maintains Invariant 2 for n.

If `such_that_r = false` then the third option is selected. The new value for `regular_r` is then `false` and `such_that_r` is assigned `true`. By the induction hypothesis, if the old value of `regular_r` was `true` then $\nexists c_i \in C, i < n \;.\; b^i_{st} = true$. Similarly, if the old value of `regular_r` was `false` then $\forall c_i \in C, i < n \;.\; (b^i_{st} = false, b^i_r = false)$. The new values are now `such_that_r = true`, `regular_r = false` which fulfills the second option of Invariant 2 for n since $\exists c_i \in C, i \leq n \;.\; (b^i_{st} = true, b^i_r = false)$ for $i = n$ and in either case for the old value of `regular_r`, $\nexists c_j \in C, j \leq n \ . \ (b^i_{st} = \mathit{true}, b^i_r = \mathit{true})$.

Case 3: `such_that_n = false`**,** `regular_n = true` Options 1 and 2 are possible in the `if/fi` structure of `combineOverload`.

If `such_that_r = true`, the second option of the `if/fi` is selected. The value for `regular_r` is then unchanged. The value for `such_that_r = true` is also unchanged on line 10. By the induction hypothesis, since `such_that_r = true`, if `regular_r = true` $\exists c_i \in C, i < n \ . \ (b^i_{st} = \mathit{true} b^i_r = \mathit{true})$ and since c_n yields $(b^n_{st} = \mathit{false}, b^i_r = \mathit{true})$, it remains unchanged for n as per the first option of Invariant 2. If `regular_r = false` $\exists c_i \in C, i < n \ . \ (b^i_{st} = \mathit{true}, b^i_r = \mathit{false})$ and $\nexists c_i \in C, i < n \ . \ (b^i_{st} = \mathit{true}, b^i_r = \mathit{true})$ which is unchanged by c_n and therefore the second option in Invariant 2 is true for n.

If `such_that_r = false`, the first option of the `if/fi` is selected. If the old value for `regular_r = true`, its new value remains `true`. If its old value is `regular_r = false`, its new value is `true`. The new value for `such_that_r` is unchanged at `false`. By the induction hypothesis, since `such_that_r = false`, if `regular_r = true` $\exists c_i \in C, i < n \ . \ (b^i_{st} = \mathit{false}, b^i_r = \mathit{true})$ and $\nexists c_j \in C, j < n \ . \ b^j_{st} = \mathit{true}$. Now since the new values are `such_that_r = false`, `regular_r = true` the third option for Invariant 2 is still valid for n since $b^n_{st} = \mathit{false}$. If the old value for `regular_r = false`, by the induction hypothesis $\forall c_i \in C, i < n (b^i_{st} = \mathit{false}, b^i_r = \mathit{false})$. Now since the new values are `such_that_r = false`, `regular_r = true` which valid under the third option for Invariant 2 for n since $\exists c_i \in C, i \leq n \ . \ (b^i_{st} = \mathit{false}, b^i_r = \mathit{true})$ for $i = n$ and $\nexists c_j \in C, i \leq n \ . \ b^i_{st} = \mathit{true}$ by the induction hypothesis.

Case 4: `such_that_n = false`**,** `regular_n = false` As in the previous case, options 1 and 2 are possible in the `if/fi` structure of `combineOverload`.

If `such_that_r = true`, the same argument for Case 3 applies since $b^n_{st} = \mathit{false}$ and $b^n_r = \mathit{false}$.

If `such_that_r = false`, the first option of the `if/fi` is selected. The new value for `regular_r` is unchanged from the old value. The new value for `such_that_r` is unchanged at `false`. By the induction hypothesis, since `such_that_r = false`, if `regular_r = true` $\exists c_i \in C, i < n \ . \ (b^i_{st} = \mathit{false}, b^i_r = \mathit{true})$ and $\nexists c_j \in C, j < n \ . \ b^j_{st} = \mathit{true}$. Now since the new values are `such_that_r = false`, `regular_r = true` the third option for Invariant 2 is still valid for n since $b^n_{st} = false$. If the old value for `regular_r = false`, by the induction hypothesis $\forall c_i \in C, i < n(b^i_{st} = false, b^i_r = false)$. Now since the new values unchanged, option four of Invariant 2 is still valid for n since $\forall c_i \in C, i \leq n(b^i_{st} = false, b^i_r = false)$. $\square$

A.3 Transaction Processing Code

In order to support transactional processing for commands, we take a snapshot of the knowledge state (*i.e.,* the global variables in Table 6.4) before each command is executed. After the command returns, if it was successful the snapshot may be forgotten. However, if it got stuck, the snapshot is used to restore the knowledge state to its pre-execution state. In this section we provide the Promela code used to support the taking of the state snapshot and the restoration of state.

A.3.1 Taking a snapshot

The code for taking a state snapshot of the global variables is shown below. It consists of the inline function `snapshot` which records the global variable state in an extra set of variables as shown.

```
inline snapshot()
{
   arrayCopy(objects, b_objects, MAXOBJECT);
   b_fnew = few;
   matrixCopy(m, b_m, MAXAGENT, MAXOBJECT);
   tagsCopy(mtags, b_mtags, MAXOBJECT);
}
```

Note that the inform and log arrays are not saved in the snapshot. Since they may be updated by the false branch of commands, they can not be saved and restored using the snapshot. Since, however, the log and inform records are not examined by any guards, extraneous records in them will not cause incorrect policy decisions.

A.3.2 Restoring a snapshot

The code for restoring a state snapshot is shown below. It consists of the inline function `restore` which restores the global variable state from a previously stored snapshot.

```
inline restore()
{
   arrayCopy(b_objects, objects, MAXOBJECT);
   few = b_fnew;
   matrixCopy(b_m, m, MAXAGENT, MAXOBJECT);
   tagsCopy(b_mtags, mtags, MAXOBJECT);
}
```

The restore code is similar to the snapshot code in Section A.3.1 and simply reverses the copying procedure performed in it. As noted there, the log and inform records are not saved or restored.

Table A.1: Judgment comparison for overloaded constraints in pre-command search

c_1			c_2			Results	
b_{scp}	b_{st}	b_r	b_{scp}	b_{st}	b_r	j	j'
T	T	T	T	T	T	A	A
T	T	T	T	T	F	A	A
T	T	T	T	F	T	A	A
T	T	T	T	F	F	A	A
T	T	F	T	T	T	A	A
T	T	F	T	T	F	F	F
T	T	F	T	F	T	F	F
T	T	F	T	F	F	F	F
T	F	T	T	T	T	A	A
T	F	T	T	T	F	F	F
T	F	T	T	F	T	I	I
T	F	T	T	F	F	I	I
T	F	F	T	T	T	A	A
T	F	F	T	T	F	F	F
T	F	F	T	F	T	I	I
T	F	F	T	F	F	I	I
F	T	T	F	T	T	I	I
F	T	T	F	T	F	I	I
F	T	T	F	F	T	I	I
F	T	T	F	F	F	I	I
F	T	F	F	T	T	I	I
F	T	F	F	T	F	I	I
F	T	F	F	F	T	I	I
F	T	F	F	F	F	I	I
F	F	T	F	T	T	I	I
F	F	T	F	T	F	I	I
F	F	T	F	F	T	I	I
F	F	T	F	F	F	I	I
F	F	F	F	T	T	I	I
F	F	F	F	T	F	I	I
F	F	F	F	F	T	I	I
F	F	F	F	F	F	I	I

Table A.2: Judgment comparison for overloaded constraints invocation

c_1			c_2			Results	
b_{scp}	b_{st}	b_r	b_{scp}	b_{st}	b_r	j	j'
T	T	T	T	T	T	A	A
T	T	T	T	T	F	A	A
T	T	T	T	F	T	A	A
T	T	T	T	F	F	A	A
T	T	F	T	T	T	A	A
T	T	F	T	T	F	F	F
T	T	F	T	F	T	F	F
T	T	F	T	F	F	F	F
T	F	T	T	T	T	A	A
T	F	T	T	T	F	F	F
T	F	T	T	F	T	DCA	DCA
T	F	T	T	F	F	DCA	DCA
T	F	F	T	T	T	A	A
T	F	F	T	T	F	F	F
T	F	F	T	F	T	DCA	DCA
T	F	F	T	F	F	DCF	DCF
F	T	T	F	T	T	A	A
F	T	T	F	T	F	A	A
F	T	T	F	F	T	A	A
F	T	T	F	F	F	A	A
F	T	F	F	T	T	A	A
F	T	F	F	T	F	F	F
F	T	F	F	F	T	F	F
F	T	F	F	F	F	F	F
F	F	T	F	T	T	A	A
F	F	T	F	T	F	F	F
F	F	T	F	F	T	DCA	DCA
F	F	T	F	F	F	DCA	DCA
F	F	F	F	T	T	A	A
F	F	F	F	T	F	F	F
F	F	F	F	F	T	DCA	DCA
F	F	F	F	F	F	DCF	DCF

Appendix B

Privacy Law Texts

This appendix contains excerpts from the privacy laws and policies discussed in the body of the dissertation. The texts are current as of the writing of this document except where noted.

This appendix is organized as follows. We first quote an excerpts from the Privacy Rule as issued by the US Department of Health and Human Services. The full text of the Rules is available on the Congress' General Printing Office web site, but for convenience relevant sections are included below. The versions of the regulatory text are as indicated in the section headings. All emphases are as in the original text in the Federal Register or Code of Federal Regulations. Section B.1 contains an excerpt from the 2000 version of the Privacy Rule. Section B.2 contains a parallel excerpt from the 2003 version. Section B.4 contains an excerpt from the Cable TV Privacy Act.

B.1 Privacy Rule December 28, 2000

The following text is as published in the Federal Register on December 28, 2000 pages 82462–82829. It is archived in the Code of Federal Regulations (CFR), Title 45, Subtitle A, Section 164 as issued on October 1, 2001.

B.1.1 §164.506

Sec. 164.506 Consent for uses or disclosures to carry out treatment, payment, or health care operations.

(a) Standard: Consent requirement.

(1) Except as provided in paragraph (a)(2) or (a)(3) of this section, a covered health care provider must obtain the individual's consent, in accordance with this section, prior to using or disclosing protected health information to carry out treatment, payment, or health care operations.

(2) A covered health care provider may, without consent, use or disclose protected health information to carry out treatment, payment, or health care operations, if:

(i) The covered health care provider has an indirect treatment relationship with the individual; or

(ii) The covered health care provider created or received the protected health information in the course of providing health care to an individual who is an inmate.

(3)(i) A covered health care provider may, without prior consent, use or disclose protected health information created or received under paragraph (a)(3)(i)(A)-(C) of this section to carry out treatment, payment, or health care operations:

(A) In emergency treatment situations, if the covered health care provider attempts to obtain such consent as soon as reasonably practicable after the delivery of such treatment;

(B) If the covered health care provider is required by law to treat the individual, and the covered health care provider attempts to obtain such consent but is unable to obtain such consent; or

(C) If a covered health care provider attempts to obtain such consent from the individual but is unable to obtain such consent due to substantial barriers to communicating with the individual, and the covered health care provider determines, in the exercise of professional judgment, that the individual's consent to receive treatment is clearly inferred from the circumstances.

(ii) A covered health care provider that fails to obtain such consent in accordance with paragraph (a)(3)(i) of this section must document its attempt to obtain consent and the reason why consent was not obtained.

(4) If a covered entity is not required to obtain consent by paragraph (a)(1) of this section, it may obtain an individual's consent for the covered entity's own use or disclosure of protected health information to carry out treatment, payment, or health care operations, provided that such consent meets the requirements of this section.

(5) Except as provided in paragraph (f)(1) of this section, a consent obtained by a covered entity under this section is not effective to permit another covered entity to use or disclose protected health information.

(b) Implementation specifications: General requirements.

(1) A covered health care provider may condition treatment on the provision by the individual of a consent under this section.

(2) A health plan may condition enrollment in the health plan on the provision by the individual of a consent under this section sought in conjunction with such enrollment.

(3) A consent under this section may not be combined in a single document with the notice required by Sec. 164.520.

(4)(i) A consent for use or disclosure may be combined with other types of written legal permission from the individual (e.g., an informed consent for treatment or a consent to assignment of benefits), if the consent under this section:

(A) Is visually and organizationally separate from such other written legal permission; and

(B) Is separately signed by the individual and dated.

(ii) A consent for use or disclosure may be combined with a research authorization under Sec. 164.508(f).

(5) An individual may revoke a consent under this section at any time, except to the extent that the covered entity has taken action in reliance thereon. Such revocation must be in writing.

(6) A covered entity must document and retain any signed consent under this section as required by Sec. 164.530(j).

(c) Implementation specifications: Content requirements. A consent under this section must be in plain language and:

(1) Inform the individual that protected health information may be used and disclosed to carry out treatment, payment, or health care operations;

(2) Refer the individual to the notice required by Sec. 164.520 for a more complete description of such uses and disclosures and state that the individual has the right to review the notice prior to signing the consent;

(3) If the covered entity has reserved the right to change its privacy practices that are described in the notice in accordance with Sec. 164.520(b)(1)(v)(C), state that the terms of its notice may change and describe how the individual may obtain a revised notice;

(4) State that:

(i) The individual has the right to request that the covered entity restrict how protected health information is used or disclosed to carry out treatment, payment, or health care operations;

(ii) The covered entity is not required to agree to requested restrictions; and

(iii) If the covered entity agrees to a requested restriction, the restriction is binding on the covered entity;

(5) State that the individual has the right to revoke the consent in writing, except to the extent that the covered entity has taken action in reliance thereon; and

(6) Be signed by the individual and dated.

(d) Implementation specifications: Defective consents. There is no consent under this section, if the document submitted has any of the following defects:

(1) The consent lacks an element required by paragraph (c) of this section, as applicable; or

(2) The consent has been revoked in accordance with paragraph (b)(5) of this section.

(e) Standard: Resolving conflicting consents and authorizations.

(1) If a covered entity has obtained a consent under this section and receives any other authorization or written legal permission from the individual for a disclosure of protected health information to carry out treatment, payment, or health care operations, the covered entity may disclose such protected health information only in accordance with the more restrictive consent, authorization, or other written legal permission from the individual.

(2) A covered entity may attempt to resolve a conflict between a consent and an authorization or other written legal permission from the individual described in paragraph (e)(1) of this section by:

(i) Obtaining a new consent from the individual under this section for the disclosure to carry out treatment, payment, or health care operations; or

(ii) Communicating orally or in writing with the individual in order to determine the individual's preference in resolving the conflict. The covered entity must document the individual's preference and may only disclose protected health information in accordance with the individual's preference.

(f)(1) Standard: Joint consents. Covered entities that participate in an organized health care arrangement and that have a joint notice under Sec. 164.520(d) may comply with this section by a joint consent.

(2) Implementation specifications: Requirements for joint consents.

(i) A joint consent must:

(A) Include the name or other specific identification of the covered entities, or classes of covered entities, to which the joint consent applies; and

(B) Meet the requirements of this section, except that the statements required by this section may be altered to reflect the fact that the consent covers more than one covered entity.

(ii) If an individual revokes a joint consent, the covered entity that receives the revocation must inform the other entities covered by the joint consent of the revocation as soon as practicable.

B.1.2 §164.530

The text in §164.530 is referenced in §164.506(b)(6) and therefore included here below for reference.

> Sec. 164.530 Administrative requirements.
>
> *(j)(1) Standard: Documentation.* A covered entity must:
>
> (i) Maintain the policies and procedures provided for in paragraph (i) of this section in written or electronic form;
>
> (ii) If a communication is required by this subpart to be in writing, maintain such writing, or an electronic copy, as documentation; and
>
> (iii) If an action, activity, or designation is required by this subpart to be documented, maintain a written or electronic record of such action, activity, or designation.
>
> *(2) Implementation specification: Retention period.* A covered entity must retain the documentation required by paragraph (j)(1) of this section for six years from the date of its creation or the date when it last was in effect, whichever is later.

B.2 Privacy Rule August 14, 2002

The following text is as published in the Federal Register on August 14, 2002, page 53268. It is archived in the Code of Federal Regulations (CFR), Title 45, Subtitle A, Section 164 as issued on October 1, 2003.

B.2.1 §164.506

> Sec. 164.506 Uses and disclosures to carry out treatment, payment, or health care operations.
>
> *(a) Standard: Permitted uses and disclosures.* Except with respect to uses or disclosures that require an authorization under Sec. 164.508(a)(2) and (3), a covered entity may use or disclose protected health information for treatment, payment, or health care operations as set forth in paragraph (c) of this section, provided that such use or disclosure is consistent with other applicable requirements of this subpart.
>
> *(b) Standard: Consent for uses and disclosures permitted.* (1) A covered entity may obtain consent of the individual to use or disclose protected health information to carry out treatment, payment, or health care operations.

(2) Consent, under paragraph (b) of this section, shall not be effective to permit a use or disclosure of protected health information when an authorization, under Sec. 164.508, is required or when another condition must be met for such use or disclosure to be permissible under this subpart.

(c) Implementation specifications: Treatment, payment, or health care operations.

(1) A covered entity may use or disclose protected health information for its own treatment, payment, or health care operations.

(2) A covered entity may disclose protected health information for treatment activities of a health care provider.

(3) A covered entity may disclose protected health information to another covered entity or a health care provider for the payment activities of the entity that receives the information.

(4) A covered entity may disclose protected health information to another covered entity for health care operations activities of the entity that receives the information, if each entity either has or had a relationship with the individual who is the subject of the protected health information being requested, the protected health information pertains to such relationship, and the disclosure is:

(i) For a purpose listed in paragraph (1) or (2) of the definition of health care operations; or

(ii) For the purpose of health care fraud and abuse detection or compliance.

(5) A covered entity that participates in an organized health care arrangement may disclose protected health information about an individual to another covered entity that participates in the organized health care arrangement for any health care operations activities of the organized health care arrangement.

B.2.2 §164.501

Sec. 164.501 Definitions.

As used in this subpart, the following terms have the following meanings:

Health care operations means any of the following activities of the covered entity to the extent that the activities are related to covered functions:

(1) Conducting quality assessment and improvement activities, including outcomes evaluation and development of clinical guidelines, provided that the obtaining of generalizable knowledge is not the primary purpose of any studies resulting from such activities; population-based activities relating to improving health or reducing health care costs, protocol development, case management and care coordination, contacting of health care providers and patients with information about treatment alternatives; and related functions that do not include treatment;

(2) Reviewing the competence or qualifications of health care professionals, evaluating practitioner and provider performance, health plan performance, conducting training programs in which students, trainees, or practitioners in areas of health care learn under supervision to practice or improve their skills as health care providers, training of non-health care professionals, accreditation, certification, licensing, or credentialing activities;

(3) Underwriting, premium rating, and other activities relating to the creation, renewal or replacement of a contract of health insurance or health benefits, and ceding, securing, or placing a contract for reinsurance of risk relating to claims for health care (including stop- loss insurance and excess of loss insurance), provided that the requirements of Sec. 164.514(g) are met, if applicable;

(4) Conducting or arranging for medical review, legal services, and auditing functions, including fraud and abuse detection and compliance programs;

(5) Business planning and development, such as conducting cost- management and planning-related analyses related to managing and operating the entity, including formulary development and administration, development or improvement of methods of payment or coverage policies; and

(6) Business management and general administrative activities of the entity, including, but not limited to:

(i) Management activities relating to implementation of and compliance with the requirements of this subchapter;

(ii) Customer service, including the provision of data analyses for policy holders, plan sponsors, or other customers, provided that protected health information is not disclosed to such policy holder, plan sponsor, or customer.

(iii) Resolution of internal grievances;

(iv) The sale, transfer, merger, or consolidation of all or part of the covered entity with another covered entity, or an entity that following such activity will become a covered entity and due diligence related to such activity; and

(v) Consistent with the applicable requirements of Sec. 164.514, creating de-identified health information or a limited data set, and fundraising for the benefit of the covered entity.

