REASONING ABOUT AUTHORIZATION POLICIES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Victoria Rebecca Weissman

May 2007
This document is in the public domain.
An authorization policy states the conditions under which an action is permitted or forbidden. In this dissertation, we use formal methods to ensure that policies written in certain languages are unambiguous and to provide provably correct algorithms for reasoning about policies. For example, we describe how questions about entailment, such as “may Alice edit the database?”, can be answered efficiently.

We begin by showing that a fragment of first-order logic can be used to represent and reason about policies. Because we use first-order logic, policies have a clear syntax and semantics. We show that further restricting the fragment results in a language that is still quite expressive yet is also tractable. More precisely, questions about entailment can be answered in time that is a low-order polynomial (indeed, almost linear in some cases), as can questions about the consistency of policy sets.

In addition to developing our own language, we have examined two policy languages, XrML and ODRL. We focused on these languages because, when we began our work, they seemed to have the strongest support from industry. We found that the specifications for both languages have significant problems, which is not surprising since neither includes formal semantics. We discussed the problems that we found with the language developers and then proposed formal semantics for each language. We present our semantics here. In addition, we consider the complexity of determining if a permission is implied by a set of statements in each language.
We prove that the general problem for XrML is undecidable and the general problem for ODRL is decidable and NP-hard. Finally, we define fragments of both languages that are fairly expressive and for which the problem is polynomial-time computable.
BIOGRAPHICAL SKETCH

Vicky Weissman was raised in a suburb of Pittsburgh, Pennsylvania. She graduated from Shady Side Academy in 1992. She then studied Electrical Engineering at Cornell University, graduating in 1996. After college, Vicky worked for two years in Seattle, Washington; first as part of the hardware design team and then on the software side of a division of Cirrus Logic that created graphics chip. She returned to Cornell for a Master’s of Engineering degree in Computer Science and decided to stay for the doctorate, which is finally done.
First and foremost, I would like to thank my advisor Joseph Halpern. Whatever research skills I have come from working closely with him. I am grateful for the high standards he has tried to instill in me. Most of all, I am thankful that he was willing to discuss everything with me, down to the last detail. To learn, I have to understand why things are done a certain way, rather than just how they are done. So, to teach me, Joe had to justify almost every suggestion he made, and he did. It is his dedication and my persistence – put another way, our endless discussions on Lithium and XrML– that taught me what I know and made Chapters 2 and 3 what they are.

Spending years debating every idea with your advisor takes emotional strength. Joe provided some of it. Most came from Riccardo Pucella. Ric bought me my first research notebook and, when my work with Joe was going so slowly that I thought I would never be published, Ric collaborated with me on a small project that lead to my first paper. In short, Ric looked for ways to give me evidence that I was on my way to becoming a good researcher. On top of that, he never seemed to get tired of reassuring me. Chapter 4 is the result of our technical discussions, which I enjoyed immensely.

While Joe, Ric, and I were the stars of my dissertation saga, many other people played important roles. Renee Kirkwood has stood by me since I met her during my freshman year of college. Every member of my committee, which included Bill Arms, Tom Bruce, Joe Halpern, and Fred Schneider, actually read my dissertation. I know this because each member provided excellent feedback. Carl Lagoze not only encouraged me, he taught me about project management and always took the time to listen to my ideas and hone them with his practical knowledge. And
Ken Birman seemed to go out of his way to let me know that I was doing well.

My last year was difficult. During that time, several people let me know that they were rooting for me. Those people include Graeme Bailey, Eric Breck, Claire Cardie, Carla Gomes, Dexter Kozen, and Fred Schneider. I wish I knew a way to tell them how much their encouragement, when I needed it the most, has meant to me.

Finally, I would like to thank the many organizations and people who supported me financially for the last several years. They include the National Science Foundation (NSF) under grants CTC-0208535, ITR-0325453, and IIS-0534064; the Office of Naval Research (ONR) under grants N00014-00-1-03-41 and N00014-01-10-511; the Department of Defense’s Multidisciplinary University Research Initiative (MURI), which is administered by ONR under grants N00014-01-1-0795 and N00014-04-1-0725; and the Air Force Office of Scientific Research (AFOSR) under grants F49620-02-1-0101 and FA9550-05-1-0055. I thank Joe for convincing these organizations of the value of our work, thereby obtaining the grants for us. I thank my family for making it clear that whatever I needed, they were happy to provide. A safety net, even when it is not in use, is much appreciated.
# TABLE OF CONTENTS

1 Introduction

2 Lithium
   2.1 A First-Order Logic for Reasoning About Policies
   2.2 Intractability Results
   2.3 Identifying Tractable Sublanguages
      - 2.3.1 A Tractable Sublanguage
      - 2.3.2 Relaxing the Restrictions
   2.4 Consistency
   2.5 Usability
   2.6 Related Work
   2.7 Conclusion

3 XrML
   3.1 Syntax
   3.2 XrML’s Authorization Algorithm
      - 3.2.1 A Description of Query
      - 3.2.2 An Analysis of Query
      - 3.2.3 A Corrected Version of Query
   3.3 Formal Semantics
   3.4 Complexity
   3.5 The Entire XrML Language
   3.6 Negation
   3.7 MPEG-21 REL
   3.8 Concluding Remarks
4 ODRL

4.1 The ODRL Language ............................................. 94
4.2 A Semantics Using First-Order Logic .......................... 107
4.3 Queries ............................................................. 116
  4.3.1 Formal Definition ............................................ 116
  4.3.2 Complexity ................................................... 119
4.4 Discussion: Improving ODRL ................................. 123
4.5 Conclusion ......................................................... 126

5 Conclusion .................................................................. 127

5.1 XACML ................................................................. 127
5.2 Future Work .......................................................... 130

A Proofs for Chapter 2 .................................................. 133

B Proofs for Chapter 3 ................................................... 153

C Proofs for Chapter 4 ................................................... 185

Bibliography ................................................................ 198
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The Query Algorithm</td>
<td>61</td>
</tr>
<tr>
<td>3.2</td>
<td>The Auth Algorithm</td>
<td>64</td>
</tr>
<tr>
<td>3.3</td>
<td>The Holds Algorithm</td>
<td>65</td>
</tr>
<tr>
<td>3.4</td>
<td>The Query2 Algorithm</td>
<td>71</td>
</tr>
<tr>
<td>3.5</td>
<td>The Auth2 Algorithm</td>
<td>72</td>
</tr>
<tr>
<td>3.6</td>
<td>The Holds2 Algorithm</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Abstract Syntax for ODRL (Agreements)</td>
<td>97</td>
</tr>
<tr>
<td>4.2</td>
<td>Abstract Syntax for ODRL (Prerequisites)</td>
<td>98</td>
</tr>
<tr>
<td>4.3</td>
<td>Abstract Syntax for ODRL (Prerequisites Components)</td>
<td>99</td>
</tr>
<tr>
<td>4.4</td>
<td>Translation of ODRL Agreements</td>
<td>110</td>
</tr>
<tr>
<td>4.5</td>
<td>Translation of ODRL Prerequisites</td>
<td>111</td>
</tr>
<tr>
<td>B.1</td>
<td>Tree Representation of $d$</td>
<td>172</td>
</tr>
<tr>
<td>B.2</td>
<td>Dag Representation of $d$</td>
<td>172</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

What happens if we do not protect intellectual property rights? A consequence suggested by the Constitution of the United States is that fewer works will be created and, as a result, society as a whole will suffer. Another stance, taken by many countries of continental Europe, is that intellectual property rights must be protected as a moral imperative [CLOO02]; so, if we do not respect such rights, then we have acted immorally. Either way, the general consensus seems to be that creators of intellectual property should have some control over their work.

The first step to protecting intellectual property rights is to state precisely what those rights are. To do this, people write authorization policies, which include legislation and contracts. Authorization policies are also used to regulate actions outside of the intellectual property domain. Some examples are given at the beginning of Chapter 2 and, we suspect, the reader can find several more in her everyday life.

An authorization policy, henceforth referred to as simply a policy, describes the conditions under which an action, such as reading a file, is permitted or forbidden. Policies are typically described informally. As a result, their meaning and consequences are not always clear to the people reading them. To better understand the problem, consider the statement “only librarians may edit the on-line catalog”. We can view this statement as a policy because it governs who may edit the catalog, based on whether the editor is a librarian. It is not clear if this policy permits librarians to make changes to the catalog or only forbids anyone who is not a librarian from doing so. The policy could be rewritten to remove this particular
ambiguity, but others are likely to exist if policies are stated in a natural language.

Of course, policies do not need to be written in a natural language. Access control lists (ACLs) [Pfl97] have been used for decades to capture simple policies in an unambiguous way. Unfortunately, ACLs lack the expressive power needed by many of today’s digital-content providers. For example, we cannot capture the policy “members are permitted to access the digital library”; the best we can do using ACLs is to maintain a list that must be updated whenever the set of members changes. Another option is to write policies in an XML-based language. Three popular choices are XrML (eXtensible rights Markup Language) [Con01], ODRL (Open Digital Rights Language) [Ian01], and XACML (eXtensible Access Control Markup Language). These languages can be given formal semantics (in part because their syntax, unlike that of natural languages, is quite restricted) and they do have more expressive power than ACLs. Prior to our work, however, none of these languages had formal semantics and, as a result, policies written in any of the languages could be ambiguous.

The formal-methods community has proposed a number of languages that have formal semantics. Many of these languages are based on some extension of Datalog [GMUW02]. The extensions are tractable fragments of first-order logic that allow a limited use of function symbols and negation. Unfortunately, the extensions do not seem to have the necessary expressive power to capture a number of policies that are currently written in English. For example, in the iTunes Terms of Sale [AC04], certain actions are explicitly forbidden and others are unregulated; most variants of Datalog cannot distinguish between the two categories. Those that can (see for example [JSS97]) take an approach similar to XACML.

The goal of this dissertation is to provide a better system in which to write
policies and reason about them. As a first step, we define a language based on the following design requirements.

1. It must have a clear syntax and semantics.

2. It must be expressive enough to capture in an easy and natural way the policies that people want to discuss.

3. It must be tractable enough to allow interesting queries about policies to be answered efficiently.

To achieve our objectives, we use a fragment of first-order logic that we call Lithium.

Because Lithium is a fragment of first-order logic, it has a clear syntax and semantics; thus, it remains to argue that the logic satisfies the second and third goals listed above. Whether a logic is sufficiently expressive to satisfy the second objective naturally depends on the application. To evaluate our approach, we collected policies from content vendors, libraries, and government legislation. (A list of the collected policies is given at http://www.cs.cornell.edu/People/vickyw/collectedPolicies.html.) We examined the policies for statements that could not be readily captured in Lithium. In addition, we compared Lithium to the industry-endorsed languages XACML, XrML, and ODRL. Both of these investigations lead to improvements in Lithium, and we now believe that the language satisfies our second goal for a wide range of applications.

For the third goal, we focus on two key queries:

- Given a set of policies and an environment that provides all relevant facts (e.g., “Alice is a librarian”, “Anyone who is a librarian for less than a year is
a novice”, etc.), does it follow that a particular action, such as Alice editing the on-line catalog, is permitted or forbidden?

- Is a set of policies consistent? (A policy set is consistent if and only if it does not imply that an action is both permitted and forbidden.) This question is particularly interesting for collaboration. For example, suppose that Alice is writing the policies for her university’s new outreach program. If the union of her policies and the university policies is consistent, then she knows that her policies do not contradict those of the university.

The answers to these queries could be used by enforcement mechanisms and individuals who want to engage in regulated activities. More importantly, we believe that the answers provide a reasonably good understanding of the policies, increasing our confidence that the formal statements capture the informal rules and the informal rules capture the policy writer’s intent.

We use the insights gained from studying Lithium to give formal semantics to XrML and ODRL, and to identify tractable fragments of those languages. We examine XrML, in part, because it is becoming an increasingly popular language in which to write software licenses. When first released in 2000, XrML received the support of many technology providers, content owners, distributors, and retailers, including Adobe Systems, Hewlett-Packard Laboratories, Microsoft, Xerox Corp., Barnesandnoble.com, and Time Warner Trade Publishing. In fact, Microsoft, OverDrive, and DMDsecure have publicly announced their agreement to build products and/or services that are XrML compliant. Currently, XrML is being used by international standard committees as the basis for application-specific languages that are designed for use across entire industries. For example, the Moving Picture Experts Group (MPEG) has selected XrML as the foundation for their
MPEG-21 Rights Expression Language, henceforth referred to as MPEG-21 (see http://www.xrml.org). It is clear that a number of industries are moving towards a standard language for writing licenses and that many of these standard languages are likely to be based on XrML.

XrML does not have formal semantics. Instead, the XrML specification [Con01] presents the semantics in two ways. First is an English description of the language. Second is an English description of an algorithm that determines if a permission follows from a set of licenses. Unfortunately, the two versions of the semantics do not agree. To make matters worse, the algorithm has unintuitive consequences that do not seem to reflect the language developers’ intent.

To address these issues, we provide formal semantics for a representative fragment of XrML. In particular, we give a translation from licenses in XrML to formulas in first-order logic extended with a validity operator. We argue that the translation preserves the meaning of the XrML statements by proving that the algorithm included in the XrML document, slightly modified to correct the unintuitive behavior, matches our semantics. More precisely, the algorithm says that a permission follows from a set of licenses if and only if the translated permission is a logical consequence of the translated licenses. We then consider the complexity of determining if a permission is implied by a set of licenses. We show that the general problem is undecidable but, for an expressive fragment of the language, it is decidable in polynomial time.

ODRL is a popular open-source alternative to XrML that has been endorsed by nearly twenty organizations including

• Nokia, a multi-industry conglomerate focused on mobile communications;

• the DAFNE project (District Architecture for Networked Editions), a re-
search project funded by the Italian Ministry of Education, University and Research to develop a prototype of the national infrastructure for electronic publishing in Italy;

- the RoMEO Project (Rights MEtadata for Open archiving), created to investigate rights management of “self-archived” research in the United Kingdom academic community.

ODRL developers are currently working with a number of communities, including Creative Commons and Dublin Core, to address their needs. The complete list of supporters and on-going projects can be found at www.odrl.net; however, this small sample already illustrates the widespread impact that ODRL has on rights management.

the ODRL specification does not include a formal semantics, and we are not aware of any attempts to give it a formal semantics other than ours. Rather than including formal semantics, the ODRL specification describes the meaning of ODRL statements in English and, as a result, agreements written in ODRL can be ambiguous. For example, suppose that Alice owns two printers, Printer One and Printer Two, and Bob is a potential user. To regulate Bob’s access to the printers, Alice and Bob write an agreement in ODRL that says only this: Bob is permitted to use Printer One or Bob is permitted to use Printer Two. The agreement clearly allows Bob to use at least one of the printers, but it does not say which one. If Alice assumes the choice is hers, since the agreement does not say otherwise, and Bob believes the choice is his, since the agreement arguably implies this, then Alice and Bob disagree on the meaning of the agreement. Moreover, because this type of underspecification is possible in ODRL, Alice and Bob cannot use the ODRL specification to resolve the dispute.
We believe that even conscientious well-trained people might write ambiguous agreements because, in practice, agreements can be fairly large and complex. The ambiguities can lead to disputes and resolving those disputes can be costly in terms of time, money, the reputation of the writers, and more. To avoid these costs, we propose the first formal semantics for ODRL and define when a permission (or prohibition) follows from a set of ODRL statements. It follows from our work that the meaning of agreements written in ODRL are not open to interpretation.

To give ODRL formal semantics, we had to resolve the ambiguities in the specification. Most of the aspects were clarified through discussions with Renato Iannella, editor of the ODRL specification and Chief Scientist at IPR systems at the time of the specification’s release. Unfortunately, Dr. Iannella could not answer all of our questions, because some of them revealed subtleties in the language that had not been considered. We highlight these ambiguities and then take what we consider to be a reasonable approach to resolving them.

We give formal semantics to ODRL by defining a translation from the key components in ODRL to formulas in a fragment of many-sorted first-order logic. The formal semantics can be used as a foundation for answering queries. For example, answering a query of the form “Does a particular permission (or prohibition) follow from a set of ODRL statements” corresponds to deciding whether the translation of the statements implies the permission (or prohibition). Answering this particular type of query is of obvious practical importance. Unfortunately, we show that the problem is NP-hard. The intractability result is due, at least in part, to a component that is not clearly defined in the specification and seems to require further consideration by the language developers. If we remove this troublesome construct, then we can answer our queries in polynomial time.
The response of industry to our work has been positive. Our interactions with the MPEG-21 development team is an example. When we first decided to give XrML formal semantics, the MPEG committee had released a beta version of its language, which was XrML with minor revisions, and was preparing the final release. We chose to give semantics to the beta language first (before analyzing the official XrML specification, as is done here), because we hoped that any problems we found would be corrected in the final version of MPEG-21. This is, in fact, what occurred. The MPEG Standards Committee released their ISO standard [MPE04] after we discussed our results with two of its members Thomas DeMartini and Xin Wang; the shortcomings that we identified are addressed in that standard. Most importantly, the standard has formal semantics. We conjecture that all of our complexity results for XrML hold with minor changes for MPEG-21, although we have not verified the details. Similar events are unfolding within the ODRL Working Group. In particular, we have discussed our findings on ODRL with Renato Iannella and Susanne Guth of the ODRL Working Group. Since then, work has begun on a new version of the language. One of the seven design requirements is that the new release must have formal semantics; we have been asked to collaborate with the group to provide the semantics.

The work presented in this dissertation is based on four papers. Lithium is introduced in “Using First-Order Logic to Reason about Policies” [HW03]. Our formal semantics of XrML was first proposed in “A Formal Foundation for XrML” [HW04]. The research presented in both of these papers are collaborations with Joseph Y. Halpern. Formal semantics for ODRL was first given in “A Formal Foundation for ODRL” [PW04]. This work was done with Riccardo Pucella. Finally, the conclusion of this dissertation is, in essence, a revision of “Towards a
Policy Language for Humans and Computers” [WL04], which is a paper written with Carl Lagoze.

Throughout the chapters, we assume knowledge of first-order logic at the level of Enderton [End72]. More specifically, we assume the reader is familiar with the syntax of first-order logic, including constants, variables, predicate symbols, function symbols, and quantification; with the semantics of first-order logic, including relational models and valuations; and with the notions of satisfiability and validity of first-order formulas. Recall that many-sorted first-order logic is first-order logic modified so that each term is associated with a sort (i.e., type); variables of sort $s$ range over the elements of sort $s$; and the signatures of predicate and function symbols restrict each argument to elements of a particular sort.

The rest of this dissertation is organized as follows. Lithium is presented in Chapter 2. In Chapters 3 and 4, we give formal semantics to XrML and ODRL, respectively. We conclude in Chapter 5.
Chapter 2

Lithium

In Chapter 1, we give two reasons why certain actions might be restricted according to a set of policies. The first is to encourage creation by giving authors some control over their works. The second is to respect the arguably moral right of a creator to the product of her labor. Other motivations depend on circumstance. For example, a hospital might adopt certain policies to protect their patients’ privacy, and to comply with government regulations. As another example, the Greek Orthodox Archdiocese of America restricts access to its website, in part, to discourage people from using the available information in a disrespectful way [NPPW03]. As a final example, Jane Hunter has documented the need for indigenous peoples, such as the Australian Aboriginal and Torres Strait Islander communities, to restrict access to digital content based on their traditional laws [Hun02]. In all of these examples, the goal is to regulate actions according to a set of policies.

To satisfy the goal, we need to determine whether a requested action is permitted or forbidden by the policies. It is easy to see that the answer to such a question might depend on the environment (i.e., the context) in which the policies are evaluated. For example, the policy “doctors are permitted to edit the patient database” implies that Alice may edit the database in an environment that implies she is a doctor. The same policy neither permits nor forbids Alice to edit the database in an environment that is silent on whether she is a doctor. In practice, we expect the application to maintain a database of policies and a database of environment facts. An individual might provide additional environment facts, including certificates such as a driver’s license or an employee badge, when requesting to perform
In this chapter, we present a fragment of first-order logic called Lithium. Lithium is a language in which people can state policies and users can query whether a specific action is permitted or forbidden in a given environment. We believe, and will argue throughout this chapter, that Lithium is sufficiently expressive to capture a wide range of queries that are of practical interest. In addition, we show that the queries can be answered efficiently. Our approach actually goes beyond stating and answering queries. We can also detect inconsistencies. That is, given a set of policies and an environment, we can determine if some action (e.g., Alice editing the database) is both permitted and forbidden. If a policy set is inconsistent in a given environment, then either the policy writer truly intends to permit and forbid the same action, which seems unlikely, or there is an error in the policies or in the environment. So, by finding inconsistencies, we can identify errors and, by notifying the policy writer, we can help her revise the policies and environment to better match her intent.

The rest of this chapter is organized as follows. In Section 2.1, we formally define our notions of policy and environment. We also give examples that illustrate how policies can be represented in an appropriate fragment of first-order logic. Sections 2.2 and 2.3 focus on queries about permissions (that is, whether a specific action is permitted given an environment and a set of policies); all our results hold with essentially no change for queries about prohibitions. We show in Section 2.2 that such queries are, in general, hard to answer. In Section 2.3, we consider some restrictions that we believe are usually satisfied in practice; under these restrictions, the queries are tractable. Lithium is the set of queries that satisfy the restrictions. We examine the consistency question (that is, whether some
action is both permitted and forbidden by a set of policies) in Section 2.4. In Section 2.5, we discuss what can be done to make Lithium accessible to users who are not conversant with first-order logic. Related work is discussed in Section 2.6 and concluding remarks are given in Section 2.7. Most of the proofs are in the appendix.

2.1 A First-Order Logic for Reasoning About Policies

We use many-sorted first-order logic with equality over some vocabulary $\Phi$ to express and reason about policies. Let $L^{fo}(\Phi)$ denote the set of first-order formulas over the vocabulary $\Phi$. For this paper, we assume that there are at least three sorts, Actions (e.g., accessing a file), Subjects (the agents that perform actions; these are sometimes called principals in the literature), and Times. While these sorts seem natural for any policy logic, other sorts may be desired for particular applications. These sorts, including objects and roles, may be added to the logic without affecting our results.

The vocabulary $\Phi$ is application-dependent; however, we assume that $\Phi$ contains a constant $\text{now}$ of sort $\text{Times}$ and a binary predicate $\text{Permitted}$ on $\text{Subjects} \times \text{Actions}$. The constant $\text{now}$ denotes the current time. In practice, a global clock would determine the interpretation of $\text{now}$. $\text{Permitted}(t, t')$ means that subject $t$ is allowed to perform action $t'$. It might be useful to add additional arguments to $\text{Permitted}$, such as when the action is permitted and who is authorizing the granting or revoking of the permission. Which arguments are added (if any) depend on the application and on other choices made in the vocabulary. For example, consider a policy $p$ that says “Alice is permitted to append image $m$ to file $f$”. We could either take $\text{Permitted}$ to be a binary predicate and $\text{append}$ to be a binary


function, and express \( p \) as “Permitted(Alice, append\((m, f)\))”; or we could take append to be a unary function and Permitted to be a ternary predicate, and express \( p \) as “Permitted(Alice, append\((m, f)\))”. Our results apply regardless of which choice is made, because they do not depend on the arity of Permitted and the other functions and predicates in the language. In fact, our results still hold even if policies refer to different variants of Permitted, with different arities.

A policy is a closed first-order formula of the form

\[
\forall x_1 \ldots \forall x_m (f \Rightarrow (\neg)\text{Permitted}(t, t')),
\]

where \( f \) is any first-order formula, \( t \) and \( t' \) are terms of sort Subject and Action respectively, and the notation \((\neg)\text{Permitted}\) indicates that the \text{Permitted} predicate may or may not be negated. Defining a policy in this way provides a structure that matches our intuition, namely, that a policy is a set of conditions under which an action is or is not permitted.

To illustrate how policies can be expressed in first-order logic, consider the following examples.

**Example 2.1.1.** The policy “only librarians may edit the catalog” can be characterized by the following two formulas:

\[
\forall x (\neg \text{Librarian}(x) \Rightarrow \neg \text{Permitted}(x, \text{edit the catalog}))
\]

\[
\forall x (\text{Librarian}(x) \Rightarrow \text{Permitted}(x, \text{edit the catalog})).
\]

(Depending on the intended meaning of the English statement, the first formula by itself may characterize the policy.)