B.2.3 §164.508

Sec. 164.508 Uses and disclosures for which an authorization is required.

(a) Standard: authorizations for uses and disclosures.

(1) Authorization required: general rule. Except as otherwise permitted or required by this subchapter, a covered entity may not use or disclose protected health information without an authorization that is valid under this section. When a covered entity obtains or receives a valid authorization for its use

or disclosure of protected health information, such use or disclosure must be consistent with such authorization.

(2) Authorization required: psychotherapy notes. Notwithstanding any provision of this subpart, other than the transition provisions in Sec. 164.532, a covered entity must obtain an authorization for any use or disclosure of psychotherapy notes, except:

(i) To carry out the following treatment, payment, or health care operations:

(A) Use by the originator of the psychotherapy notes for treatment;

(B) Use or disclosure by the covered entity for its own training programs in which students, trainees, or practitioners in mental health learn under supervision to practice or improve their skills in group, joint, family, or individual counseling; or

(C) Use or disclosure by the covered entity to defend itself in a legal action or other proceeding brought by the individual; and

(ii) A use or disclosure that is required by Sec. 164.502(a)(2)(ii) or permitted by Sec. 164.512(a); Sec. 164.512(d) with respect to the oversight of the originator of the psychotherapy notes; Sec. 164.512(g)(1); or Sec. 164.512(j)(1)(i).

(3) Authorization required: Marketing.

(i) Notwithstanding any provision of this subpart, other than the transition provisions in Sec. 164.532, a covered entity must obtain an authorization for any use or disclosure of protected health information for marketing, except if the communication is in the form of:

(A) A face-to-face communication made by a covered entity to an individual; or

(B) A promotional gift of nominal value provided by the covered entity.

(ii) If the marketing involves direct or indirect remuneration to the covered entity from a third party, the authorization must state that such remuneration is involved.

B.3 Insurance Council of Australia Privacy Code

In Section 7.2 we consider the Insurance Council of Australia's Privacy Code. The rules in Chapter 2 regarding disclosure of personal information by a health service provider are as follows:

Disclosure by a health service provider

2.4 Despite the rules on use and disclosure of personal information in Privacy Principle 2.1, an organisation that provides a health service to an individual may disclose health information about the individual to a person who is responsible for the individual if:

(a) the individual:

(i) is physically or legally incapable of consenting to the disclosure; or

(ii) physically cannot communicate consent to the disclosure; and

(b) a natural person (the carer) providing the health service for the organisation is satisfied that either:

(i) the disclosure is necessary to provide appropriate care or treatment of the individual; or

(ii) the disclosure is made for compassionate reasons; and

(c) the disclosure is not contrary to any wish:

(i) expressed by the individual before the individual became unable to give or communicate consent; and

(ii) of which the carer is aware, or of which the carer could reasonably be expected to be aware; and

(d) the disclosure is limited to the extent reasonable and necessary for a purpose mentioned in paragraph (b).

2.5 For the purposes of 2.4, a person is responsible for an individual if the person is:

(a) a parent of the individual; or

(b) a child or sibling of the individual and at least 18 years old; or

(c) a spouse or de facto spouse of the individual; or

(d) a relative of the individual, at least 18 years old and a member of the individuals household; or

(e) a guardian of the individual; or

(f) exercising an enduring power of attorney granted by the individual that is exercisable in relation to decisions about the individuals health; or

(g) a person who has an intimate personal relationship with the individual; or

(h) a person nominated by the individual to be contacted in case of emergency.

2.6 For the purposes of 2.5:

child of an individual includes an adopted child, a step-child and a foster-child, of the individual.

parent of an individual includes a step-parent, adoptive parent and a foster-parent, of the individual.

relative of an individual means a grandparent, grandchild, uncle, aunt, nephew or niece of the individual.

sibling of an individual includes a half-brother, half-sister, adoptive brother, adoptive sister, step-brother, step-sister, foster-brother and foster-sister, of the individual.

B.4 Cable TV Privacy Act of 1984

The Cable TV Privacy Act of 1984 is recorded in US Code Title 47, Chapter 5, Subchapter V-A, Part IV, Section 551. An excerpt of the regulatory text from the current version of the Act is as follows.

> *(a) Notice to subscriber regarding personally identifiable information; definitions.*
>
> (1) At the time of entering into an agreement to provide any cable service or other service to a subscriber and at least once a year thereafter, a cable operator shall provide notice in the form of a separate, written statement to such subscriber which clearly and conspicuously informs the subscriber of —
>
> (A) the nature of personally identifiable information collected or to be collected with respect to the subscriber and the nature of the use of such information;
>
> (B) the nature, frequency, and purpose of any disclosure which may be made of such information, including an identification of the types of persons to whom the disclosure may be made;
>
> (C) the period during which such information will be maintained by the cable operator;
>
> (D) the times and place at which the subscriber may have access to such information in accordance with subsection (d) of this section; and
>
> (E) the limitations provided by this section with respect to the collection and disclosure of information by a cable operator and the right of the subscriber under subsections (f) and (h) of this section to enforce such limitations.
>
> In the case of subscribers who have entered into such an agreement before the effective date of this section, such notice shall be provided within 180 days of such date and at least once a year thereafter.
>
> (2) For purposes of this section, other than subsection (h) of this section —
>
> (A) the term "personally identifiable information" does not include any record of aggregate data which does not identify particular persons;
>
> (B) the term "other service" includes any wire or radio communications service provided using any of the facilities of a cable operator that are used in the provision of cable service; and
>
> (C) the term "cable operator" includes, in addition to persons within the definition of cable operator in section 522 of this title, any person who
>
> (i) is owned or controlled by, or under common ownership or control with, a cable operator, and
>
> (ii) provides any wire or radio communications service.
>
> *(b) Collection of personally identifiable information using cable system.*

(1) Except as provided in paragraph (2), a cable operator shall not use the cable system to collect personally identifiable information concerning any subscriber without the prior written or electronic consent of the subscriber concerned.

(2) A cable operator may use the cable system to collect such information in order to —

(A) obtain information necessary to render a cable service or other service provided by the cable operator to the subscriber; or

(B) detect unauthorized reception of cable communications.

(c) Disclosure of personally identifiable information.

(1) Except as provided in paragraph (2), a cable operator shall not disclose personally identifiable information concerning any subscriber without the prior written or electronic consent of the subscriber concerned and shall take such actions as are necessary to prevent unauthorized access to such information by a person other than the subscriber or cable operator.

(2) A cable operator may disclose such information if the disclosure is —

(A) necessary to render, or conduct a legitimate business activity related to, a cable service or other service provided by the cable operator to the subscriber;

(B) subject to subsection (h) of this section, made pursuant to a court order authorizing such disclosure, if the subscriber is notified of such order by the person to whom the order is directed; or

(C) a disclosure of the names and addresses of subscribers to any cable service or other service, if —

(i) the cable operator has provided the subscriber the opportunity to prohibit or limit such disclosure, and

(ii) the disclosure does not reveal, directly or indirectly, the —

(I) extent of any viewing or other use by the subscriber of a cable service or other service provided by the cable operator, or

(II) the nature of any transaction made by the subscriber over the cable system of the cable operator.

. . .

(h) Disclosure of information to governmental entity pursuant to court order. A governmental entity may obtain personally identifiable information concerning a cable subscriber pursuant to a court order only if, in the court proceeding relevant to such court order —

(1) such entity offers clear and convincing evidence that the subject of the information is reasonably suspected of engaging in criminal activity and that the information sought would be material evidence in the case; and

(2) the subject of the information is afforded the opportunity to appear and contest such entity's claim.

B.5 TiVo Privacy Policy

The privacy policy for TiVo Corporation lists its business practices with respect to the subscriber information that it collects. The most current edition of the policy can be found on the company's web site `www.tivo.com`. For reference we include a selection of the full policy which is current as of this writing. It was last updated in May 2006. We use the policy selection below in a case study in Section 7.3.

Section 1 of the privacy policy includes definitions of terms used in the policy.

> 1. Our User Information Definitions
>
> In discussing the kind of information TiVo collects, it is important to distinguish between anonymous information and other information that specifically identifies you or your household. Too often people refer to "personal information" or "personally identifiable information" without really making clear what they mean. TiVo has developed the following definitions to help clarify this important issue. We use "User Information" as a general term that refers to any information relating to you or your use of the TiVo service. The following are more specific types of User Information.
>
> 1.1 "Account Information" means information about you and your TiVo DVR, including your Contact Information (defined below) and other information linked to your Contact Information such as the model and Service Number of your TiVo DVR, your ZIP code, software version used, your TV programming source (cable, satellite or an antenna), the type of cable hook-up (digital or analog) and level of TiVo service (TiVo Basic, TiVo Plus, premium services, etc.), privacy preferences, and the cable or satellite box model that you use. You provide us with this information when (a) you set up your TiVo DVR with TiVo Basic service or (b) when you register as a user of the TiVo Plus service. This Service Information (defined below) is sent to TiVo on an ongoing basis to enable TiVo to provide the TiVo service to your TiVo DVR. Account Information also includes information we may receive about you in a communication from you or a Third Party. Account Information does not include any Personally Identifiable Viewing Information, or Anonymous Viewing Information, as defined below.
>
> a "Contact Information" means information that allows someone to identify or contact you, including, for example: your name, address, telephone number, and e-mail address. Contact Information is a subset of Account Information and is thus linked to your TiVo DVR's Service Number. Your ZIP code by itself, while part of your address, is not Contact Information because your ZIP code alone does not allow someone to identify or contact you. NOTE: If your TiVo DVR is receiving the TiVo Basic service, you are not required to provide TiVo with any Contact Information from that TiVo DVR.

b “Service Information” means information necessary for TiVo to provide service to your TiVo DVR. Examples of Service Information include your software version number, your TV programming source, level of service, and the success status of the last attempted service connection (e.g., periodic call). This information is always transmitted to TiVo when connected to the TiVo servers.

1.2 “Personally Identifiable Viewing Information” means information about the viewing choices that you and those in your household make while using your TiVo DVR, if that information is linked to or associated with your Account Information. Your TiVo DVR stores your viewing information so that it may recommend viewing choices and personalize your viewing experience. We have worked very hard to design our system to ensure that no Personally Identifiable Viewing Information may be sent to, or collected by, TiVo without your express consent. If you use the TiVo Plus service, you may choose to consent to TiVo’s collection of Personally Identifiable Viewing Information by changing your privacy preferences. NOTE: If your TiVo DVR is receiving the TiVo Basic service, you may not be able to consent to TiVo’s collection of Personally Identifiable Viewing Information from that TiVo DVR.

1.3 “Anonymous Viewing Information” means information about viewing choices that you and those in your household make while using your TiVo DVR, but is not associated with or linked to any Contact Information. Your TiVo DVR sends Anonymous Viewing Information to TiVo on an ongoing basis. This information allows TiVo to know that a TiVo service user from a particular ZIP code watched certain programming but we are unable to associate those viewing choices with you. If you use the TiVo Plus service, you may request that TiVo block the collection of Anonymous Viewing Information from your TiVo DVR. NOTE: If your TiVo DVR is receiving the TiVo Basic service, you may not opt out of TiVo’s collection of Anonymous Viewing Information from that TiVo DVR. . . .

Section 3 of the policy describes the situations when TiVo discloses user information.

3. Disclosure of User Information

3.1 Generally. We disclose aggregated Account Information and aggregated Anonymous Viewing Information and any reports or analyses derived therefrom, to unaffiliated third parties including advertisers, broadcasters, consumer and market research companies and other organizations (“Third Parties”).

3.2 Manufacturing and Service Provider Partners. In certain instances we will disclose to our hardware manufacturing partners and service provider partners (for example DIRECTV) the Account Information of users who have a DVR made by that manufacturing partner or receive a service from that service provider partner. However, TiVo contractually binds our manufacturing and service provider partners to comply with the provisions of this Privacy Policy. Our manufacturing and service provider partners are legally liable for misuse of User Information.

3.3 Contractors and Vendors. We use contractors to help with some of our operations. Some of these contractors will have access to our databases of User Information on a temporary basis for specific tasks. TiVo also uses vendors to help with certain aspects of its operations, which may require disclosure of your User Information to them. For example, TiVo may use a vendor to communicate with you (via telephone, e-mail, or letter) about your TiVo service or upcoming features or services, to mail rebate checks, to generate demographic profiles based on User Information of current TiVo service users, and to perform other work that we may need to outsource. If you are a TiVo Plus service user, TiVo may additionally use a vendor to process and collect payment for your TiVo Plus service via your credit card. TiVo contractually binds these contractors and vendors to use your User Information only as necessary to perform the services they are asked to perform. Such contractors and vendors are legally liable for misuse of User Information.

3.4 The "Corporate Family." TiVo may share some or all of your User Information with any parent company, subsidiaries, joint ventures, or other companies under a common control (collectively "Affiliates"). In such event, TiVo will require its Affiliates to honor this Privacy Policy. If another company acquires TiVo, or acquires assets of TiVo that comprise or include your User Information, that company will possess the User Information collected by TiVo and it will assume the rights and obligations regarding your User Information as described in this Privacy Policy.

3.5 Commerce Partners. When you elect to participate in a special offer or engage in a transaction with TiVo or a Third Party advertiser or promoter, TiVo will collect and disclose your Commerce Information to the commerce partner sponsoring and/or fulfilling the promotion. In addition to fulfilling your request, that commerce partner may also use your Commerce Information to send you other information in which you might be interested consistent with its own privacy policies. This information is disclosed only upon your affirmative response to an offer. NOTE: Depending on your level of TiVo service and the model of your TiVo DVR, such features may not be available to you.

3.6 Factors Beyond Our Control. Your privacy is very important to us. Due to factors beyond our control, however, we cannot fully ensure that your User Information will not be disclosed to Third Parties. For example, we may be legally obligated to disclose User Information to local, state or federal governmental agencies or Third Parties under certain circumstances (including in response to a subpoena), or Third Parties may circumvent our security measures to unlawfully intercept or access your User Information.

Appendix C

Full Privacy APIs for Case Studies

This chapter contains the Privacy APIs for the case studies presented above in Chapter 7. The formal code in this chapter is derived from the legal texts considered in the case studies and is designed to model it directly. The roles, rights, tags, and purposes for each Privacy API are discussed above in the relevant case study.

C.1 Privacy Rule 2000 Privacy API

The Privacy API as derived for the Privacy Rule [§164.506, v.2000] section on consent for treatment, payment, and health care operations is as follows. For reference, the full regulatory text for the section is presented in Section B.1.

C.1.1 Tags

Due to the large number of tags in the Privacy API, we list them here in Table C.1 rather than in Section 7.1.2. They are the members of *Tag* for the Privacy API for the 2000 Privacy Rule.

Table C.1: Tags for the HIPAA 2000 Privacy API

attempted-consent	barriers-to-communication
combined-research	consent-changed-to-multiple-entities
consent-combined	consent-in-writing
consent-includes-classes	consent-includes-names
consent-includes-specific-identifiers	consent-is-dated
consent-is-signed	consent-refers-to-notice
consent-request-consistent-with-164.506	consent-separately-dated
consent-separately-signed	
consent-states-entity-not-required-to-agree	consent-visibly-separate
created-while-inmate	
consent-states-if-entity-agrees-must-comply	consent-states-individual-may-revoke
consent-states-may-request-restriction	
consent-states-revocation-only-for-future	consent-states-right-to-review
describes-how-to-obtain-new	health-plan-member
in-writing	
informs-may-be-used-for-health-care-operations	informs-may-be-used-for-payment
informs-may-be-used-for-treatment	joint-notice
must-be-in-writing	must-obtain-consent
policies-mentioned-in-530i	
professional-judgment-indicates-consent	protected-health-information
received-while-inmate	record-electronically
record-in-writing	refuse-without-consent
reserves-right-to-update-notice	revocation-in-writing
revoked-consent	states-right-to-modify
will-obtain-consent-asap	will-record

C.1.2 Privacy API

§164.506(a)(1)

CST	Permitted506a1(a, s, r, P, f, f', msg)
Scope	{TreatmentUse506a1, PaymentUse506a1, HealthCareOperationsUse506a1, TreatmentDisclose506a1, PaymentDisclose506a1, HealthCareOperationsDisclose506a1}
Such That	individual **in** Roles(s)
and	f.protected-health-information = true
if	Permitted506a2(a, s, r, P, f, f', msg) $\in$ {Allow}
then	**return true**
else	**return false**

```
      CST   Permitted506a1(a, s, r, P, f, f', msg)
    Scope   {TreatmentUse506a1, PaymentUse506a1,
            HealthCareOperationsUse506a1, TreatmentDisclose506a1,
            PaymentDisclose506a1, HealthCareOperationsDisclose506a1}
Such That   individual in Roles(s)
      and   f.protected-health-information = true
       if   Permitted506a3(a, s, r, P, f, f', msg) ∈ {Allow}
     then   return true
     else   return false

      CST   Permitted506a1(a, s, r, P, f, f', msg)
    Scope   {TreatmentUse506a1, PaymentUse506a1,
            HealthCareOperationsUse506a1, TreatmentDisclose506a1,
            PaymentDisclose506a1, HealthCareOperationsDisclose506a1}
Such That   individual in Roles(s)
      and   f.protected-health-information = true
       if   consent in (a, s)
     then   return true
     else   return false

CMD    TreatmentUse506a1 (a, s, r, P, f, f', msg)
  if   Permitted506a1(a, s, r, P, f, f', msg) ∈ {Allow}
 and   individual in Roles(s)
 and   healthCareProvider in Roles(a)
 and   local in (a, f)
 and   treatment $\mathbf{in}_f$ P
 and   use $\mathbf{in}_a$ P
then   insert treatment in (a, s)
 and   return true
else   return false
```

```
CMD   PaymentUse506a1 (a, s, r, P, f, f', msg)
 if   Permitted506a1(a, s, r, P, f, f', msg) $\in$ {Allow}
and   individual in Roles(s)
and   healthCareProvider in Roles(a)
and   local in (a, f)
and   payment $\mathbf{in}_f$ P
and   use $\mathbf{in}_a$ P
then  insert payment in (a, s)
and   return true
else  return false

CMD   HealthCareOperationsUse506a1 (a, s, r, P, f, f', msg)
 if   Permitted506a1(a, s, r, P, f, f', msg) $\in$ {Allow}
and   individual in Roles(s)
and   healthCareProvider in Roles(a)
and   local in (a, f)
and   healthCareOperations $\mathbf{in}_f$ P
and   use $\mathbf{in}_a$ P
then  insert healthCareOperations in (a, s)
and   return true
else  return false
```

```
CMD   TreatmentDisclose506a1 (a, s, r, P, f, f’, msg)
 if   Permitted506a1(a, s, r, P, f, f’, msg) $\in$ {Allow}
and   individual in Roles(s)
and   healthCareProvider in Roles(a)
and   local in (a, f)
and   treatment $\mathbf{in}_f$ P
and   disclose $\mathbf{in}_a$ P
then  insert local in (r, f)
and   return true
else  return false

CMD   PaymentDisclose506a1 (a, s, r, P, f, f’, msg)
 if   Permitted506a1(a, s, r, P, f, f’, msg) $\in$ {Allow}
and   individual in Roles(s)
and   healthCareProvider in Roles(a)
and   local in (a, f)
and   payment $\mathbf{in}_a$ P
and   disclose $\mathbf{in}_a$ P
then  insert local in (r, f)
and   return true
else  return false
```

```
CMD   HealthCareOperationsDisclose506a1 (a, s, r, P, f, f', msg)
 if   Permitted506a1(a, s, r, P, f, f', msg) $\in$ {Allow}
and   individual in Roles(s)
and   healthCareProvider in Roles(a)
and   local in (a, f)
and   healthCareOperations $\mathbf{in}_a$ P
and   disclose $\mathbf{in}_a$ P
then  insert local in (r, f)
and   return true
else  return false

CMD   GrantConsent506a(a, s, r, P, f, f', msg)
 if   individual in Roles(s)
and   healthCareProvider in Roles(a)
then  insert consent in (a, s)
and   RecordConsent506b6 (a, s, r, P, f, f', msg)
and   return true
else  return false
```