**Example 2.1.2.** The policy “a customer may download any article if she has paid a fee within the past six weeks” can be rewritten as “if an individual \( i \) has paid
the fee within the past six weeks, \( i \) is a customer, and \( a \) is some article, then \( i \) may download \( a \)”.

The policy can be encoded readily as

\[
∀i∀t∀a((\text{PaidFee}(i, t) \land (\text{now} - 6 < t < \text{now})) \\
\land \text{Customer}(i, \text{now}) \land \text{Article}(a)) \\
\Rightarrow \text{Permitted}(i, \text{download}(a))). \]

**Example 2.1.3.** The policy set “anyone may sing” and “anyone who is allowed to sing may dance” can be characterized by the following two formulas:

\[
∀x(\text{Permitted}(x, \text{sing})) \\
∀x(\text{Permitted}(x, \text{sing}) \Rightarrow \text{Permitted}(x, \text{dance})).
\]

To determine the consequences of policies, we need to know which facts are true in the *environment* (i.e., the context in which the policies are applied). For example, if the environment implies that Alice is a librarian, then the policies in Example 2.1.1 imply that she may edit the catalog. If the environment is silent as to whether Alice is a librarian, then the policies in Example 2.1.1 do not regulate her actions. The *environment* may include specific statements such as “Alice is a librarian”, “*The Cat in the Hat* is a children’s book”, or “Sally has a junior library card”. General statements may also be included, such as the conditions under which a customer is considered to be in good standing and “at all times, there is a senior staff member who is on call”. All the examples we have considered so far confirm our belief that first-order logic is sufficiently expressive to capture most environments that are likely to arise in practice. Thus, we formally define an environment to be a closed first-order formula that does not contain the *Permitted* predicate. The requirement that the environment not contain *Permitted* encourages the intuitive separation between the environment, which is a description of reality, and the policies, which are the rules governing that reality.
The two types of queries discussed in the introduction can now be formalized. The first query, is an individual \( t \) permitted to perform an action \( t' \) (where \( t \) and \( t' \) are closed terms) given an environment \( E \) and some policies \( p_1, \ldots, p_n \), amounts to asking if the formula \( E \land p_1 \land \ldots \land p_n \Rightarrow \text{Permitted}(t, t') \) is valid. (Similarly, \( t \) is forbidden to do \( t' \) if and only if \( E \land p_1 \land \ldots \land p_n \Rightarrow \neg\text{Permitted}(t, t') \) is valid.) The second query, “Are the policies consistent?”, asks if the formula \( E \land p_1 \land \ldots \land p_n \) is satisfiable. For ease of exposition, we focus on determining if an action is permitted (or forbidden). As we show, it is easy to modify our techniques to handle the consistency question.

### 2.2 Intractability Results

In general, the two types of queries in which we are interested cannot be answered efficiently. Indeed, the problem in its full generality is easily seen to be undecidable if the vocabulary \( \Phi \) has at least one binary predicate other than \text{Permitted} (and closed terms \( t \) and \( t' \) of sort \text{Subjects} and \text{Actions}, respectively, so that it is possible to actually form queries). To see this, let \( f \) be an arbitrary formula that does not contain \text{Permitted}. Consider the policy \( f \Rightarrow \text{Permitted}(t, t') \), and let the environment be empty (i.e., \text{true}). Standard manipulations show that

\[
\text{true} \land (f \Rightarrow \text{Permitted}(t, t')) \Rightarrow \text{Permitted}(t, t')
\]

is equivalent to

\[
f \lor \text{Permitted}(t, t').
\]

Since \( f \) does not mention \text{Permitted}, the last formula is valid iff \( f \) is valid. The validity problem for first-order formulas is well known to be undecidable, even if we restrict to formulas whose only nonlogical symbol is a binary predicate.
In fact, undecidability holds if we further restrict to formulas that have a single alternation of quantifiers (i.e., formulas of the form $Q_1 x_1 \ldots Q_n x_n R_1 y_1 \ldots R_m y_m f$, where $Q_i = \exists$ and $R_j = \forall$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$ or vice-versa, and $f$ is quantifier-free) [BGG97]. So, in general, we cannot determine whether a single policy implies a permission if writing the policy as a first-order formula requires an alternation of quantifiers and a binary predicate other than $Permitted$. It turns out that undecidability holds even without the assumption that $\Phi$ has a binary predicate other than $Permitted$.

**Theorem 2.2.1.** Let $L_0$ be the set of closed formulas of the form

$$(f \Rightarrow Permitted(c, c')) \Rightarrow Permitted(c, c'),$$

where $c$ and $c'$ are constants of the appropriate sorts, $f$ has a single alternation of quantifiers, and the only nonlogical symbol in $f$ is $Permitted$. The validity question for $L_0$ is undecidable.

Not surprisingly, similar undecidability results hold if we allow formulas in the environment to involve an alternation of quantifiers (provided that there is a binary predicate in the language other than $Permitted$, since we do not allow $Permitted$ in the environment). Given Theorem 2.2.1, it seems that our only hope is to forbid any alternation of quantifiers.

How much quantification do we really need? A quantifier-free environment suffices to capture simple databases. Many applications, however, need a richer environment that includes general properties, such as “men are not women” and “a senior citizen is anyone over sixty-five years old”. For these applications, universal quantification is needed in the environment. In addition, almost all applications need quantification in their policies. To see why, notice that if we do not allow
a policy to have any quantification (i.e., define a policy to have the form \( f \Rightarrow \text{Permitted}(t, t') \) where \( t \) and \( t' \) are closed terms and \( f \) is quantifier-free), then each policy must govern a specific individual and action. For example, we can say “If Alice is good, she may play outside”, but we cannot say “All good children may play outside”. Because policies typically permit an individual to do an action based on the attributes of that individual, we must allow policies to be universally quantified.

All policies expressible in XrML and in ODRL, as well as the policies that we have collected from libraries and government databases, can be written as universal formulas (i.e., as formulas that can be written in the form \( \forall x_1 \ldots \forall x_n f \), where \( f \) is quantifier-free). Some of the policies that we collected may appear to need existential quantification, but they can be converted to equivalent universal formulas. Example 2.2.2 illustrates how we can apply standard first-order transformations to do the conversion.

**Example 2.2.2.** Consider the policy “anyone who is accompanied by a librarian may enter the stacks”. A natural way to state this in first-order logic is

\[
\forall x_1 (\exists x_2 (\text{Librarian}(x_2) \land \text{Accompanies}(x_2, x_1)) \Rightarrow \text{Permitted}(x_1, \text{enter(stacks)}))
\]

This formula is logically equivalent to

\[
\forall x_1 \forall x_2 ((\text{Librarian}(x_2) \land \text{Accompanies}(x_2, x_1)) \Rightarrow \text{Permitted}(x_1, \text{enter(stacks)}))
\]

which uses only universal quantification. \(\blacksquare\)

Note that \textit{enter} is a function in Example 2.2.2. Unfortunately, it is well known that the validity problem for existential formulas with function symbols is undecidable, even if we restrict to formulas with only two existentials and one unary
function symbol [BGG97]. The following strengthening of Theorem 2.2.1 is almost immediate.

**Theorem 2.2.3.** Let $\mathcal{L}_1$ be the set of closed formulas of the form

$$\forall x_1 \forall x_2 (f \Rightarrow \text{Permitted}(c, c')) \Rightarrow \text{Permitted}(c, c'),$$

where $c$ and $c'$ are constants of the appropriate sort and $f$ is a quantifier-free formula whose only nonlogical symbols are $\text{Permitted}$ and a unary function. The validity problem for $\mathcal{L}_1$ is undecidable.

Theorem 2.2.3 suggests that even if we drastically reduce quantification, we still need to disallow functions to get decidability. Once we severely restrict quantification and remove functions entirely, then we do get a decidable fragment, but it is not tractable. Recall that $\Pi^P_2$ is the second level of the polynomial hierarchy, and represents languages that can be decided in co-NP with an NP oracle.

**Theorem 2.2.4.** Let $\Phi$ be a vocabulary that contains $\text{Permitted}$, constants $c$ and $c'$ of sorts Subjects and Actions, respectively, and possibly other predicate and constant symbols (but no function symbols). Assume that there is a bound on the arity of the predicate symbols in $\Phi$ (that is, there exists some $N$ such that all predicate symbols in $\Phi$ have arity at most $N$). Finally, let $\mathcal{L}_2$ be the set of all closed formulas in $\mathcal{L}^{fo}(\Phi)$ of the form $E \land p_1 \land \ldots \land p_n \Rightarrow \text{Permitted}(c, c')$ such that $E$ is a conjunction of quantifier-free and universal formulas and each policy $p_1, \ldots, p_n$ has the form $\forall x_1 \ldots \forall x_m (f \Rightarrow \text{Permitted}(t_1, t_2))$, where $t_1$ and $t_2$ are terms of the appropriate sort and $f$ is quantifier-free.

(a) The validity problem for $\mathcal{L}_2$ is in $\Pi^P_2$.

(b) If $\mathcal{L}_3$ is the set of formulas in $\mathcal{L}_2$ in which every policy’s antecedent is a conjunction of literals, then the validity problem for $\mathcal{L}_3$ is $\Pi^P_2$ hard.
(c) If $\mathcal{L}_4$ is the set of $\mathcal{L}_2$ formulas in which $E$ is quantifier-free, then the validity problem for $\mathcal{L}_4$ is both NP-hard and co-NP hard.

If we do not require the arity of the predicate symbols in $\Phi$ to be bounded, then we must replace $\Pi_2^P$ by co-NEXPTIME (co-nondeterministic exponential time) in parts (a) and (b) [BGG97].

Theorems 2.2.1, 2.2.3, and 2.2.4 seem to suggest that the questions we are interested in are hopelessly intractable. Fortunately, things are not nearly as bad as they seem.

2.3 Identifying Tractable Sublanguages

The work on Datalog and its variants mentioned in the introduction demonstrates that there are useful, tractable fragments of first-order logic. In this section we define Lithium, a fragment of first-order logic characterized by a different set of restrictions than those considered by the Datalog community, show that these restrictions lead to tractability, and argue that they are particularly well-suited to reasoning about policies.

As a first step towards defining Lithium, we characterize the classes of environments and policies that are likely to occur in practice. A basic environment is an environment that is a conjunction of ground literals. Basic environments are sufficiently expressive to capture the information in databases and certificates. While this is adequate for many applications, basic environments cannot represent general properties such as “every citizen of Germany is a member of the European Union”. To capture these, we define a standard environment to be an environment that is a conjunction of ground literals and closed formulas of the form $\forall x_1 \ldots \forall x_n (\ell_1 \land \ldots \land \ell_k \Rightarrow \ell_{k+1})$, where $\ell_1, \ldots, \ell_{k+1}$ are literals. Each
conjunct of a standard environment is an *environment fact*. Note that every basic environment is a standard environment. A *standard policy* is a policy of the form $\forall x_1 \ldots \forall x_n (\ell_1 \land \ldots \land \ell_k \Rightarrow \text{Permitted}(t_1, t_2))$, where $\ell_1, \ldots, \ell_{k+1}$ are literals and both $t_1$ and $t_2$ are terms of the appropriate sort. Standard environments and standard policies are sufficiently expressive for all of the applications that we have considered. A *simple policy* is a standard policy where none of the literals in the antecedent mentions *Permitted*. For example, $\text{Permitted}(t_1, t_2) \Rightarrow \text{Permitted}(t_1, t_3)$ is not a simple policy.

A *policy base* is a formula of the form $E \land P$, where $E = E_0 \land E_1$ is a standard environment, $E_0$ is a conjunction of ground literals, $E_1$ is a conjunction of universally quantified formulas, and $P$ is a conjunction of standard policies. In the rest of the paper, when we write standard queries, we assume that the formulas $E, E_0, E_1,$ and $P$ satisfy these constraints (that is, $E$ is a standard environment of the form $E_0 \land E_1$; $E_0$ is a basic environment; and so on). We are interested in characterizing policy bases $E \land P$ for which it is tractable to determine whether the query $E \land P \Rightarrow \text{Permitted}(t, t')$ is valid, where $t$ and $t'$ are terms of the appropriate sort. We call such a query a *standard query*.

In the next section, we define a set of restrictions on standard queries that guarantee that validity can be determined quickly. After presenting the restrictions, we evaluate the likelihood that the restrictions will hold in practice. In subsequent sections, we relax each of the restrictions to accommodate a wider range of applications without sacrificing tractability. Roughly speaking, Lithium, which is formally defined in Section 2.3.2, is the set of standard queries that satisfy the relaxed restrictions.
2.3.1 A Tractable Sublanguage

We use the following terms to define the initial set of restrictions. A variable \( v \) is constrained in a clause \( c \) if \( v \) appears as an argument to \textbf{Permitted} in \( c \).

For example, both \( x \) and \( y \) are constrained in the clause \( \forall x \forall y \forall z (\neg R(x, z) \lor \text{Permitted}(x, y)) \); \( z \) is not constrained. Two literals \( \ell \) and \( \ell' \) are unifiable if there are variable substitutions \( \sigma \) and \( \sigma' \) such that \( \ell \sigma = \ell' \sigma' \). For example, \( R(x, c_1) \) and \( R(c_2, y) \) are unifiable by substituting \( c_2 \) for \( x \) and \( c_1 \) for \( y \), while \( R(x, c_1) \) and \( R(y, c_2) \) are not unifiable (if \( c_1 \) and \( c_2 \) are distinct constants).

Let \( f \) be a formula in CNF\(^1\) and let \( \ell \) be a literal in \( f \). We say that \( \ell \) is bipolar in \( f \) if there is another literal \( \ell' \) in \( f \) such that \( \ell \) and \( \neg \ell' \) are unifiable. The pair \( \ell, \ell' \) is called a bipolar pair. For example, consider the formula \( f = \forall x (\text{Permitted}(x, \text{nap}) \Rightarrow \text{Permitted}(\text{Advisor}(x), \text{nap})) \), which in CNF is \( \forall x (\neg \text{Permitted}(x, \text{nap}) \lor \text{Permitted}(\text{Advisor}(x), \text{nap})) \). Because \( \neg \text{Permitted}(x, \text{nap})[x/\text{Advisor}(y)] = \neg \text{Permitted}(\text{Advisor}(x), \text{nap})[x/y] \), the literals \( \neg \text{Permitted}(x, \text{nap}) \) and \( \text{Permitted}(\text{Advisor}(x), \text{nap}) \) are bipolar in \( f \); together they form a bipolar pair.

\textbf{Theorem 2.3.1.} Let \( \mathcal{L}_5 \) consist of all standard queries of the form \( E \land P \Rightarrow \text{Permitted}(t, t') \) such that

\begin{enumerate}
  \item \( E \) is basic (i.e., \( E \) is a conjunction of ground literals),
  \item there are no bipolar literals in \( P \),
  \item equality is not mentioned in \( E \land P \), and
\end{enumerate}

\(^1\)We say that a first-order formula is in CNF if it has the form \( c_1 \land \ldots \land c_n \), where each \( c_i \) has the form \( Q_1 x_1 \ldots Q_m x_m(\varphi) \), each \( Q_j \in \{\forall, \exists\} \), and \( \varphi \) is a (quantifier-free) disjunction of literals, for \( i = 1, \ldots, n \) and \( j = 1, \ldots, m \). Each \( \varphi \) is called a clause. We sometimes identify a universal formula in CNF with its set of clauses.
(4) every variable appearing in a conjunct \( p \) of \( P \) is constrained in \( p \).

The validity of formulas in \( \mathcal{L}_5 \) can be determined in time \( O(\|P\| |\text{Permitted}(t, t')| + |E| \log |E|) \), where \( |\varphi| \) denotes the length of \( \varphi \), when viewed as a string of symbols.

The language \( \mathcal{L}_5 \) includes formulas such as

\[
\text{Student}(\text{Alice}) \land \text{Good}(\text{Alice}) \land \\
\forall x (\text{Student}(x) \Rightarrow \text{Permitted}(x, \text{work})) \land \\
\forall x (\text{Student}(x) \land \text{Good}(x) \\
\Rightarrow \text{Permitted}(x, \text{play})) \Rightarrow \text{Permitted}(\text{Alice}, \text{play})
\]

(may Alice play given that Alice is a student, Alice is good, all students may work, and all good students may play). Unlike Theorem 2.2.4(c), function symbols are allowed by Theorem 2.3.1. Moreover, there is no assumption that the arity of predicates and functions in the vocabulary is bounded. The price we pay for this added generality and for cutting the complexity to linear in the number of policies (which could well be large), linear in the length of the permission being considered (which is almost certainly small), and not much more than linear in the size of the database (which we expect to be relatively small, particularly if the user is expected to provide most of the environment facts when making a request) is the four restrictions. We now discuss the likelihood that the restrictions will hold in practice; in subsequent sections we consider how the restrictions can be relaxed.

As we have already said, basic environments are sufficiently expressive to capture the facts stored in databases and certificates. They are also sufficiently expressive for library applications that we considered and for policies that can be written in XrML or ODRL, since both languages assume a minimal environment containing facts such as the number of times that a particular subject has done
a specific action (e.g., printing a file) and the current time. It is true, however, that basic environments are not always enough. For example, the documents that describe who may collect Social Security define an aged person to be anyone 65 years old or older, who is a resident of the U.S., and is either a citizen or an alien residing in the U.S. both legally and permanently. A basic environment cannot capture this definition.

The second restriction, that there are no bipolar literals in $P$, is likely to hold if all the policies are permitting policies (that is, their conclusions have the form $\text{Permitted}(t_1, t_2)$) or all are denying policies (that is, their conclusions have the form $\neg \text{Permitted}(t_1, t_2)$).

To see why, recall that a permitting policy says ‘if the following conditions hold, then a particular action is permitted’. These conditions typically include requirements that someone possess one or more credentials, such as a library card or a driver’s license. It is fairly rare that not having a credential, such as not having a driver’s license, increases an individual’s rights. Therefore, we do not expect credentials to correspond to bipolars. Similar arguments may be made for other types of information.

If the policy set includes a mix of permitting and denying policies, then it seems less likely that the bipolar restriction will hold. For example, suppose that an individual may smoke if and only if she is over eighteen years old. We could write this statement as two policies

$$p_1 = \forall x(\text{GreaterThan}(\text{age}(x), 18) \Rightarrow \text{Permitted}(x, \text{smoke}))$$

$$p_2 = \forall x(\neg \text{GreaterThan}(\text{age}(x), 18) \Rightarrow \neg \text{Permitted}(x, \text{smoke})).$$

Note that $p_1$ is a permitting policy, $p_2$ is a denying policy, and every literal in $p_1 \land p_2$ is bipolar in $p_1 \land p_2$. 
The third restriction, that equality is not used, is satisfied by most of the policies and environment facts that we collected. However, the restriction is violated by threshold policies (e.g., “if Alice is blackballed by at least two people, then she may not join the club”) and by statements that say two distinct names refer to the same individual (e.g., “Alice Smith = wifeOf(Bob Smith)”)

The last restriction, that every variable appearing in a policy $p$ is constrained in $p$, holds if an individual is granted or denied permission based solely on her attributes and the attributes of the regulated action. Notice that the policies in Examples 2.1.1 and 2.1.3 have this form, but the policies in Examples 2.1.2 and 2.2.2 do not. In particular, whether the policy in Example 2.2.2 allows $x_1$ to enter the stacks depends on an attribute of some other individual $x_2$.

Before relaxing the restrictions, we briefly discuss why they are sufficient for tractability. The first three restrictions allow us to consider each policy individually, that is, $E \land P \Rightarrow \text{Permitted}(t, t')$ is valid iff $E \land p \Rightarrow \text{Permitted}(t, t')$ is valid for some conjunct $p$ in $P$.

**Proposition 2.3.2.** Suppose that $E \land P \Rightarrow \text{Permitted}(t, t')$ is a standard query in which $E$ is basic, the equality symbol is not mentioned in $E \land P$, and there are no bipolars in $P$. Then $E \land P \Rightarrow \text{Permitted}(t, t')$ is valid iff there is a conjunct $p$ of $P$ such that $E \land p \Rightarrow \text{Permitted}(t, t')$ is valid.

If the last restriction holds, then we can determine quickly whether $E \land p \Rightarrow \text{Permitted}(t, t')$ is valid for some conjunct $p$ of $P$.

### 2.3.2 Relaxing the Restrictions

In this section, we consider the extent to which we can relax the four restrictions given in Theorem 2.3.1, while still maintaining tractability. We consider each of
Beyond Basic Environments

There is an obvious generalization of Theorem 2.3.1: we simply remove the first restriction and replace every reference to $P$ with $E_1 \land P$, where $E_1$ is the conjunction of universal statements in $E$. This results in three restrictions: there are no bipolar literals in $E_1 \land P$, equality is not mentioned in $E_1 \land P$, and every variable appearing in a conjunct $c$ of $E_1 \land P$ is constrained in $c$. Unfortunately, because $\text{Permitted}$ does not appear in the environment, the variable restriction holds only if the environment has no quantification. In addition, we can prove that, if there are no bipolar literals in $E_1 \land P$, then $E \land P \Rightarrow \text{Permitted}(t, t')$ is valid if and only if $E$ is inconsistent or $E_0 \land P \Rightarrow \text{Permitted}(t, t')$ is valid, where $E_0$ is the conjunction of ground literals in $E$. This means that a universal statement in the environment can affect the validity of a query only if it makes the environment inconsistent.

To support interesting universal statements in the environment, we must relax the restrictions on bipolar literals and variables, which we do in Sections 2.3.2 and 2.3.2, respectively.

Relaxing the Bipolar Restriction

If we allow bipolar literals in $E_1 \land P$, then a permission might follow from a set of policies without following from any single policy. In other words, the conclusion of Proposition 2.3.2 might not hold.

**Example 2.3.3.** Consider two policies $p_1$ and $p_2$, where $p_1$ says “Alice may cry if she is happy” and $p_2$ says “Alice may cry if she is not happy”. Formally,

$$p_1 = \text{Happy}(\text{Alice}) \Rightarrow \text{Permitted}(\text{Alice}, \text{cry})$$

and
\[ p_2 = \neg \text{Happy}(\text{Alice}) \Rightarrow \text{Permitted}(\text{Alice, cry}). \]

Clearly, \( p_1 \Rightarrow \text{Permitted}(\text{Alice, cry}) \) is not valid, because Alice might not be happy. Similarly, \( p_2 \Rightarrow \text{Permitted}(\text{Alice, cry}) \) is not valid, because Alice might be happy. But \( p_1 \land p_2 \Rightarrow \text{Permitted}(\text{Alice, cry}) \) is valid, because Alice is either happy, in which case she may cry by \( p_1 \), or she is not happy, in which case she may cry by \( p_2 \). So Alice’s right to cry doesn’t follow from either policy individually, but follows from both policies together, essentially because \( p_1 \land p_2 \) includes the bipolar pair \((\text{Happy}(\text{Alice}), \neg \text{Happy}(\text{Alice}))\).

Example 2.3.3 shows how we can use bipolar literals to infer a statement, namely Alice may cry, from two clauses, namely \( p_1 \) and \( p_2 \). Resolution [NS97] generalizes the reasoning in this example. To understand how resolution works, let \( c \) be the clause \( \forall x_1 \ldots \forall x_n (\ell \Rightarrow d) \) and let \( c' \) be the clause \( \forall x'_1 \ldots \forall x'_m (\ell' \Rightarrow d') \), where \( \ell \) and \( \ell' \) are literals. Suppose that \( \sigma \) and \( \sigma' \) are variable substitutions such that \( \ell \sigma = \neg \ell' \sigma' \). It is easy to see that \( c \land c' \Rightarrow d \sigma \lor d' \sigma' \) is valid. Using standard terminology, we call \( c \) and \( c' \) the parents of the resolvent \( d \sigma \lor d' \sigma' \), and we say that \( c \) and \( c' \) resolve on \( \ell \sigma \) to create \( d \sigma \lor d' \sigma' \).\(^2\)

The closure under resolution of a universal formula \( f \), denoted \( R(f) \), is the smallest set of clauses that includes the clauses in \( f \) (when \( f \) is in CNF) and is closed under resolution, that is if \( e \) is the resolvent of two distinct clauses in \( R(f) \), then \( e \) is in \( R(f) \). Roughly speaking, the resolvents in \( R(f) \) are all the clauses that can be inferred from the clauses in \( f \).