§164.506(a)(2)

```
      CST   Permitted506a2(a, s, r, P, f, f', msg)
    Scope   {TreatmentUse506a2, PaymentUse506a2,
            HealthCareOperationsUse506a2, TreatmentDisclose506a2,
            PaymentDisclose506a2, HealthCareOperationsDisclose506a2}
Such That   individual in Roles(s)
      and   f.protected-health-information = true
       if   Permitted506a2i (a, s, r, P, f, f', msg)
     then   return true
     else   return false
```

```
       CST   Permitted506a2(a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506a2, PaymentUse506a2,
             HealthCareOperationsUse506a2, TreatmentDisclose506a2,
             PaymentDisclose506a2, HealthCareOperationsDisclose506a2}
 Such That   individual in Roles(s)
       and   f.protected-health-information = true
        if   Permitted506a2ii (a, s, r, P, f, f', msg)
      then   return true
      else   return false

CMD   TreatmentUse506a2 (a, s, r, P, f, f', msg)
  if  Permitted506a2(a, s, r, P, f, f', msg) $\in$ {Allow}
 and  local in (a, f)
 and  treatment $\mathbf{in}_f$ P
 and  use $\mathbf{in}_a$ P
then  insert treatment in (a, s)
 and  return true
else  return false

CMD   PaymentUse506a2 (a, s, r, P, f, f', msg)
  if  Permitted506a2(a, s, r, P, f, f', msg) $\in$ {Allow}
 and  local in (a, f)
 and  payment $\mathbf{in}_f$ P
 and  use $\mathbf{in}_a$ P
then  insert payment in (a, s)
 and  return true
else  return false
```

```
CMD   HealthCareOperationsUse506a2 (a, s, r, P, f, f', msg)
 if   Permitted506a2(a, s, r, P, f, f', msg) $\in$ {Allow}
and   local in (a, f)
and   healthCareOperations $\mathbf{in}_f$ P
and   use $\mathbf{in}_a$ P
then  insert healthCareOperations in (a, s)
and   return true
else  return false

CMD   TreatmentDisclose506a2 (a, s, r, P, f, f', msg)
 if   Permitted506a2(a, s, r, P, f, f', msg) $\in$ {Allow}
and   local in (a, f)
and   treatment $\mathbf{in}_f$ P
and   disclose $\mathbf{in}_a$ P
then  insert local in (r, f)
and   return true
else  return false

CMD   PaymentDisclose506a2 (a, s, r, P, f, f', msg)
 if   Permitted506a2(a, s, r, P, f, f', msg) $\in$ {Allow}
and   local in (a, f)
and   payment $\mathbf{in}_f$ P
and   disclose $\mathbf{in}_a$ P
then  insert local in (r, f)
and   return true
else  return false
```

```
       CMD  HealthCareOperationsDisclose506a2 (a, s, r, P, f, f', msg)
        if  Permitted506a2(a, s, r, P, f, f', msg) $\in$ {Allow}
       and  local **in** (a, f)
       and  healthCareOperations $\mathbf{in}_f$ P
       and  disclose $\mathbf{in}_a$ P
      then  **insert** local **in** (r, f)
       and  **return true**
      else  **return false**
```

§164.502(a)(2)(i)

```
       CST  Permitted506a2i ( a, s, r, P, f, f', msg)
     Scope  {TreatmentUse506a2, PaymentUse506a2,
            HealthCareOperationsUse506a2, TreatmentDisclose506a2,
            PaymentDisclose506a2, HealthCareOperationsDisclose506a2}
 Such That  individual in Roles(s)
       and  f.protected-health-information = true
        if  healthCareProvider in Roles(a)
       and  indirectTreatment $\mathbf{in}_a$ P
       and  indirect in (a, s)
      then  return true
      else  return false
```

§162.506(a)(2)(ii)

```
       CST   Permitted506a2ii ( a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506a2, PaymentUse506a2,
             HealthCareOperationsUse506a2, TreatmentDisclose506a2,
             PaymentDisclose506a2, HealthCareOperationsDisclose506a2}
 Such That   individual in Roles(s)
       and   f.protected-health-information = true
        if   f.received-while-inmate = true
       and   healthCareProvider in Roles(a)
      then   return true
      else   return false

       CST   Permitted506a2ii ( a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506a2, PaymentUse506a2,
             HealthCareOperationsUse506a2, TreatmentDisclose506a2,
             PaymentDisclose506a2, HealthCareOperationsDisclose506a2}
 Such That   individual in Roles(s)
       and   f.protected-health-information = true
        if   f.created-while-inmate = true
       and   healthCareProvider in Roles(a)
      then   return true
      else   return false
```

§164.506(a)(3)

```
       CST   Permitted506a3(a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506a3, PaymentUse506a3,
             HealthCareOperationsUse506a3, TreatmentDisclose506a3,
             PaymentDisclose506a3, HealthCareOperationsDisclose506a3}
 Such That   individual in Roles(s)
       and   f.protected-health-information = true
        if   Permitted508a3i(a, s, r, P, f, f', msg) ∈ {Allow}
      then   return true
      else   return false

  CMD   TreatmentUse506a3 (a, s, r, P, f, f', msg)
   if   Permitted506a3 (a, s, r, P, f, f', msg) ∈ {Allow}
  and   individual in Roles(s)
  and   healthCareProvider in Roles(a)
  and   local in (a, f)
  and   treatment $\mathbf{in}_a$ P
  and   use $\mathbf{in}_a$ P
 then   insert treatment in (a, s)
  and   invoke noteAttempt506a3ii(a, s, r, P, f, f', msg)
  and   return true
 else   return false
```

```
CMD   PaymentUse506a3 (a, s, r, P, f, f', msg)
  if  Permitted506a3 (a, s, r, P, f, f', msg) $\in$ {Allow}
 and  individual in Roles(s)
 and  healthCareProvider in Roles(a)
 and  local in (a, f)
 and  payment $\mathbf{in}_a$ P
 and  use $\mathbf{in}_a$ P
then  insert payment in (a, s)
 and  invoke noteAttempt506a3ii(a, s, r, P, f, f', msg)
 and  return true
else  return false

CMD   HealthCareOperationsUse506a3 (a, s, r, P, f, f', msg)
  if  Permitted506a3 (a, s, r, P, f, f', msg) $\in$ {Allow}
 and  individual in Roles(s)
 and  healthCareProvider in Roles(a)
 and  local in (a, f)
 and  healthCareOperations $\mathbf{in}_a$ P
 and  use $\mathbf{in}_a$ P
then  insert healthCareOperations in (a, s)
 and  invoke noteAttempt506a3ii(a, s, r, P, f, f', msg)
 and  return true
else  return false
```

```
CMD   TreatmentDisclose506a3 (a, s, r, P, f, f', msg)
 if   Permitted506a3 (a, s, r, P, f, f', msg) $\in$ {Allow}
and   individual in Roles(s)
and   healthCareProvider in Roles(a)
and   local in (a, f)
and   treatment $\textbf{in}_a$ P
and   disclose $\textbf{in}_a$ P
then  insert local in (r, f)
and   invoke noteAttempt506a3ii(a, s, r, P, f, f', msg)
and   return true
else  return false

CMD   PaymentDisclose506a3 (a, s, r, P, f, f', msg)
 if   Permitted506a3 (a, s, r, P, f, f', msg) $\in$ {Allow}
and   individual in Roles(s)
and   healthCareProvider in Roles(a)
and   local in (a, f)
and   payment $\textbf{in}_a$ P
and   disclose $\textbf{in}_a$ P
then  insert local in (r, f)
and   invoke noteAttempt506a3ii(a, s, r, P, f, f', msg)
and   return true
else  return false
```

```
       CMD  HealthCareOperationsDisclose506a3 (a, s, r, P, f, f', msg)
        if  Permitted506a3 (a, s, r, P, f, f', msg) $\in$ {Allow}
       and  individual **in** Roles(s)
       and  healthCareProvider **in** Roles(a)
       and  local **in** (a, f)
       and  healthCareOperations $\mathbf{in}_a$ P
       and  disclose $\mathbf{in}_a$ P
      then  **insert** local **in** (r, f)
       and  **invoke** noteAttempt506a3ii(a, s, r, P, f, f', msg)
       and  **return true**
      else  **return false**
```

§164.506(a)(3)(i)

```
        CST  Permitted506a3i(a, s, r, P, f, f', msg)
      Scope  {TreatmentUse506a3, PaymentUse506a3,
             HealthCareOperationsUse506a3, TreatmentDisclose506a3,
             PaymentDisclose506a3, HealthCareOperationsDisclose506a3}
  Such That  individual **in** Roles(s)
        and  f.protected-health-information = true
         if  Permitted506a3iA (a, s, r, P, f, f', msg) $\in$ {Allow}
       then  **return true**
       else  **return false**
        CST  Permitted506a3i(a, s, r, P, f, f', msg)
      Scope  {TreatmentUse506a3, PaymentUse506a3,
             HealthCareOperationsUse506a3, TreatmentDisclose506a3,
             PaymentDisclose506a3, HealthCareOperationsDisclose506a3}
  Such That  individual in Roles(s)
        and  f.protected-health-information = true
         if  Permitted506a3iB (a, s, r, P, f, f', msg) $\in$ {Allow}
       then  return true
       else  return false
```

```
       CST  Permitted506a3i(a, s, r, P, f, f', msg)
     Scope  {TreatmentUse506a3, PaymentUse506a3,
            HealthCareOperationsUse506a3, TreatmentDisclose506a3,
            PaymentDisclose506a3, HealthCareOperationsDisclose506a3}
 Such That  individual in Roles(s)
       and  f.protected-health-information = true
        if  Permitted506a3iC (a, s, r, P, f, f', msg) $\in$ {Allow}
      then  return true
      else  return false
```

§164.506(a)(3)(i)(A)

```
       CST  Permitted506a3iA(a, s, r, P, f, f', msg)
 Such That  emergency $\mathbf{in}_a$ P
       and  individual in Roles(s)
       and  f.protected-health-information = true
       and  healthCareProvider in Roles(a)
        if  a.will-obtain-consent-asap = true
       and  return true
      else  return false
```

§164.506(a)(3)(i)(B)

```
       CST  Permitted506a3iB (a, s, r, P, f, f', msg)
 Such That  requiredToTreat in (a, s)
       and  individual in Roles(s)
       and  f.protected-health-information = true
       and  healthCareProvider in Roles(a)
        if  a.attemptedConsent = true
      then  return true
      else  return false
```

§164.506(a)(3)(i)(C)

```
       CST  Permitted506a3iC (a, s, r, P, f, f', msg)
 Such That  s.barriers-to-communication = true
       and  s.professional-judgment-indicates-consent = true
       and  individual in Roles(s)
       and  f.protected-health-information = true
       and  healthCareProvider in Roles(a)
        if  a.attemptedConsent = true
      then  return true
      else  return false
```

§164.506(a)(3)(ii)

```
 CMD  noteAttempt506a3ii (a, s, r, P, f, f', msg)
  if  a.attemptedConsent = true
then  insert attemptedConsent in (a, s)
 and  insert "Attempted consent and failed" in log
 and  return true
else  return false
```

§**164.506(a)(4)**

```
        CST   VoluntaryConsent506a4(a, s, r, P, f, f’, msg)
      Scope   {OptionalConsent506a4}
  Such That   Consent506a1(a, s, r, P, f, f’, msg) $\in$ {Forbid}
        and   individual in Roles(s)
         if   voluntaryConsent $\textbf{in}_a$ P
        and   a.consent-request-consistent-with-164.506 = true
        and   healthCareProvider in Roles(a)
        and   treatment $\textbf{in}_a$ P
       then   return true
       else   return false

        CST   VoluntaryConsent506a4(a, s, r, P, f, f’, msg)
      Scope   {OptionalConsent506a4}
  Such That   Consent506a1(a, s, r, P, f, f’, msg) $\in$ {Forbid}
         if   voluntaryConsent $\textbf{in}_a$ P
        and   a.consent-request-consistent-with-164.506 = true
        and   individual in Roles(s)
        and   healthCareProvider in Roles(a)
        and   payment $\textbf{in}_a$ P
       then   return true
       else   return false
```

```
CST         VoluntaryConsent506a4(a, s, r, P, f, f', msg)
Scope       {OptionalConsent506a4}
Such That   Consent506a1(a, s, r, P, f, f', msg) $\in$ {Forbid}
  if        voluntaryConsent in$_a$ P
  and       a.consent-request-consistent-with-164.506 = true
  and       individual in Roles(s)
  and       healthCareProvider in Roles(a)
  and       healthCareOperations in$_a$ P
  then      return true
  else      return false

CMD   OptionalConsent506a4 (a, s, r, P, f, f', msg)
  if    VoluntaryConsent506a4(a, s, r, P, f, f', msg) $\in$ {Allow}
  and   local in (a, f)
  and   use in$_a$ P
  then  insert consent in (a, s)
  and   RecordConsent506b6 (a, s, r, P, f, f', msg)
  and   return true
  else  return false
```

§164.506(a)(5)

```
       CST   TransferConsent506a5 (a, s, r, P, f, f’, msg)
     Scope   {OtherConsent506a5}
 Such That   transfer-consent in$_a$ P
       and   consent in (a, s)
       and   coveredEntity in Roles(a)
       and   coveredEntity in Roles(r)
       and   individual in Roles(s)
        if   JointConsent506f1 (a, s, r, P, f, f’, msg) $\in$ {Allow}
       and   local in (a, f)
      then   return true
      else   return false
```

```
 CMD   OtherConsent506a5 (a, s, r, P, f, f’, msg)
  if   TransferConsent506a5(r, s, a, P, f, f′, msg) $\in$ {Allow}
then   insert consent in (a, s)
 and   return true
else   return false
```

§164.506(b)(1)

```
       CST   Condition506b1(a, s, r, P, f, f’, msg)
     Scope   {BeginTreatment506b1}
 Such That   a.refuse-without-consent = true
       and   individual in Roles(s)
        if   consent in (a, s)
      then   return true
      else   return false
```

```
CMD        BeginTreatment506b1 (a, s, r, P, f, f', msg)
  if       Condition506b1(a, s, r, P, f, f', msg) $\in$ {Allow, Don't Care/Allow,
           Don't Care/Forbid}
  and      healthCareProvider in Roles(a)
  and      treatment $\mathbf{in}_a$ P
  then     insert beginTreat in (a, s)
  and      return true
  else     return false
```

§164.506(b)(2)

```
CST          Condition506b2(a, s, r, P, f, f', msg)
Scope        {Enroll506b2}
Such That    a.refuse-without-consent = true
  and        consent in (a, s)
  and        individual in Roles(s)
  if         true
  then       return true
  else       return false
```

```
CMD        Enroll506b2 (a, s, r, P, f, f', msg)
  if       Condition506b2(a, s, r, P, f, f', msg) $\in$ {Allow, Don't Care/Allow,
           Don't Care/Forbid}
  and      healthPlan in Roles(a)
  and      enrollment $\mathbf{in}_a$ P
  then     set s.healthPlan-member = true
  and      return true
  else     return false
```

§164.506(b)(3)

No Commands

§164.506(b)(4)(i)

```
      CST  PermittedCombine506b4i(a, s, r, P, f, f', msg)
    Scope  {CombineConsent506b4i}
Such That  s.consent-combined = true
       if  PermittedCombine506b4iA(a, s, r, P, f, f', msg) ∈ {Allow}
      and  PermittedCombine506b4iB(a, s, r, P, f, f', msg) ∈ {Allow}
     then  return true
     else  return false
```

```
  CMD  CombineConsent506b4i (a, s, r, P, f, f', msg)
   if  PermittedCombined506b4i(a, s, r, P, f, f', msg) ∈ {Allow, Don't Care/Allow}
  and  s.consent-in-writing = true
 then  insert consent in (a, s)
  and  recordConsent506b6 (a, s, r, P, f, f', msg)
  and  return true
 else  return false
```

§164.506(b)(4)(i)(A)

```
      CST  PermittedCombine506b4iA(a, s, r, P, f, f', msg)
    Scope  {CombineConsent506b4i}
Such That  s.consent-combined = true
       if  s.consent-visibly-separate = true
     then  return true
     else  return false
```

§164.506(b)(4)(i)(B)

```
       CST  PermittedCombine506b4iB(a, s, r, P, f, f', msg)
     Scope  {CombineConsent506b4i}
 Such That  s.consent-combined = true
        if  s.consent-separately-signed = true
       and  s.consent-separately-dated = true
      then  return true
      else  return false
```

§164.506(b)(4)(ii)

```
       CST  PermittedCombine506b4ii(a, s, r, P, f, f', msg)
     Scope  {CombinedConsent506b4ii}
 Such That  s.consent-combined = true
       and  s.combined-research = true
       and  research $\mathbf{in}_a$ P
        if  PermittedResearch508f(a, s, r, P, f, f', msg) $\in$ {Allow,
            Don't Care/Allow}
      then  return true
      else  return false

 CMD  CombinedConsent506b4ii (a, s, r, P, f, f', msg)
  if  PermittedCombine506b4ii(a, s, r, P, f, f', msg) $\in$ {Alow, Don't Care/Allow}
then  insert consent in (a, s)
 and  insert research in (a, s)
 and  RecordConsent506b6 (a, s, r, P, f, f', msg)
 and  return true
else  return false
```

§164.506(b)(5)

```
CMD         RevokeConsent506b5 (a, s, r, P, f, f', msg)
   if       s.revocation-in-writing = true
   and      consent in (a, s)
   and      revoke $\mathbf{in}_f$ P
   then     delete consent from (a, s)
   and      invoke Record530j(a, s, r, P, f, f', msg)
   and      return true
   else     return false
```

§164.506(b)(6)

```
CST         ReqRecordConsent506b6 (a, s, r, P, f, f', msg)
Scope       {GrantConsent506a, OptionalConsent506a4, CombineConsent506b4i,
            CombinedConsent506b4ii, GrantJointConsent506f1}
Such That   coveredEntity in Roles(a)
   if       grant-consent $\mathbf{in}_a$ P
   and      s.consent-in-writing = true
   and      a.will-record = true
   and      Maintain530j(a, s, r, P, f, f', msg) $\in$ {Allow, Don't Care/Allow,
            Don't Care/Forbid}
   then     return true
   else     return false
```

```
CMD         RecordConsent506b6 (a, s, r, P, f, f', msg)
   if       consent $\mathbf{in}_a$ P
   then     insert "Consent granted" in log
   and      invoke Record530j(a, s, r, P, f, f', msg)
   and      return true
   else     return false
```

§164.506(c)

```
CST    ConsentContent506c (a, s, r, P, f, f’, msg)
Scope  {GrantConsent506a, OptionalConsent506a4,
       CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
if     a.consent-plain-language = true
and    ConsentContent506c1(a, s, r, P, f, f’, msg) ∈ {Allow}
and    ConsentContent506c2(a, s, r, P, f, f’, msg) ∈ {Allow}
and    ConsentContent506c3(a, s, r, P, f, f’, msg) ∈ {Allow}
and    ConsentContent506c4(a, s, r, P, f, f’, msg) ∈ {Allow}
and    ConsentContent506c5(a, s, r, P, f, f’, msg) ∈ {Allow}
and    ConsentContent506c6(a, s, r, P, f, f’, msg) ∈ {Allow}
then   return true
else   return false
```

§164.506(c)(1)

```
CST    ConsentContent506c1 (a, s, r, P, f, f’, msg)
Scope  {GrantConsent506a, OptionalConsent506a4,
       CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
if     a.informs-may-be-used-for-treatment = true
and    a.informs-may-be-used-for-payment = true
and    a.informs-may-be-used-for-health-care-operations = true
then   return true
else   return false
```