Our interest in resolution is motivated in part because we can prove that a sta-

\(^2\)Actually, the resolvent is created using a particular substitution, called a most general unifier, which is essentially the substitution that replaces variables with constants only when necessary. For example, the most general unifier for \( c = \forall y (\neg \text{R}(y) \Rightarrow \text{S}(y)) \) and \( c' = \forall x (\text{R}(\text{f}(x)) \Rightarrow \text{S}(\text{g}(x))) \) substitutes \( \text{f}(x) \) for \( y \), instead of substituting \text{Alice} for \( x \) and \( \text{f}(\text{Alice}) \) for \( y \). So, the resolvent of \( c \) and \( c' \) is \( \forall x (\text{S}(\text{f}(x)) \lor \text{S}(\text{g}(x))) \). (See [NS97] for details.)
standard query $q$ of the form $E_0 \land E_1 \land P \Rightarrow \text{Permitted}(t, t')$ that does mention equality is valid iff there is a clause $c \in R(E_1 \land P)$ such that $E_0 \land c \Rightarrow \text{Permitted}(t, t')$ is valid. The role of the bipolar restriction in the language $L_5$ is also best understood in the context of resolution. Part of our approach to guaranteeing tractability involves keeping $R(E_1 \land P)$ small. If there are no bipolar literals in $E_1 \land P$, then $R(E_1 \land P)$ includes only the conjuncts of $E_1$ and $P$; there are no resolvents. We can also prove that $R(E_1 \land P)$ is still fairly small if each conjunct in $E \land P$ has at most one bipolar literal. As a result, we maintain tractability if there is at most one bipolar literal in each conjunct (see Theorem 2.3.7). However, if even a single conjunct of $E_1 \land P$ has two bipolars, and the other conjuncts have at most one bipolar each, then $R(E_1 \land P)$ can be infinite.

Example 2.3.4. Suppose we have two policies; the first is “Alice may play” and the second is “for all individuals $x_1$ and $x_2$, if $x_1$ may play and $x_2$ is $x_1$’s boss, then $x_2$ may play”. We can write these policies as

\[
p_1 = \text{Permitted}(\text{Alice}, \text{play})
\]
\[
p_2 = \forall x_1 \forall x_2 (\text{Permitted}(x_1, \text{play}) \land \text{BossOf}(x_2, x_1) \Rightarrow \text{Permitted}(x_2, \text{play}))
\]

It is not hard to see that for any integer $n$, the closure of $p_1 \land p_2$ includes the clause

\[
(\bigvee_{i=1,\ldots,n} \neg \text{BossOf}(x_i, x_{i+1})) \lor \neg \text{BossOf}(x_0, \text{Alice}) \lor \text{Permitted}(x_n, \text{play}),
\]

which says that if $x_0$ is Alice’s boss, $x_1$ is $x_0$’s boss, $\ldots$, and $x_n$ is $x_{n-1}$’s boss, then $x_n$ may play. $\blacksquare$

While many policy bases that arise in practice have no more than one bipolar literal in each clause, we have found two relatively common situations in which this is not the case. The first is when policies refer to properties that are, intuitively, defined in the environment. The second is when the policy set includes both
permitting and denying policies (that is, the set has policies with \textbf{Permitted} in the conclusion and policies with \textbf{¬Permitted} in the conclusion).

To see why the bipolar restriction might be violated in the presence of definitions, consider a video store that has three types of customers: regular, gold, and platinum. Every adult member is permitted to send queries to the store’s helpdesk, where adulthood is defined by the state in which the individual resides. In New York, an individual is an adult if she is over twenty-one years old. In Alaska, an individual is an adult if she is over eighteen. Formally,

\begin{align*}
p_1 &= \forall x (\text{Adult}(x) \land \text{Member}(x) \Rightarrow \text{Permitted}(x, \text{query helpdesk})) \\
e_1 &= \forall x (\text{Over21}(x) \land \text{InNY}(x) \Rightarrow \text{Adult}(x)) \\
e_2 &= \forall x (\text{Over18}(x) \land \text{InAK}(x) \Rightarrow \text{Adult}(x)) \\
e_3 &= \forall x (\text{RegMember}(x) \Rightarrow \text{Member}(x)) \\
e_4 &= \forall x (\text{GoldMember}(x) \Rightarrow \text{Member}(x)) \\
e_5 &= \forall x (\text{PlatinumMember}(x) \Rightarrow \text{Member}(x))
\end{align*}

Roughly speaking, \(e_1\) and \(e_2\) define the notion of being an adult, while \(e_3\), \(e_4\), and \(e_5\) define the notion of being a member. These definitions are used in \(p_1\) to regulate who may send queries to the helpdesk. It is easy to see that \(p_1\) has two bipolar literals in \(p_1 \land e_1 \land \ldots \land e_5\), namely \text{Adult}(x) and \text{Member}(x). Therefore, the bipolar restriction does not hold in this example. More generally, if a policy \(p\) mentions \(k\) terms that are defined in the environment, then \(p\) will include \(k\) bipolar literals.

Definitions in this spirit arise frequently in government legislation, including the US Code (see \url{http://www4.law.cornell.edu/uscode/} for an electronic version) and the Privacy Rule \cite{oHS00}. For example, Title 1 Chapter 1 Section 1 of the US Code says “the words ‘insane’ and ‘insane person’ and ‘lunatic’ shall include
every idiot, lunatic, insane person, and person non compos mentis”. (Observe
that $\forall x (\text{Idiot}(x) \Rightarrow \text{Insane}(x))$ is part of this definition; the rest can be similarly
encoded.) So handling definitions is a matter of practical importance. Perhaps the
simplest approach is to rewrite the policy $p_1$ so as to replace the defined predicates
in the antecedent by their definitions. This will result in an equivalent policy base
with no bipolars. The effect of replacing Adult and Member by their definitions
in our example is to replace $p_1$ by the six policies in $P_{NY} \cup P_{AK}$, where

$$P_{NY} = \{ \forall x (\text{Over21}(x) \land \text{InNY}(x) \land \text{Pr}(x) \\
\Rightarrow \text{Permitted}(x, \text{query helpdesk})) : \\
\text{Pr} \in \{\text{RegMember}, \text{GoldMember}, \text{PlatinumMember}\}) \}
$$

$$P_{AK} = \{ \forall x (\text{Over18}(x) \land \text{InAK}(x) \land \text{Pr}(x) \\
\Rightarrow \text{Permitted}(x, \text{query helpdesk})) : \\
\text{Pr} \in \{\text{RegMember}, \text{GoldMember}, \text{PlatinumMember}\}) \}
$$

Notice that there are no bipolars in $\bigwedge_{p \in P_{NY} \cup P_{AK}} p$ and the policies permit the same
actions as $p_1 \land e_1 \land \ldots \land e_5$.

Our translation illustrates a potential problem with this approach: it can blow
up the size of the policy set. Suppose that a policy $p$ has $m$ bipolar literals and
that literal $i$ is defined using $c_i$ clauses. Rewriting would result in replacing policy
$p$ by $c_1 \times \cdots \times c_m$ policies. Each of the new policies can also be longer than $p$, although the total length of each one can be no more than $|E_1|$, where $E_1$ is the
first-order part of the environment. Is this so bad? Examples in the social security
database and in the Privacy Rule suggest that typically $m$ is less than three and $i$
is less than five, in which case definitions do not significantly reduce the efficiency
of our procedures.
In practice, we can often improve efficiency by removing definitions that are irrelevant when answering queries in a given environment. Continuing our earlier example, suppose that $E_0$ is the environment that results from Alice by presenting certificates that show she is a regular member who is over eighteen and in Alaska (i.e., $E_0 = \text{RegMember}(Alice) \land \text{Over18}(Alice) \land \text{InNY}(Alice)$). It is easy to see that we can remove $e_1$, $e_4$, and $e_5$ without changing the set of permissions that are implied by the policy base. In practice, we believe that this single optimization will usually result in each $c_i$ being one (i.e., every predicate is defined by at most one clause), in which case our approach to handling definitions does not increase the number of policies mentioned in the query. As an aside, this optimization is one of many that are well-known in the theorem-proving community. We suspect that, by applying the appropriate optimizations, we can answer queries substantially faster than is indicated by the worst-case complexity results given in Theorem 2.3.7.

We next show how we can deal with policy bases that have both permitting and denying policies. This task would be easy if we could consider only the permitting policies (ignoring the denying policies) when determining if an action is permitted. Unfortunately, if we do this, then we might not answer queries correctly.

To see why, consider an environment $E$ that says “Alice is a student” and a policy set $\mathcal{P} = \{p_1, p_2, p_3\}$, where $p_1$ says “faculty members may chair committees”, $p_2$ says “students may not chair committees”, and $p_3$ says “anyone who is not a faculty member may take naps”. We can write these policies as

\[
p_1 = \forall x (\text{Faculty}(x) \Rightarrow \text{Permitted}(x, \text{chair committees})) ,
\]
\[
p_2 = \forall x (\text{Student}(x) \Rightarrow \neg \text{Permitted}(x, \text{chair committees})) ,
\]
\[
p_3 = \forall x (\neg \text{Faculty}(x) \Rightarrow \text{Permitted}(x, \text{nap})) .
\]
Clearly, \(p_1\) and \(p_3\) are permitting policies and \(p_2\) is a denying policy. Because \(p_1\) is equivalent to \(\forall x (\neg \text{Permitted}(x, \text{chair committees}) \Rightarrow \neg \text{Faculty}(x))\), \(p_1\) and \(p_2\) together imply that no student is a faculty member. (Intuitively, students cannot be faculty members, because no one can be both permitted and not permitted to chair committees.) Because students are not faculty members, Alice, being a student, is not a faculty member and, by \(p_3\), may take a nap. We cannot determine that Alice may nap if we consider only the permitting policies, because to derive the permission we need the environment fact that is implied by \(p_1 \land p_2\).

If each fact implied by a permitting and denying policy together were derivable from either the environment or a single policy, then we could separate the permitting policies from the denying policies. Intuitively, this is because the interaction would not provide any information that was not already known. To formalize this intuition, note that each implied fact corresponds to a resolvent of a permitting and denying policy. In the previous example, the implied fact that students are not faculty members corresponds to the resolvent of \(p_1\) and \(p_2\), namely \(\forall x (\text{Faculty}(x) \Rightarrow \neg \text{Student}(x))\). Therefore, if every resolvent of a permitting and denying policy is already implied by the environment or a single policy, then we can separate the policies. Continuing our example, we could separate the policies if the environment said that students were not faculty members. A closer analysis shows that we need to consider only those resolvents that are created by resolving on a literal that mentions \textbf{Permitted}.

To formalize all of this, we need to discuss permitting and denying policies in a bit more detail. Observe that a policy such as \(\forall x (\text{Permitted}(\text{Alice}, a) \Rightarrow \text{Permitted}(\text{Bob}, a))\) is logically equivalent to both a permitting policy and a denying policy, where \(\forall x (\neg \text{Permitted}(\text{Bob}, a) \Rightarrow \neg \text{Permitted}(\text{Alice}, a))\) is the deny-
ing policy. We say that a policy is \textit{pure} if it is not logically equivalent to both a permitting and a denying policy. For example, policies that do not mention \textbf{Permitted} in the antecedent (which is the case for almost all the policies we have collected) are guaranteed to be pure.

\textbf{Theorem 2.3.5.} Suppose that $E$ is a standard environment, $P$ is a conjunction of pure permitting policies, and $D$ is a conjunction of (not necessarily pure) denying policies such that, for every resolvent $f$ created by resolving a conjunct of $P$ and a conjunct of $D$ on a literal that mentions \textbf{Permitted}, either $E \Rightarrow f$ is valid or $q \Rightarrow f$ is valid for some conjunct $q$ of $P \land D$. Then, for all terms $t$ and $t'$ of the appropriate sort, $E \land P \land D \Rightarrow \textbf{Permitted}(t, t')$ is valid iff $E \land P \Rightarrow \textbf{Permitted}(t, t')$ is valid.

We can always add clauses to a policy base to obtain an equivalent policy base that satisfies the antecedent of Theorem 2.3.5. Therefore, the key question is not “how likely are these conditions to hold in practice”, but “how many clauses are we going to have to add in practice so that these conditions hold”. Example 2.3.4 shows that we may need to add an infinite number of policies to the set. However, in practice, policies are often simple. (Recall that a policy $p$ is simple if the antecedent of $p$ does not mention \textbf{Permitted}.) If every policy in a policy base is simple, then every resolvent is an environment fact and there is, at most, one resolvent per pair of permitting and denying policies. So, if the policy base mentions $n$ policies, all simple, then we can satisfy the antecedent of Theorem 2.3.5 by adding at most $n^2$ clauses to the environment.

Adding clauses to the environment, however, can have an unfortunate consequence. Suppose that $E$, $P$, and $D$ are as defined in Theorem 2.3.5 and $E'$ is $E$ extended so that the antecedent of Theorem 2.3.5 holds. Then the policy
bases $E' \land P$ and $E' \land D$ might violate the bipolar restriction, even if $E \land P$ and $E \land D$ do not. To illustrate the problem, recall our earlier example in which policy $p_1$ says “faculty members may chair committees”, $p_2$ says “students may not chair committees”, and $p_3$ says “anyone who is not a faculty member may take a nap”. Consider a policy set that consists of $p_1$, $p_2$, $p_3$, and a policy $p_4$ that says “anyone who is not a student may not enter the student-only website” $(\forall x(\neg \text{Student}(x) \Rightarrow \neg \text{Permitted}(x, \text{enter student site})))$. To satisfy the conditions of Theorem 2.3.5, we could add a clause $e$ to the environment that says “students are not faculty” ($e = \forall x(\text{Students}(x) \Rightarrow \neg \text{Faculty}(x))$). By Theorem 2.3.5, we can now separate the permitting and denying policies. However, determining if a permission is denied might be an intractable problem because the denying policies together with $e$ violate the bipolar restriction; $e$ has two bipolar literals in $e \land p_2 \land p_4$.

In this example, we can avoid the problem by satisfying the antecedent of Theorem 2.3.5 in another way. Rather than adding $e$ to the environment, we could replace $p_1$ by the policy $p_1' = \forall x(\text{Faculty}(x) \land \neg \text{Student}(x) \Rightarrow \text{Permitted}(x, \text{chair committees}))$, which says that faculty members who are not students may chair committees. Note that every clause in $p_1' \land p_3$ has at most one literal that is bipolar in $p_1' \land p_3$, and every clause in $p_2 \land p_4$ has at most one literal that is bipolar in $p_2 \land p_4$. So the antecedent of Theorem 2.3.5 holds and the resulting policy bases satisfy the bipolar restriction. We suspect that, in practice, a policy base $E \land P \land D$ either satisfies the bipolar restriction or can be converted to an equivalent policy base $E' \land P \land D$ that satisfies the antecedent of Theorem 2.3.5 and has the property that both $E' \land P$ and $E' \land D$ satisfy the bipolar restriction. We have not, however, done an extensive check.
Instead of adding these clauses to the environment automatically, it might be better to verify the changes with the policy maker. To see why, recall the two policies “faculty members may chair committees” and “students may not chair committees”. We could satisfy the antecedent of Theorem 2.3.5 by adding the fact “no student is a faculty member” to the environment. But suppose that there is (or could one day be) a student who is also a faculty member. Then the policy maker may want to revise the policies to take this into account, rather than allowing the environment to (possibly) become inconsistent. In general, we expect that the additional facts needed to satisfy the antecedent of Theorem 2.3.5 will be ones that either the user would agree should have been there all along or are ones that should not be there and in fact suggest that the policies should be rewritten.

Dealing With the Equality Restriction

To explain how we can relax the equality restriction, we need two definitions. We say that a standard query \( q \) of the form \( E_0 \land E_1 \land P \Rightarrow \text{Permitted}(t, t') \) is equation-free if no conjunct of \( E_0 \land E_1 \land P \), when written in CNF, has a disjunct of the form \( t = t' \). (Note that an equation-free query may mention equality in its antecedent, but only in the scope of negation when the antecedent is written in CNF. Thus, for example, the query \( a \neq b \land (c = d \Rightarrow \text{Permitted}(t, t')) \Rightarrow \text{Permitted}(t, t') \) is equation-free.) It is easy to see that Theorem 2.3.1 applies to equation-free queries; it is only positive occurrences of = that cause problems.

We can actually go slightly beyond equation-free queries. If \( F_0 \) is the conjunction of equality statements in \( E_0 \), then \( q \) is equality-safe provided that \( E_1 \land P \) (when written in CNF) has no clause with a disjunct of the form \( t = t' \) and it is not the case that \( F_0 \Rightarrow t = t' \) is valid, where \( t \) and \( t' \) are closed terms that
appear in $E_0$ and either $t$ is a subterm of $t'$ or both $t$ and $t'$ mention function symbols. For example, $q$ is not equality-safe if $E_0$ includes the conjunct $c = f(c)$, the conjunct $f(c) = f'(c')$, or both $f(c) = c'$ and $c' = f'(c')$ (since these together imply $f(c) = f'(c')$).

Note that the notion of equality-safe is a generalization of equation-free; an equality-safe query can have some conjuncts in $E_0$ where equality does not appear in the scope of a negation, as long as not too much can be inferred from those equality statements. The following proposition shows that we can efficiently convert an equality-safe query $q$ to an equation-free query $q'$ such that $q$ is valid iff $q'$ is valid. Thus, we can determine the validity of an equality-safe query by first transforming it to an equation-free query and then applying the techniques discussed previously.

**Proposition 2.3.6.** If $q$ is an equality-safe standard query, then there is a standard query $q'$ of the form $E'_0 \land E'_1 \land P' \Rightarrow \text{Permitted}(t, t')$ such that (a) $q$ is valid iff $q'$ is valid, (b) $q'$ is equation-free, and (c) $|q'| = O(|q||L'_q|)$, where $L'_q$ is the length of the longest term in $q$. Moreover, we can find such a $q'$ in time $O(|q|)$.

Example A.0.9 in the appendix illustrates the procedure for converting $q$ to $q'$, and shows problems that arise if we allow queries that are not equality-safe. The example also shows that the transformation procedure can increase the number of bipolar literals. Since we need to restrict the number of bipolars for tractability, our theorems must refer to the number of bipolars after the transformation. We say that $\ell$ and $\ell'$ are unifiable relative to a set $E$ of equality statements if there are variable substitutions $\sigma$ and $\sigma'$ such that it follows from $E$ that $\ell\sigma = \ell'\sigma'$. For example, $P(a)$ and $P(b)$ are unifiable relative to $a = b$, and $\text{Permitted}(\text{Alice}, \text{nap})$ and $\text{Permitted}(\text{wifeOf}(x), \text{nap})$ are unifiable relative to $\text{Alice} = \text{wifeOf}(\text{Bob})$. 
Similarly, we can talk about a literal $\ell$ being bipolar in a formula $f$ relative to $E$. If every conjunct in $E_1 \land P$ has at most one literal that is bipolar in $E_1 \land P$ relative to the equality statements in $E_0$, then after the transformation each conjunct will have at most one bipolar literal, which is what we need for tractability.

As we show in Theorem 2.3.7, we can handle equality-safe formulas. This suffices to handle the use of equality in all of the library and government policies that we collected, as well as the uses of equality in XrML and ODRL.

**The Effect of Unconstrained Variables**

Let $q$ be a standard query of the form $E_0 \land E_1 \land P \Rightarrow \text{Permitted}(t, t')$ that satisfies the four restrictions of Theorem 2.3.1 (possibly relaxed as discussed in Sections 2.3.2 and 2.3.2). These restrictions essentially guarantee that (a) $q$ is valid if and only if there is a clause $c$ in $R(E_1 \land P)$ such that $E_0 \land c \Rightarrow \text{Permitted}(t, t')$ is valid and (b) $R(E_1 \land P)$ is relatively small. (This is made precise in Section 2.3.2.) The role of the variable restriction is to ensure that, for each $c$ in $R(E_1 \land P)$, we can quickly determine whether $E_0 \land c \Rightarrow \text{Permitted}(t, t')$ is valid. We now relax the variable restriction in a way that preserves this property.

Let $c$ be a conjunct of $E_1 \land P$. A variable $v$ is *constrained in $c$ relative to $q$* if $v$ appears as an argument to a literal that mentions $\text{Permitted}$, is a disjunct of $c$, and is not bipolar in $E_1 \land P$ relative to the equality statements in $E_0$. For example, consider the query “may Alice read file $A$” given that Alice is Ms. Jones, Alice may copy any file to any destination, and if Ms. Jones may copy a file to a destination, then she may read that file. We can write this query as

$$q = (\text{Alice} = \text{Ms. Jones}) \land p_1 \land p_2 \Rightarrow \text{Permitted}(\text{Alice}, \text{Read(file A)}),$$
where
\[ p_1 = \forall x_1 \forall x_2(\text{Permitted}(\text{Alice}, \text{copySrcDst}(x_1, x_2))), \text{ and} \]
\[ p_2 = \forall x_1 \forall x_2(\text{Permitted}(\text{Ms. Jones}, \text{copySrcDst}(x_1, x_2)) \Rightarrow \text{Permitted}(\text{Ms. Jones}, \text{Read}(x_1))). \]

Note that \( \text{Permitted}(\text{Ms. Jones, Read}(x_1)) \) is the only literal in \( p_1 \land p_2 \) that is not bipolar in \( p_1 \land p_2 \) relative to \( \text{Alice} = \text{Ms. Jones} \). It follows that no variable is constrained in \( p_1 \) relative to \( q \); \( x_1 \) is constrained in \( p_2 \) relative to \( q \); and \( x_2 \) is not constrained in \( p_2 \) relative to \( q \).

If every literal in every conjunct \( c \) of \( E_1 \land P \) mentions at most one variable that is not constrained in \( c \) relative to \( q \), then it is not hard to show that every literal in every clause \( c' \) in \( R(E_1 \land P) \) mentions at most one variable that is not constrained in \( c' \). (Recall that a variable is constrained in a clause \( c \), as opposed to being constrained in \( c \) relative to a query, if it appears in \( c \) as an argument to \( \text{Permitted} \)). It turns out that this property is sufficient to ensure that we can quickly determine the validity of \( E_0 \land c \Rightarrow \text{Permitted}(t, t') \) for each \( c \) in \( R(E_1 \land P) \). It can also be shown that if every conjunct \( c \) of \( E_1 \land P \) mentions at most \( k \) variables that are not constrained in \( c \) relative to \( q \), then every clause \( c \) in \( R(E_1 \land P) \) mentions at most \( 2k \) variables that are not constrained in \( c \). In this case, it can be shown that the validity of \( E_0 \land c \Rightarrow \text{Permitted}(t, t') \) for a clause \( c \) in \( R(E_1 \land P) \) can be determined in time exponential in \( 2k \). (All these claims are made precise in Theorem 2.3.7.) It follows that if \( k \) is less than three, which is likely to be the case in practice, then we can remove the variable restriction entirely and still answer queries in a reasonable period of time.
Putting It All Together

With all this machinery, we can finally define the fragment of first-order logic that we believe to be appropriate for expressing policies. Lithium consists of all equality-safe standard queries \( E_0 \land E_1 \land P \Rightarrow \text{Permitted}(t, t') \) such that every conjunct in \( E_1 \land P \) has at most one literal that is bipolar in \( E_1 \land P \) relative to the equality statements in \( E_0 \).

We can quickly determine whether a query \( q \) is in Lithium. To determine if \( q \) is equality-safe, we create equivalence classes for the terms in \( E_0 \), which takes linear time, and then verify that each class has at most one term that is not a constant, which also takes linear time. To determine if the bipolar restriction holds, we choose a term from each equivalence class to represent the class (choosing a term that is not a constant if possible) and replace each term in the query by its representative. The bipolar restriction holds if each conjunct \( c \) of \( E_1 \land P \) has at most one literal that is bipolar in \( E_1 \land P \), which we can check in quadratic time.