§164.506(c)(2)

```
CST        ConsentContent506c2 (a, s, r, P, f, f', msg)
Scope      {GrantConsent506a, OptionalConsent506a4,
           CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
if         a.consent-refers-to-notice = true
and        a.consent-states-right-to-review = true
then       return true
else       return false
```

§164.506(c)(3)

```
CST        ConsentContent506c3 (a, s, r, P, f, f', msg)
Scope      {GrantConsent506a, OptionalConsent506a4, CombineConsent506b4i,
           CombinedConsent506b4ii, GrantJointConsent506f1}
Such That  a.reserves-right-to-update-notice = true
if         a.states-right-to-modify = true
and        a.describes-how-to-obtain-new = true
then       return true
else       return false
```

§164.506(c)(4)

```
CST        ConsentContent506c4 (a, s, r, P, f, f', msg)
Scope      {GrantConsent506a, OptionalConsent506a4,
           CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
if         ConsentContent506c4i(a, s, r, P, f, f', msg) ∈ {Allow}
and        ConsentContent506c4ii(a, s, r, P, f, f', msg) ∈ {Allow}
and        ConsentContent506c4iii(a, s, r, P, f, f', msg) ∈ {Allow}
then       return true
else       return false
```

§**164.506(c)(4)(i)**

```
 CST   ConsentContent506c4i (a, s, r, P, f, f', msg)
Scope  {GrantConsent506a, OptionalConsent506a4,
       CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if  a.consent-states-may-request-restriction = true
 then  return true
 else  return false
```

§**164.506(c)(4)(ii)**

```
 CST   ConsentContent506c4ii (a, s, r, P, f, f', msg)
Scope  {GrantConsent506a, OptionalConsent506a4,
       CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if  a.consent-states-entity-not-required-to-agree = true
 then  return true
 else  return false
```

§**164.506(c)(4)(iii)**

```
 CST   ConsentContent506c4iii (a, s, r, P, f, f', msg)
Scope  {GrantConsent506a, OptionalConsent506a4,
       CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if  a.consent-states-if-entity-agrees-must-comply = true
 then  return true
 else  return false
```

§164.506(c)(5)

```
CST    ConsentContent506c5 (a, s, r, P, f, f', msg)
Scope  {GrantConsent506a, OptionalConsent506a4,
       CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if  a.consent-states-individual-may-revoke = true
  and  a.consent-states-revocation-only-for-future = true
 then  return true
 else  return false
```

§164.506(c)(6)

```
CST    ValidConsent506c6 (a, s, r, P, f, f', msg)
Scope  {GrantConsent506a, OptionalConsent506a4,
       CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if  a.consent-is-signed = true
  and  a.consent-is-dated = true
 then  return true
 else  return false
```

§164.506(d)

```
CST    DefectiveConsent506d (a, s, r, P, f, f', msg)
Scope  {GrantConsent506a, OptionalConsent506a4,
       CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if  DefectiveConsent506d1(a, s, r, P, f, f', msg)
   or  DefectiveConsent506d2(a, s, r, P, f, f', msg)
 then  return true
 else  return false
```

§164.506(d)(1)

```
  CST   DefectiveConsent506d1 (a, s, r, P, f, f', msg)
Scope   {GrantConsent506a, OptionalConsent506a4,
        CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if   ConsentContent506c (a, s, r, P, f, f', msg) $\in$ {Allow}
 then   return true
 else   return false
```

§164.506(d)(2)

```
  CST   DefectiveConsent506d2 (a, s, r, P, f, f', msg)
Scope   {GrantConsent506a, OptionalConsent506a4,
        CombineConsent506b4i, CombinedConsent506b4ii, GrantJointConsent506f1}
   if   s.revoked-consent = false
 then   return true
 else   return false
```

§164.506(e)

No Commands

§164.506(f)(1)

```
  CST   JointConsent506f1 (a, s, r, P, f, f', msg)
Scope   {GrantJointConsent506f1}
   if   organizedHealthCareArrangement in (a, r)
  and   a.joint-notice = true
  and   r.joint-notice = true
 then   return true
 else   return false
```

```
     CMD   GrantJointConsent506f1 (a, s, r, P, f, f’, msg)
      if   JointConsent506f1 (a, s, r, P, f, f’, msg) $\in$ {Allow}
     and   JointConsentContent506f2(a, s, r, P, f, f’, msg) $\in$ {Allow}
     and   jointConsent **in**$_a$ P
    then   **insert** consent **in** (a, s)
     and   **insert** consent **in** (r, s)
     and   **insert** “Joint consent granted” **in log**
     and   RecordConsent506b6 (a, s, r, P, f, f’, msg)
     and   **return true**
    else   **return false**
```

§164.506(f)(2)

```
     CST   JointConsentContent506f2 (a, s, r, P, f, f’, msg)
   Scope   {GrantJointConsent506f1}
      if   JointConsentContent506f2iA (a, s, r, P, f, f’, msg) $\in$ {Allow}
     and   JointConsentContent506f2iB (a, s, r, P, f, f’, msg) $\in$ {Allow,
           Don’t Care/Allow}
    then   **return true**
    else   **return false**
```

§164.506(f)(2)(i)(A)

```
         CST   JointConsentContent506f2iA (a, s, r, P, f, f’, msg)
       Scope   {GrantJointConsent506f1}
   Such That   jointConsent **in**$_a$ P
          if   a.consent-includes-names = true
        then   **return true**
        else   **return false**
         CST   JointConsentContent506f2iA (a, s, r, P, f, f', msg)
       Scope   {GrantJointConsent506f1}
   Such That   jointConsent $\textbf{in}_a$ P
          if   a.consent-includes-specific-identifiers = true
        then   return true
        else   return false
```

```
       CST   JointConsentContent506f2iA (a, s, r, P, f, f', msg)
     Scope   {GrantJointConsent506f1}
 Such That   jointConsent $\textbf{in}_a$ P
        if   a.consent-includes-classes = true
      then   return true
      else   return false
```

§164.506(f)(2)(i)(B)

```
       CST   JointConsentContent506f2iB (a, s, r, P, f, f', msg)
     Scope   {GrantJointConsent506f1}
 Such That   jointConsent $\textbf{in}_a$ P
       and   a.consent-changed-to-multiple-entities = true
        if   ConsentContent506c (a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false
```

§**164.506(f)(2)(ii)**

```
CMD    RevokeJointConsent506f2ii (a, s, r, P, f, f', msg)
  if   revoke in_a P
 and   jointConsent in_a P
then   invoke RevokeConsent506b5 (a, s, r, P, f, f', msg)
 and   inform r of "Consent revoked"
 and   invoke Record530j(a, s, r, P, f, f', msg)
 and   return true
else   return false
```

§**164.530(j)**

```
  CST   Maintain530j(a, s, r, P, f, f', msg)
Scope   {RevokeConsent506b5, RevokeJointConsent506f2ii, GrantConsent506a,
        OptionalConsent506a4, CombineConsent506b4i, CombinedConsent506b4ii,
        GrantJointConsent506f1}
   if   Policies530ji(a, s, r, P, f, f', msg) ∈ {Allow, Don't Care/Allow,
        Don't Care/Forbid}
  and   MaintainWritten530jii(a, s, r, P, f, f', msg) ∈ {Allow, Don't Care/Allow,
        Don't Care/Forbid}
 then   return true
 else   return false
```

§**164.530(j)(i)**

```
      CST   Policies530ji(a, s, r, P, f, f', msg)
Such That   a.policies-mentioned-in-530i = true
       if   a.maintain-policies-in-electronic-form = true
     then   return true
     else   return false
```

§164.530(j)(ii)

```
       CST  MaintainWritten530jii(a, s, r, P, f, f', msg)
     Scope  {RevokeConsent506b5, RevokeJointConsent506f2ii, GrantConsent506a,
            OptionalConsent506a4, CombineConsent506b4i,
            CombinedConsent506b4ii, GrantJointConsent506f1}
 Such That  a.in-writing = true
       and  communication $\mathbf{in}_f$ P
       and  a.must-be-in-writing = true
        if  a.will-record = true
       and  a.record-in-writing = true
      then  return true
      else  return false

       CST  MaintainWritten530jii(a, s, r, P, f, f', msg)
     Scope  {RevokeConsent506b5, RevokeJointConsent506f2ii, GrantConsent506a,
            OptionalConsent506a4, CombineConsent506b4i,
            CombinedConsent506b4ii, GrantJointConsent506f1}
 Such That  a.in-writing = true
       and  communication $\mathbf{in}_f$ P
       and  a.must-be-in-writing = true
        if  a.will-record = true
       and  a.record-electronically = true
      then  return true
      else  return false

 CMD  Record530j(a, s, r, P, f, f', msg)
  if  true
then  insert msg in log
 and  return true
else  return false
```

C.2 Privacy Rule 2003 Privacy API

The Privacy API as derived for the Privacy Rule [§164.506, v.2000] section on consent for treatment, payment, and health care operations is as follows. For reference, the full regulatory text for the section and other essential texts are presented in Section B.2.

C.2.1 Health Care Operations Hierarchy

The purpose hierarchy for health care operations as mentioned above in Section 7.1.5 is as follows. Note that the top level purpose health care operations is the parent of Paragraphs 1–6 below although it is not listed. The hierarchy is as defined in Section B.2.2.

Paragraph 1
- (a) Conducting quality assessment and improvement activities
  - i. Outcomes evaluation
  - ii. Development of clinical guidelines
- (b) Population-based activities relating to improving health or reducing health care costs
- (c) Protocol development
- (d) Case management and care coordination
- (e) Contacting of health care providers and patients with information about treatment alternatives
- (f) Related functions that do not include treatment

Paragraph 2
- (a) Reviewing the competence or qualifications of health care professionals
- (b) Evaluating practitioner and provider performance
- (c) Health plan performance
- (d) Conducting training programs in which students, trainees, or practitioners in areas of health care learn under supervision to practice or improve their skills as health care providers
- (e) Training of non-health care professionals
- (f) Accreditation

(g) Certification

(h) Licensing

(i) Credentialing activities

Paragraph 3 (a) Underwriting

(b) Premium rating

(c) Other activities relating to the creation, renewal or replacement of a contract of health insurance or health benefits

(d) Ceding, securing, or placing a contract for reinsurance of risk relating to claims for health care provided that the requirements of Sec. 164.514(g) are met, if applicable;

i. Stop-loss insurance and excess of loss insurance

Paragraph 4 (a) Conducting or arranging for medical review

(b) Conducting or arranging for legal services

(c) Conducting or arranging for auditing functions

i. Fraud and abuse detection

ii. Compliance programs

Paragraph 5 Business planning and development

(a) Conducting cost-management and planning-related analyses related to managing and operating the entity

(b) Formulary development and administration

(c) Development or improvement of methods of payment or coverage policies

Paragraph 6 Business management and general administrative activities of the entity

Subparagraph (i) i. Management activities relating to implementation of and compliance with the requirements of this subchapter;

Subparagraph (ii)	i. Customer service, including the provision of data analyses for policy holders, plan sponsors, or other customers, provided that protected health information is not disclosed to such policy holder, plan sponsor, or customer.
Subparagraph (iii)	i. Resolution of internal grievances;
Subparagraph (iv)	i. The sale, transfer, merger, or consolidation of all or part of the covered entity with another covered entity, or an entity that following such activity will become a covered entity and due diligence related to such activity
Subparagraph (v)	i. Consistent with the applicable requirements of Sec. 164.514, creating de-identified health information or a limited data set, and fundraising for the benefit of the covered entity.

C.2.2 Commands and Constraints

The commands and constraints for the Privacy Rule 2003 are as follows.

§164.506(a)

CST	Permitted506a(a, s, r, P, f, f', msg)
Scope	{TreatmentUse506c1, PaymentUse506c1, HealthCareOperationsUse506c1, TreatmentDisclose506c1, PaymentDisclose506c1, TreatmentDisclose506c1, HealthCareOperationsDisclose506c1, PaymentDisclose506c3, Disclose506c4, Disclose506c5}
Such That	Permitted508a2(a, s, r, P, f, f', msg) $\in$ {Allow, Don't Care/Allow, Don't Care/Forbid}
and	Permitted508a3(a, s, r, P, f, f', msg) $\in$ {Allow, Don't Care/Allow, Don't Care/Forbid}
and	f.protected-health-information = true
and	coveredEntity **in** Roles(a)
if	Permitted506c(a, s, r, P, f, f', msg) $\in$ {Allow}
then	**return true**
else	**return false**

§164.506(b)(1)

CST	OptionalConsent506b1(a, s, r, P, f, f', msg)
Scope	{OptionalUseConsent506b1, OptionalDiscloseConsent506b1}
Such That	a.optional-consent = true
and	coveredEntity **in** Roles(a)
if	treatment $\textbf{in}_a$ P
and	consent **in** (a, s)
and	**return true**
else	**return false**

```
CMD   OptionalUseConsent506b1 (a, s, r, P, f, f', msg)
 if   individual in Roles(s)
and   local in (a, f)
and   use $\mathbf{in}_a$ P
then  insert consent in (a, s)
and   return true
else  return false
```

```
CMD   OptionalDiscloseConsent506b1 (a, s, r, P, f, f', msg)
 if   individual in Roles(s)
and   local in (a, f)
and   disclose $\mathbf{in}_a$ P
then  insert consent in (a, s)
and   return true
else  return false
```

164.506(b)(2)

```
      CST   ConsentValidAsIn506b (a, s, r, P, f, f', msg)
    Scope   {OptionalUseConsent506b1, OptionalDiscloseConsent506b1}
Such That   consent $\mathbf{in}_a$ P
       if   Permitted508 (a, s, r, P, f, f', msg) $\in$ {Allow, Don't Care/Allow,
            Don't Care/Forbid}
     then   return true
     else   return false
```

§164.506(c)

```
 CST   Permitted506c (a, s, r, P, f, f', msg)
Scope  {TreatmentUse506c1, PaymentUse506c1, HealthCareOperationsUse506c1,
       TreatmentDisclose506c1, PaymentDisclose506c1,
       HealthCareOperationsDisclose506c1, TreatmentDisclose506c1,
       PaymentDisclose506c3, Disclose506c4, Disclose506c5}
   if  Permitted506c1 (a, s, r, P, f, f', msg) ∈ {Allow}
 then  return true
 else  return false

 CST   Permitted506c (a, s, r, P, f, f', msg)
Scope  {TreatmentUse506c1, PaymentUse506c1, HealthCareOperationsUse506c1,
       TreatmentDisclose506c1, PaymentDisclose506c1,
       HealthCareOperationsDisclose506c1, TreatmentDisclose506c1,
       PaymentDisclose506c3, Disclose506c4, Disclose506c5}
   if  Permitted506c2 (a, s, r, P, f, f', msg) ∈ {Allow}
 then  return true
 else  return false

 CST   Permitted506c (a, s, r, P, f, f', msg)
Scope  {TreatmentUse506c1, PaymentUse506c1, HealthCareOperationsUse506c1,
       TreatmentDisclose506c1, PaymentDisclose506c1,
       HealthCareOperationsDisclose506c1, TreatmentDisclose506c1,
       PaymentDisclose506c3, Disclose506c4, Disclose506c5}
   if  Permitted506c3 (a, s, r, P, f, f', msg) ∈ {Allow}
 then  return true
 else  return false
```

```
      CST   Permitted506c (a, s, r, P, f, f', msg)
    Scope   {TreatmentUse506c1, PaymentUse506c1, HealthCareOperationsUse506c1,
            TreatmentDisclose506c1, PaymentDisclose506c1,
            HealthCareOperationsDisclose506c1, TreatmentDisclose506c1,
            PaymentDisclose506c3, Disclose506c4, Disclose506c5}
       if   Permitted506c4 (a, s, r, P, f, f', msg) $\in$ {Allow}
     then   return true
     else   return false

      CST   Permitted506c (a, s, r, P, f, f', msg)
    Scope   {TreatmentUse506c1, PaymentUse506c1, HealthCareOperationsUse506c1,
            TreatmentDisclose506c1, PaymentDisclose506c1,
            HealthCareOperationsDisclose506c1, TreatmentDisclose506c1,
            PaymentDisclose506c3, Disclose506c4, Disclose506c5}
       if   Permitted506c5 (a, s, r, P, f, f', msg) $\in$ {Allow}
     then   return true
     else   return false
```

§164.506(c)(1)

```
         CST   Permitted506c1 (a, s, r, P, f, f', msg)
       Scope   {TreatmentUse506c1, PaymentUse506c1,
               HealthCareOperationsUse506c1, TreatmentDisclose506c1,
               PaymentDisclose506c1, HealthCareOperationsDisclose506c1}
   Such That   f.protected-health-information = true
         and   coveredEntity in Roles(a)
         and   individual in Roles(s)
          if   own $\text{in}_a$ P
        then   return true
        else   return false
```

```
CMD   TreatmentUse506c1 (a, s, r, P, f, f', msg)
  if  Permitted506c1 (a, s, r, P, f, f', msg) $\in$ {Allow}
 and  local in (a, f)
 and  use $\mathbf{in}_a$ P
 and  treatment $\mathbf{in}_a$ P
then  insert treatment in (a, s)
 and  return true
else  return false

CMD   PaymentUse506c1 (a, s, r, P, f, f', msg)
  if  Permitted506c1 (a, s, r, P, f, f', msg)
 and  local in (a, f)
 and  use $\mathbf{in}_a$ P
 and  payment $\mathbf{in}_a$ P
then  insert payment in (a, s)
 and  return true
else  return false

CMD   HealthCareOperationsUse506c1 (a, s, r, P, f, f', msg)
  if  Permitted506c1 (a, s, r, P, f, f', msg)
 and  local in (a, f)
 and  use $\mathbf{in}_a$ P
 and  healthCareOperations $\mathbf{in}_a$ P
then  insert healthCareOperations in (a, s)
 and  return true
else  return false
```

```
CMD   TreatmentDisclose506c1 (a, s, r, P, f, f', msg)
 if   Permitted506c1(a, s, r, P, f, f', msg) $\in$ {Allow}
and   coveredEntity in Roles(a)
and   local in (a, f)
and   treatment $\mathbf{in}_a$ P
and   disclose $\mathbf{in}_a$ P
then  insert local in (r, f)
and   return true
else  return false

CMD   PaymentDisclose506c1 (a, s, r, P, f, f', msg)
 if   Permitted506c1(a, s, r, P, f, f', msg) $\in$ {Allow}
and   coveredEntity in Roles(a)
and   local in (a, f)
and   payment $\mathbf{in}_a$ P
and   disclose $\mathbf{in}_a$ P
then  insert local in (r, f)
and   return true
else  return false

CMD   HealthCareOperationsDisclose506c1 (a, s, r, P, f, f', msg)
 if   Permitted506c1(a, s, r, P, f, f', msg) $\in$ {Allow}
and   coveredEntity in Roles(a)
and   local in (a, f)
and   healthCareOperations $\mathbf{in}_a$ P
and   disclose $\mathbf{in}_a$ P
then  insert local in (r, f)
and   return true
else  return false
```

§164.506(c)(2)

```
       CST  Permitted506c2 (a, s, r, P, f, f', msg)
     Scope  {TreatmentDisclose506c2}
 Such That  coveredEntity in Roles(a)
       and  f.protected-health-information = true
        if  healthCareProvider in Roles(r)
       and  treatment $\mathbf{in}_a$ P
       and  recipient $\mathbf{in}_a$ P
      then  return true
      else  return false

 CMD   TreatmentDisclose506c2 (a, s, r, P, f, f', msg)
   if  Permitted506c2 (a, s, r, P, f, f', msg)$\in$ {Allow}
  and  local in (a, f)
  and  treatment $\mathbf{in}_a$ P
  and  recipient $\mathbf{in}_a$ P
  and  disclose $\mathbf{in}_a$ P
 then  insert treatment in (a, s)
  and  insert local in (r, f)
  and  return true
 else  return false
```