The discussion in Sections 2.3.2, 2.3.2, and 2.3.2 suggests that queries in Lithium are tractable. The following theorem makes this precise. We prove the theorem only for equation-free Lithium queries, but by Proposition 2.3.6, it applies to equality-safe queries as well (although the complexity statements would have to be changed to take into account the possible increase in size when converting from equality-safe to equation-free queries).

Let \( L_f \) be the length of the longest clause in a CNF formula \( f \), and let \( L'_f \) be the length of the longest term in \( f \).

**Theorem 2.3.7.** The validity of an equation-free Lithium query \( q = E_0 \land E_1 \land P \Rightarrow \text{Permitted}(t, t') \) with \( m \) terms in \( E_0 \) can be determined in time \( O((|E_0| + T|E_1 \land P|^2) \log |E_0|)) \), where \( T = mL_{E_1 \land P}L'_{E_1 \land P} |\text{Permitted}(t, t')| \) if every literal in every
conjunct \( c \) of \( E_1 \land P \) mentions at most one variable that is not constrained in \( c \) relative to \( q \); otherwise, 
\[ T = m^{2k}L_{E_1 \land P}L'_{E_1 \land P}[\text{Permitted}(t, t')] \]
where every conjunct \( c \) of \( E_1 \land P \) has at most \( k \) variables that are not constrained in \( c \) relative to \( q \).

Theorem 2.3.7 shows that Lithium is tractable. Is it sufficiently expressive? The bipolar restriction holds in all of the applications that we considered, provided that definitions and mixed policy sets are handled as described in Section 2.3.2. We believe that our examples are representative, and that in fact the restriction will hold in practice. The restriction to equality-safe queries is likely to hold for applications that do not include a threshold policy; that is, a policy of the form “if \( k \) instances of \( P \) hold then subject \( t_1 \) may do action \( t' \)”.

If an application includes threshold policies, then the restriction is still likely to hold provided that the environment stores the number of relevant instances of \( P \) that hold rather than the instances themselves. For example, the threshold policy “if two people blackball Alice, then she may not join the club” can be written in Lithium if the environment stores the number of people who blackball Alice (e.g., \( \text{numOfBlackballers} = 2 \)), instead of who blackballs Alice (e.g., \( \text{Blackballs}(\text{Bob, Alice}) \land \text{Blackballs}(\text{Carol, Alice}) \)).

### 2.4 Consistency

Recall that a policy set is inconsistent if it both permits and forbids the same action. By detecting inconsistencies, we can warn policy writers that their policies probably do not match their intentions. We expect that this ability will be particularly important if the policy set is large or if it is created and maintained by more than one person. In addition, we can verify that a policy base \( P \) is consistent with a policy base \( P' \) by checking that \( P \cup P' \) is consistent. For example, suppose that
access to a patient’s medical file is regulated by the hospital’s policies and state law. If the union of the two policy bases is consistent, then the hospital’s policies do not contradict state law. (Note that the converse is not necessarily true.)

Clearly, \( E \land P \) is not consistent iff both \( E \land P \Rightarrow \text{Permitted}(c, c') \) and \( E \land P \Rightarrow \neg\text{Permitted}(c, c') \) are valid, for arbitrary constants \( c \) and \( c' \). Thus, if the two queries are in Lithium, then we can apply our previous techniques to show that we can efficiently check consistency. However, we can say even more. If the condition of Theorem 2.3.5 (or the corresponding condition for determining prohibitions) is met, then we automatically have consistency, provided that \( E \) is consistent.

**Theorem 2.4.1.** Suppose that \( E \) is an environment, \( P \) is a conjunction of pure permitting policies, and \( D \) is a conjunction of (not necessarily pure) denying policies such that the antecedent of Theorem 2.3.5 holds. Then \( E \land P \land D \) is satisfiable iff \( E \) is satisfiable.

Thus, in addition to making it feasible to check the consequences of policies, our conditions essentially prevent users from writing inconsistent policies. This is a major benefit of adhering to these restrictions!

2.5 Usability

In this section, we consider ways to make Lithium accessible to people who are not conversant with first-order logic. The restrictions on bipolars and equality in Lithium might be difficult to explain to non-logicians, but we suspect that teaching people to write standard queries can be done quickly, particularly if syntactic sugar is used to help the medicine do down.

We are currently designing usability tests to verify that computer programmers
can learn to translate English sentences to standard (sugared) queries quickly. The “sugaring” involves, for example, rewriting “∀x₁⋯∀xₙ(ℓ₁ ∧⋯∧ℓₖ ⇒ ℓₖ₊₁)” as “type₁ x₁;⋯, type₁ xₙ; if ℓ₁ and ⋯ and ℓₖ then ℓₖ₊₁”, where type₁ is the sort of variable xᵢ. We are focusing on programmers because, if this community can read and write queries, then they can build user interfaces for other communities, along with translators that convert user input to queries. Of course, input entered through a user interface can also be translated directly to a (non-sugared) standard query. For example, it should be possible to write a form-based interface that allows users to enter queries, which can then be translated directly to Lithium. Such a form-based interface was sketched in the conference version of the paper [HW03]. We have not pursued it because we feel it is better to write an interface for programmers.

The key question is how we should explain the bipolar and equality restrictions to policy writers. One option is to define a fragment of Lithium that is easy to explain to non-logicians and fairly expressive. For example, let S be the set of standard queries in which the environment is basic (a conjunction of Permitted-free literals), the policies are simple (Permitted is not mentioned in the antecedent), and the antecedents of policies are negation-free. It is easy to see that every query in S is in Lithium. Another example is Rosetta [WL04]. Rosetta is a fragment of (somewhat stilted) English in which queries can be written. All queries that can be expressed in Rosetta are guaranteed to be convertible to Lithium. Finally, graphical interfaces can be designed in such a way that every query written using the interface can be translated to Lithium. We conjecture that the Information Rights Management system that is part of Microsoft’s Office, Professional Edition 2003 [Mic03] is an example of this approach, although we have not verified
that all policies written through these interfaces satisfy the bipolar and equality restrictions. In short, we believe that, for many applications, there is a fragment of Lithium that is both sufficiently expressive and accessible to users with minimal training. Which fragment is appropriate depends on the capabilities of the users and the needs of the application.

Another approach is to give policy writers guidelines and tools to help them write policy bases that satisfy our requirements. For example, we might suggest that policy writers try to minimize their use of negation, equality, and universal formulas in the environment. We can provide tools to check if proposed policy bases are likely to lead to queries that are in Lithium. In practice, we expect policy writers to define the universal formulas in the environment and the policies (i.e., $E_1 \land P$); individuals then present certain credentials (i.e., $E_0$) along with a request (i.e., $\text{Permitted}(t, t')$). In this setting, we can check if the bipolar restriction and equality restriction are satisfied by $E_1 \land P$ and, if so, we can conclude that every query of the form $E_0 \land E_1 \land P \Rightarrow \text{Permitted}(t, t')$ is in Lithium provided that $E_0$ is equality-free. This allows us to identify potential problems at “compile time” and alert the policy writer, who might then choose to change the policies and environment to more closely adhere to the guidelines.

Perhaps the simplest solution is to not do anything at all. We believe, and have argued throughout this paper, that queries in practice are likely to be in Lithium. So users might not need to understand the restrictions on bipolar literals and equality, because they will naturally write queries that satisfy our requirements. We can build a verifier to check that a user’s query is either in Lithium or can be converted to Lithium using the techniques discussed in Section 2.3.2. If a query is in Lithium, then the user is assured that her question will be answered efficiently.
Otherwise, the verifier issues a warning. The warning could be ignored since our algorithm for answering queries might still run efficiently or, since warnings are likely to be rare, an expert could be consulted.

We expect that all of these strategies will allow naive users to express their queries in Lithium easily. It is up to the application developers to decide which approach is best in their setting.

2.6 Related Work

There has been a great deal of work on policy languages. Since we cannot hope to review all of the work in only a few pages, we restrict our attention to some of the best-known approaches and to those that seem most similar to Lithium.

The classic approach in the Computer Science community is arguably the one taken by UNIX. Every policy in UNIX can be expressed as a formula of the form \( \forall x(R(x, r) \Rightarrow \text{Permitted}(x, \text{act}(r))) \), where \( R \in \{\text{User}, \text{Group}, \text{Other}\} \), \( \text{act} \in \{\text{read}, \text{write}, \text{execute}\} \), and \( r \) is a constant typically representing a file or directory. The corresponding environment can be written as a conjunction of ground literals. It is easy to see that every query in UNIX can be written in Lithium. However, UNIX follows the Principle of Fail-safe Defaults [SS75], so the UNIX approach to answering a query is somewhat different than that taken by Lithium. In particular, UNIX assumes that every action not explicitly permitted is forbidden. Thus, with an empty environment (\( \text{true} \)), the UNIX response to the query \( \neg \text{Permitted}(\text{Alice}, \text{read}(\text{file f})) \) would be yes, while the Lithium response would be no (it does not logically follow from \( \text{true} \) that Alice is not permitted to edit file f. We can modify Lithium to give the same answers to queries as UNIX, simply by saying that the answer to a query of the form \( \neg \text{Permitted}(t, t') \) given
a policy base \( b \) is yes iff \( b \Rightarrow Permitted(t, t') \) is not acceptably valid; that is, a prohibition holds if and only if the corresponding permission does not. Since we know how to determine whether a permission holds, we can determine if a prohibition holds according to the revised definition. This modified version of Lithium can also capture the way policies are evaluated using access control lists. We believe that we can also capture whether a permission follows in SPKI/SDSI [EFL\textsuperscript{+99a}, EFL\textsuperscript{+99b}] from a collection of certificates in Lithium, although we have not checked the details.

Perhaps the most talked-about policy language in industry today is the XML-based language XACML [Mos05]. Every XACML query can be written as a standard query in which all policies are simple (the antecedents of policies are \textit{Permitted}-free) and the environment is basic (a conjunction of \textit{Permitted}-free ground literals). There are two significant differences between XACML and Lithium. The first is that users of XACML are expected to provide an algorithm for determining whether a permission is granted, denied, or unregulated by a policy base, as a function of whether the permission is granted, denied, or unregulated by the individual polices in that policy base. For example, the \textit{deny-overrides} algorithm (which is one of the built-in algorithms provided by XACML) says a permission is denied if it is denied by any single policy, is permitted if it is not denied by any single policy and is permitted by at least one, and is unregulated otherwise. Lithium essentially allows only one algorithm, which is logical consequence (a choice which cannot in fact be expressed in XACML, since it may depend on the interaction between the policies in a policy base). We could, of course, modify the way Lithium handles queries to match any particular algorithm, although doing this may result in losing many of the unique features of Lithium.
The second key difference between XACML and Lithium is the treatment of negation. In XACML, the semantics of negation is somewhat nonstandard. For example, in XACML, the policies “if Alice is good, then she may play” and “if Alice is not good, then she may play” together do not necessarily imply that Alice may play. The policies imply the permission only if the environment says either that Alice is good or that she is not good. So, given a set of XACML policies, we can replace every literal of the form \( \neg R(t_1, \ldots, t_n) \) by \( \text{NotR}(t_1, \ldots, t_n) \), where \( \text{NotR} \) is a fresh predicate symbol, without changing the meaning of the policies. Thus, although XACML seems to allow the unrestricted use of negation, it is actually less expressive than Lithium in its use of negation. Moreover, we believe that the nonstandard usage of negation may well confuse users.

Another XML-based language that has received widespread support in industry is XrML [Con01]. XrML and Lithium are incomparable in expressive power. XrML is less expressive in that it does not allow negation. This means in particular that it cannot express denying policies and cannot capture a policy that grants a permission based on whether a condition does not hold. In addition, the conclusion of every environment fact that is not a ground literal is of the form \( R(p) \), where \( R \) is a unary predicate symbol and \( p \) is a principal. On the other hand, XrML is more expressive than Lithium in that a policy can grant a permission based on the answers to various queries. For example, in XrML, Alice’s babysitter can write the policy “Alice is permitted to do some action \( a \) if the permission follows from her mother’s policies and from her father’s policies”. We can extend Lithium to include such policies as well. Let Lithium\(^+\) be Lithium extended with a \( \text{Val} \) operator, where \( \text{Val}(\varphi) \) is true if \( \varphi \) is valid. We can write the babysitter’s policy in Lithium\(^+\) as \( \forall x (\text{Val}(E_M \land P_M \rightarrow \text{Permitted}(\text{Alice}, x)) \land \text{Val}(E_D \land P_D \rightarrow \)
\textbf{Permitted}(Alice, x) \Rightarrow \textbf{Permitted}(Alice, x), \text{ where } E_M \land P_M \text{ and } E_D \land P_D \text{ are the policy bases of Alice’s mother and father respectively. We can place restrictions on Lithium$^+$ similar in spirit to those on Lithium to ensure that it is tractable, yet expressive enough to capture the policies that users want in practice; see [HW04] for details.}

The policy languages that are perhaps closest in spirit to Lithium are the approaches that are based on some variant of Datalog. Examples of such languages include Delegation Logic [LGF03], the RT (Role-based Trust-management) framework [LMW02], Binder [DeT02], SD3 [Jim01], FAF (Flexible Authorization Framework) [JSSS01], and Cassandra [BS04]). Datalog is an efficient well-understood reasoning engine that is restricted to function-free negation-free Horn clauses; these restrictions are made to ensure tractability. The variants, such as \textit{safe stratified Datalog} [GMUW02] or \textit{Datalog with constraints}, allow limited use of functions and negation while preserving tractability.

The main difference between Lithium and these Datalog-based languages is in the use of functions and negation. There are relatively few policy languages that include functions symbols, but those that do (e.g. [BBFS98, LM03, BS04]) seem to favor Datalog with constraints. By using this variant of Datalog, many structured resources, such as directories, can be expressed using functions. However, function symbols may not appear in intentional predicates (predicates whose relations are computed by applying Datalog rules, as opposed to being stored in a database). For example, the policy “every authorized individual may copy a classified file from one secure server to another” when written as

$$\forall x_1 \ldots \forall x_4 (\text{Auth}(x_1) \land \text{Classified}(x_2) \land \text{SecureSrvr}(x_3) \land \text{SecureSrvr}(x_4)$$

$$\Rightarrow \textbf{Permitted}(x_1, \text{copySrcDst}(x_2, x_3, x_4)))$$
is not in Datalog with constraints. Also, for tractability, additional restrictions are often made. For example, Li and Mitchell [LM03] do not allow formulas in constraints to have more than one variable and Becker and Sewell [BS04] require that every argument of a function in a query be variable-free.

There are a number of policy languages that allow a limited use of negation. Jajodia, Samarati, Sapino, and Subrahmanian [JSSS01] base their policy language on Datalog with negation, which is a variant of Datalog that allows unrestricted use of negation in the body of rules. Datalog with negation is tractable because it makes the closed-world assumption: if we cannot prove that a positive literal is true, we take it to be false. Unfortunately, the closed-world assumption can lead to unintuitive (and probably unintended) results. For example, consider the policy “if Alice does not have bad credit, then she may apply for a loan”, and suppose that the reasoning engine determines whether an individual has bad credit by reviewing her credit report. If Alice has bad credit and does not present her credit report, then a reasoning engine that makes the closed-world assumption will incorrectly assume that Alice does not have bad credit and thus will allow her to make a loan application.

Several policy languages (e.g. [DeT02, LGF03, LMW02, Jim01]) are based on safe stratified Datalog. Safe stratified Datalog allows some use of negation in the body of rules and does not make the closed world assumption. However, the restrictions on negation still prevent it from capturing some permitting policies of interest. For example, the policy

$$\forall x (\neg \text{BadCredit}(x) \Rightarrow \text{Permitted}(x, \text{apply for loan}))$$

(anyone without bad credit may apply for a loan) cannot be expressed. More importantly, denying policies cannot be expressed in safe stratified Datalog because
the language does not allow negation in the conclusion of rules.

This limitation may not seem to be particularly troublesome. After all, the standard approach is to assume that every permission not explicitly granted is denied. (For example, this is done in relational databases [GW76], almost all of the Datalog-based languages, UNIX, SPKI/SDSI [RL96, EFL+99a, EFL+99b], and KeyNote [BFL96].) However, in many contexts, it is difficult to believe that policymakers really want to forbid every action that they do not explicitly permit, so there is a mismatch between a policymaker’s intentions and the interpretation of the policy base. This becomes a problem when different policymakers want to compare policy bases or combine them. The following examples illustrate the concern.

Example 2.6.1. Suppose that a hospital wants to verify that its policies comply with federal regulations; that is, the hospital wants to check that, if the government permits an action, then the hospital permits it and, if the government forbids an action, then the hospital forbids it. If the policies are written in a language that captures only permissions, assuming all other actions are forbidden, then compliance checking is essentially impossible. In particular, if the hospital permits any action that is not regulated by the government (e.g., nurses may park in Lot A, all staff are welcome to drink the coffee in the lounge), then the hospital will appear to be non-compliant because it permits an action that is not explicitly permitted by the government and, thus, is implicitly forbidden. In short, because we cannot distinguish forbidden actions from unregulated ones, compliance checking reduces to determining whether one policy set is essentially identical to another. 

Example 2.6.2. Consider a group of libraries that want to merge their policies so that patrons are governed by the same regulations, regardless of which library they
visit. When merging the policy sets, we clearly want to detect conflicts (e.g. one library lets minors check out adult books and another does not). Unfortunately, if a language can state only what is permitted, then this will be impossible. If we put the permitting policies from each library into one large set, then that set will be consistent (it is satisfied in the model that permits everything), regardless of which policies are in the set. Alternatively, we could require that no library permits an action that another forbids (which is what we want to do) under the assumption that every unregulated action is forbidden. It is not hard to see that this approach will always detect a conflict between sets of library policies unless the sets are essentially identical.

The issues involved with comparing and merging policy bases have by and large been ignored, but we believe they will become increasingly significant. It seems unlikely that a policy language will be able to support these features unless the language can express both permitting and denying policies.

Although we do not know of a Datalog variant that allows negation in the conclusions of rules (thereby allowing denying policies), some languages seem to capture something comparable. For example, in FAF, actions are either positive or negative; the statement “principal \( p \) can do negative action \( \text{act} \)” means \( p \) is forbidden to do \( \text{act} \). Another option in the same spirit is to have the predicate symbol \textbf{Forbidden} in the language, in addition to \textbf{Permitted}. A consequence of this approach is that it is not logically inconsistent for an action to be both permitted and forbidden. (Note that this is also the case for XACML, due to its nonstandard interpretation of negation.) To handle inconsistencies, FAF expects the policy writer to create overriding policies such as “if an action is both permitted and forbidden, then it is forbidden”. If an inconsistency is detected when answering
a query, then the overriding policy is applied. Similar approaches are taken by Chomicki et al. [CLN00] and Ioannides and Selis [IS92]. The main problem with capturing prohibitions in this way is that the answers to queries might not match a policy writer’s expectations. Policy writers typically do not intend to write policies that both permit and forbid the same action. Rather than identifying such policies and alerting the policy writer, potential errors are patched with overriding policies. In addition, these overriding policies are required even for consistent policy bases, which seems rather burdensome.

Lithium deals with this issue in what is arguably a better way. Given a policy base written in Lithium, we can detect conflicts and determine why they occur at “compile time” rather than at “run time”, when a particular query is evaluated. Using this information, the policy writer can modify the environment and policies to more closely match her intentions. Moreover, if the antecedent of Theorem 2.4.1 holds, then the policy base is inconsistent if and only if the environment alone is inconsistent. Thus, we can often determine when there is no conflict that needs to be addressed.

The use of function symbols and negation is not the only difference between Lithium and other policy languages. Unlike Lithium, many languages have explicit support for *groups* and *roles*. A group is a set of subjects such that if a group has a property, then every member of the group has the property (cf. [ABLP93, JSSS01]). In role-based access control models [FBK99, HV01, LMW02, SCFY96] roles are an intermediary between individuals and rights. More specifically, an individual obtains a right by assuming a role that is associated with that right. For example, Alice may need to assume the role of Department Chair in order to obtain the budget.
Predicate symbols can be used to capture groups and roles in first-order logic. For example, if we want to say that Alice is a member of the faculty and any faculty member may chair committees, then we can represent the group using the predicate $\text{Faculty}$. The environment fact is encoded as $\text{Faculty}(\text{Alice})$; the policy is then

$$ \forall x ( \text{Faculty}(x) \Rightarrow \text{Permitted}(x, \text{chair committees})) $$

Similarly, the policy “Alice, acting as the Department Chair, may sign the budget” can be written as

$$ \text{Dept\_Chair}(\text{Alice}) \Rightarrow \text{Permitted}(\text{Alice}, \text{sign the budget}) $$

The fact $\text{Dept\_Chair}(\text{Alice})$ would be added to the environment when Alice assumes the role and would be removed when she relinquishes it. Alternatively, we could add a sort $\text{Roles}$ to our logic along with the predicate $\text{As}$ (as suggested by Lampson, Abadi, Burrows, and Wobber [LABW92]), where $\text{As}(e, r)$ means that entity $e$ is acting as role $r$ (in other words, $e$ has assumed role $r$). Continuing our example, “Alice, acting as the Department Chair, may sign the budget” could be written in the logic as

$$ \text{As}(\text{Alice}, \text{Dept\_Chair}) \Rightarrow \text{Permitted}(\text{Alice}, \text{sign the budget}) $$

The second encoding for roles may be more in keeping with the spirit of the role-based model, but we believe that both approaches are reasonable (and our results apply to both choices). In short, Lithium supports groups and roles implicitly.

Lithium, as well as the Datalog variants, all use a fragment of first-order logic to express policies. Other approaches use a modal logic. Formal work on deontic logic (the logic of “obligation” and “permission”) goes back to von Wright [Wri51].
Glasgow, MacEwen, Panangaden [GMP92] were the first to base a formal logic of
security on deontic logic. The logic of access control consider by Lampson et al. and
Abadi et al. [LABW92, ABLP93] can also be viewed as a modal logic, with a says
operator. These approaches can be translated into first-order logic, but they have
features that take them beyond Lithium. For example, Abadi et al. have a calculus
of principals; Glasgow, MacEwen, and Panangaden deal with obligation as well as
permission. We believe that many of these features could be added to Lithium,
but we have not explored this issue.

The KeyNote system [BFIK98], which is based on PolicyMaker [BFL96], is
more flexible than Lithium in that the application can invoke policies written in
a number of different languages. There are programs that determine if a policy
applies to a query. Because KeyNote essentially views these programs as black
boxes, it is quite limited in its ability to reason about policies. As discussed by
Blaze, Feigenbaum, and Strauss [BFS98], the system needs to put restrictions on
the programs to ensure correct analysis. This is in fact done in KeyNote, but at
the price of a substantial reduction in the expressive power of the language.

Finally, we remark that the design of Lithium was heavily influenced by the
work of Halpern, van der Meyden, and Schneider [HvdMS99]. They identify some
key issues that must be addressed when developing a policy language, evaluate
various solutions that have been proposed in the literature, and recommend direc-
tions for future research. Our design incorporates three of their suggestions. In
particular, we write policies in first-order logic; define sorts for principals, actions,
and time; and use a Permitted predicate that takes an individual and an action
argument. (This usage of Permitted is much in the spirit of how it is used in
modal deontic logic.)
2.7 Conclusion

We have presented a fragment of first-order logic called Lithium that seems well-suited to reasoning about policies. Unlike previous approaches, Lithium allows nearly unrestricted use of function symbols while still preserving tractability. Moreover, Lithium can express prohibitions explicitly, making it possible to detect certain errors, namely policies that are inconsistent or imply a fact that is missing from the environment, to determine if one policy set complies with another, and to capture the merger of policies. We are currently working with the Naval Research Laboratory to build a policy engine that is based on Lithium. The engine will answer queries about whether an action is permitted and will check policy compliance.

We would like to extend the work in at least four ways. First, we suspect that we can increase the expressive power of Lithium by further relaxing the bipolar restriction. The general idea is to answer queries without computing the entire closure of the policies and environment under resolution (i.e. \( R(E \land P) \)). Second, we would like to extend Lithium to capture obligations, as well as permissions. Third, we want to find ways to answer queries about policies written in Lithium that change over time. For example, we would like to be able to determine which permissions change in response to a particular change in a policy set, and we would like to be able to detect if, by modifying the policies in a specific way during runtime, a sequence of actions is permitted that is not permitted under the original or revised policies. Finally, we would like to conduct a formal usability study to test our belief that a large fragment of Lithium can be made accessible to non-logicians by an appropriate use of syntactic sugar.
Chapter 3

XrML

XrML, like Lithium, is a language for reasoning about policies. XrML differs from Lithium in four essential ways. The first is that negation is not mentioned in the antecedent or in the conclusion of XrML policies. The second is that XrML does not make the same distinction between environment and policies that is maintained in Lithium. In particular, an XrML policy can say that a principal has a property (e.g., is a student) if certain conditions hold and an XrML environment gives only basic facts; that is, it can be represented as a conjunction of positive Permitted-free ground literals. The third significant difference is that an XrML policy can say that, if certain conditions hold, then a principal $p$ is permitted to issue a policy $pol$. If $p$ exercises this right, then $pol$ is added to the set of policies used during query evaluation. Finally, an XrML policy can grant a permission based on who has issued certain policies. For example, in XrML, we can write the policy “if Cornell University says that Alice is a student, then she is permitted to register for classes”, where Cornell University says that Alice is a student if the policies issued by Cornell University imply this fact. In addition to these fundamental differences, XrML and Lithium use different terminology. In particular, a policy in XrML is called a grant, and a grant $g$ together with a principal who issued $g$ is called a license.

The XrML specification introduces the components of the language and gives an algorithm for determining whether a particular permission follows from a set of grants and licenses in a given environment. Both the language components and the algorithm are described in English and, as a result, they are open to interpre-
In such circumstances, the hope is that any ambiguity in the description of the components can be resolved by looking at the algorithm and vice-versa. Unfortunately, some aspects of the language are clearly inconsistent and even the algorithm alone has certain unintuitive consequences. The goal of this chapter is to address these issues.

The rest of the chapter is organized as follows. In the next section we present a representative fragment of XrML. In Section 3.2 we review XrML’s algorithm for answering queries. After considering some examples in which the algorithm’s behavior is unintuitive and almost certainly unintended, we propose corrections that we believe captures the designers’ intent. Formal semantics for XrML are given in Section 3.3, and the revised algorithm is shown to be sound and complete with respect to the semantics. In Section 3.4 we show that the problem of determining if a permission follows from a set of licenses is undecidable. We also discuss a fragment of the language that is both tractable and relatively expressive. In Section 3.5 we outline how our results can be modified to apply to the entire language, including extensions that are within the XrML framework. MPEG-21 REL, which is an international standard based on XrML, is described in Section 3.7. We conclude in Section 3.8. All of the proofs are in the appendix.

3.1 Syntax

XrML is an XML-based language; it follows XML-conventions. Rather than present that syntax, we use an alternative syntax that is more concise and, we believe, more intuitive. In this section, we introduce our syntax for a representative fragment of XrML (the rest of the language is discussed in Section 3.5) and describe the key differences between the syntax used in the XrML specification
At the heart of XrML is the notion of a license. A license is a (principal, grant) pair, where the license \((p, g)\) means \(p\) issues (i.e., says) \(g\). For example, the license \((\text{Alice}, \text{Bob is smart})\) means “Alice says ‘Bob is smart’.”

A grant has the form \(\forall x_1 \ldots \forall x_n (\text{condition} \rightarrow \text{conclusion})\), which intuitively means that the condition implies the conclusion under all appropriate substitutions. Conditions and conclusions are defined as follows.

- A condition has the form \(d_1 \land \ldots \land d_n\), where each \(d_i\) is either \(\text{true}\) or \(\text{Said}(p, e)\) for some principal \(p\) and conclusion \(e\). Roughly speaking, the condition \(\text{true}\) always holds and the condition \(\text{Said}(p, e)\) holds if \(p\) issues a grant that says \(e\) holds if a condition \(d\) holds, and \(d\) does, in fact, hold.

- A conclusion has either the form \(\text{Permitted}(p, r, s)\) or the form \(\text{Pr}(p)\), where \(\text{Pr}\) is a property, \(p\) is a principal, \(r\) is a right (i.e., an action), and \(s\) is a resource. The conclusion \(\text{Permitted}(p, r, s)\) means \(p\) may exercise \(r\) over \(s\). For example, \(\text{Permitted}(\text{Bob}, \text{edit}, \text{budget report})\) means Bob may edit the budget report. The conclusion \(\text{Pr}(p)\) means \(p\) has the property \(\text{Pr}\). For example, the conclusion \(\text{Attractive}(\text{Bob})\) means Bob is attractive.

We abbreviate the grant \(\forall x_1 \ldots \forall x_n (\text{true} \rightarrow e)\) as \(\forall x_1 \ldots \forall x_n e\). Also, we try to consistently use \(d\), possibly subscripted, to denote a generic condition and \(e\), possibly subscripted, to denote a generic conclusion.

Consider the following example. Suppose that Alice issues the grant “Bob is smart” and Amy issues the grant “if Alice says that Bob is smart, then he is attractive”. We can write the first license in our syntax as \((\text{Alice}, g_1)\), where \(g_1 = \text{Smart}(\text{Bob})\) (recall that this is an abbreviation for \(\text{true} \rightarrow \text{Smart}(\text{Bob})\)),
and we can write the second as \((Amy, g_2)\), where \(g_2 = \text{Said}(Alice, \text{Smart}(Bob)) \rightarrow \text{Attractive}(Bob)\). Because \((Alice, g_1)\) is in the set of issued licenses, the condition \(\text{Said}(Alice, \text{Smart}(Bob))\) holds. It follows from this fact and the license \((Amy, g_2)\) that \(\text{Said}(Amy, \text{Attractive}(Bob))\) holds as well.

The sets of principals, properties, rights, and resources depend on the particular application. For example, a multimedia application might have a principal for each employee and each customer; properties such as “hearing impaired” and “manager”; rights such as “edit” and “download”; and a resource for each object such as a movie. We assume the application gives us a finite set \(\text{primitivePrin}\) of principals and a finite set \(\text{primitiveProp}\) of properties. We then define the components in our language as follows.

- The set \(P\) of principals is the result of closing \(\text{primitivePrin}\) under union. (Here and elsewhere we identify a principal \(p \in \text{primitivePrin}\) with the singleton \(\{p\}\) and write \(\{p_1, \ldots, p_n\}\) rather than \(\{p_1\} \cup \ldots \cup \{p_n\}\).) The interpretation of a principal \(\{p_1, \ldots, p_n\}\) depends on context; that is, the interpretation depends on whether the principal appears as the first argument in a \(\text{Said}\) condition, in a conclusion, or in a license. We discuss this later in the paper (primarily in Section 3.4).

- The set of properties is \(\text{primitiveProp}\). We assume that every property in \(\text{primitiveProp}\) takes a single argument and that argument is of sort \(Princ\). For example, \(\text{primitiveProp}\) can include the property \(\text{Employee}\), where \(\text{Employee}(x)\) means principal \(x\) is an employee, but it cannot include the property \(\text{MotherOf}\), where \(\text{MotherOf}(x, y)\) means principal \(x\) is the mother of principal \(y\), nor can it include the property \(\text{Vehicle}\), where \(\text{Vehicle}(x)\) means resource \(x\) is a vehicle (e.g., a motorcycle, car, or truck). The results
in this paper continue to hold if we extend the language to include properties that take multiple arguments of various sorts (i.e., principals, rights, and resources). It is also easy to show that closing $\text{primitiveProp}$ under conjunction adds no expressive power to the language. Closing under negation does add expressive power; we return to this issue in Section 3.6.

- The only right in our language is $\text{issue}$ and the only resources are grants. Intuitively, if a principal $p$ has the right to issue a grant $g$, and $p$ does issue $g$, then $g$ is a true statement. Including additional rights and resources in our language does not significantly affect the discussion.

We formally define the syntax according to the following grammar.

\[
\begin{align*}
\text{license} & ::= (\text{princ, grant}) \\
\text{grant} & ::= \forall \text{var} \ldots \forall \text{var}(\text{cond} \rightarrow \text{conc}) \\
\text{var} & ::= x_p | x_r \\
\text{cond} & ::= \text{true} | \text{Said}(\text{princ}, \text{conc}) | \text{cond} \land \text{cond} \\
\text{conc} & ::= \text{Pr}(\text{princ}) | \text{Permitted}(\text{princ}, \text{right}, rsrc) \\
\text{princ} & ::= \{p\} | \{x_p\} | \text{princ} \cup \text{princ} \\
\text{right} & ::= \text{issue} \\
\text{rsrc} & ::= \text{grant} | x_r,
\end{align*}
\]

where $\text{Pr}$ is an element of $\text{primitiveProp}$, $p$ is an element of $\text{primitivePrin}$, $x_p$ is an element of $\text{prinVar}$, which is the set of variables ranging over primitive principles, and $x_r$ is an element of $\text{rsrcVar}$, which is the set of variables ranging over resources. For the remainder of this chapter we assume that the first argument in a license is a singleton. Because the XrML document treats the license $(\{p_1, \ldots, p_n\}, g)$ as an abbreviation for the set of licenses $\{(p, g) \mid p \in \{p_1, \ldots, p_n\}\}$, it is easy to modify our discussion to support all of the licenses included in the grammar.
As mentioned at the beginning of this section, the grammar presented here is not identical to that described in the XrML document. The main differences are listed below.

- Instead of assuming that the application provides a set of primitive principals, XrML assumes that the application provides a set $K$ of cryptographic keys; the set of primitive principals is $\{\text{KeyHolder}(k) \mid k \in K\}$. We could take $\text{primitivePrin}$ to be this set; however, our more general approach leads to a simpler discussion. Moreover, our results do not change if we restrict primitive principals to those of the form $\text{KeyHolder}(k)$.

- XrML does not have conclusions of the form $\text{Pr}(p)$. To capture properties, XrML uses a right called $\text{PossessProperty}$ and considers the properties given by the application to be resources. The conclusion $\text{Pr}(p)$ in our grammar corresponds to the conclusion $\text{Permitted}(p, \text{PossessProperty}, \text{Pr})$ in XrML. We have two types of conclusions because we believe the grammar should help distinguish the conceptually different notions of permissions and properties, rather than confounding them.

- Instead of writing $\text{AllPrincipals}(p_1, \ldots, p_n)$, $\text{allConds}(c_1, \ldots, c_n)$, and $\text{allConds}()$, we use the more standard notations $\{p_1, \ldots, p_n\}$, $c_1 \land \ldots \land c_n$, and true, respectively. Instead of writing $\text{PrerequisiteRight}(p, e)$, we use the shorter and, we believe, more appropriate notation $\text{Said}(p, e)$.

- As discussed previously, XrML abbreviates a set of licenses $\{(p_i, g_j) \mid i \leq n, j \leq m\}$ as the single license $\{(p_1, \ldots, p_n), \{g_1, \ldots, g_m\}\}$. For ease of exposition, we do not do this.
3.2 XrML’s Authorization Algorithm

The XrML document includes a procedure that we call \textbf{Query} to determine if a conclusion follows from a set of licenses (and some additional input that is discussed below). In this section we present and analyze the parts of the algorithm that pertain to our fragment.

Before describing the algorithm, we note that some aspects of \textbf{Query} are inefficient. This is acknowledged in the XrML document, which explains that \textbf{Query} was designed with clarity as the primary goal; it is the responsibility of the language implementors to create efficient algorithms with the same input/output behavior as \textbf{Query}. (In Section 3.4, we show that it is highly unlikely that such an efficient algorithm exists.)

3.2.1 A Description of Query

The input to \textbf{Query} is a closed conclusion \( e \) (i.e., a conclusion with no free variables), a set \( L \) of licenses \((p, g)\) such that \( p \) is variable-free, and a set \( R \) of grants; \textbf{Query} returns \textit{true} if \( e \) is implied by \( L \) and \( R \), and returns \textit{false} otherwise. To explain the intuition behind \( L \) and \( R \), we first note that the procedure treats a predefined set of principals as trusted. If a trusted principal issues the grant \( g \), then \( g \) is in \( R \) and it is assumed to be true. If the license \((p, g)\) is in \( L \), then \( p \) issued \( g \) (i.e., \( p \) says \( g \)) and \( p \) is not an implicitly trusted principal. To clarify the inferences that are drawn from \( R \) and \( L \), suppose that the grant \( g \) is \textbf{QueenOfSiam}(Alice), which means Alice is Queen of Siam, and the grant \( g' \) is \textbf{Permitted}(Alice, issue, g), which means Alice may issue \( g \). If \( g \in R \), then we assume that Alice really is queen. If \((Alice, g)\) is in \( L \), then Alice says that she is

...
the queen, but we cannot conclude that she is royalty from this statement alone. If \((Alice, g)\) is in \(L\) and \(g'\) is in \(R\), then we assume that Alice has the authority to declare herself queen, because \(g' \in R\); we assume that she exercises that authority, because \((Alice, g) \in L\); and we conclude that Alice is queen, because this follows from the two assumptions.

**Query** begins by calling the **Auth** algorithm. **Auth** takes \(e, L, \text{ and } R\) as input; it returns a set \(D\) of closed conditions (i.e., conditions with no free variables). Roughly speaking, a closed condition \(d\) is in \(D\) if \(d, L, \text{ and } R\) together imply \(e\). To determine if a condition in \(D\) holds, **Query** relies on the **Holds** algorithm. The input to **Holds** is a closed condition \(d\) and a set \(L\) of licenses; \(\text{Holds}(d, L)\) returns true if the licenses in \(L\) imply \(d\), and returns false otherwise. If \(\text{Holds}(d, L)\) returns true for some \(d\) in \(D\), then **Query** returns true, indicating that \(L\) implies \(e\). **Query** is summarized in Figure 3.1.

We now discuss **Auth** and **Holds** in some detail. To define **Auth**, we first consider the case where \(L = \emptyset\). Define a *closed substitution* to be a mapping from variables to closed expressions of the appropriate sort. Given a closed substitution \(\sigma\) and an expression \(t\), let \(t\sigma\) be the expression that arises after all free variables
\(x\) in \(t\) are replaced by \(\sigma(x)\). Roughly speaking, \(\text{Auth}(e, \emptyset, R)\) returns the set \(D\) of closed conditions such that each condition in \(D\), in conjunction with the grants in \(R\), implies \(e\). That is, \(d \in D\) iff there is a grant \(g = \forall x_1 \ldots \forall x_n(d_g \rightarrow e_g)\) in \(R\) and a closed substitution \(\sigma\) such that \(d = d_g \sigma\) and \(e_g\) implies \(e\). \(\text{Auth}\) determines whether \(e_g\) implies \(e\) in a somewhat nonstandard way. In particular, it makes the subset assumption, which says that any property or permission attributed to a principal \(p\) is attributed to every principal that includes \(p\). In other words, if \(p \subseteq p'\), then \(\Pr(p)\) implies \(\Pr(p')\) and \(\text{Permitted}(p, r, s)\) implies \(\text{Permitted}(p', r, s)\). Thus,

\[
\text{Auth}(\Pr(p), \emptyset, R) = \{d | \text{for some } g = \forall x_1 \ldots \forall x_n(d_g \rightarrow \Pr(p_g)) \in R \text{ and closed substitution } \sigma, d_g \sigma = d \text{ and } p_g \sigma \subseteq p\} \text{ and }
\]

\[
\text{Auth}(\text{Permitted}(p, r, s), \emptyset, R) = \{d | \text{for some } g = \forall x_1 \ldots \forall x_n(d_g \rightarrow \text{Permitted}(p_g, r_g, s_g)) \in R \text{ and closed substitution } \sigma, d_g \sigma = d, p_g \sigma \subseteq p, r_g \sigma = r, \text{ and } s_g \sigma = s\}.
\]

Suppose that \(L \neq \emptyset\). Then we reduce to the previous case by taking \(\text{Auth}(e, L, R) = \text{Auth}(e, \emptyset, R')\), where, intuitively, \(R'\) is the set of legitimate grants; that is, \(R'\) consists of the grants in \(R\) and the grants issued by someone who has the authority to do so. It seems reasonable to call \(\text{Query}(\text{Permitted}(p, \text{issue}, g), L, R)\) to determine if a principal \(p\) has the authority to issue a grant \(g\). However, if \(\text{Auth}\) calls \(\text{Query}(\text{Permitted}(p, \text{issue}, g), L, R)\) to construct \(R'\), then the algorithm will not terminate, because \(\text{Query}\) calls \(\text{Auth}\), leading to an infinite call tree. So, instead of calling \(\text{Query}(\text{Permitted}(p, \text{issue}, g), L, R)\), the XrML algorithm determines if \(p\) is permitted to issue \(g\) by checking if \(\text{Holds}(d, L) = \text{true} \text{ for some } d\) in the set \(\text{Auth}(\text{Permitted}(p, \text{issue}, g), L - \{(p, g)\}, R)\). We discuss the consequences.
of this solution in Section 3.2.2. In summary,

\[ R' = R \cup R'' , \]
\[ R'' = \{ g \mid \text{for some } (p, g) \in L \text{ and condition } d, \]
\[ d \in \text{Auth}(\text{Permitted}(p, \text{issue}, g), L - \{(p, g)\}, R) \text{ and } \]
\[ \text{Holds}(d, L) = \text{true} \} \]

Pseudocode for \text{Auth} is given in Figure 3.2.

We define \text{Holds}(d, L) by induction on the structure of \( d \). If \( d \) is \text{true}, then \( \text{Holds}(d, L) = \text{true} \). If \( d = \text{Said}(p, e) \), then \( \text{Holds}(d, L) = \text{true} \) iff \( p \) issues a grant \( \forall x_1 \ldots \forall x_n(d_g \rightarrow e_g) \) such that, for some substitution \( \sigma \), \( e_g \sigma = e \) and \( \text{Holds}(d_g \sigma, L) = \text{true} \). In this context, a principal \( \{p_1, \ldots, p_n\} \) issues a grant \( g \) if \( p_i \) issues \( g \) for some \( i = 1, \ldots, n \). If \( d = d_1 \wedge \ldots \wedge d_n \), where each \( d_i \) is \text{true} or a \text{Said} condition, then \( \text{Holds}(d, L) = \bigwedge_{i=1,\ldots,n} \text{Holds}(d_i, L) \). Pseudocode for \text{Holds} is given in Figure 3.3.

**Example 3.2.1.** In Section 3.1, we argued informally that Amy says Bob is attractive if the set of licenses is \( L = \{(Alice, g_1), (Amy, g_2)\} \), where \( g_1 = \text{Smart}(Bob) \) and \( g_2 = \text{Said}(Alice, \text{Smart}(Bob)) \rightarrow \text{Attractive}(Bob) \). The formal algorithm gives the same conclusion. Specifically, \( \text{Holds}(\text{Said}(Amy, \text{Attractive}(Bob)), L) \) sets \( R_{Amy} = \{g_2\} \) and calls \( \text{Holds}(\text{Said}(Alice, \text{Smart}(Bob)), L) \). During this call \( R_{Alice} \) is set to \( \{g_1\} \) and \( \text{Holds}(\text{true}, L) \) is called. Because \( \text{Holds}(\text{true}, L) = \text{true} \), \( \text{Holds}(\text{Said}(Alice, \text{Smart}(Bob)), L) = \text{true} \) and, thus, \( \text{Holds}(\text{Said}(Amy, \text{Attractive}(Bob)), L) = \text{true} \).

Suppose that a trusted principal says that Amy has the authority to issue \( g_2 \) (i.e., if Amy says \( g_2 \), then \( g_2 \) holds). Then we can conclude that Bob really is attractive, because \( \text{Query}(\text{Attractive}(Bob), L, R) = \text{true} \), where \( R = \{\text{Permitted}(Amy, \text{issue}, g_2)\} \). Specifically, \( \text{Query} \) begins by calling
\textbf{Auth}(e, L, R):

\[ D := \emptyset \]

\textbf{if} \( L = \emptyset \) \textbf{then}

\% Find \( D \), the conditions under which \( R \) implies \( e \)

\textbf{if} \( e = \Pr(p) \)

\textbf{for} each grant \( \forall x_1 \ldots \forall x_n(d_g \rightarrow \Pr(p_g)) \in R \)

\[ D := D \cup \{d | d_g\sigma = d \text{ and } p_g\sigma \subseteq p, \text{ for some closed substitution } \sigma\} \]

\textbf{if} \( e = \text{Permitted}(p, r, s) \)

\textbf{for} each grant \( \forall x_1 \ldots \forall x_n(d_g \rightarrow \text{Permitted}(p_g, r_g, s_g)) \in R \)

\[ D := D \cup \{d | d_g\sigma = d, p_g\sigma \subseteq p, r_g\sigma = r, \text{ and } s_g\sigma = s, \text{ for some closed substitution } \sigma\} \]

\textbf{else}

\% Find \( R' \)

\[ R' := R \]

\textbf{for} each license \( (p, g) \in L \)

\[ L' := L - \{(p, g)\} \]

\[ D' := \text{Auth}(\text{Permitted}(p, \text{issue}, g), L', R) \]

\textbf{if} \( \text{Holds}(d, L) = \text{true} \) \textbf{for} a condition \( d \in D' \)

\textbf{then} \( R' := R' \cup \{g\} \)

\% Find \( D \), the conditions under which \( R' \) implies \( e \)

\[ D := \text{Auth}(e, \emptyset, R') \]

\textbf{return} \( D \)

Figure 3.2: The \textbf{Auth} Algorithm
Holds\((d, L)\):

if \(d = true\)
then return \(true\)

if \(d = Said(p, e)\)
then

\(R_p = \{g | \text{for some principal } p', (p', g) \in L \text{ and } p' \in p\}\)
\(D := \{d' | \text{for some grant } \forall x_1 \ldots \forall x_n(d_g \rightarrow e_g) \in R_p \text{ and } d_g \sigma = d' \text{ and } e_g \sigma = e\}\)

if \(Holds(d', L) = true\) for a condition \(d' \in D\)
then return \(true\)
else return \(false\)

if \(d = d_1 \land \ldots \land d_n\), where each \(d_i\) is \(true\) or a \(Said\) condition
then return \(\land_{i=1,...,n} Holds(d_i, L)\)

Figure 3.3: The \textbf{Holds} Algorithm
Auth(Attractive(Bob), L, R) in turn, calls Auth(Attractive(Bob), ∅, R'), where R' = \{g_2, \text{Permitted}(Amy, \text{issue}, g_2)\}. Auth(Attractive(Bob), ∅, R') = \{\text{Said}(Alice, \text{Smart}(Bob))\}. So, Bob is attractive if the condition Said(Alice, Smart(Bob)) holds. To determine if the condition holds, Query calls Holds(Said(Alice, Smart(Bob)), L). We have already shown that Holds(Said(Alice, Smart(Bob)), L) = true; we evaluated this call during our analysis of Holds(Said(Amy, Attractive(Bob)), L). So Bob is indeed attractive.

Query as described here and in the XrML specification is somewhat ambiguous. For example, the specification does not say in which order the conditions in D should be tested to see if at least one condition in D holds. As a result, there are a number of possible executions of a call Query(e, L, R), depending on the implementation of Query. It is easy to see that, for a particular input, every execution that terminates returns the same output. However, as we show in Example 3.2.4, whether Query terminates can depend on how it is implemented. A similar issue arises with Auth and Holds. We talk about an execution of Query, Auth, or Holds only if the choice of execution affects whether the algorithm terminates. For example, we write Query(e, L, R) = true if every execution of Query(e, L, R) returns true.

3.2.2 An Analysis of Query

In this section we present five examples in which Query gives unexpected results. Example 3.2.2 reveals a mismatch between Query and the informal language description; the discrepancy exists because Auth makes the subset assumption and the informal language description does not. Example 3.2.3 demonstrates that a
license \((p, g)\) should not be removed from the set of licenses when determining if \(p\) is permitted to issue \(g\). Examples 3.2.4, 3.2.5, and 3.2.6, show that a reasonable implementation of \textbf{Query} does not terminate on all inputs, for three quite different reasons: Example 3.2.4 shows that on some inputs \textbf{Holds} makes infinitely many identical calls, Example 3.2.5 shows that on some inputs the call tree for \textbf{Query} includes an infinite path of distinct nodes; and Example 3.2.6 shows that on some inputs the call tree for \textbf{Query} includes a node with infinitely many distinct children.