§164.506(c)(3)

```
       CST   Permitted506c3 (a, s, r, P, f, f', msg)
     Scope   {PaymentDisclose506c3}
 Such That   coveredEntity in Roles(a)
       and   f.protected-health-information = true
        if   coveredEntity in Roles(r)
       and   payment $\mathbf{in}_a$ P
       and   recipient $\mathbf{in}_a$ P
      then   return true
      else   return false

       CST   Permitted506c3 (a, s, r, P, f, f', msg)
     Scope   {PaymentDisclose506c3}
 Such That   coveredEntity in Roles(a)
       and   f.protected-health-information = true
        if   healthCareProvider in Roles(r)
       and   payment $\mathbf{in}_a$ P
       and   recipient $\mathbf{in}_a$ P
      then   return true
      else   return false

 CMD   PaymentDisclose506c3 (a, s, r, P, f, f', msg)
  if   Permitted506c3 (a, s, r, P, f, f', msg)$\in$ {Allow}
 and   local in (a, f)
 and   payment $\mathbf{in}_a$ P
 and   recipient $\mathbf{in}_a$ P
 and   disclose $\mathbf{in}_a$ P
then   insert local in (r, f)
 and   insert payment in (r, f)
 and   return true
else   return false
```

§164.506(c)(4)

```
       CST   Permitted506c4 (a, s, r, P, f, f', msg)
     Scope   {Disclose506c4}
 Such That   f.protected-health-information = true
       and   coveredEntity in Roles(a)
       and   coveredEntity in Roles(r)
       and   relationship in (a, s)
       and   relationship in (r, s)
        if   Permitted506c4i (a, s, r, P, f, f', msg) $\in$ {Allow}
       and   healthCareOperations $\mathbf{in}_a$ P
       and   recipient $\mathbf{in}_a$ P
       and   relatedRelationship $\mathbf{in}_a$ P
      then   return true
      else   return false

       CST   Permitted506c4 (a, s, r, P, f, f', msg)
     Scope   {Disclose506c4}
 Such That   f.protected-health-information = true
       and   coveredEntity in Roles(a)
       and   coveredEntity in Roles(r)
       and   relationship in (a, s)
       and   relationship in (r, s)
        if   Permitted506c4ii (a, s, r, P, f, f', msg)$\in$ {Allow}
       and   healthCareOperations $\mathbf{in}_a$ P
       and   recipient $\mathbf{in}_a$ P
       and   relatedRelationship $\mathbf{in}_a$ P
      then   return true
      else   return false
```

```
       CST  Permitted506c4 (a, s, r, P, f, f', msg)
     Scope  {Disclose506c4}
 Such That  f.protected-health-information = true
       and  coveredEntity in Roles(a)
       and  coveredEntity in Roles(r)
       and  a.past-relationship = true
       and  relationship in (r, s)
        if  Permitted506c4i (a, s, r, P, f, f', msg) $\in$ {Allow}
       and  healthCareOperations $\mathbf{in}_a$ P
       and  recipient $\mathbf{in}_a$ P
       and  relatedRelationship $\mathbf{in}_a$ P
      then  return true
      else  return false

       CST  Permitted506c4 (a, s, r, P, f, f', msg)
     Scope  {Disclose506c4}
 Such That  f.protected-health-information = true
       and  coveredEntity in Roles(a)
       and  coveredEntity in Roles(r)
       and  a.past-relationship = true
       and  relationship in (r, s)
        if  Permitted506c4ii (a, s, r, P, f, f', msg) $\in$ {Allow}
       and  healthCareOperations $\mathbf{in}_a$ P
       and  recipient $\mathbf{in}_a$ P
       and  relatedRelationship $\mathbf{in}_a$ P
      then  return true
      else  return false
      CST  Permitted506c4 (a, s, r, P, f, f', msg)
    Scope  {Disclose506c4}
Such That  f.protected-health-information = true
      and  coveredEntity **in** Roles(a)
      and  coveredEntity **in** Roles(r)
      and  relationship **in** (a, s)
      and  r.past-relationship = true
       if  Permitted506c4i (a, s, r, P, f, f', msg)$\in$ {Allow}
      and  healthCareOperations $\mathbf{in}_a$ P
      and  recipient $\mathbf{in}_a$ P
      and  relatedRelationship $\mathbf{in}_a$ P
     then  return true
     else  return false

      CST  Permitted506c4 (a, s, r, P, f, f', msg)
    Scope  {Disclose506c4}
Such That  f.protected-health-information = true
      and  coveredEntity **in** Roles(a)
      and  coveredEntity **in** Roles(r)
      and  relationship **in** (a, s)
      and  r.past-relationship = true
       if  Permitted506c4ii (a, s, r, P, f, f', msg)$\in$ {Allow}
      and  healthCareOperations $\mathbf{in}_a$ P
      and  recipient $\mathbf{in}_a$ P
      and  relatedRelationship $\mathbf{in}_a$ P
     then  return true
     else  return false
```

```
       CST   Permitted506c4 (a, s, r, P, f, f', msg)
     Scope   {Disclose506c4}
 Such That   f.protected-health-information = true
       and   coveredEntity in Roles(a)
       and   coveredEntity in Roles(r)
       and   a.past-relationship = true
       and   r.past-relationship = true
        if   Permitted506c4i (a, s, r, P, f, f', msg) ∈ {Allow}
       and   healthCareOperations $\textbf{in}_a$ P
       and   recipient $\textbf{in}_a$ P
       and   relatedRelationship $\textbf{in}_a$ P
      then   return true
      else   return false
```

```
       CST   Permitted506c4 (a, s, r, P, f, f', msg)
     Scope   {Disclose506c4}
 Such That   f.protected-health-information = true
       and   coveredEntity in Roles(a)
       and   coveredEntity in Roles(r)
       and   a.past-relationship = true
       and   r.past-relationship = true
        if   Permitted506c4ii (a, s, r, P, f, f', msg) ∈ {Allow}
       and   healthCareOperations $\textbf{in}_a$ P
       and   recipient $\textbf{in}_a$ P
       and   relatedRelationship $\textbf{in}_a$ P
      then   return true
      else   return false
```

```
CMD   Disclose506c4 (a, s, r, P, f, f', msg)
 if   Permitted506c4 (a, s, r, P, f, f', msg) ∈ {Allow}
and   local in (a, f)
and   coveredEntity in Roles(a)
and   coveredEntity in Roles(r)
then  insert local in (r, f)
and   insert healthCareOperations in (r, f)
and   return true
else  return false
```

§164.506(c)(4)(i)

```
  CST   Permitted506c4i (a, s, r, P, f, f', msg)
Scope   {}
   if   Paragraph1 $\textbf{in}_a$ P
 then   return true
 else   return false

  CST   Permitted506c4i (a, s, r, P, f, f', msg)
Scope   {}
   if   Paragraph2 $\textbf{in}_a$ P
 then   return true
 else   return false
```

§164.506(c)(4)(ii)

```
  CST   Permitted506c4ii (a, s, r, P, f, f', msg)
Scope   {}
   if   healthCareFraudAbuseDetection $\textbf{in}_a$ P
 then   return true
 else   return false
```

```
       CST   Permitted506c4ii (a, s, r, P, f, f', msg)
     Scope   {}
        if   compliance in_a P
      then   return true
      else   return false
```

§164.506(c)(5)

```
       CST   Permitted506c5 (a, s, r, P, f, f', msg)
     Scope   {Disclose506c5}
 Such That   organizedHealthCareArrangement in (a, r)
       and   f.protected-health-information = true
       and   organizedArrangement in_a P
        if   healthCareOperations in_a P
       and   individual in Roles(s)
       and   coveredEntity in Roles(a)
       and   coveredEntity in Roles(r)
      then   return true
      else   return false

  CMD   Disclose506c5 (a, s, r, P, f, f', msg)
   if   Permitted506c5 (a, s, r, P, f, f', msg) ∈ {Allow}
  and   local in (a, f)
  and   healthCareOperations in_a P
 then   insert local in (r, f)
  and   return true
 else   return false
```

§164.508(a)(2)

```
       CST   Permitted508a2 (a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506c1, PaymentUse506c1,
             HealthCareOperationsUse506c1, TreatmentDisclose506c1,
             PaymentDisclose506c1, TreatmentDisclose506c1,
             HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
             Disclose506c4, Disclose506c5}
 Such That   f.psychotherapy-notes = true
       and   use $\mathbf{in}_a$ P
        if   Transition532(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false

       CST   Permitted508a2 (a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506c1, PaymentUse506c1,
             HealthCareOperationsUse506c1, TreatmentDisclose506c1,
             PaymentDisclose506c1, TreatmentDisclose506c1,
             HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
             Disclose506c4, Disclose506c5}
 Such That   f.psychotherapy-notes = true
       and   disclose $\mathbf{in}_a$ P
        if   Transition532(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false
```

```
CST        Permitted508a2 (a, s, r, P, f, f', msg)
Scope      {TreatmentUse506c1, PaymentUse506c1,
           HealthCareOperationsUse506c1, TreatmentDisclose506c1,
           PaymentDisclose506c1, TreatmentDisclose506c1,
           HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
           Disclose506c4, Disclose506c5}
Such That  f.psychotherapy-notes = true
and        use in_a P
if         authorization in (a, s)
then       return true
else       return false
```

```
CST        Permitted508a2 (a, s, r, P, f, f', msg)
Scope      {TreatmentUse506c1, PaymentUse506c1,
           HealthCareOperationsUse506c1, TreatmentDisclose506c1,
           PaymentDisclose506c1, TreatmentDisclose506c1,
           HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
           Disclose506c4, Disclose506c5}
Such That  f.psychotherapy-notes = true
and        disclose in_a P
if         authorization in (a, s)
then       return true
else       return false
```

```
       CST   Permitted508a2 (a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506c1, PaymentUse506c1,
             HealthCareOperationsUse506c1, TreatmentDisclose506c1,
             PaymentDisclose506c1, TreatmentDisclose506c1,
             HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
             Disclose506c4, Disclose506c5}
 Such That   f.psychotherapy-notes = true
       and   use $\mathbf{in}_a$ P
        if   Permitted508a2i(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false

       CST   Permitted508a2 (a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506c1, PaymentUse506c1,
             HealthCareOperationsUse506c1, TreatmentDisclose506c1,
             PaymentDisclose506c1, TreatmentDisclose506c1,
             HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
             Disclose506c4, Disclose506c5}
 Such That   f.psychotherapy-notes = true
       and   disclose $\mathbf{in}_a$ P
        if   Permitted508a2i(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false
```

```
       CST   Permitted508a2 (a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506c1, PaymentUse506c1,
             HealthCareOperationsUse506c1, TreatmentDisclose506c1,
             PaymentDisclose506c1, TreatmentDisclose506c1,
             HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
             Disclose506c4, Disclose506c5}
 Such That   f.psychotherapy-notes = true
       and   use $\mathbf{in}_a$ P
        if   Permitted508a2ii(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false
```

```
       CST   Permitted508a2 (a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506c1, PaymentUse506c1,
             HealthCareOperationsUse506c1, TreatmentDisclose506c1,
             PaymentDisclose506c1, TreatmentDisclose506c1,
             HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
             Disclose506c4, Disclose506c5}
 Such That   f.psychotherapy-notes = true
       and   disclose $\mathbf{in}_a$ P
        if   Permitted508a2ii(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false
```

§**164.508(a)(2)(i)**

```
       CST   Authorization508a2i (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
        if   Notes508a2iA(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false
       CST   Authorization508a2i (a, s, r, P, f, f', msg)
     Scope   {}
 Such That  f.psychotherapy-notes = true
        if   Notes508a2iB(a, s, r, P, f, f', msg)$\in$ {Allow}
      then   return true
      else   return false
```

```
     CST   Authorization508a2i (a, s, r, P, f, f', msg)
   Scope   {}
Such That  f.psychotherapy-notes = true
      if   Notes508a2iC(a, s, r, P, f, f', msg)$\in$ {Allow}
    then   return true
    else   return false
```

§164.508(a)(2)(i)(A)

```
     CST   Notes508a2iA (a, s, r, P, f, f', msg)
   Scope   {}
Such That  f.psychotherapy-notes = true
      if   originator in (a, f)
     and   treatment $\text{in}_a$ P
     and   use $\text{in}_a$ P
    then   return true
    else   return false
```

§**164.508(a)(2)(i)(B)**

```
CST         Notes508a2iB (a, s, r, P, f, f', msg)
Scope       {}
Such That   f.psychotherapy-notes = true
if          local in (a, f)
and         coveredEntity in Roles(a)
and         own $\mathbf{in}_a$ P
and         improveCounseling $\mathbf{in}_a$ P
and         use $\mathbf{in}_a$ P
then        return true
else        return false
```

```
CST         Notes508a2iB (a, s, r, P, f, f', msg)
Scope       {}
Such That   f.psychotherapy-notes = true
if          local in (a, f)
and         coveredEntity in Roles(a)
and         own $\mathbf{in}_a$ P
and         improveCounseling $\mathbf{in}_a$ P
and         disclose $\mathbf{in}_a$ P
then        return true
else        return false
```

§164.508(a)(2)(i)(C)

```
CST         Notes508a2iC (a, s, r, P, f, f', msg)
Scope       {}
Such That   f.psychotherapy-notes = true
      and   defendFromIndividual in$_a$ P
      and   legalAction in$_a$ P
      and   coveredEntity in Roles(a)
       if   local in (a, f)
      and   use in$_a$ P
     then   return true
     else   return false
```

```
CST         Notes508a2iC (a, s, r, P, f, f', msg)
Scope       {}
Such That   f.psychotherapy-notes = true
      and   defendFromIndividual in$_a$ P
      and   legalAction in$_a$ P
      and   coveredEntity in Roles(a)
       if   local in (a, f)
      and   disclose in$_a$ P
     then   return true
     else   return false
```

```
       CST   Notes508a2iC (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   defendFromIndividual $\mathbf{in}_a$ P
       and   proceedings $\mathbf{in}_a$ P
       and   coveredEntity in Roles(a)
        if   local in (a, f)
       and   use $\mathbf{in}_a$ P
      then   return true
      else   return false
```

```
       CST   Notes508a2iC (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   defendFromIndividual $\mathbf{in}_a$ P
       and   proceedings $\mathbf{in}_a$ P
       and   coveredEntity in Roles(a)
        if   local in (a, f)
       and   disclose $\mathbf{in}_a$ P
      then   return true
      else   return false
```

§**164.508(a)(2)(ii)**

```
       CST   Notes508a2ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   Required502a2ii (a, s, r, P, f, f', msg) $\in$ {Allow}
        if   use $\mathbf{in}_a$ P
      then   return true
      else   return false
```

```
       CST   Notes508a2ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   Required502a2ii (a, s, r, P, f, f', msg) ∈ {Allow}
        if   disclose $\mathbf{in}_a$ P
      then   return true
      else   return false

       CST   Notes508a2ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   Permitted512a (a, s, r, P, f, f', msg) ∈ {Allow}
        if   use $\mathbf{in}_a$ P
      then   return true
      else   return false

       CST   Notes508a2ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   Permitted512a (a, s, r, P, f, f', msg) ∈ {Allow}
        if   disclose $\mathbf{in}_a$ P
      then   return true
      else   return false

       CST   Notes508a2ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   Permitted512d (a, s, r, P, f, f', msg) ∈ {Allow}
       and   notesOriginator $\mathbf{in}_a$ P
        if   use $\mathbf{in}_a$ P
      then   return true
      else   return false
```

```
       CST   Notes508a2ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   Permitted512d (a, s, r, P, f, f', msg) $\in$ {Allow}
       and   notesOriginator $\mathbf{in}_a$ P
        if   disclose $\mathbf{in}_a$ P
      then   return true
      else   return false

       CST   Notes508a2ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   Permitted512g1 (a, s, r, P, f, f', msg) $\in$ {Allow}
        if   use $\mathbf{in}_a$ P
      then   return true
      else   return false

       CST   Notes508a2ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   Permitted512g1 (a, s, r, P, f, f', msg) $\in$ {Allow}
        if   disclose $\mathbf{in}_a$ P
      then   return true
      else   return false

       CST   Notes508a2ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   f.psychotherapy-notes = true
       and   Permitted512j1i (a, s, r, P, f, f', msg) $\in$ {Allow}
        if   use $\mathbf{in}_a$ P
      then   return true
      else   return false
      CST   Notes508a2ii (a, s, r, P, f, f', msg)
    Scope   {}
Such That   f.psychotherapy-notes = true
      and   Permitted512j1i (a, s, r, P, f, f', msg) $\in$ {Allow}
       if   disclose $\mathbf{in}_a$ P
     then   return true
     else   return false
```

§164.508(a)(3)(i)

```
      CST   Marketing508a3i (a, s, r, P, f, f', msg)
    Scope   {TreatmentUse506c1, PaymentUse506c1,
            HealthCareOperationsUse506c1, TreatmentDisclose506c1,
            PaymentDisclose506c1, TreatmentDisclose506c1,
            HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
            Disclose506c4, Disclose506c5}
Such That   f.protected-health-information = true
      and   coveredEntity in Roles(a)
      and   use $\mathbf{in}_a$ P
      and   marketing $\mathbf{in}_f$ P
       if   Transition532(a, s, r, P, f, f', msg) $\in$ {Allow}
     then   return true
     else   return false
```

```
       CST   Marketing508a3i (a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506c1, PaymentUse506c1,
             HealthCareOperationsUse506c1, TreatmentDisclose506c1,
             PaymentDisclose506c1, TreatmentDisclose506c1,
             HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
             Disclose506c4, Disclose506c5}
 Such That   f.protected-health-information = true
       and   coveredEntity in Roles(a)
       and   disclose $\mathbf{in}_a$ P
       and   marketing $\mathbf{in}_f$ P
        if   Transition532(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false

       CST   Marketing508a3i (a, s, r, P, f, f', msg)
     Scope   {TreatmentUse506c1, PaymentUse506c1,
             HealthCareOperationsUse506c1, TreatmentDisclose506c1,
             PaymentDisclose506c1, TreatmentDisclose506c1,
             HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
             Disclose506c4, Disclose506c5}
 Such That   f.protected-health-information = true
       and   coveredEntity in Roles(a)
       and   use $\mathbf{in}_a$ P
       and   marketing $\mathbf{in}_f$ P
        if   authorization in (a, s)
      then   return true
      else   return false
```

```
CST         Marketing508a3i (a, s, r, P, f, f', msg)
Scope       {TreatmentUse506c1, PaymentUse506c1,
            HealthCareOperationsUse506c1, TreatmentDisclose506c1,
            PaymentDisclose506c1, TreatmentDisclose506c1,
            HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
            Disclose506c4, Disclose506c5}
Such That   f.protected-health-information = true
and         coveredEntity in Roles(a)
and         disclose $\mathbf{in}_a$ P
and         marketing $\mathbf{in}_f$ P
if          authorization in (a, s)
then        return true
else        return false

CST         Marketing508a3i (a, s, r, P, f, f', msg)
Scope       {TreatmentUse506c1, PaymentUse506c1,
            HealthCareOperationsUse506c1, TreatmentDisclose506c1,
            PaymentDisclose506c1, TreatmentDisclose506c1,
            HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
            Disclose506c4, Disclose506c5}
Such That   f.protected-health-information = true
and         coveredEntity in Roles(a)
and         use $\mathbf{in}_a$ P
and         marketing $\mathbf{in}_f$ P
if          Marketing508a3iA (a, s, r, P, f, f', msg) $\in$ {Allow}
then        return true
else        return false
```

```
      CST   Marketing508a3i (a, s, r, P, f, f', msg)
    Scope   {TreatmentUse506c1, PaymentUse506c1,
            HealthCareOperationsUse506c1, TreatmentDisclose506c1,
            PaymentDisclose506c1, TreatmentDisclose506c1,
            HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
            Disclose506c4, Disclose506c5}
Such That   f.protected-health-information = true
      and   coveredEntity in Roles(a)
      and   disclose $\mathbf{in}_a$ P
      and   marketing $\mathbf{in}_f$ P
       if   Marketing508a3iA (a, s, r, P, f, f', msg) $\in$ {Allow}
     then   return true
     else   return false

      CST   Marketing508a3i (a, s, r, P, f, f', msg)
    Scope   {TreatmentUse506c1, PaymentUse506c1,
            HealthCareOperationsUse506c1, TreatmentDisclose506c1,
            PaymentDisclose506c1, TreatmentDisclose506c1,
            HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
            Disclose506c4, Disclose506c5}
Such That   f.protected-health-information = true
      and   coveredEntity in Roles(a)
      and   use $\mathbf{in}_a$ P
      and   marketing $\mathbf{in}_f$ P
       if   Marketing508a3iB (a, s, r, P, f, f', msg) $\in$ {Allow}
     then   return true
     else   return false
```

```
        CST   Marketing508a3i (a, s, r, P, f, f', msg)
      Scope   {TreatmentUse506c1, PaymentUse506c1,
              HealthCareOperationsUse506c1, TreatmentDisclose506c1,
              PaymentDisclose506c1, TreatmentDisclose506c1,
              HealthCareOperationsDisclose506c1, PaymentDisclose506c3,
              Disclose506c4, Disclose506c5}
  Such That   f.protected-health-information = true
        and   coveredEntity in Roles(a)
        and   disclose $\mathbf{in}_a$ P
        and   marketing $\mathbf{in}_f$ P
         if   Marketing508a3iB(a, s, r, P, f, f', msg) $\in$ {Allow}
       then   return true
       else   return false
```