\textbf{Example 3.2.2.} Suppose that Alice is quietly walking beside her two giggling daughters, Betty and Bonnie. Are the three of them a quiet group? Intuitively, they are not, because Betty and Bonnie are giggling. According to \textbf{Query}, however, the answer is yes. Since Alice is quiet and \textbf{Auth} makes the subset assumption, \textbf{Query} concludes that the principal \(\{Alice, Betty, Bonnie\}\) is quiet; that is, 

\[\text{Query}(\text{Quiet}(\{Alice, Betty, Bonnie\}), \emptyset, \{\text{Quiet}(Alice)\}) = \text{true}.\]

\textbf{Example 3.2.3.} Suppose that Alice says that she is smart, and if Alice says that she is smart, then she is permitted to say that she is smart. Is Alice smart? Intuitively, she is, because Alice is permitted to say that she is smart and she does so. But consider \textbf{Query}(\text{Smart}(Alice), L, R), where \(L = \{(Alice, g)\}\), \(R = \{\text{Said}(Alice, \text{Smart}(Alice)) \rightarrow \text{Permitted}(Alice, \text{issue}, g)\}\), and \(g = \text{Smart}(Alice)\). \textbf{Query}(\text{Smart}(Alice), L, R) begins by calling \textbf{Auth}(\text{Smart}(Alice), L, R). \textbf{Auth} checks whether or not Alice is permitted to issue \(g\). It determines that Alice may not issue \(g\), because the permission does not follow from \(R\) and \(L - \{(Alice, g)\}\). Since Alice is not permitted to issue \(g\), \textbf{Auth} sets \(R' = R\) and returns \(\emptyset\). Because \textbf{Auth} returns \(\emptyset\), \textbf{Query} returns \textit{false}. \textit{\[\square\]}
Example 3.2.4. Suppose that Alice issues the grant “if I say Bob is smart, then he is” and Alice is permitted to issue this grant. Can we conclude that Bob is smart?

To answer the question using Query, let $e = \text{Smart}(Bob)$, $g = \text{Said}(Alice, e) \Rightarrow e$, $L = \{(Alice, g)\}$, and $R = \{\text{Permitted}(Alice, \text{issue}, g)\}$. We are interested in the output of Query($e, L, R$). Query($e, L, R$) begins by calling Auth($e, L, R$), which returns the set $D = \{\text{Said}(Alice, e)\}$. Query then calls Holds(Said($Alice, e), L$), which sets $R_Alice = \{g\}$ and calls Holds(Said($Alice, e), L$) again. It is easy to see that an infinite number of calls to Holds(Said($Alice, e), L$) are made during the execution of Query($e, L, R$) and thus the execution does not terminate.

It is tempting to conclude that a set $L$ of licenses and a set $R$ of grants imply a conclusion $e$ only if Query($e, L, R$) terminates and returns true. Unfortunately, whether Query($e, L, R$) terminates can depend on the order in which the calls to Holds are made. To see why, consider a slight modification of the previous example where we add the grant $\{\text{Smart}(Bob)\}$ to $R$. Intuitively, this means that an implicitly trusted principal says that Bob is smart. It now seems reasonable to expect that every execution of Query($e, L, R'$) returns true, where $R' = R \cup \{e\}$, and $e$, $L$, and $R$ are as defined in the original example. Surely the issued grants imply that Bob is smart, since a grant issued by a trusted principal says just that! However, only some of the executions terminate. Every execution of Query begins by calling Auth($e, L, R'$), and every execution of Auth($e, L, R'$) returns $\{\text{Said}(Alice, e), true\}$. If an execution of Query next calls Holds(true, L), then that execution of Query returns true. On the other hand, if the execution calls Holds(Said($Alice, e), L$) and then waits for the call to return before calling Holds(true, L), then the execution does not terminate for the same reason that every execution of Query($e, L, R$) does not terminate. □
Example 3.2.5. Suppose that Alice says “for all grants $g$, if I say I am allowed to issue the grant $\text{Permitted}(\text{Alice}, \text{issue}, g)$, then I am allowed to issue $g$”, and Alice is allowed to issue that statement. Is Alice allowed to issue the grant $\text{Nap}(\text{Alice})$? To answer this question using $\text{Query}$, some abbreviations are useful. For all grants $g$, we abbreviate the condition $\text{Said}(\text{Alice}, \text{Permitted}(\text{Alice}, \text{issue}, \text{Permitted}(\text{Alice}, \text{issue}, g)))$ as $d(g)$ and we abbreviate the grant $\text{Permitted}(\text{Alice}, \text{issue}, g)$ as $h(g)$. We execute $\text{Query}(e, L, R)$, where $e = \text{Permitted}(\text{Alice}, \text{issue}, \text{Nap}(\text{Alice}))$, $R = \{ \text{Permitted}(\text{Alice}, \text{issue}, \forall x(d(x) \Rightarrow \text{Permitted}(\text{Alice}, \text{issue}, x))) \}$, and $L = \{ (\text{Alice}, \forall x(d(x) \Rightarrow \text{Permitted}(\text{Alice}, \text{issue}, x))) \}$. $\text{Query}(e, L, R)$ begins by calling $\text{Auth}(e, L, R)$, which returns $D = \{ d(g) \mid g \text{ is a grant} \}$. We show below that $D$ is an infinite set, so every execution of $\text{Auth}$ that tries to compute $D$ does not terminate. Even if $D$ is defined without explicitly listing all of its elements, $\text{Query}$ must determine if some element in $D$ holds. In fact, none do. Thus, any approach to testing if some

Example 3.2.6. Suppose that Alice may say that she is trusted if Bob says that Alice may issue some grant (any grant at all). May Alice say that she is trusted? To answer this question using $\text{Query}$, we run $\text{Query}(e, \emptyset, R)$, where $e = \text{Permitted}(\text{Alice}, \text{issue}, \text{Trusted}(\text{Alice}))$, $R = \{ \forall x(d(x) \rightarrow e) \}$, and $d(x) = \text{Said}(\text{Bob}, \text{Permitted}(\text{Alice}, \text{issue}, x))$. $\text{Query}$ begins by calling $\text{Auth}(e, \emptyset, R)$, which returns $D = \{ d(g) \mid g \text{ is a grant} \}$. We show below that $D$ is an infinite set, so every execution of $\text{Auth}$ that tries to compute $D$ does not terminate. Even if $D$ is defined without explicitly listing all of its elements, $\text{Query}$ must determine if some element in $D$ holds. In fact, none do. Thus, any approach to testing if some
condition in $D$ holds by explicitly testing each condition will not terminate.

It remains to show that $D = \{d(g) \mid g \text{ is a grant}\}$ is an infinite set. The key observation is that infinitely many distinct grants can be expressed in the language, even if the vocabulary consists of only one property $Pr$ and one principal $p$. To see why, define grants $g_n$, $n \geq 1$, inductively by taking $g_1 = true \rightarrow Pr(p)$ and $g_{n+1} = Said(p, Permitted(p, issue, g_n)) \rightarrow Pr(p)$ for all $n > 0$. Since each of these grants is clearly distinct, $D$ is infinite. 

3.2.3 A Corrected Version of Query

In this section we revise Query to correct the problems observed in Section 3.2.2. One of the corrections is fairly straightforward. We resolve the mismatch illustrated in Example 3.2.2 by removing the subset assumption from Auth. We note that the language is sufficiently expressive to force the subset assumption, if desired, by including the following grants in $R$:

$$g = \forall x_1 \forall x_2 \forall x_3(Permitted(x_1, issue, x_2) \rightarrow Permitted(x_1 \cup x_3, issue, x_2))$$

$$g_i = \forall x_1 \forall x_2(Pr_i(x_1) \rightarrow Pr_i(x_1 \cup x_2))$$, for $i = 1, \ldots, n$,

where $x_1$, $x_2$, and $x_3$ are variables of the appropriate sorts and $Pr_1, \ldots, Pr_n$ are the properties in the language. We now consider Examples 3.2.3, 3.2.4, 3.2.5, and 3.2.6, in turn.

The problem illustrated in Example 3.2.3 lies in the definition of $R'$. Recall that we define $Auth(e, L, R) = Auth(e, \emptyset, R')$. Roughly speaking, $R'$ should consist of the set of grants in $R$ together with those issued by someone who has the authority to do so. In other words, $R'$ should be $R \cup \{g \mid \text{ for some principal } p, (p, g) \in L \text{ and } Query(Permitted(p, issue, g), L, R) = true\}$. However, when computing $Query(Permitted(p, issue, g), L, R)$, $Auth$ is given the argument $L - \{(p, g)\}$.
rather than $L$. Our solution is to do the “right” thing here, that is, we compute $\text{Query}(\text{Permitted}(p, i, s, g), L, R)$. But now we have to deal with the problem of termination, since a consequence of our change is that $\text{Query}(e, L, R)$ terminates only if the set $L = \emptyset$. To ensure termination, we modify $\text{Auth}$ so that no call is evaluated twice. Specifically, the revised $\text{Auth}$ takes a fourth argument $E$ that is the set of closed conditions that have been the first argument to a previous call; $\text{Auth}(e, L, R, E)$ returns $\emptyset$ if $e \in E$. Because the revised $\text{Auth}$ calls $\text{Query}$, which calls $\text{Auth}$, we modify $\text{Query}$ to take $E$ as its fourth argument. A closed condition $e$ is implied by a set $L$ of licenses and a set $R$ of grants if the modified $\text{Query}$ algorithm returns $true$ on input $(e, L, R, \emptyset)$. Pseudocode for the revised version of $\text{Query}$, which we call $\text{Query2}$, and for the revised version of $\text{Auth}$, which we call $\text{Auth2}$, are given in Figures 3.4 and 3.5, respectively. $\text{Query2}$ refers to the algorithm $\text{Holds2}$, which is $\text{Holds}$ modified to correct the behavior seen in Example 3.2.4 (discussed below).

The type of nontermination seen in Example 3.2.4 occurs because $\text{Query}$ tries to verify that a condition of the form $\text{Said}(p, e)$ holds by checking if $\text{Said}(p, e)$ holds. To correct the problem, we modify $\text{Holds}$ to take a third argument $S$ that
**Auth2**\( (e, L, R, E) \):

\[
\begin{align*}
\text{if } e & \in E \\
\text{then } & \text{return } \emptyset \\
\text{else} \\
E' & := E \cup \{e\} \\
R' & := R \\
\text{for each license } (p, g) \in L \\
\quad \text{if } \text{Query2}(\text{Permitted}(p, \text{issue}, g), L, R, E') = \text{true} \\
\quad \text{then } R' := R' \cup \{g\} \\
D & := \emptyset \\
\text{for each grant } \forall x_1 \ldots \forall x_n(d_g \rightarrow e_g) \in R' \\
\quad D := D \cup \{d \mid d_g\sigma = d \text{ and } e_g\sigma = e, \text{ for some closed substitution } \sigma\}
\end{align*}
\]

return \( D \)

Figure 3.5: The **Auth2** Algorithm

is the set of Said conditions that have been the first argument to a previous call; that is, \( S \) is the set of Said conditions that are currently being evaluated. If the revised Holds is called with a first argument \( d \) that is in \( S \) (which means that the call was made when trying to determine whether \( d \) holds), then the algorithm returns false, thereby halting the cycle. Pseudocode for the revised version of Holds, which we call Holds2, is given in Figure 3.6.

It is easy to see that the problem illustrated by Example 3.2.4 does not occur during the execution of Holds2. Moreover, the following theorem shows that Holds2 is correct in the sense that every execution of Holds and Holds2 have
\textbf{Holds2}(d, L, S):

if $d = \text{true}$
then return \text{true}

if $d = d_1 \land \ldots \land d_n$
then return $\bigwedge_{i=1,\ldots,n} \text{Holds2}(d_i, L, S)$

if $d = \text{Said}(p, e)$ and $d \in S$
then return \text{false}

if $d = \text{Said}(p, e)$ and $d \notin S$
then

$S' = S \cup \{d\}$

$R_p = \{g \mid \text{for some principal } p', (p', g) \in L \text{ and } p' \in p\}$

$D := \{d' \mid \text{for some grant } \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in R_p \text{ and closed substitution } \sigma, d_g\sigma = d' \text{ and } e_g\sigma = e\}$

if $\text{Holds2}(d', L, S') = \text{true}$ for a condition $d' \in D$
then return \text{true}
else return \text{false}

Figure 3.6: The \textbf{Holds2} Algorithm
the same input/output behavior on the inputs for which both executions terminate
and, if an execution of \textit{Holds} terminates for a particular input \((d, L)\), then some
execution of \textit{Holds2}(d, L, \emptyset) terminates as well.

\textbf{Proposition 3.2.7.} For all closed conditions \(d\) and sets \(L\) of licenses,

(a) every execution of \textit{Holds}(d, L) that terminates returns the same output,

(b) every execution of \textit{Holds2}(d, L, \emptyset) that terminates returns the same output,

(c) if an execution of \textit{Holds}(d, L) terminates by returning the truth value \(t\), then
an execution of \textit{Holds2}(d, L, \emptyset) terminates by returning \(t\).

Now consider Examples 3.2.5 and 3.2.6. To address the type of nontermination
seen in these examples, we might hope to find an algorithm \textit{Query3} that returns
the same output as \textit{Query2} on inputs for which an execution of \textit{Query2} ter-
minalizes and returns \textit{false} on all other inputs. Returning \textit{false} when no execution
of \textit{Query2} terminates gives an intuitively reasonable answer; moreover, this ap-
proach is essentially what is done in MPEG-21 REL (see Section 3.7 for details).
Unfortunately, as we show shortly (see Theorem 3.4.1) this approach will not work
in general; there is no algorithm \textit{Query3} with these properties, since whether
\textit{Query2} terminates on a given input is undecidable.

Since we cannot “fix” \textit{Query2}, the best we can do is define some restrictions
such that, if the restrictions hold for a particular query, then the problems seen
in Examples 3.2.5 and 3.2.6 do not occur for that query. We now describe some
conditions that are sufficient and that we suspect often hold in practice.

To describe our approach for avoiding the problem seen in Example 3.2.5, let \(g\)
and \(g'\) be the grants \(\forall x_1 \ldots \forall x_n (d_g \rightarrow e_g)\) and \(\forall x_1 \ldots \forall x_m (d_{g'} \rightarrow e_{g'})\) respectively.
The license \((p, g)\) affects the license \((p', g')\) if and only if there are closed substitutions \(\sigma\) and \(\sigma'\) such that a condition of the form \(\text{Said}(p'', e, g\sigma)\) is mentioned in \(d_{g'}\sigma'\) and \(p \subseteq p''\). For example, consider the license set \(L = \{(\text{Alice}, g_1), (\text{Amy}, g_2)\}\), where \(g_1 = \text{Smart}(\text{Bob})\) and \(g_2 = \forall x (\text{Said}(\text{Alice}, \text{Smart}(x)) \Rightarrow \text{Attractive}(x))\).

The license \((\text{Alice}, g_1)\) affects the license \((\text{Amy}, g_2)\) because the conditions are satisfied if \(\sigma\) is a closed substitution and \(\sigma'\) is a closed substitution such that \(\sigma'(x) = \text{Bob}\). A set \(L\) of licenses is hierarchical if there exists a strict partial order \(\prec\) on the licenses in \(L\) such that, for all license \(\ell, \ell' \in L\), if \(\ell\) affects \(\ell'\) then \(\ell \prec \ell'\). Continuing our example, \(L\) is hierarchical because the ordering \((\text{Alice}, g_1) \prec (\text{Amy}, g_2)\) satisfies the requirements. Observe that no hierarchical license set includes the license \((\text{Alice}, \text{Said}(\text{Alice}, e) \Rightarrow e)\) because this license affects itself. The license set in Example 3.2.5 is not hierarchical for essentially the same reason. It is not hard to see that by restricting the set of queries \((e, L, R, E)\) to those in which \(L\) is hierarchical, we avoid the type of circularity that causes the problem seen in Example 3.2.5. In the next result and elsewhere, we use \(\#(X)\) to denote the cardinality of a set \(X\).

**Proposition 3.2.8.** If \(d\) is a closed condition, \(L\) is a hierarchical set of licenses, \(S\) is a set of closed Said conditions, and \(T\) is the call tree of an execution of \(\text{Holds2}(d, L, S)\), then the height of \(T\) is at most \(2\#(L) + 1\).

We further restrict the language to avoid the problem seen in Example 3.2.6. To understand our restriction, recall that \(\text{Auth}(e, L, R)\) first extends \(R\) to \(R'\) by adding all the grants that are issued by someone who has the authority to do so. Since all the grants in \(R' - R\) are in \(L\), the set \(R'\) must be finite. Then \(\text{Auth}\) creates the possibly infinite set \(R_\Sigma\) consisting of all substitution instances of grants in \(R'\), and returns \(\{d \mid d \rightarrow e \in R_\Sigma\}\). (For simplicity here, we are assuming that
\textbf{Auth} does not use the subset assumption; the subset assumption does not affect our discussion.) Since \textbf{Auth} considers only the grants in \( R_\Sigma \) whose conclusion matches the first input to \textbf{Auth}, we could certainly replace \( R_\Sigma \) by \( R'_\Sigma \), where

\[
R'_\Sigma = \{ d_g \sigma \to e \mid \forall x_1 \ldots \forall x_n (d_g \to e_g) \in R', \sigma \text{ is a closed substitution, and } e_g \sigma = e \}.
\]

Because \( e \) is closed, \( R'_\Sigma \) is finite if, for every grant \( g \) in \( R' \), if the condition of \( g \) mentions a free variable \( x \), then either \( x \) ranges over a finite set or \( x \) appears in the conclusion of \( g \). Our solution is simply to restrict the language so that every grant has this property. Since, in our fragment, there are infinitely many resources (grants) and only finitely many principles, this amounts to restricting the language so that if \( \forall x_1 \ldots \forall x_n (d_g \to e_g) \) is a grant, then every free variable of sort \( R_{src} \) that appears in \( d_g \) also appears in \( e_g \). We call a grant \emph{restrained} if it has this property; we call a license \((p, g)\) restrained if \( g \) is restrained. Thus, for example, \( \forall x \forall y(\text{Said}(\emptyset, \text{Permitted}(x, \text{issue}, y)) \rightarrow \text{Permitted}(Alice, \text{issue}, y)) \) is restrained, but neither

\[
\forall y \forall z(\text{Said}(\emptyset, \text{Permitted}(Alice, \text{issue}, y)) \rightarrow \text{Permitted}(Alice, \text{issue}, z))
\]

nor the grant \( \forall x (d(x) \Rightarrow e) \) in Example 3.2.6 is restrained. It is easy to see that, for all restrained grants \( g = \forall x_1 \ldots \forall x_n (d_g \to e_g) \) and closed conclusions \( e \), if \( n \) is the number of primitive principals in the language and \( |g| \) is the length of \( g \), then there are at most \( n^{|g|} \) grants of the form \( d_g \sigma \to e_g \sigma \) such that \( \sigma \) is a closed substitution and \( e_g \sigma = e \). Thus, by considering only restrained grants and licenses, we solve the problem raised in Example 3.2.6.
3.3 Formal Semantics

In this section we provide formal semantics for the XrML fragment described in the previous section; we translate licenses in the grammar to formulas in a modal many-sorted first-order logic. The logic has three sorts: \( \text{Princ} \), \( \text{Right} \), and \( \text{Rsnc} \). The vocabulary includes the following symbols, where \( \text{primitivePrin} \) is the application-provided set of primitive principals and \( \text{primitiveProp} \) is the application-provided set of properties:

- a constant \( p \) of sort \( \text{Princ} \) for every principal \( p \in \text{primitivePrin} \);
- a constant \( \text{issue} \) of sort \( \text{Right} \);
- a ternary predicate \( \text{Permitted} \) that takes arguments of sort \( \text{Princ} \), \( \text{Right} \), and \( \text{Rsnc} \);
- a unary predicate \( \text{Pr} \) that takes an argument of sort \( \text{Princ} \) for each property \( \text{Pr} \in \text{primitiveProp} \);
- a function \( \cup : \text{Princ} \times \text{Princ} \rightarrow \text{Princ} \);
- a function \( f_g : s_1 \times \ldots \times s_n \rightarrow \text{Rsnc} \) for each grant \( g \) in the language; if \( x_1, \ldots, x_n \) are the free variables in \( g \), then \( x_i \) is of sort \( s_i \), for \( i = 1, \ldots, n \). If \( g \) is closed, then the corresponding function is a constant that we denote as \( c_g \); and
- a modal operator \( \text{Val} \) that takes a formula as its only argument.

Intuitively, \( \text{Pr}(p) \) means principal \( p \) has property \( \text{Pr} \), and \( \text{Val}(\varphi) \) means formula \( \varphi \) is valid.
Notice that every principal in the grammar corresponds to a term in the language, because $\cup$ is a function symbol.

The semantics of our language is just the standard semantics for first-order logic, extended to deal with $\text{Val}$. We restrict attention to models for which $\cup$ satisfies the following standard properties:

U1. $\forall x((x \cup x) = x)$

U2. $\forall x_1 \forall x_2((x_1 \cup x_2) = (x_2 \cup x_1))$

U3. $\forall x_1 \forall x_2 \forall x_3((x_1 \cup (x_2 \cup x_3)) = ((x_1 \cup x_2) \cup x_3))$

U4. $\forall x((x \cup \emptyset) = x)$

We call such models acceptable. $\text{Val}(\varphi)$ is true in a model $m$ if $\varphi$ is true in all acceptable models. If a formula $\varphi$ is true in all acceptable models, then we say that $\varphi$ is acceptably valid. Thus, $\text{Val}(\varphi)$ is true in an acceptable model iff $\varphi$ is acceptably valid.

The translation takes four finite sets as parameters. They are a set $L$ of licenses, a set $A$ of closed resources, a set $S$ of closed Said conditions, and a set $E$ of closed conclusions. Roughly speaking, $L$ is the set of licenses that have been issued; $A$ is the set of resources that are relevant to a particular application; $S$ is the set of Said conditions that are assumed not to hold; and $E$ is the set of conclusions that are assumed not to hold. (The roles of each parameter should become clearer in the course of defining the translation and the subsequent discussion.) The translation is defined below, where $s^{L,A,S,E}$ is the translation of the string $s$ given input $L$, $A$, $S$, and $E$.

- If $\text{Permitted}(p, \text{issue}, g) \in E$ or $(p, g) \notin L$, then $(p, g)^{L,A,S,E} = \text{true}$.  

• If $\text{Permitted}(p, \text{issue}, g) \notin E$ and $(p, g) \in L$, then

$$(p, g)^{L,A,S,E} = \text{Permitted}(p, \text{issue}, c_g) \Rightarrow g^{L,A,S,E}.$$ 

Note that we assume $g$ is closed, because this assumption is built into Query.

• $$(d_g \rightarrow e_g)^{L,A,S,E} = ((\bigwedge_{e \in E} \neg \text{Val}(e^{L,A,S,E} \iff e_g^{L,A,S,E})) \land d_g^{L,A,S,E}) \Rightarrow e_g^{L,A,S,E}.$$ 

• $$(\forall x \varphi)^{L,A,S,E} = \bigwedge_{t \in T}(\varphi[x/t])^{L,A,S,E},$$ where $T = A$ if $x$ is of sort $Rsrc$, and $T = P$ if $x$ is of sort $Princ$. (Recall that $P$ is the set of principals.)