§164.508(a)(3)(i)(A)

```
        CST   Marketing508a3iA (a, s, r, P, f, f', msg)
      Scope   {}
  Such That   individual in Roles(s)
        and   coveredEntity in Roles(a)
         if   communication $\mathbf{in}_a$ P
        and   faceToFace $\mathbf{in}_a$ P
       then   return true
       else   return false
```

§164.508(a)(3)(i)(B)

```
       CST   Marketing508a3iB (a, s, r, P, f, f', msg)
     Scope   {}
        if   promotionalGift $\mathbf{in}_a$ P
       and   nominalValue $\mathbf{in}_a$ P
       and   coveredEntity **in** Roles(a)
      then   **return true**
      else   **return false**
```

§164.508(a)(3)(ii)

```
       CST   AuthValidAsIn508a3ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   marketing $\mathbf{in}_f$ P
       and   a.direct-remuneration = true
       and   a.from-third-party = true
        if   a.authorization-states-remuneration = true
      then   **return true**
      else   **return false**
```

```
       CST   AuthValidAsIn508a3ii (a, s, r, P, f, f', msg)
     Scope   {}
 Such That   marketing $\mathbf{in}_f$ P
       and   indirectRemuneration $\mathbf{in}_f$ P
       and   fromThirdParty $\mathbf{in}_f$ P
        if   a.authorization-states-remuneration = true
      then   **return true**
      else   **return false**
```

C.3 Insurance Council of Australia Privacy Code

The commands and constraints from the Insurance Council of Australia's Privacy Code section 2.4 are as follows. Section 2.4 deals with disclosure of personal information by health service providers. Health service providers may make such disclosures to people who are responsible for an individual (*i.e.,* the subject of the data) in cases where the individual is unable to communicate consent. Section 2.5 lists the people who are considered responsible for an individual. Section 2.6 widens some of the definitions of 2.5. The source text for the Privacy API in this section is found in Appendix B.3.

2.4

CST	Permitted2.4(a, s, r, P, f, f', msg)
Scope	{Disclose2.4}
Such That	healthProvider **in** Roles(a)
and	individual **in** Roles(s)
and	f.health-information = true
if	Permitted2.4a(a, s, r, P, f, f', msg) $\in$ {Allow}
and	Permitted2.4b(a, s, r, P, f, f', msg) $\in$ {Allow}
and	Permitted2.4c(a, s, r, P, f, f', msg) $\in$ {Allow}
and	responsible **in** (r, s)
and	disclose **in**$_a$ P
then	**return true**
else	**return false**

```
   CMD   Disclose2.4(a, s, r, P, f, f', msg)
    if   Permitted2.4(a, s, r, P, f, f', msg)
   and   individual in Roles(s)
   and   healthProvider in Roles(a)
   and   responsible in (r, s)
   and   disclose $\mathbf{in}_a$ P
   and   f.health-information = true
   and   local in (a, f)
  then   insert local in (r, f)
   and   return true
  else   return false
```

2.4(a)

```
   CST   Permitted2.4a(a, s, r, P, f, f', msg)
 Scope   {Disclose2.4}
    if   Permitted2.4ai(a, s, r, P, f, f', msg) $\in$ {Allow}
  then   return true
  else   return false

   CST   Permitted2.4a(a, s, r, P, f, f', msg)
 Scope   {Disclose2.4}
    if   Permitted2.4aii(a, s, r, P, f, f', msg) $\in$ {Allow}
  then   return true
  else   return false
```

2.4(a)(i)

```
  CST   Permitted2.4ai(a, s, r, P, f, f', msg)
Scope   {}
   if   physicallyIncapable in Roles(s)
 then   return true
 else   return false

  CST   Permitted2.4ai(a, s, r, P, f, f', msg)
Scope   {}
   if   legallyIncapable in Roles(s)
 then   return true
 else   return false
```

2.4(a)(ii)

```
  CST   Permitted2.4aii(a, s, r, P, f, f', msg)
Scope   {}
   if   cannotPhysicallyCommunicateConsent in Roles(s)
 then   return true
 else   return false
```

2.4(b)

```
  CST   Permitted2.4b(a, s, r, P, f, f', msg)
Scope   {Disclose2.4}
   if   naturalPerson in Roles(a)
  and   providingHealthService in Roles(a)
  and   Permitted2.4bi(a, s, r, P, f, f', msg) ∈ {Allow}
 then   return true
 else   return false
```

```
      CST   Permitted2.4b(a, s, r, P, f, f', msg)
    Scope   {Disclose2.4}
       if   naturalPerson in Roles(a)
      and   providingHealthService in Roles(a)
      and   Permitted2.4bii(a, s, r, P, f, f', msg) $\in$ {Allow}
     then   return true
     else   return false
```

2.4(b)(i)

```
      CST   Permitted2.4bi(a, s, r, P, f, f', msg)
    Scope   {}
       if   treatment $\textbf{in}_a$ P
      and   a.satisfied-disclosure-necessary-to-provide-appropriate-care = true
     then   return true
     else   return false

      CST   Permitted2.4bi(a, s, r, P, f, f', msg)
    Scope   {}
       if   treatment $\textbf{in}_a$ P
      and   a.satisfied-disclosure-necessary-to-provide-appropriate-treatment = true
     then   return true
     else   return false
```

2.4(b)(ii)

```
       CST   Permitted2.4bii(a, s, r, P, f, f', msg)
     Scope   {}
 Such That   disclose $\textbf{in}_a$ P
       and   compassion $\textbf{in}_a$ P
      then   return true
      else   return false
```

2.4(c)

```
       CST  Permitted2.4c(a, s, r, P, f, f', msg)
     Scope  {Disclose2.4}
 Such That  disclose $\mathbf{in}_a$ P
        if  s.wish-contrary = false
       and  Permitted2.4ci(a, s, r, P, f, f', msg) $\in$ {Allow}
       and  Permitted2.4cii(a, s, r, P, f, f', msg) $\in$ {Allow}
      then  return true
      else  return false
```

2.4(c)(i)

```
  CST  Permitted2.4cii(a, s, r, P, f, f', msg)
Scope  {}
   if  s.wish-expressed-before-unable-to-consent = false
 then  return true
 else  return false
```

2.4(c)(ii)

```
  CST  Permitted2.4cii(a, s, r, P, f, f', msg)
Scope  {}
   if  a.aware-of-wish = false
  and  a.reasonably-expected-to-be-aware-of-wish = false
 then  return true
 else  return false
```

2.4(d)

```
       CST   Permitted2.4d(a, s, r, P, f, f', msg)
     Scope   {Disclose2.4}
        if   reasonable $\mathbf{in}_a$ P
       and   necessary $\mathbf{in}_a$ P
       and   Permitted2.4b(a, s, r, P, f, f', msg) $\in$ {Allow, Don't Care/Allow}
       and   disclose $\mathbf{in}_a$ P
      then   return true
      else   return false
```

2.5

```
       CST   Responsible2.5(a, s, r, P, f, f', msg)
     Scope   {Permitted2.4}
 Such That   responsible in (r, s)
       and   individual in Roles(s)
        if   Responsible2.5a(a, s, r, P, f, f', msg) in {Allow}
      then   return true
      else   return false

       CST   Responsible2.5(a, s, r, P, f, f', msg)
 Such That   responsible in (r, s)
       and   individual in Roles(s)
        if   responsible in (r, s)
       and   Responsible2.5b(a, s, r, P, f, f', msg) in {Allow}
      then   return true
      else   return false
      CST  Responsible2.5(a, s, r, P, f, f', msg)
    Scope  {Permitted2.4}
Such That  responsible in (r, s)
      and  individual in Roles(s)
       if  Responsible2.5c(a, s, r, P, f, f', msg) in {Allow}
     then  return true
     else  return false

      CST  Responsible2.5(a, s, r, P, f, f', msg)
    Scope  {Permitted2.4}
Such That  responsible in (r, s)
      and  individual in Roles(s)
       if  Responsible2.5d(a, s, r, P, f, f', msg) in {Allow}
     then  return true
     else  return false

      CST  Responsible2.5(a, s, r, P, f, f', msg)
    Scope  {Permitted2.4}
Such That  responsible in (r, s)
      and  individual in Roles(s)
       if  Responsible2.5e(a, s, r, P, f, f', msg) in {Allow}
     then  return true
     else  return false

      CST  Responsible2.5(a, s, r, P, f, f', msg)
    Scope  {Permitted2.4}
Such That  responsible in (r, s)
      and  individual in Roles(s)
       if  Responsible2.5f(a, s, r, P, f, f', msg) in {Allow}
     then  return true
     else  return false
```

```
        CST   Responsible2.5(a, s, r, P, f, f', msg)
      Scope   {Permitted2.4}
  Such That   responsible in (r, s)
        and   individual in Roles(s)
         if   Responsible2.5g(a, s, r, P, f, f', msg) in {Allow}
       then   return true
       else   return false

        CST   Responsible2.5(a, s, r, P, f, f', msg)
      Scope   {Permitted2.4}
  Such That   responsible in (r, s)
        and   individual in Roles(s)
         if   Responsible2.5h(a, s, r, P, f, f', msg) in {Allow}
       then   return true
       else   return false
```

2.5(a)

```
        CST   Responsible2.5a(a, s, r, P, f, f', msg)
      Scope   {}
  Such That   individual in Roles(s)
         if   parent in (r, s)
       then   return true
       else   return false

        CST   Responsible2.5a(a, s, r, P, f, f', msg)
      Scope   {}
  Such That   individual in Roles(s)
         if   Parent2.6(a, s, r, P, f, f', msg) ∈ {Allow}
       then   return true
       else   return false
```

2.5(b)

```
         CST  Responsible2.5b(a, s, r, P, f, f', msg)
       Scope  {}
   Such That  individual in Roles(s)
          if  child in (r, s)
         and  r.at-least-18 = true
        then  return true
        else  return false

         CST  Responsible2.5b(a, s, r, P, f, f', msg)
       Scope  {}
   Such That  individual in Roles(s)
          if  Child2.6(a, s, r, P, f, f', msg) ∈ {Allow}
         and  r.at-least-18 = true
        then  return true
        else  return false

         CST  Responsible2.5b(a, s, r, P, f, f', msg)
       Scope  {}
   Such That  individual in Roles(s)
          if  sibling in (r, s)
         and  r.at-least-18 = true
        then  return true
        else  return false

         CST  Responsible2.5b(a, s, r, P, f, f', msg)
       Scope  {}
   Such That  individual in Roles(s)
          if  Sibling2.6 ∈ {Allow}
         and  r.at-least-18 = true
        then  return true
        else  return false
```

2.5(c)

```
       CST   Responsible2.5c(a, s, r, P, f, f', msg)
     Scope   {}
 Such That   individual in Roles(s)
        if   spouse in (r, s)
      then   return true
      else   return false

       CST   Responsible2.5c(a, s, r, P, f, f', msg)
     Scope   {}
 Such That   individual in Roles(s)
        if   defactoSpouse in (r, s)
      then   return true
      else   return false
```

2.5(d)

```
       CST   Responsible2.5d(a, s, r, P, f, f', msg)
     Scope   {}
 Such That   individual in Roles(s)
        if   relative in (r, s)
       and   r.at-least-18 = true
       and   householdMember in (r, s)
      then   return true
      else   return false
```

```
        CST  Responsible2.5d(a, s, r, P, f, f', msg)
      Scope  {}
  Such That  individual in Roles(s)
         if  Relative2.6(a, s, r, P, f, f', msg) $\in$ {Allow}
        and  r.at-least-18 = true
        and  householdMember in (r, s)
       then  return true
       else  return false
```

2.5(e)

```
        CST  Responsible2.5e(a, s, r, P, f, f', msg)
      Scope  {}
  Such That  individual in Roles(s)
         if  guardian in (r, s)
       then  return true
       else  return false
```

2.5(f)

```
        CST  Responsible2.5f(a, s, r, P, f, f', msg)
      Scope  {}
  Such That  individual in Roles(s)
         if  powerOfAttorney in (r, s)
        and  r.power-granted-by-individual = true
        and  r.power-exercisable-to-health = true
       then  return true
       else  return false
```

2.5(g)

```
         CST   Responsible2.5g(a, s, r, P, f, f', msg)
       Scope   {}
   Such That   individual in Roles(s)
          if   intimatePersonalRelationship in (r, s)
        then   return true
        else   return false
```

2.5(h)

```
         CST   Responsible2.5h(a, s, r, P, f, f', msg)
       Scope   {}
   Such That   individual in Roles(s)
          if   nominatedEmergencyContact in (s, r)
         and   r.nominated-by-individual = true
        then   return true
        else   return false
```

2.6

Child

```
         CST   Child2.6(a, s, r, P, f, f', msg)
       Scope   {}
   Such That   individual in Roles(s)
          if   adoptedChild in (r,s)
        then   return true
        else   return false
       CST  Child2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  stepChild in (r,s)
      then  return true
      else  return false

       CST  Child2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  fosterChild in (r,s)
      then  return true
      else  return false
```

Parent

```
       CST  Parent2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  stepParent in (r,s)
      then  return true
      else  return false

       CST  Parent2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  adoptiveParent in (r,s)
      then  return true
      else  return false
      CST  Parent2.6(a, s, r, P, f, f’, msg)
    Scope  {}
Such That  individual in Roles(s)
       if  fosterParent in (r,s)
     then  return true
     else  return false
```

Relative

```
      CST  Relative2.6(a, s, r, P, f, f’, msg)
    Scope  {}
Such That  individual in Roles(s)
       if  grandparent in (r,s)
     then  return true
     else  return false

      CST  Relative2.6(a, s, r, P, f, f’, msg)
    Scope  {}
Such That  individual in Roles(s)
       if  grandchild in (r,s)
     then  return true
     else  return false

      CST  Relative2.6(a, s, r, P, f, f’, msg)
    Scope  {}
Such That  individual in Roles(s)
       if  uncle in (r,s)
     then  return true
     else  return false
```

```
       CST  Relative2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  aunt in (r,s)
      then  return true
      else  return false

       CST  Relative2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  nephew in (r,s)
      then  return true
      else  return false

       CST  Relative2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  niece in (r,s)
      then  return true
      else  return false
```

Sibling

```
       CST  Sibling2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  halfBrother in (r,s)
      then  return true
      else  return false
```

```
       CST   Sibling2.6(a, s, r, P, f, f', msg)
     Scope   {}
 Such That   individual in Roles(s)
        if   halfSister in (r,s)
      then   return true
      else   return false

       CST   Sibling2.6(a, s, r, P, f, f', msg)
     Scope   {}
 Such That   individual in Roles(s)
        if   adoptiveBrother in (r,s)
      then   return true
      else   return false

       CST   Sibling2.6(a, s, r, P, f, f', msg)
     Scope   {}
 Such That   individual in Roles(s)
        if   adoptiveSister in (r,s)
      then   return true
      else   return false

       CST   Sibling2.6(a, s, r, P, f, f', msg)
     Scope   {}
 Such That   individual in Roles(s)
        if   stepBrother in (r,s)
      then   return true
      else   return false
```

```
       CST  Sibling2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  stepSister in (r,s)
      then  return true
      else  return false

       CST  Sibling2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  fosterBrother in (r,s)
      then  return true
      else  return false

       CST  Sibling2.6(a, s, r, P, f, f', msg)
     Scope  {}
 Such That  individual in Roles(s)
        if  fosterSister in (r,s)
      then  return true
      else  return false
```

C.4 Cable TV Act Privacy API

§551(a)(1)

```
       CST   MustNotify551a(a, s, r, P, f, f', msg)
 Such That   s.entering-agreement = true
        if   cableService in P
       and   a.written-statement = true
       and   a.separate-statement = true
       and   a.clear-conspicuous = true
       and   Notice551a1A(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Notice551a1b(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Notice551a1C(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Notice551a1D(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Notice551a1E(a, s, r, P, f, f', msg) ∈ {Allow}
      then   return true
      else   return false

       CST   MustNotify551a(a, s, r, P, f, f', msg)
 Such That   yearlyNotice $\mathbf{in}_a$ P
        if   cableService in P
       and   a.written-statement = true
       and   a.separate-statement = true
       and   a.clear-conspicuous = true
       and   Notice551a1A(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Notice551a1b(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Notice551a1C(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Notice551a1D(a, s, r, P, f, f', msg) ∈ {Allow}
       and   Notice551a1E(a, s, r, P, f, f', msg) ∈ {Allow}
      then   return true
      else   return false
```

```
CST   MustNotify551a(a, s, r, P, f, f', msg)
 if   s.entered-before-effective = true
and   subscriber in Roles(s)
and   a.written-statement = true
and   a.separate-statement = true
and   a.clear-conspicuous = true
and   Notice551a1A(a, s, r, P, f, f', msg) $\in$ {Allow}
and   Notice551a1b(a, s, r, P, f, f', msg) $\in$ {Allow}
and   Notice551a1C(a, s, r, P, f, f', msg) $\in$ {Allow}
and   Notice551a1D(a, s, r, P, f, f', msg) $\in$ {Allow}
and   Notice551a1E(a, s, r, P, f, f', msg) $\in$ {Allow}
then  return true
else  return false
```

```
CMD   Noice551a1(a, s, r, P, f, f', msg)
 if   MustNotify551a(a, s, r, P, f, f', msg) $\in$ {Allow}
and   subscriber in Roles(s)
and   cableOperator in Roles(a)
then  inform s of "notice"
and   return true
else  return false
```

§551(a)(1)(A)

```
CST   Notice551a1A(a, s, r, P, f, f', msg)
 if   a.nature-personally-identifiable-info = true
and   a.nature-use-information = true
then  return true
else  return false
```

§551(a)(1)(B)

```
 CST   Notice551a1B(a, s, r, P, f, f', msg)
  if   a.nature-disclosure = true
 and   a.frequency-disclosure = true
 and   a.purpose-disclosure = true
 and   a.identification-types-persons-disclosure = true
then   return true
else   return false
```

§551(a)(1)(C)

```
 CST   Notice551a1C(a, s, r, P, f, f', msg)
  if   a.maintained-period = true
 and   cableOperator in Roles(a)
then   return true
else   return false
```

§551(a)(1)(D)

```
 CST   Notice551a1D(a, s, r, P, f, f', msg)
  if   a.times-place-subscriber-access = true
 and   subscriber in Roles(s)
then   return true
else   return false
```

§551(a)(1)(E)

```
 CST   Notice551a1E(a, s, r, P, f, f', msg)
  if   a.limitations-collection-disclosure = true
 and   a.right-subscriber-f-h-enforce = true
then   return true
else   return false
```

§551(a)(2)

Definitions section, no commands.