• $true^{L,A,S,E} = true.$

• If $\text{Said}(p, e)^{L,A,S,E} \in S$, then $\text{Said}(p, e)^{L,A,S,E} = false.$

• If $\text{Said}(p, e)^{L,A,S,E} \notin S$, then $\text{Said}(p, e)^{L,A,S,E} = \text{Val}((\bigwedge_{g \in R_p} g^{L,A,S',\emptyset}) \Rightarrow e^{L,A,S',\emptyset}),$ where $R_p = \{ g \mid (p', g) \in L \text{ for a } p' \in p \}$ and $S' = S \cup \{\text{Said}(p, e)\}.$

• $$(d_1 \land d_2)^{L,A,S,E} = d_1^{L,A,S,E} \land d_2^{L,A,S,E}.$$ 

• $\text{Permitted}(p, r, s)^{L,A,S,E} = \text{Permitted}(p, r, s^*),$ where $s^* = s$ if $s$ is a variable of sort $Rsrc$, $s^* = c_s$ if $s$ is a closed grant, and $s^* = f_s(x_1, \ldots, x_n)$ if $s$ is an open grant with free variables $x_1, \ldots, x_n.$

• $\text{Pr}(p)^{L,A,S,E} = \text{Pr}(p).$

• for every principal $p$, $\{p\}^{L,A,S,E} = p.$

Note that $\text{Said}(p, e)^{L,A,S,E}$ does not depend on $E$. This matches our intuition that the meaning of a $\text{Said}$ condition depends only on what principals have said, rather than on what is actually true. By adding $\text{Said}(p, e)$ to $S$, we ensure that the meaning of the condition does not depend on itself. Finally, observe that
Said\(^{(p, e)^{L,A,S,E}}\) is defined in terms of the translation of potentially more complex expressions. Nevertheless, the following result shows that the translation is well defined.

**Theorem 3.3.1.** For all strings \(s\) in the language and all finite sets \(L\) of licenses, \(A\) of closed resources, \(S\) of closed Said conditions, and \(E\) of closed conclusions, \(s^{L,A,S,E}\) is well defined.

We believe that our semantics captures the intended meaning of XrML expressions, as implied by the specification. To make this precise, we show that \texttt{Query2} agrees with the semantics on all queries. Specifically, we show that for all terminating executions \(X\) of \texttt{Query2}(\(e, L, R, E\)), \(X\) returns \(\text{true}\) iff

\[
\bigwedge_{\ell \in L} \ell^{L,A,\emptyset,E} \land \bigwedge_{g \in R} g^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E}
\]

is acceptably valid, where \(A = A(e, L, R, E, X)\) is the set of closed resources that appear in the first argument of a call to \texttt{Query2}, \texttt{Auth2}, or \texttt{Holds2} during execution \(X\). Intuitively, \(A\) is the set of resources relevant to answering the query \((e, L, R, E)\). For example, suppose that, during a particular execution \(X\) of \texttt{Query2}(\(e, L, R, E\)), \texttt{Holds2}(\texttt{Said}(\(p, \text{Permitted}(p', \text{issue}, \text{Permitted}(p'', \text{issue}, g))))), \(L, S\)) is called. Then \(A(e, L, R, E, X)\) includes \texttt{Permitted}(\(p'', \text{issue}, g\)) and \(g\). Notice that if \(X\) is a terminating execution, then \(A(e, L, R, E, X)\) is finite.

**Theorem 3.3.2.** Suppose that \((e, L, R, E)\) is a query and \(X\) is a terminating execution of \texttt{Query2}(\(e, L, R, E\)). Then \(X\) returns \(\text{true}\) iff

\[
\bigwedge_{\ell \in L} \ell^{L,A,\emptyset,E} \land \bigwedge_{g \in R} g^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E}
\]

is acceptably valid, where \(A = A(e, L, R, E, X)\).
3.4 Complexity

To answer a query \((e, L, R, E)\), we need to determine whether an execution of \textbf{Query2} \((e, L, R, E)\) returns \textit{true}. We claimed earlier that the problem of answering queries is, in general, undecidable. We now formalize this claim. Recall that a grant \(g\) is restrained if every variable of sort \(Rs\)c mentioned in the antecedent of \(g\) is mentioned in the conclusion of \(g\). We say that a grant \(g\) is in a set \(L\) of licenses if \((p, g) \in L\) for some principal \(p\). A grant \(g\) is in \(R \cup L\), for some set \(R\) of grants, if \(g\) is in \(R\) or \(g\) is in \(L\).

\textbf{Theorem 3.4.1.} Determining whether some execution of \textbf{Query2} \((e, L, R, E)\) returns \textit{true} is undecidable for the set of queries \((e, L, R, E)\) such that at most one grant in \(R \cup L\) is not restrained.

Let \(\mathcal{L}_0\) be the set of queries \((e, L, R, E)\) such that every grant in \(R \cup L\) is restrained. In this section, we examine the computational complexity of answering queries for fragments of \(\mathcal{L}_0\).

We first show that the problem of answering queries for the full language \(\mathcal{L}_0\) is NP hard for two quite different reasons. The first stems from the fact that, if there are \(n\) primitive principals, we can construct \(2^n\) principals using the \(\cup\) operator. The second is that, to answer a query, we might need to determine if exponentially many closed \textbf{Said} conditions hold.

We use the following definitions to state our results. \(\mathcal{L}_1\) is the set of queries that do not mention the \(\cup\) operator. A grant \(g\) is \(n\)-\textit{restricted} if the number of variables of sort \(Princ\) that are mentioned in the antecedent of \(g\) and not in the conclusion of \(g\) is at most \(n\). \(\mathcal{L}_2^n\) is the set of queries \((e, L, R, E)\) such that all grants in \(R \cup L\) are \(n\)-restricted. A call \textbf{Holds2} \((d, L, S)\) is \(h\)-\textit{bounded} if the call tree for every execution
of $\text{Holds2}(d, L, S)$ has height at most $h$. Note that Proposition 3.2.8 shows that if $L$ is a hierarchical set of licenses, then $\text{Holds2}(d, L, S)$ is $(2\#(L) + 1)$-bounded. $\mathcal{L}_3^h$ is the set of queries $(e, L, R, E)$ such that if an execution of $\text{Query2}(e, L, R, E)$ calls $\text{Holds2}(d, L, S)$, then $\text{Holds2}(d, L, S)$ is $h$-bounded. The next result shows that deciding if at least one execution of $\text{Query2}$ returns $\text{true}$ is hard, even if we restrict to queries in $\mathcal{L}_0$ that satisfy any two of the following: the union operator is not mentioned (i.e., restrict to $\mathcal{L}_1$), the query is $n$-restricted for some fixed $n$, or all calls made during an execution of the query are $h$-bounded for some fixed $h$. (We show shortly that the set of queries in $\mathcal{L}_0$ that satisfy all three restrictions is tractable.)

For a formula $\varphi$, let $|\varphi|$ be the length of $\varphi$ when viewed as a string of symbols. For a set $S$, let $|S|$ be the length of $S$; that is $|S| = \sum_{s \in S} |s|$. Finally, we abbreviate $\text{primitivePrin}$, the set of primitive principals, as $P_0$.

**Theorem 3.4.2.** The problem of deciding whether at least one execution of $\text{Query2}(e, L, R, E)$ returns $\text{true}$ for $(e, L, R, E) \in \mathcal{L}_0 \cap \mathcal{L} \cap \mathcal{L}'$ is NP-hard for $\mathcal{L}, \mathcal{L}' \in \{\mathcal{L}_1, \mathcal{L}_2^0, \mathcal{L}_3^h\}$.

If we make all three restrictions (that is, restrict to queries in $\mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2^0 \cap \mathcal{L}_3^h$, for some fixed $n$ and $h$), then determining whether a query returns $\text{true}$ is decidable in polynomial time. However, as we might expect in light of Theorem 3.4.2, the degree of the polynomial depends on $n$ and $h$, and the polynomial involves constants that are exponential in $n$ and $h$. Note that, for queries in $\mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2^0 \cap \mathcal{L}_3^h$, all executions of $\text{Query2}$ terminate and return the same answer. Termination is fairly easy to show since every call tree of an execution of $\text{Query2}(e, L, R, E)$ has a finite branching factor if $(e, L, R, E) \in \mathcal{L}_0$, and has finite height if $(e, L, R, E) \in \mathcal{L}_3^h$. The fact that all executions of $\text{Query2}(e, L, R, E)$ return the same output for all
queries \((e, L, R, E)\in \mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2^n \cap \mathcal{L}_3^h\) follows easily from Proposition 3.2.7(b).

**Theorem 3.4.3.** For fixed \(n\) and \(h\), if \((e, L, R, E)\in \mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2^n \cap \mathcal{L}_3^h\) then determining whether \(\text{Query2}(e, L, R, E)\) returns true takes time \(O(|L||E| + (|R| + |L|)(|L|^{h-1}(|L| + |R| + |e|)^2))\).

The big-O notation is hiding some rather complex (and uninformative) terms that are functions of \(n\) and \(h\); we spell these out in the appendix.

In practice, we believe that queries are often in \(\mathcal{L}_0\) and, as shown in Proposition 3.2.8, if we restrict to queries where the set \(L\) of licenses has size at most \(h\) and is hierarchical (which we expect in practice will often be the case), than all call trees that arise are guaranteed to have height at most \(2h + 1\). Thus, in practice, we expect that we can restrict to queries in \(\mathcal{L}_2^n\) and \(\mathcal{L}_3^h\) for relatively small values of \(n\) and \(h\). Moreover, even for larger values of \(n\) and \(h\) (say, as large as 10), as long as the union operator does not appear, we expect that queries can be answered efficiently, because the upper bound is quite conservative.

How reasonable is it to restrict to queries in \(\mathcal{L}_1\) that do not mention the \(\cup\) operator? We believe that XrML without the \(\cup\) operator is sufficiently expressive for many applications. To examine the effect of not using the \(\cup\) operator, note that principals appear as the first argument in a license, in a **Said** condition, and in a conclusion.

- According to the XrML documentation, the license \((\{p_1, \ldots, p_n\}, g)\) is an abbreviation for the set of licenses \(\{(p, g) \mid p \in \{p_1, \ldots, p_n\}\}\). It follows that we can restrict the first argument of licenses to primitive principals and variables without sacrificing any expressive power. (In fact, we can restrict the first argument of licenses to only primitive principals, because **Query** assumes that if \((p, g)\) is a license in \(L\), then \(p\) is variable-free.)
• We can replace a condition of the form \textbf{Said}(\{p_1, \ldots, p_n\}, e), where \(p_1, \ldots, p_n\) are primitive principals, by \textbf{Said}(\{p_1, \ldots, p_n\}^*, e), where \(\{p_1, \ldots, p_n\}^*\) is a new primitive principal, and then expand the set \(L\) of issued licenses by adding a new license \((\{p_1, \ldots, p_n\}^*, g)\) for every license \((p, g)\) already in \(L\), where \(p \in \{p_1, \ldots, p_n\}\). It is not hard to show that this results in at most a quadratic increase in the number of grants. Thus, as long as the first argument to \textbf{Said} is variable-free, we can express it without using \(\cup\).

• To understand the impact of our restriction on conclusions, we need to consider the meaning of statements such as \textbf{Trust}(\{Alice, Bob\}) and \textbf{Permitted}(\{Alice, Bob\}, issue, g). According to the XrML document, \textbf{Trust}(\{Alice, Bob\}) means Alice and Bob together (i.e., when viewed as a single entity) is trusted; \textbf{Permitted}(\{Alice, Bob\}, issue, g) means Alice and Bob is permitted to issue \(g\). However, the XrML document does not explain precisely what it means for Alice and Bob to be viewed as a single entity. Indeed, it seems to treat this notion somewhat inconsistently (recall the inconsistent use of the subset assumption). There are other difficulties with sets. Notice that if \{Alice, Bob\} is permitted to issue a grant, then presumably \(g\) holds if \{Alice, Bob\} issues \(g\). However, according to the XrML documentation, the license \((\{Alice, Bob\}, g)\) is simply an abbreviation for the set of licenses \(((\{Alice\}, g), (\{Bob\}, g))\). So it is unclear whether a principal that is not a singleton can issue a license. Furthermore, if principals that are not singletons can issue grants and \{Alice, Bob\} is permitted to issue a grant \(g\), then it seems reasonable to conclude that \(g\) holds if \(g\) is issued by both Alice and Bob, but it is not clear whether \(g\) holds if it is issued by only Alice (or by only Bob).
There may well be applications for which these notions have an obvious and clear semantics. But we suspect that such applications typically include only a relatively small set of groups of interest. In that case, it may be possible to simply take these groups to be new primitive principals, and express the relationship between the group and its elements in the language. (This approach has the added advantage of forcing license writers to be clear about the semantics of groups.)

In short, we are optimistic that many applications do not need the union function.

3.5 The Entire XrML Language

XrML has several components that are not in our fragment. Most have been excluded simply for ease of exposition. In this section we list the main omissions, briefly discussing each one.

- XrML supports *patterns*, where a pattern restricts the terms over which a variable ranges. For example, if the variable $x$ is restricted to the pattern “ends in Simpson”, then $x$ ranges over the terms that meet this syntactic constraint (e.g., $x$ ranges over \{HomerSimpson, MargeSimpson, \ldots\}). Patterns in XrML correspond to properties in our fragment. We could represent the example in our fragment by having the property *Simpson* in the language and having the set of grants determine which terms have the property. XrML also allows a pattern to be a set of patterns. We can express a set of patterns as a conjunction of patterns. Since we can express conjunctions of properties in our fragment, we can also capture sets of patterns.

- XrML supports *delegable grants*. A delegable grant $g$ can be viewed as a con-
junction of a grant $g'$ in our fragment and a set $G$ of grants that, essentially, allow other principals to issue $g'$. For example, the delegable grant “Doctor Alice may view Charlie’s medical file and she may also give the right to view the file to her colleague, Doctor Bob” can be viewed as the conjunction of the grant “Doctor Alice may view Charlie’s medical file” and the grant “Alice is permitted to issue the grant ‘Doctor Bob may view Charlie’s medical file’ ”. Thus, we can express delegable grants in our framework.

- XrML supports $grantGroups$, where a grantGroup is a set of grants. We can extend our syntax to support grantGroups by closing the set of grants (as currently defined) under the union operator. Note that our proposed treatment of grantGroups is quite similar to our current treatment of principals.

- XrML includes rights, resources, and conditions that are not in our fragment. There should be no difficulty in extending our translation to handle these new features, and proving an analogue of Theorem 3.3.2. But we might not be able to answer queries in the extended language. The problem is that XrML allows resource terms to be formed by applying functions other than $\cup$. For example, MPEG-21 REL extends XrML by defining a $container$ resource that is a sequence of resources. This naturally translates to a function $container:Rscc \times Rscc \rightarrow Rscc$, so that the container $\langle s_1, s_2, s_3 \rangle$ is translated as $container(s_1, container(s_2, s_3))$. Allowing such functions makes the problem of deciding if a conclusion follows from a set of XrML licenses and grants undecidable, for much the same reason that the validity problem for negation-free Datalog with function symbols is undecidable [NS97].

- XrML allows an application to define additional principals, rights, resources,
and conditions within the XrML framework. Obviously, we cannot analyze terms that have yet to be defined; however, we do not anticipate any difficulty in extending the translation to deal with these terms and getting an analogue of Theorem 3.3.2.

- XrML allows licenses to be encrypted and supports abbreviations via the Inventory component. However, the XrML procedure for determining if a permission follows from a set of licenses assumes that all licenses are unencrypted and all abbreviations have been replaced by the statements for which they stand. In other words, these features are engineering conveniences that are not part of understanding or reasoning about licenses.

3.6 Negation

We believe that many license writers will find it important to deny permissions explicitly and to state conclusions based on whether a permission is granted, denied, or neither granted nor denied by a particular principal. For example, Alice’s mother might want to say “Alice is not permitted to enter the adult website”, a teacher might want to say “if the university does not object, then Alice is permitted to audit the class”, and a lawyer might want to say “if the hospital permits an action that the government forbids, then the hospital is not compliant”.

We can write these statements in XrML by using special “negated predicates”. For example, we can write \texttt{Prohibited}(Alice, enter, adult website) to capture “Alice is not permitted to enter the adult website”\textsuperscript{1}; we can write

\textsuperscript{1}Since XrML allows the application to define only additional principals, rights, resources, and conditions, we cannot add \texttt{Prohibited} to XrML without extending the framework, but the extension is so minor that we ignore it here; moreover, there are no implications as far as complexity goes.
NotSaid(University, Prohibited(Alice, audit, class)) to capture “the university does not say that Alice is not permitted to audit the class” (i.e., the university does not object to Alice auditing); and we can write NotCompliant(Hospital) to capture “the hospital is not compliant”. We remark that this approach of using “negated predicates” has appeared before in the literature [JSS97, BS04]; it is essentially the technique used by XACML [Mos05], another popular license language.

Adding negated predicates to XrML is straightforward; reasoning about statements in the extended language is not. One problem is that we have to handle statements that are intuitively inconsistent. For example, consider the grants Permitted(Alice, issue, g) and Prohibited(Alice, issue, g), which say that Alice is permitted and prohibited to issue the grant g. It is not clear what we should conclude from these grants. In particular, it is not clear if Alice should be allowed to issue g. (The languages that include negated predicates typically require the policy writer to specify how inconsistencies should be resolved.)

Other problems arise if we extend XrML so that the set of conditions includes Pr(p) and NotPr(p), in addition to Said(p, e) and true.

**Example 3.6.1.** Suppose that a company allows employees to access their server and allows nonemployees access if they sign a nondisclosure agreement. If Alice cannot prove that she is an employee, can she still get access to the server by signing a nondisclosure agreement? Intuitively, she should be able to, because Alice is either an employee, in which case she has permission, or she is not an employee, in which case she still has permission because she signed the waiver. However, if we express the query in the obvious way (using negated predicates),
then Alice is not permitted, because

\[ \text{SignedWaiver}(\text{Alice}) \]

\[ \land \forall x (\text{Employee}(x) \Rightarrow \text{Permitted}(x, \text{access}, \text{server})) \]

\[ \land \forall x (\neg \text{Employee}(x) \land \text{SignedWaiver}(x) \Rightarrow \text{Permitted}(x, \text{access}, \text{server})) \]

\[ \Rightarrow \text{Permitted}(\text{Alice}, \text{access}, \text{server}) \]

is not valid. ■

To address the unintuitive behavior shown in Example 3.6.1, we could replace the negated predicates by a negation operator, which is the standard approach in logic. Let XrML\(^{-}\) be XrML extended so that the set of conditions includes \(\neg \text{Said}(p, e)\) as well as \(\text{Said}(p, e)\), and the set of conclusions includes

\(\neg \text{Pr}(p)\) and \(\neg \text{Permitted}(p, r, s)\), as well as \(\text{Pr}(p)\) and \(\text{Permitted}(p, r, s)\). There is no problem extending the semantics of XrML to XrML\(^{-}\). Moreover, by replacing \(\text{NotEmployee}\) in Example 3.6.1 by \(\neg \text{Employee}\), we get the intuitively correct answer. The downside of allowing negation is intractability. Recall that \(\mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2 \cap \mathcal{L}_3\) is a small fragment of XrML: the licenses in this fragment do not mention the \(\cup\) operator, every variable in the antecedent of a grant appears in its conclusion, and the execution tree for all calls to \(\text{Holds2}\) has height at most two. Theorem 3.4.2 shows that queries in \(\mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2 \cap \mathcal{L}_3\) are tractable; however, as we now show, adding negation to this relatively small language makes it intractable.

**Theorem 3.6.2.** Let \((e, L, R, E)\) be a tuple in \(\mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2 \cap \mathcal{L}_3\) extended to include negated \text{Said} conditions and negated conclusions. The problem of deciding whether

\[ \bigwedge_{\ell \in L} \ell^{L,A,S,E} \land \bigwedge_{g \in R} g^{L,A,S,E} \Rightarrow e^{L,A,S,E} \]
is valid is NP-hard. This result holds even if $e$, all of the licenses in $L$, and all of the conclusions in $E$ are in XrML, all but one of the grants in $R$ is in XrML, and the one grant that is in XrML $\neg XrML$ is of the form $\forall x_1 \ldots \forall x_n(\neg e)$.

We are currently investigating whether there is a tractable fragment of XrML that is sufficiently expressive to capture the grants and licenses that are of practical importance. We expect that some ideas from our work on Lithium [HW03] will prove useful in this regard.

### 3.7 MPEG-21 REL

MPEG-21 is an international standard that is based on XrML. In [HW04], we give semantics to a beta version of MPEG-21. All of the problems discussed in Section 3.2.2 are present in the beta version. We reported these issues to Xin Wang and Thomas DeMartini of the MPEG-21 working group before the final version was released, and our concerns were addressed in the final version (although not exactly as specified in Section 3.2.3).

The key differences between XrML and MPEG-21 are as follows.

- **MPEG-21** (consistently) makes the subset assumption; that is, a principal \( \{p_1, \ldots, p_n\} \) has all of the properties and permissions of principal \( p_i \), for \( i = 1, \ldots, n \).

- A **Said** condition takes a trustRoot $s$ and a conclusion $e$. No definition of trustRoot is given in the specification; rather, it is assumed that the application will associate with every trustRoot $s$, set $L$ of licenses, and set $R$ of grants a set $G(s, L, R)$ of grants. Said\((s, e)\) holds if the set $L$ of issued
licenses and $G(s, L, R)$ together imply $e$, where $R$ is the set of grants that implicitly hold.

- Rather than defining an algorithm, MPEG-21 says that $L$ and $R$ imply $e$ if there is a proof tree that shows the result holds. Roughly speaking, a proof tree $t$ shows that $L$ and $R$ imply $e$ if (a) $t$ includes a grant $g$ that implies $e$ if certain conditions hold; (b) for each of these conditions, $t$ includes a proof tree showing that the condition does, in fact, hold, and (c) either $g$ is in $R$ or, for some principal $p$, $(p, g)$ is in $L$ and $t$ includes a proof tree showing that $p$ is permitted to issue $g$.

We believe that the translation and corresponding proof of correctness given in Section 3.3 can be modified in a straightforward way to apply to MPEG-21. If this is indeed the case, then an appropriately modified Query2 can be used to answer queries about licenses and grants that are written in MPEG-21.

### 3.8 Concluding Remarks

XrML is a popular language that does not have formal semantics. Since there are no formal semantics, we cannot argue that the XrML algorithm is incorrect, but its behavior on certain input does seem unreasonable. To address the problem, we modified the algorithm, provided formal semantics for XrML in a way that we believe captures the designers’ intent, and showed that the modified algorithm corresponds to our semantics in a precise sense.

In a broader context, our work emphasizes the need for license languages to have formal semantics. Our analysis of XrML shows that even carefully crafted languages are prone to ambiguities and inconsistencies if they do not have formal
We have examined only a fragment of XrML. A key reason for XrML’s popularity is that the framework is extensible; applications can define new components (i.e., principals, rights, resources, and conditions) to suit their needs. We do not believe there will be any difficulty in giving semantics to the extended language. The real question is whether we can find useful tractable extensions. As we have already seen, functions pose no semantic difficulties, but adding them makes the problem of answering queries in XrML undecidable. Another obvious and desirable feature is negation. Currently, XrML does not support negation in either the condition or conclusion of grants. This is a significant expressive weakness. Without negation, license writers cannot forbid an action explicitly nor can they say that a conclusion holds if a permission is denied or unregulated by a particular principal. While it is easy to extend XrML to include negation, doing so without placing further restrictions on the language makes it intractable. We suspect that we can use our earlier work [HW03] to find a fragment of XrML with negation that is tractable and substantially more expressive. We hope to pursue this in future work.
Chapter 4

ODRL

ODRL, at a high level, is simply another language for reasoning about policies. ODRL is different from both XrML and Lithium in that, in ODRL, every policy is part of an agreement. An agreement says that the owner of a specific resource \( r \) gives a particular set of principals access to \( r \) and that access is governed by a set of policies.

The agreement framework leads to other differences between the languages. In particular, since every ODRL policy gives the members of a particular set of users access to a specific resource, we cannot write certain policies in ODRL such as “everyone is permitted to download the movie trailer” and “Alice is permitted to download every resource that is in the public domain”. In addition, because the resource owner only grants rights to certain users, denying policies (i.e., policies that forbid actions) are fairly rare; they occur only when a set of users is granted an exclusive right. For example, an agreement can say that Pixar gives Disney the exclusive right to distribute the movie “Cars”. The agreement implies that Disney may distribute “Cars” and everyone else is forbidden to distribute the movie.

ODRL has other unique features. For example, the antecedent of an ODRL policy can mention disjunction and negation without restriction. So ODRL is not comparable, and in particular is not subsumed, by either Lithium or XrML.