§551(b)(1)

```
       CST   Collect551b1(a, s, r, P, f, f’, msg)
     Scope   {Collect551b1}
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   collectPII $\mathbf{in}_f$ P
        if   Collect551b2(a, s, r, P, f, f’, msg) $\in$ {Allow}
      then   return true
      else   return false

       CST   Collect551b1(a, s, r, P, f, f’, msg)
     Scope   {Collect551b1}
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   collectPII $\mathbf{in}_f$ P
        if   writtenConsent in (a, s)
      then   return true
      else   return false

       CST   Collect551b1(a, s, r, P, f, f’, msg)
     Scope   {Collect551b1}
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   collectPII $\mathbf{in}_f$ P
        if   electronicConsent in (a, s)
      then   return true
      else   return false
```

```
CMD         CollectInformation551b1(a, s, r, P, f, f', msg)
 if         Collect551b1(a, s, r, P, f, f', msg) $\in$ {Allow}
 and        cableOperator in Roles(a)
 then       insert collectPII in (a, s)
 and        return true
 else       return false
```

§551(b)(2)

```
CST         Collect551b2(a, s, r, P, f, f', msg)
Scope       {CollectInformation551b2A, CollectInformation551b2B}
Such That   cableOperator in Roles(a)
 and        collectPII $\text{in}_a$ P
 if         Collect551b2A(a, s, r, P, f, f', msg) $\in$ {Allow}
 then       return true
 else       return false

CST         Collect551b2(a, s, r, P, f, f', msg)
Scope       {CollectInformation551b2A, CollectInformation551b2B}
Such That   cableOperator in Roles(a)
 and        collectPII $\text{in}_a$ P
 if         Collect551b2B(a, s, r, P, f, f', msg) $\in$ {Allow}
 then       return true
 else       return false
```

§**551(b)(2)(A)**

```
       CST   Collect551b2A(a, s, r, P, f, f', msg)
     Scope   {CollectInformation551b2A}
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   collectPII $in_a$ P
        if   obtainNecessaryInfo $in_a$ P
       and   cableService $in_a$ P
      then   return true
      else   return false

       CST   Collect551b2A(a, s, r, P, f, f', msg)
     Scope   {CollectInformation551b2A}
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   collectPII $in_a$ P
        if   obtainNecessaryInfo $in_a$ P
       and   otherService $in_a$ P
      then   return true
      else   return false

 CMD   CollectInformation551b2A(a, s, r, P, f, f', msg)
  if   Collect551b2A(a, s, r, P, f, f', msg) $\in$ {Allow}
 and   collectPII $in_a$ P
then   insert collect in (a, s)
 and   insert cableSystem in (a, s)
 and   return true
else   return false
```

§**551(b)(2)(B)**

```
       CST   Collect551b2B(a, s, r, P, f, f', msg)
     Scope   {CollectInformation551b2B}
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
        if   detectUnauthorized $\textbf{in}_a$ P
       and   cableCommunications $\textbf{in}_a$ P
       and   collectPII $\textbf{in}_a$ P
      then   return true
      else   return false
```

```
 CMD   CollectInformation551b2B(a, s, r, P, f, f', msg)
   if  Collect551b2B(a, s, r, P, f, f', msg) $\in$ {Allow}
  and  collect $\textbf{in}_a$ P
 then  insert collect in (a, s)
  and  insert cableSystem in (a, s)
  and  return true
 else  return false
```

§**551(c)(1)**

```
       CST   Permitted551c1(a, s, r, P, f, f', msg)
     Scope   {Disclose551c1, Disclose551c2A, Disclose551c2B, Disclose551c2C }
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   f.personally-identifiable-information = true
       and   f.subject-subscriber = true
       and   disclose $\textbf{in}_a$ P
        if   Permitted551c2(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false
```

```
       CST   Permitted551c1(a, s, r, P, f, f', msg)
     Scope   {Disclose551c1, Disclose551c2A, Disclose551c2B, Disclose551c2C }
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   f.personally-identifiable-information = true
       and   f.subject-subscriber = true
       and   disclose $\mathbf{in}_a$ P
        if   consent in (a, s)
       and   written in (a, s)
      then   return true
      else   return false

       CST   Permitted551c1(a, s, r, P, f, f', msg)
     Scope   {Disclose551c1, Disclose551c2A, Disclose551c2B, Disclose551c2C }
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   f.personally-identifiable-information = true
       and   f.subject-subscriber = true
       and   disclose $\mathbf{in}_a$ P
        if   consent in (a, s)
       and   electronic in (a, s)
      then   return true
      else   return false
```

```
       CMD   Disclose551c1(a, s, r, P, f, f', msg)
        if   Permitted551c1(a, s, r, P, f, f', msg) ∈ {Allow}
       and   f.personally-identifiable-information = true
       and   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   disclose $\textbf{in}_a$ P
      then   insert local in (r, f)
       and   return true
      else   return false
```

```
        CST   Protect551c1(a, s, r, P, f, f', msg)
      Scope   { }
  Such That   cableOperator in Roles(a)
         if   a.takes-actions-prevents-unauthorized = true
       then   return true
       else   return false
```

§**551(c)(2)**

```
        CST   Permitted551c2(a, s, r, P, f, f', msg)
      Scope   {Disclose551c2A, Disclose551c2B, Disclose551c2C }
  Such That   cableOperator in Roles(a)
        and   f.personally-identifiable-information = true
        and   disclose $\textbf{in}_a$ P
         if   Permitted551c2A(a, s, r, P, f, f', msg) ∈ {Allow}
       then   return true
       else   return false
        CST   Permitted551c2(a, s, r, P, f, f', msg)
      Scope   {Disclose551c2A, Disclose551c2B, Disclose551c2C }
  Such That   cableOperator in Roles(a)
        and   f.personally-identifiable-information = true
        and   disclose $\mathbf{in}_a$ P
         if   Permitted551c2B(a, s, r, P, f, f', msg) $\in$ {Allow}
       then   return true
       else   return false

        CST   Permitted551c2(a, s, r, P, f, f', msg)
      Scope   {Disclose551c2A, Disclose551c2B, Disclose551c2C }
  Such That   cableOperator in Roles(a)
        and   f.personally-identifiable-information = true
        and   disclose $\mathbf{in}_a$ P
         if   Permitted551c2C(a, s, r, P, f, f', msg) $\in$ {Allow}
       then   return true
       else   return false
```

§551(c)(2)(A)

```
       CST   Permitted551c2A(a, s, r, P, f, f', msg)
     Scope   {Disclose551c2A}
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   disclose $\mathbf{in}_a$ P
        if   businessActivity $\mathbf{in}_a$ P
       and   legitimate $\mathbf{in}_a$ P
       and   render $\mathbf{in}_a$ P
       and   cableService $\mathbf{in}_a$ P
      then   return true
      else   return false
```

```
       CST   Permitted551c2A(a, s, r, P, f, f', msg)
     Scope   {Disclose551c2A}
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   disclose $\mathbf{in}_a$ P
        if   businessActivity $\mathbf{in}_a$ P
       and   legitimate $\mathbf{in}_a$ P
       and   conduct $\mathbf{in}_a$ P
       and   cableService $\mathbf{in}_a$ P
      then   return true
      else   return false

       CST   Permitted551c2A(a, s, r, P, f, f', msg)
     Scope   {Disclose551c2A}
 Such That   cableOperator in Roles(a)
       and   subscriber in Roles(s)
       and   disclose $\mathbf{in}_a$ P
        if   businessActivity $\mathbf{in}_a$ P
       and   legitimate $\mathbf{in}_a$ P
       and   render $\mathbf{in}_a$ P
       and   otherService $\mathbf{in}_a$ P
      then   return true
      else   return false
       CST  Permitted551c2A(a, s, r, P, f, f’, msg)
     Scope  {Disclose551c2A}
 Such That  cableOperator in Roles(a)
       and  subscriber in Roles(s)
       and  disclose $\mathbf{in}_a$ P
        if  businessActivity $\mathbf{in}_a$ P
       and  legitimate $\mathbf{in}_a$ P
       and  conduct $\mathbf{in}_a$ P
       and  otherService $\mathbf{in}_a$ P
      then  return true
       else  return false

 CMD  Disclose551c2A(a, s, r, P, f, f’, msg)
   if  Permitted551c2A(a, s, r, P, f, f’, msg) $\in$ {Allow}
  and  cableOperator in Roles(a)
  and  subscriber in Roles(s)
  and  local in (a, f)
  and  disclose $\mathbf{in}_a$ P
 then  insert local in (r, f)
  and  return true
 else  return false
```

§551(c)(2)(B)

```
CST        Permitted551c2B(a, s, r, P, f, f', msg)
Scope      {Disclose551c2B }
Such That  Permitted551h(a, s, r, P, f, f', msg) $\in$ {Allow}
and        disclose in$_a$ P
if         pursuantCourtOrder in$_a$ P
and        orderAuthorizes in$_a$ P
and        notified in (a, s)
then       return true
else       return false
```

```
CMD   Disclose551c2B(a, s, r, P, f, f', msg)
if    Permitted551c2B(a, s, r, P, f, f', msg) $\in$ {Allow}
and   cableOperator in Roles(a)
and   subscriber in Roles(s)
and   local in (a, f)
and   disclose in$_a$ P
then  insert local in (r, f)
and   return true
else  return false
```

```
     CMD   NotifySubscriber551c2B(a, s, r, P, f, f', msg)
      if   Permitted551h(a, s, r, P, f, f', msg) $\in$ {Allow}
     and   cableOperator in Roles(a)
     and   subscriber in Roles(s)
     and   pursuantCourtOrder $\text{in}_a$ P
     and   orderAuthorizes $\text{in}_a$ P
     and   a.message-notifies = true
     and   disclose $\text{in}_a$ P
    then   inform s of msg
     and   insert notified in (a, s)
     and   return true
    else   return false
```

§551(c)(2)(C)

```
        CST   Permitted551c2C(a, s, r, P, f, f', msg)
      Scope   {Dislclose551c2C }
  Such That   f.names = true
        and   f.addresses = true
        and   f.subject-subscriber = true
        and   Permitted551c2Ci(a, s, r, P, f, f', msg) $\in$ { Allow}
        and   Permitted551c2Cii(a, s, r, P, f, f', msg) $\in$ { Allow}
         if   disclose $\text{in}_a$ P
        and   cableSubscriber in Roles(s)
       then   return true
       else   return false
        CST   Permitted551c2C(a, s, r, P, f, f', msg)
      Scope   {Dislclose551c2C }
  Such That   f.names = true
        and   f.addresses = true
        and   f.subject-subscriber = true
        and   Permitted551c2Ci(a, s, r, P, f, f', msg) ∈ { Allow}
        and   Permitted551c2Cii(a, s, r, P, f, f', msg) ∈ { Allow}
         if   disclose in_a P
        and   otherServiceSubscriber in Roles(s)
       then   return true
        else   return false
```

```
  CMD   Dislclose551c2C(a, s, r, P, f, f', msg)
   if   Permitted551c2C(a, s, r, P, f, f', msg) ∈ {Allow}
  and   cableOperator in Roles(a)
 then   insert local in (r, f)
  and   return true
 else   return false
```

§551(c)(2)(C)(i)

```
      CST   Permitted551c2Ci(a, s, r, P, f, f', msg)
    Scope   { }
Such That   cableOperator in Roles(a)
      and   subscriber in Roles(s)
       if   opportunityProhibit in (a, s)
      and   !prohibit in (a, s)
     then   return true
     else   return false
```

```
       CST  Permitted551c2Ci(a, s, r, P, f, f', msg)
     Scope  {Dislclose551c2C }
 Such That  cableOperator in Roles(a)
       and  subscriber in Roles(s)
        if  opportunityLimit in (a, s)
       and  !limit in (a, s)
      then  return true
      else  return false

CMD   ProvideProhibit551c2Ci(a, s, r, P, f, f', msg)
  if  cableOperator in Roles(a)
 and  subscriber in Roles(s)
 and  provideProhibit in_a P
then  inform s of msg
 and  insert opportunityProhibit in (a, s)
 and  return true
else  return false

CMD   ProvideLimit551c2Ci(a, s, r, P, f, f', msg)
  if  cableOperator in Roles(a)
 and  subscriber in Roles(s)
 and  provideLimit in_a P
then  inform s of msg
 and  insert opportunityLimit in (a, s)
 and  return true
else  return false
```

§551(c)(2)(C)(ii)

```
       CST   Permitted551c2Cii(a, s, r, P, f, f', msg)
     Scope   {Dislclose551c2C }
        if   disclose $\mathbf{in}_a$ P
       and   notDirectlyReveal $\mathbf{in}_f$ P
       and   notIndirectlyReveal $\mathbf{in}_f$ P
       and   Permitted551c2CiiI(a, s, r, P, f, f', msg) $\in$ {Allow}
       and   Permitted551c2CiiII(a, s, r, P, f, f', msg) $\in$ {Allow}
      then   return true
      else   return false
```

§551(c)(2)(C)(ii)(I)

```
       CST   Permitted551c2CiiI(a, s, r, P, f, f', msg)
     Scope   { }
 Such That   subscriber in Roles(s)
       and   f.cable-service = true
        if   f.extent-of-viewing = false
       and   f.extent-of-use = false
      then   return true
      else   return false

       CST   Permitted551c2CiiI(a, s, r, P, f, f', msg)
     Scope   { }
 Such That   subscriber in Roles(s)
       and   f.other-service = true
        if   f.extent-of-viewing = false
       and   f.extent-of-use = false
      then   return true
      else   return false
```

§551(c)(2)(C)(ii)(II)

```
       CST  Permitted551c2CiII(a, s, r, P, f, f', msg)
     Scope  { }
 Such That  subscriber in Roles(s)
        if  f.nature-of-transactions-over-system = false
      then  return true
      else  return false
```

§551(h)

```
       CST  Permitted551h(a, s, r, P, f, f', msg)
     Scope  {Disclose551h }
 Such That  governmentEntity in Roles(r)
       and  pursuantCourtOrder $\mathbf{in}_a$ P
       and  f.personally-identifiable-information = true
        if  subscriber in Roles(s)
       and  Permitted551h1(a, s, r, P, f, f', msg) $\in$ {Allow}
       and  Permitted551h2(a, s, r, P, f, f', msg) $\in$ {Allow}
      then  return true
      else  return false

  CMD  Disclose551h(a, s, r, P, f, f', msg)
   if  Permitted551h(a, s, r, P, f, f', msg) $\in$ {Allow}
  and  local in (a, f)
 then  insert local in (r, f)
  and  return true
 else  return false
```

§551(h)(1)

```
  CST  Permitted551h1(a, s, r, P, f, f', msg)
Scope  {Disclose551h }
   if  r.clear-convincing-evidence = true
  and  r.subject-suspected-criminal-activity = true
  and  r.information-material-evidence = true
 then  return true
 else  return false
```

§551(h)(2)

```
  CST  Permitted551h2(a, s, r, P, f, f', msg)
Scope  {Disclose551h }
   if  s.opportunity-to-appear = true
  and  s.opportunity-to-contest = true
 then  return true
 else  return false
```

Appendix D

Glossary of Sets, Functions, Relations, and Notation

Given the extensive use of notation and shorthand in this dissertation, we present a glossary of terms used in this appendix. Where applicable, a reference to the full definition of a particular term is given.

D.1 General Notation

We use the following mathematical notation consistently throughout this work. In general, we use lower case variable names (*e.g.,* a, b) for individuals and upper case variable names (*e.g.,* A, B) for sets or collections of individuals.

pwr The power set operator.

$=$ The mathematical equality operator.

`=` The C-style assignment operator.

`==` The C-style equality operator.

&&, $\wedge$ The logical AND operator.

! The logical NOT operator.

$||$, $\vee$ The logical OR operator.

$-$ The set difference operator.

ϵ The empty string.

$|A|$ The cardinality of set A.

$+$ The string concatenation operator.

D.2 Fundamental Sets and Types

We use the following sets in our presentation of Privacy Commands. See Section 5.1 for formal definitions and usage.

Agent Agents.

Object Objects. $\mathit{Agent} \subseteq \mathit{Object}$.

Role Roles held by agents. Roles are implemented as atomic properties of agents. $\mathit{Roles}(a) = \{k | a.k = \mathit{true}, \forall k \in \mathit{Role}\}$.

Tag Tags which are meta-data properties of objects. $\mathit{tags}(a) = \{t | a.t = \mathit{true}, \forall t \in \mathit{Tag}\}$.

Log Append-only log of notes and messages to inform agents.

State Knowledge state. $\mathit{State} \subseteq \mathit{Agent} \times \mathit{Object} \times \mathit{Matrix} \times \mathit{Log}$.

Operations, guards, commands, and constraints are the fundamental pieces of Privacy APIs. We use the following sets and symbols for items in the sets.

$\mathit{Operation}, \omega, \overline{\omega}$ The set of operations, a single operation, and an operation sequence. See Section 5.2.1 for definitions, Table 5.3 for structure definition, and Section 5.3 for semantics.

$\mathit{Guard}, \psi, \overline{\psi}$ The set of guards, a single guard, and a guard sequence. See Section 5.2.1 for definitions, Table 5.2 for structure definition, and Section 5.3 for semantics.

Command, $e, \overline{e}$ The set of commands, a single command, and a command sequence. See Section 5.2.3 for definition, Table 5.6 for BNF grammar definition, and Section 5.3 for semantics.

Constraint, $c, \overline{c}$ The set of constraints, a single constraint, and a constraint sequence. See Section 5.2.4 for definition, Table 5.6 for BNF grammar definition, and Section 5.3 for semantics.

Policy, ϕ The set of Privacy APIs and a single Privacy API. We often refer to a Privacy API as a *policy.* See Definition 5.2.7 for the definition of a well formed Privacy API.

We represent parameters to commands and constraints using tuples built from the above types.

ParametersE The set of parameters used for a command. They are denoted a, s, r, P, f, f', msg and are typed as in Table D.1.

ParametersC The set of parameters used for a constraint. They are denoted $a, s, r, P, f, f', msg, e$ and are typed as in Table D.1.

D.3 Variable Name Bindings

Throughout this dissertation we use single and multiple letter variable names for ranging over values of different types. We reference we include the commonly used variable names in Table D.1. As a rule, variable names which begin with lower case character are scalars and those that begin with upper case letters are sets. Variable names which end in a trailing asterisk * and those indicated with top bar (*ex.* $\overline{\psi}$) are ordered series of scalar values.

D.4 Transitions

Commands and constraints accept parameters a, s, r, P, f, f', msg for commands and $a, s, r, P, f, f', msg, e$ for constraints which are used by guards during evaluation. The parameters, their types, and informal names are shown in Table D.1.

Table D.1: Variable Names

Name	Type	Usage
a	Agent	Actor
$a_1, a_2, \ldots$	Agent	Agents
A	pwr(Agent)	Set of agents
b	Bool	A boolean
c	Constraint	A constraint
C	pwr(Constraint)	Set of constraints
c^*	Constraint^*	Series of constraints
d	Right	A right
e	Command	A command
E	pwr(Command)	A set of commands
e^*	Command^*	A series of commands
f	Object	An object
f'	Object	A fresh object
g	$\text{Agent} \times \text{Agent} \times \text{Agent} \times \text{Purpose}^* \times \text{Object} \times \text{Object} \times \text{String}$	Command parameters
h	$\text{Agent} \times \text{Agent} \times \text{Agent} \times \text{Purpose}^* \times \text{Object} \times \text{Object} \times \text{String} \times \text{Command}$	Constraint parameters
j	Judgment	A judgment
J	pwr(Judgment)	A set of judgments
k	Role	A role
l	Log	A log
m	$\text{Agent} \times \text{Object} \rightarrow \text{Right}^*$	Rights matrix
msg	String	A message string
$o, o_1, o_2, \ldots$	Object	An object
O	pwr(Object)	A set of objects
p	Purpose	A purpose
P	pwr(Purpose)	A set of purposes
r	Agent	Recipient
s	Agent	Subject
t	Tag	An object tag

We use the $\longrightarrow$ symbol to denote transitions between members of *State* via commands. As defined in Section 5.3, if e is a valid command and accepts parameter list $a \in \mathit{ParametersE}$, we write $(A, O, m, l) \xrightarrow{e(a)} (A', O', m', l')$ when e is run. The updates applied for $(A, O, m, l) \xrightarrow{e(a)} (A', O', m', l')$ are from one of the following cases. The function $\mathsf{most\text{-}strict}$ is a summary of the behavior of the operational semantics shown there:

1. If $\mathsf{most\text{-}strict}(\{c | e \in c.E\}) \in \{\text{Allow, Ignore (Allow)}\}$, $\bigwedge_{\psi \in \overline{\psi}} \psi = \mathit{true}$ for a then $(A, O, m, l) \xrightarrow{\overline{\omega^t}} (A', O', m', l')$ for a and the return value is true.
2. If $\mathsf{most\text{-}strict}(\{c | e \in c.E\}) = \text{Forbid}$ or $\bigwedge_{\psi \in \overline{\psi}} \psi = \mathit{false}$ for a then $(A, O, m, l) \xrightarrow{\overline{\omega^f}} (A', O', m', l')$ and the return value is false.

D.5 Purposes

We use the following functions and set inclusion operators for purposes. See Section 5.1 for formal definitions and usage.

$\mathbf{parent}(p)$ Parent purpose of p.

$\mathbf{children}(p)$ The direct children of p.

$\mathbf{ancestors}(p)$ Transitive and reflexive closure of $\mathsf{parent}(p)$. $p \in \mathsf{ancestors}(p)$.

$\mathbf{descendants}(p)$ Transitive and reflexive closure of $\mathsf{children}(p)$. $p \in \mathsf{descendants}(p)$.

$p\ \mathbf{in}_a\ P$ Set inclusion using allowed semantics. True iff $\exists p' \in P\ .\ p' \in \mathsf{descendants}(p)$.

$p\ \mathbf{in}_f\ P$ Set inclusion using forbidden semantics. True iff $\exists p' \in P\ .\ p' \in \{\mathsf{ancestors}(p) \cup \mathsf{descendants}(p)\}$.

D.6 Functions

We use the following functions for deriving sets and describing transformations.

Pr The Promela translation of formal artifact. The function is defined over the following domains:

- *State* Translation of a knowledge state snapshot into Promela. Defined in Section 6.4.
- *Constraint* Translation of a single constraint or overloaded constraint into a Promela process. Defined in Lemma 6.4.7.
- *Constraint** Translation of a set of constraints into a list of Promela processes. Defined in Theorem 2
- *Command* Translation of a single command into a Promela process. Defined in Lemma 6.4.6.
- *Command** Translation of a set of commands into a list of Promela processes. Defined in Theorem 2

deletedo The objects deleted from an initial state to reach a final state. Defined in Definition 5.5.5.

created The objects added to an initial state to reach a final state. Defined in Definition 5.5.6.

tagsm Tags modified in an initial state to reach a final state. Defined in Definition 5.5.7.

inserted The rights inserted in the rights matrix of an initial state to reach a final state. Defined in Definition 5.5.8.

deletedr The rights deleted from the rights matrix of an initial state to reach a final state. Defined in Definition 5.5.9.

prefix Determines whether one log is subsumed by another by comparing the entries in each without respect to the order of the entries. Defined in Definition 5.5.10.

noconflict Relationship between commands and commands series that the updates performed by one do not conflict with the updates performed by the other. Defined in Section 5.5.2.

Roles(a) Derives the roles currently held by the agent $a : \textsc{Agent}$.

Bibliography

[1] Alan S. Abrahams. *Developing and Executing Electronic Commerce Applications with Occurrences.* PhD thesis, University of Cambridge, 2002.

[2] Alan S. Abrahams, David M. Eyers, and Jean M. Bacon. An asynchronous rule-based approach for business process automation using obligations. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Rule-Based Programming (RULE'02)*, pages 93–103, Pittsburgh, Pennsylvania, October 2002.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. An XPath-based preference language for P3P. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 629–639, New York, NY, USA, 2003. ACM Press.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 439–450. ACM Press, 2000.

[5] Ross J. Anderson. A security policy model for clinical information systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 30–43. IEEE, May 1996.

[6] Annie I. Antón, Julia B. Earp, Matthew W. Vail, Neha Jain, Carrie Gheen, and Jack M. Frink. An analysis of web site privacy policy evolution in the presence of hipaa. Technical Report TR-2004-21, North Carolina State University, 24 July 2004.

[7] Grigoris Antoniou. A tutorial on default logics. *ACM Comput. Survey.*, 31(4):337–359, 1999.

[8] Paul Ashley, Santoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. Enterprise Privacy Authorization Langugae (EPAL 1.2). W3C Member Submission, `www.w3c.org/Submission/EPAL`, November 2003.

[9] Paul Ashley, Satoshi Hada, Günter Karjoth, and Matthias Schunter. E-P3P privacy policies and privacy authorization. In *Proceeding of the ACM workshop on Privacy in the Electronic Society*, pages 103–109. ACM Press, 2002.

[10] Paul Ashley, Calvin Powers, and Matthias Schunter. From privacy promises to privacy management: a new approach for enforcing privacy throughout an enterprise. In *Proceedings of the 2002 Workshop on New Security Paradigms*, pages 43–50. ACM Press, 2002.

[11] Michael Backes, Markus Dürmuth, and Günter Karjoth. Unification in privacy policy evaluation - translating epal into prolog. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*, 2004.

[12] Michael Backes, Markus Dürmuth, and Rainer Steinwandt. An algebra for composing enterprise privacy policies. In *European Symposium on Research in Computer Security (ESORICS)*, 2004.

[13] Michael Backes, Günter Karjoth, Walid Bagga, and Matthias Schunter. Efficient comparison of enterprise privacy policies. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 375–382, New York, NY, USA, 2004. ACM Press.

[14] Michael Backes, Birgit Pfitzmann, and Matthias Schunter. A toolkit for managing enterprise privacy policies. In *European Symposium on Research in Computer Security (ESORICS)*, 2003.

[15] Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2006.

[16] Moritz Y. Becker and Peter Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks*, June 2004.

[17] Moritz Y. Becker and Peter Sewell. Cassandra: flexible trust management, applied to electronic health records. In *17th IEEE Computer Security Foundations Workshop*, June 2004.

[18] D. Bell and L. La Padula. Secure computing systems: Mathematical foundations and model. Technical Report MTR–2547, MITRE Corp, 1973.

[19] Artur Bergman. Law is code. *O'Reilly Radar*, 6 August 2007. `radar.oreilly.com/archives/2007/08/code_looks_like.html`.

[20] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 41–52, New York, NY, USA, 2001. ACM Press.

[21] Claudio Bettini, Sushil Jajodia, X. Sean Wang, and Duminda Wijesekera. Provisions and obligations in policy management and security applications. In *28th Conference on Very Large Data Bases (VLDB'02)*, Aug 2002.

[22] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, chapter The Role of Trust Management in Distributed Systems Security, pages 185–210. Lecture Notes in Computer Science State-of-the-Art Series. Springer-Verlag Berlin, 1999.

[23] Jan Bormans and Keith Hill. Mpeg-21 multimedia framework. Standard 5, Motion Picture Experts Group, 2002.

[24] Travis Breaux and Annie I. Antón. Mining rule semantics to understand legislative compliance. In *CCS Workshop on Privacy in the Electronic Society*. ACM, Nov 2005.

[25] D. Brewer and M. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[26] Elaine Callas and Karl Brockmeier. HIPAA compliance readiness assessment: A case study. *Healthcare Financial Management*, Oct 2001.

[27] Marco Casassa-Mont. Dealing with privacy obligations in enterprises. Tech Report HPL-2004-109, HP Labs, 2004.

[28] Marco Casassa-Mont. A system to handle privacy obligations in enterprises. Tech Report HPL-2005-180, HP Labs, 2005.

[29] Electronic Privacy Information Center. Double trouble. `www.epic.org/privacy/doubletrouble/`, March 2000.

[30] Electronic Privacy Information Center and Junkbusters. Pretty poor privacy: An assessment of P3P and internet privacy. `www.epic.org/reports/prettypoorprivacy.html`, 2000.

[31] Inc. Certus Software. Certus 404/302. `www.certus.com`, 2007.

[32] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194. IEEE, 1987.

[33] Roger Clarke. Platform for Privacy Preferences: A critique. `www.anu.edu.au/people/Roger.Clarke/DV/P3PCrit.html`, 1998.

[34] Federal Trade Commission. Children's online privacy protection act of 1998 (COPPA). *Federal Register*, 64(212):59888–59915, 3 November 1999. 16 CFR Part 312.

[35] Senate Banking Committee. Financial modernization act of 1999 (Gramm-Leach-Bliley act) – disclosure of nonpublic personal information. *15 USC, Subchapter I, Sec. 6801-6809*, 1999.

[36] Lorrie Cranor. *Web privacy with P3P.* O'Reilly and Associates, 2002.

[37] Nikhil Dinesh, Aravind Joshi, Insup Lee, and Bonnie Webber. Extracting formal specifications from natural language regulatory documents. In *Inference in Computational Semantics*, Buxton, England, 20–21 April 2006.

[38] Julia B. Earp, Annie I. Antón, and Olli Jarvinen. A social, technical and legal framework for privacy management and policies. In *Americas Conference on Information Systems (AMCIS)*, pages 605–612, August 2002.

[39] David Ferrailo and Richard Kuhn. Role–based access control. In *Proceedings of the 15th National Computer Security Conference*, 1992.

[40] Simone Fischer-Hübner and Amon Ott. From a formal privacy model to its implementation. In *Proceedings of the 21st National Information Systems Security Conference*, Oct 1998.

[41] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 196–205, New York, NY, USA, 2005. ACM Press.

[42] Office for Civil Rights. Standards for privacy of individually identifiable health information; final rule. *Federal Register*, 65(250):82462–82829, December 28 2000. Unamended Version.

[43] Office for Civil Rights. Standards for privacy of individually identifiable health information; final rule. *Federal Register*, 67(157):53182–53273, August 14 2002. Final modifications to privacy rule.

[44] Office for Civil Rights. *Fact Sheet: Protecting the privacy of patients' health information*, 14 April 2003. `www.hhs.gov/news/facts/privacy.html`.

[45] Office for Civil Rights. Health insurance reform: Security standards; final rule. *Federal Register*, 68(34):8334–8381, 20 February 2003. 45 CFR Parts 160, 162, and 164.

[46] Office for Civil Rights. Standards for privacy of individually identifiable health information. Regulation Text (Unofficial Version) 45 CFR Parts 160 and 164, U.S. Department of Health and Human Services, August 2003. As amended: May 31, 2002, Aug 14, 2002, Feb 20, 2003, and Apr 17, 2003.

[47] Christopher Giblin, Alice Y. Liu, Samuel Müller, Birgit Pfitzmann, and Xin Zhou. Regulations expressed as logical models (realm). In M.-F. Moens and P. Spyns, editors, *Legal Knowledge and Information Systems*, pages 37–48. IOS Press, 2005.

[48] US Government. Cable tv privacy act of 1984 (as amended). *47 USC, Chapter 5, Subchapter V-A, Part IV, Sec. 551*, 1984.

[49] G. S. Graham and P. J. Denning. Protection: Principles and Practices. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 40, pages 417–429. AFIPS Press, Montvale, N.J., 1972.

[50] Richard Graubart. On the need for a third form of access control. In *12th National Computer Security Conference Proceedings*, pages 296–303, October 1989.

[51] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, May 2000.

[52] Carl A. Gunter, Michael J. May, and Stuart Stubblebine. A formal privacy system and its application to location based services. In *Privacy Enhancing Technologies (PET)*, Toronto, May 2004.

[53] Susanne Guth, Gustaf Neumann, and Mark Strembeck. Experiences with the enforcement of access rights extracted from odrl-based digital contracts. In *DRM '03: Proceedings of the 3rd ACM workshop on Digital rights management*, pages 90–102, New York, NY, USA, 2003. ACM Press.

[54] Satoshi Hada and Michiharu Kudo. XML access control language (XACL): Provisional authorization for XML documents. Standard, OASIS, 2001. `xml.coverpages.org/xacl.html`.

[55] Joseph Y. Halpern and Vicky Weissman. A formal foundation for XrML. In *Proceedings of the Seventeenth IEEE Computer Security Foundations Workshop (CSFW-04)*, pages 251–263. IEEE, 2004.

[56] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[57] Katia Hayati and Martín Abadi. Language-based enforcement of privacy policies. In *Privacy Enhancing Technologies (PET)*, May 2004.

[58] Manuel Hilty, David Basin, and Alexander Pretschner. On obligations. In *European Symposium on Research in Computer Security (ESORICS)*, 2005.

[59] Giles Hogben. A technical analysis of problems with P3P 1.0 and possible solutions. In *W3C Workshop on the Future of P3P*. World Wide Web Consortium, Dulles, Virginia, November 2002. `www.w3.org/2002/p3p-ws/pp/jrc.html`.

[60] Renato Iannella. Open digital rights management language (ODRL) 1.1. W3C note, World Wide Web Consortium, 2002. `odrl.net`.

[61] ContentGuard Inc. eXtensible rights Markup Language 2.0 (XrML2.0). Technical report, ContentGuard, Inc., 2001. `www.xrml.org`.

[62] TiVo Inc. TiVo privacy policy. Privacy policy, `www.tivo.com/abouttivo/policies/tivoprivacypolicy.html`, May 2006. Accessed Nov 2007.

[63] Sushil Jajodia, Michiharu Kudo, and V.S. Subrahmanian. Provisional authorizations. In *Recent Advances in Secure and Private E-Commerce*, pages 133–159. Kluwer Academic Publishers, 2001.

[64] Günter Karjoth, Matthias Schunter, and Els Van Herreweghen. Translating privacy practices into privacy promises–how to promise what you can keep. In *Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 135–146, 2003.

[65] Bradley A. Malin. An evaluation of the current state of genomic data privacy protection technology and a roadmap for the future. *Journal of the American Medical Informatics Association*, 12(1):28–34, Jan/Feb 2005.

[66] Bradley A. Malin. *Trail Re-identification and Unlinkability in Distributed Databases*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2006.

[67] Stony Hill Management. GetHIP: Automated HIPAA compliance solution. `www.hipaahomecare.com`, 2006.

[68] David Martin. Tivo's data collection and privacy practices. Web page accessed Nov 06, Privacy Foundation, 2001. `www.cs.uml.edu/~dm/pubs/TiVo%20report.htm`.

[69] Timothy D. Miller and Ravinder J. Singh. A five phase process for HIPAA compliance: A case study in process. Phoenix Health Systems, Nov 2000. `www.hipaadvisory.com/action/CaseStudy.htm`.

[70] Tim Moses. eXtensible Access Control Markup Language (XACML). Standard Version 2.0, OASIS, February 2005.

[71] Commonwealth of Australia. Privacy act 1988 (as amended up to act no. 25 of 2006. *Act No. 119 of 1988*, 1988.

[72] Insurance Council of Australia. General insurance information privacy code. online, Australia, April 2002. `app01.ica.com.au/PrivacyPrinciples/default.jsp`.

[73] Department of Justice. The privacy act (as amended). *5 USC, Section 522a*, 1964.

[74] Association of Market and Social Research Organizations. Market and social research privacy principles. `www.amro.com.au/index.cfm?p=1635`, Australia, September 2003.

[75] Government of Ontario. Freedom of information and protection of privacy act. *Revised Statues of Ontario 1990, Chapter F.31*, 2005. `www.e-laws.gov.on.ca/DBLaws/Statutes/English/90f31_e.htm`.

[76] Jaehong Park and Ravi Sandhu. Towards usage control models: beyond traditional access control. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 57–64. ACM Press, 2002.

[77] Jaehong Park and Ravi Sandhu. The $\mathrm{UCON}_{\mathrm{abc}}$ usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.

[78] Jaehong Park and Ravi Sandhu. The $UCON_{ABC}$ usage control model. *ACM Trans. Information. and Systems. Security.*, 7(1):128–174, 2004.

[79] European Parliament. Directive 95/46/ec on the protection of individuals with regard to the processing of personal data and on the free movement of such data. *Official Journal of the European Communities L 281*, 24 October 1995.

[80] European Parliament. Directive 2002/58/ec concerning the processing of personal data and the protection of privacy in the electronic communications sector (directive on privacy and electronic communications). *Official Journal of the European Communities L 201*, 12 July 2002.

[81] Calvin Powers, Steve Adler, and Bruce Wishart. EPAL translation of the freedom of information and protection of privacy act. `www.ipc.on.ca/docs/EPAL%20FI1.pdf`, March 2004. White Paper.

[82] VNU Business Publications. Doubleclick apologises for privacy outcry. *Global News Wire*, 3 March 2000.

[83] Riccardo Pucella and Vicky Weissman. A formal foundation for ODRL. In *Workshop on Issues in the Theory of Security (WITS-04)*, 2004.

[84] Health Resources and Services Administration. Health insurance portability and accountability act of 1996. *Public Law 104-191*, 1996. H.R. 3103.

[85] Reuters. Privacy groups see danger in a merger. *The New York Times*, 22 June 1999. Section C, Page 6, Column 3.

[86] Matthias Schunter and Els Van Herreweghen. Enterprise privacy practices vs. privacy promises–how to promise what you can keep. Technical Report RZ 3452 (#93771), IBM Research, September 2002.

[87] Richard Shim. Tivo, comcast reach DVR deal. *News.com accessed Nov 2006*, 15 March 2005. `news.com.com/TiVo,+Comcast+reach+DVR+deal/2100-1041_3-5616961.html`.

[88] Einar Snekkenes. Concepts for personal location privacy policies. In *EC '01: Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 48–57, New York, NY, USA, 2001. ACM Press.

[89] William E. Spangler, Kathleen S. Hartzel, and Mordechai Gal-Or. Exploring the privacy implications of addressable advertising and viewer profiling. *Commun. ACM*, 49(5):119–123, 2006.

[90] Robert Thibadeau. A critique of P3P: Privacy on the Web. `dollar.ecom.cmu.edu/p3pcritique/`, 2000.

[91] W3C. XML path language (XPath). Recommendation, World Wide Web Consortium, Nov 1999. `www.w3c.org/TR/XPath`.

[92] W3C. A P3P preference exchange language 1.0 (APPEL1.0). Working draft, World Wide Web Consortium, April 2002. `www.w3.org/TR/P3P-preferences`.

[93] W3C. The Platform for Privacy Preferences 1.1 (P3P1.1). Working draft, World Wide Web Consortium, Feb 2006. `www.w3c.org/P3P`.

[94] Xin Wang, Guillermo Lao, Thomas DeMartini, Hari Reddy, Mai Nguyen, and Edgar Valenzuela. XrML – eXtensible rights Markup Language. In *XMLSEC '02: Proceedings of the 2002 ACM workshop on XML security*, pages 71–79, New York, NY, USA, 2002. ACM Press.

[95] Samuel D. Warren and Louis D. Brandeis. The right to privacy. *Harvard Law Review*, IV(5), December 1890.

[96] Duminda Wijesekera and Sushil Jajodia. Policy algebras for access control: the propositional case. In *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 38–47, New York, NY, USA, 2001. ACM Press.

[97] Duminda Wijesekera and Sushil Jajodia. A propositional policy algebra for access control. *ACM Trans. Information. and System. Security.*, 6(2):286–325, 2003.

[98] Ting Yu, Ninghui Li, and Annie I. Antón. A formal semantics for P3P. In *SWS '04: Proceedings of the 2004 workshop on secure web services*, pages 1–8, New York, NY, USA, 2004. ACM Press.