The ODRL specification consists of an English description of the language components. It does not include formal semantics nor is an algorithm given for determining if a set of agreements imply a particular permission in a given environment. In this chapter, we rectify both of these omissions.
The rest of the chapter is organized as follows. In the next section, we present a representative fragment of ODRL. In Section 4.2, we give a semantics to this fragment by translating expressions in the language to formulas in first-order logic. In Section 4.3, we define when a set of ODRL statements implies a permission (or prohibition); show that determining whether a particular implication holds is, in general, NP-hard; and find a tractable fragment of the language. We give a general critique of ODRL, along with suggested improvements, in Section 4.4. We conclude in Section 4.5.

4.1 The ODRL Language

We describe ODRL by giving an abstract syntax for a representative fragment of the language. Using this abstract syntax, rather than the XML-based syntax of ODRL, simplifies the presentation and discussion of our semantics. To illustrate the differences between the two notations, consider the statement “If Mary Smith pays five dollars, then she is allowed to print the eBook ‘Treasure Island’ twice and she is allowed to display it on her computer as many times as she likes”. (A similar expression is discussed in [GNS03].) We can write the statement in ODRL as

```xml
<agreement>
  <asset>
    <context>
      <uid> Treasure Island </uid>
    </context>
  </asset>
  <permission>
</agreement>
```
<display>
  <constraint>
    <cpu>
      <context>
        <uid> Mary’s computer </uid>
      </context>
    </cpu>
  </constraint>
</display>

<print>
  <constraint> <count> 2 </count> </constraint>
</print>

<requirement>
  <prepay>
    <payment>
      <amount currency="AUD"> 5.00</amount>
    </payment>
  </prepay>
</requirement>

<permission>
  <party>
    <context>
      <name> Mary Smith </name>
    </context>
  </party>
</permission>
In our syntax, we write the statement as

\[
\text{agreement for Mary Smith about Treasure Island with prePay}[5.00] \rightarrow \text{and}[\text{cpu}[Mary's Computer] \Rightarrow \text{display}, \\
\text{count}[2] \Rightarrow \text{print}].
\]

Our syntax is given in Figures 4.1, 4.2, and 4.3. We now discuss its main features and then present a summary of the key differences between our syntax and that of ODRL.

The central construct of ODRL is an agreement. An agreement says that a principal (i.e., an agent or a group) prin\textsubscript{u} is allowed to access an asset according to a set of policies (i.e., rules). Typically, prin\textsubscript{u} is called the agreement’s user. For example, suppose that an agreement says “Alice is allowed to play ‘Finding Nemo’, if she first pays five dollars”. Then, the user is Alice, the asset is ‘Finding Nemo’, and the policy is “The user may play the asset, if she pays five dollars”.

The set of principals and assets is application-dependent. For example, a digital library might have a principal for each patron and an asset for each publication. We assume that the application provides a set Assets of assets, as well as a set Subjects of subjects. The set of principals is defined inductively: every subject in Subjects is a principal and every group (i.e., set) of principals is a principal. Roughly speaking, if a policy applies to a principal prin, then the policy applies to every subject in prin.

Every agreement includes a policy set. A policy set consists of a prerequisite and a policy. Roughly speaking, if the prerequisite holds, then the policy holds;
agr ::= agreement

agreement for prin

about a with ps

prin ::= principal

s subject

{prin₁, ..., prinₘ} group

a ∈ Assets asset

s ∈ Subjects subject

ps ::= policy set

prq → p primitive policy set

prq ← p primitive exclusive policy set

and[ps₁, ..., psₘ] conjunction (m ≥ 1)

p ::= policy

prq ⇒ᵯ act primitive policy

and[p₁, ..., pₘ] conjunction (m ≥ 1)

act ::= action

play play asset

print print asset

display display asset

id ∈ PolIds policy identifier

---

Figure 4.1: Abstract Syntax for ODRL (Agreements)
A policy consists of a prerequisite, an action, and a unique identifier. The policy says that, if the prerequisite holds, then the agreement’s user may perform the action to the agreement’s asset. (We use the identifiers to simplify the translation. They are optional in ODRL.) The set of policies is closed under conjunction. For
\begin{itemize}
\item \textit{cons} ::= \text{constraint}
\item \textit{prin} \quad \text{principal}
\item \textbf{forEachMember}[\textit{prin}; \textit{cons}_1, \ldots, \textit{cons}_m] \quad \text{constraint distribution} \ (m \geq 1)
\item \textit{count}[n] \quad \text{number of uses} \ (n \in \mathbb{N})
\item \textit{prin}(\textit{count}[n]) \quad \text{number of uses by} \ \textit{prin} \ (n \in \mathbb{N})
\item \textit{req} ::= \text{requirement}
\item \textit{prePay}[r] \quad \text{prepayment} \ (r \in \mathbb{R}_+)
\item \textit{attribution}[s] \quad \text{attribution to subject} \ s
\item \textit{inSeq}[\textit{req}_1, \ldots, \textit{req}_m] \quad \text{ordered constraints} \ (m \geq 1)
\item \textit{anySeq}[\textit{req}_1, \ldots, \textit{req}_m] \quad \text{unordered constraints} \ (m \geq 1)
\item \textit{cond} ::= \text{condition}
\item \textit{not}[\textit{ps}] \quad \text{suspending policy set}
\item \textit{not}[\textit{cons}] \quad \text{suspending constraint}
\end{itemize}

Figure 4.3: Abstract Syntax for ODRL (Prerequisites Components)
simplicity, we often omit the identifier if it is not relevant to our examples and we restrict the set of actions to play, print, and display.

A prerequisite is either true, a constraint, a requirement, or a condition. The prerequisite true always holds. For simplicity, we abbreviate policy sets of the form true → p as p, and we abbreviate policies of the form true ⇒ act as act. Constraints are facts that are outside the user’s influence. For example, there is nothing that Alice can do to meet the constraint “The user is Bob”. Requirements are facts that are typically within the user’s power to meet. For example, Alice can meet the requirement “The user has paid five dollars” by making the payment. Although the distinction between constraints and requirements is not relevant when answering questions about what is and is not permitted, it is relevant for other types of queries. In particular, it provides key information when determining what a principal can do to obtain a permission. Finally, conditions are constraints that must not hold. The statement “The user is not Bob” is an example of a condition.

The set of prerequisites is closed under conjunction, disjunction, and exclusive disjunction (i.e., under and, or, and xor). Conjunction allows a single policy or policy set to have multiple prerequisites. For example, we use conjunction to write the policy “If the user pays one dollar and acknowledges Alice as the creator of file f, then the user may copy f”. Disjunction and exclusive disjunction are used to abbreviate policies and policy sets in a natural way. For example, consider the policy “If the user pays five dollars then the user may watch the movie and if the user is Alice, then the user may watch the movie”. Using disjunction, we can abbreviate the policy as “If the user pays five dollars or the user is Alice, then the user may watch the movie”.
Our fragment of ODRL includes two primitive forms of constraints: *user constraints* and *count constraints*. A *user constraint* is a principal prin; a subject s meets the constraint if s ∈ prin. A *count constraint* refers to a set P of policies, and is parameterized by an integer n. The constraint holds if n is greater than the number of times the user of the agreement has invoked the policies in P to justify her actions. If the constraint appears in a policy p, then P = {p}. Otherwise, the constraint appears in some policy set ps and P is the set of policies mentioned in ps.

**Example 4.1.1.** Consider the following agreement:

\[
\text{agreement for } \{Alice, Bob\} \text{ about The Report with and}[p_1, p_2],
\]

where \(p_1\) is \(\text{count}[5] \rightarrow_{d_1} \text{print}\) and \(p_2\) is \(\text{and}[Alice, \text{count}[2]] \rightarrow_{d_2} \text{print}\). (Recall that \(\text{and}[p_1, p_2]\) is an abbreviation for the policy set \(\text{true} \rightarrow \text{and}[p_1, p_2]\).) The agreement says that asset *The Report* may be printed a total of five times by either Alice or Bob, and twice more by Alice. That is, if Alice and Bob have used policy \(p_1\) to justify their printing of *The Report* \(a_1\) and \(b_1\) times, respectively, then either may do so again if \(a_1 + b_1 < 5\). Similarly, if Alice and Bob have used the policy \(p_2\) to justify printing \(a_2\) and \(b_2\) times, respectively, then Alice may do so again if \(a_2 + b_2 < 5\). Note that, since Bob does not satisfy the constraint of being Alice, \(b_2\) is 0, so the second policy amounts to giving Alice the permission to print *The Report* twice (in addition to any printings made by invoking other policies).

A count constraint that appears in a policy set is interpreted in a similar way.

**Example 4.1.2.** Consider the following agreement:

\[
\text{agreement for } \{Alice, Bob\} \text{ about The Report with } \text{count}[5] \rightarrow \text{and}[p_1, p_2],
\]
where \( p_1 \) is \textit{print} and \( p_2 \) is \textit{display}. The agreement says that Alice and Bob may invoke policies \( p_1 \) and \( p_2 \) a total of five times to justify the printing or displaying of asset \textit{The Report}. That is, if Alice and Bob have used policy \( p_1 \) to justify the \textit{print} action \( a_p \) and \( b_p \) times respectively, and have used policy \( p_2 \) to justify the \textit{display} action \( a_d \) and \( b_d \) times respectively, then either of them may print or display again if \( a_p + b_p + a_d + b_d < 5 \).

The constraint \texttt{forEachMember} takes a principal \textit{prin} (usually a group) and a list \( L \) of constraints; it holds if each principal in \textit{prin} satisfies each constraint in \( L \).

ODRL supports nested constraints, where a constraint is used to modify another constraint. To illustrate how our approach can accommodate nested constraints, consider the constraint \( \textit{prin}\langle \texttt{count}[n]\rangle \), which is interpreted like a \texttt{count}[n] constraint, except that it applies to the principal \textit{prin} rather than to the user of the agreement. Thus, the constraint holds if \( n \) is greater than the number of times \textit{prin} has used the policies to justify her actions.

**Example 4.1.3.** Consider the following agreement:

\[
\text{agreement for } \{Alice, Bob\} \textit{ about } The \textit{Report} \textit{ with } ps,
\]

where \( ps \) is \texttt{true} \( \rightarrow \) \( p \) and \( p \) is \( Alice\langle \texttt{count}[1]\rangle \) \( \Rightarrow \) \textit{print}. The agreement says that if Alice has not invoked policy \( p \) to print asset \textit{The Report}, then she may do so; until she does, Bob may use \( p \) to print \textit{The Report} any number of times.

**Example 4.1.4.** Consider the following agreement:

\[
\text{agreement for } \{Alice, Bob, Charlie\} \textit{ about } The \textit{Report} \textit{ with } ps,
\]

where \( ps \) is \texttt{and}[(\{Alice, Bob\}, \{Alice, Bob\}\langle \texttt{count}[5]\rangle)] \( \rightarrow \) \texttt{and}[(p_1, p_2), p_1 \textit{ is } \textit{print}, and \( p_2 \) is \textit{display}. The agreement says that Alice and Bob may invoke policies \( p_1 \)
and $p_2$ a total of five times to justify printing and displaying asset *The Report*. Since Charlie does not satisfy the prerequisite $\{Alice, Bob\}$, he cannot invoke $p_1$ or $p_2$.

There are two primitive requirements, *prePay* and *attribution*. The *prePay* requirement takes an amount of money as a parameter; it holds if the user pays the specified amount. The *attribution* requirement takes a subject $s$ as a parameter; it holds if $s$ is properly acknowledged (e.g., as the writer, producer, etc.). The set of requirements is closed under the *inSeq* construct, which says the requirements must be met in a particular order (e.g., acknowledge, then pay), and under the *anySeq* construct, which says the requirements can be met in any order.

Finally, there are two types of conditions, negated constraints and negated policy sets. The condition $\text{not}[^{\text{cons}}]$ holds if and only if the constraint $\text{cons}$ does not hold. For example, $\text{not}[^{\text{Alice}}]$ holds if and only if the user is not Alice. Similarly, the condition $\text{not}[^{\text{ps}}]$ holds if and only if the policy set $\text{ps}$ does not hold. But what does it mean that a policy set (and, in particular, a policy) does not hold? Consider the policy “If Alice pays five dollars, then she is permitted to play ‘Finding Nemo’”. There are at least two reasonable interpretations of when the policy does not hold. Under the first interpretation, the policy does not hold if Alice cannot get the permission by paying five dollars. In other words, we could interpret $\text{not}[^{\text{ps}}]$ to mean that a certain set of agreements does not imply $\text{ps}$. A problem with this interpretation is that we do not know which agreements should be used to evaluate the condition. Under the second interpretation, which we favor, the policy does not hold if Alice has paid five dollars and is not permitted to play the movie. In other words, the condition amounts to the logical negation of the policy. We choose this interpretation because it is simple, fairly intuitive, and, as we shall see,
leads to semantics that matches the semantics for negated constraints. (This is encouraging because, in the ODRL specification, the discussion of negated policy sets is essentially identical to the discussion of negated constraints.)

Example 4.1.5. Consider the following agreement:

\[
\text{agreement} \\
\text{for } \{Alice, Bob\} \\
\text{about } ebook \\
\text{with } \text{count}[10] \rightarrow \\
\text{and}(\text{forEachMember}[\{Alice, Bob\}; \text{count}[5]] \Rightarrow_{id_1} \text{display}, \\
\text{forEachMember}[\{Alice, Bob\}; \text{count}[1]] \Rightarrow_{id_2} \text{print}).
\]

The agreement says that Alice and Bob may each display the asset \textit{ebook} up to five times, and they may each print it once. However, the total number of actions, either displays or prints, done by Alice and Bob may be at most ten.

Example 4.1.6. Consider the following agreement:

\[
\text{agreement} \\
\text{for } \{Alice, Bob\} \\
\text{about } \textit{latestJingle} \\
\text{with } \text{inSeq}[\text{prePay}[5.00], \text{attribution}[Charlie]] \rightarrow \\
(Alice(\text{count}[10]) \Rightarrow_{id} \text{play}).
\]

\[\text{It is worth noting that we could modify our interpretation without contradicting the specification. Continuing with our example, one variation is to have the condition hold if Alice paid five dollars and is not explicitly permitted to play the movie. Another variation is to have the condition hold if Alice paid five dollars and is explicitly forbidden to play the movie. We can handle all of the variations by extending our semantics in a fairly straightforward way (using a validity operator); see [HW04] for details.}\]
The agreement says that after paying five dollars and then acknowledging Charlie, Alice is permitted to play the asset *latestJingle* up to ten times. Moreover, any subject that is neither Alice nor Bob is forbidden from playing *latestJingle*. (Bob’s right is unregulated.)

As mentioned at the beginning of this section, the syntax presented here differs from the one described in the ODRL specification. The key differences are discussed below.

**Authorship.** An ODRL agreement includes a principal called the owner. Roughly speaking, the owner is the principal who is granting the permissions that are mentioned in the agreement. While this information can be useful in practice (e.g., for auditing), our syntax does not mention the owner of an agreement because the identity of the owner does not affect the legitimacy of an ODRL agreement—an agreement holds regardless of who created it.

**Offers.** In addition to agreements, ODRL includes offers, which are essentially agreements without users. Intuitively, an offer is a contract (governing the use of an asset) that does not apply until it is accepted by a user; once accepted, it becomes an agreement. We can interpret offers much as we do agreements.

**Permissions versus Policies.** The ODRL specification uses the term permission to refer to actions, policies, and policy sets, as defined here. We introduce the distinction to clarify the exposition and to emphasize the two-tiered structure of ODRL. Notice that it is the two layers in the framework that allow a prerequisite to apply to multiple policies.

**Contexts.** ODRL uses contexts to assign additional information to agreements, prerequisites, and other entities. A context might include a unique identifier, a human-readable name, an expiration date, and so on. We represent the
context elements that are included in our fragment directly in the syntax. Adding
full contexts to our syntax is straightforward, but it does not add any insight.
Moreover, we believe it obscures the main issues.

**Prerequisites.** Payments and other requirements in ODRL take a number
of arguments. For instance, payments can take an amount and a percentage to
be collected for taxes. We restrict every prerequisite to at most one argument
for simplicity; it is easy to extend our approach to include multiple arguments.
As we have already mentioned, ODRL supports nested constraints. These can be
handled in a manner similar to that used for \(\text{prin}\langle\text{count}[n]\rangle\).

(Sequences and Boolean Connectives.) In ODRL, sequences (\text{inSeq}, \text{anySeq})
and Boolean connectives (\text{and}, \text{or}, \text{xor}), which are called containers in the ODRL
specification, apply to a number of entities. For simplicity, we associate the three
containers with prerequisites, and associate sequences with requirements. The gen-
eral case is a straightforward extension. In particular, the extension of containers
to policies in the obvious way helps resolve the ambiguity discussed in the intro-
duction; the policy “Bob may use Printer One or Bob may use Printer Two” gives
Bob the right to use either printer as he chooses. According to discussions with
Renato Iannella, this is the interpretation intended by the language developers.

**Right Holders.** In ODRL, right holders have a royalty annotation, indicating
the amount of royalty that they receive. This does not reflect an obligation on the
part of the agreement’s user, since payment obligations are captured by require-
ments. Instead, the annotations record how the payments are distributed. Since
we are primarily interested in capturing permissions, we do not consider royalty
annotations, and as a result, do not distinguish right holders from other principals.
**Revocation.** Finally, the ODRL specification mentions revocation, however it is not clearly defined. A revocation invalidates a previously established agreement. Unfortunately, answers to key questions, such as who can revoke an agreement, under what conditions, and subject to what penalties, are not discussed in the ODRL specification. As it stands, a revocation simply indicates that an agreement has been nullified, and thus may be ignored.

### 4.2 A Semantics Using First-Order Logic

In this section, we formalize the intuitive description of ODRL given in Section 4.1. Specifically, we present a translation from agreements to formulas in many-sorted first-order logic with equality. We assume sorts *Actions, Subjects, Assets, PolIds,* and *SetPolIds* (for sets of policy identifiers), and deliberately identify a sort with the set of values of that sort. We further assume sorts *Reals* and *Times; Real* to represent real numbers, and *Times* to represent time. We interpret real numbers in the standard way. For simplicity, we take sort *Times* to be the nonnegative real numbers extended with the special constant $\infty$ representing infinity. Again, we interpret such extended nonnegative real numbers in the standard way; in particular, $t < \infty$ for every nonnegative real number $t$ different from $\infty$.

The vocabulary includes:

- A predicate **Permitted** on *Subjects $\times$ Actions $\times$ Assets*, where the literal $\text{Permitted}(s, act, a)$ means $s$ is permitted to perform action $act$ on asset $a$.

- A predicate **Paid** on *Reals $\times$ SetPolIds $\times$ Times*. The literal $\text{Paid}(r, I, t)$ means an amount $r$ was paid towards the policies corresponding to the set $I$ of policy identifiers at time $t$. 
• A predicate **Attributed** on **Subjects** × **Times**. The literal **Attributed**(s, t) means s was acknowledged at time t.

• Constants of sort **PolIds**, **SetPolIds**, **Subjects**, and **Assets**; we also assume constants **play**, **display**, and **print** of sort **Actions**.

• Variables of sort **Times** and a variable x of sort **Subjects**.

• A function **count** : **Subjects** × **PolIds** → **Reals**. Intuitively, **count**(s, id) is the number of times subject s used the policy with identifier id to justify an action.

• Standard functions for addition (+) and comparison (<, ≤) of real numbers and extended real numbers.

Before presenting the translation, we define some useful auxiliary functions.

The function **subjects** returns the set all subjects appearing in a principal:

\[
subjects(s) \triangleq \{s\}
\]

\[
subjects(\{\text{prin}_1, \ldots, \text{prin}_k\}) \triangleq \bigcup_{i=1}^{k} subjects(\text{prin}_i).
\]

The function **principals** returns the set of principals that are members of a given principal; if the principal is a subject, the function returns the singleton set consisting of that subject:

\[
principals(s) \triangleq \{s\}
\]

\[
principals(\{\text{prin}_1, \ldots, \text{prin}_k\}) \triangleq \{\text{prin}_1, \ldots, \text{prin}_k\}.
\]

The function **ids** takes a policy p, and returns the set of policy identifiers that are
mentioned in $p$:

$$ids(pr_1 \ldots pr_m \Rightarrow_{\text{id}} act) \triangleq \{\text{id}\}$$

$$ids(\text{and}[p_1, \ldots, p_m]) \triangleq \bigcup_{i=1}^m ids(p_i).$$

The translation proceeds by induction on the structure of the agreement. The translation is given in Figures 4.4 and 4.5; we discuss its key features below.

An agreement is translated into a conjunction of formulas of the form:

$$\forall x(prerequisites(x) \Rightarrow P(x)),$$

where $P(x)$ is itself a conjunction of formulas of the form

$$prerequisites(x) \Rightarrow (\neg)\text{Permitted}(x, act, a)$$

and $x$ is a variable of sort $Subjects$ that is free in $P(x)$. (We use the notation $(\neg)\text{Permitted}(-)$ to indicate that the formula $\text{Permitted}(-)$ might be negated.)

The translation of a policy set $ps$ is a formula $[ps]_{prin_u,a}$, where $prin_u$ is the agreement’s user and $a$ is the asset. A (nonexclusive) primitive policy set $prq \rightarrow p$ translates to an implication: if the user is in $prin_u$ and the prerequisite holds, then the policy holds. An exclusive primitive policy set is translated as a nonexclusive primitive policy set in conjunction with a clause that captures the prohibition (i.e., every subject that is not mentioned in the agreement’s user is forbidden from performing the actions). Conjunctions of policy sets translate to conjunctions of the corresponding formulas. (In the translation, we follow the convention that $\bigwedge_{i=1}^m f_i$ is true when $m = 0$.) Note that the translation of a policy set is defined in terms of a check that the user is in $prin_u$, the translation of a policy, and the translation of a prerequisite. We now consider each of these in turn. The formula
[agreement for prin\textsubscript{u} about a with ps] \triangleq [ps]\textsubscript{prin\textsubscript{u},a}

\[ [prq \longrightarrow p]^{prin\textsubscript{u},a} \triangleq \forall x(([prin\textsubscript{u}]x \land [prq]\textsubscript{x}^{ids(p),prin\textsubscript{u},a}) \Rightarrow [p]\textsubscript{x}^{+prin\textsubscript{u},a}) \]

\[ [prq \longrightarrow p]^{prin\textsubscript{u},a} \triangleq \forall x(([prin\textsubscript{u}]x \land [prq]\textsubscript{x}^{ids(p),prin\textsubscript{u},a}) \Rightarrow [p]\textsubscript{x}^{+prin\textsubscript{u},a}) \]

\[ \land \forall x(\neg [prin\textsubscript{u}]x \Rightarrow [p]\textsubscript{x}^{-a}) \]

\[ [\text{and}[ps_1, \ldots, ps_m]]^{prin\textsubscript{u},a} \triangleq \bigwedge_{i=1}^{m} [ps_i]\textsubscript{prin\textsubscript{u},a} \]

\[ [s]_x \triangleq x = s \]

\[ [(\text{prin}_1, \ldots, \text{prin}_k)]_x \triangleq ([\text{prin}_1]_x \lor \ldots \lor [\text{prin}_k]_x) \]

\[ [prq \longrightarrow \text{act}]^{+prin\textsubscript{u},a} \triangleq ([prq]\textsubscript{x}\textsubscript{\{id\},prin\textsubscript{u},a}) \Rightarrow \text{Permitted}(x, [\text{act}], a) \]

\[ [\text{and}[p_1, \ldots, p_m]]^{+prin\textsubscript{u},a} \triangleq \bigwedge_{i=1}^{m} [p_i]\textsubscript{x}^{+prin\textsubscript{u},a} \]

\[ [prq_1 \ldots prq_m \longrightarrow \text{act}]^{-a} \triangleq \neg \text{Permitted}(x, [\text{act}], a) \]

\[ [\text{and}[p_1, \ldots, p_m]]^{-a} \triangleq \bigwedge_{i=1}^{m} [p_i]\textsubscript{x}^{-a} \]

\[ [\text{play}] \triangleq \text{play} \]

\[ [\text{display}] \triangleq \text{display} \]

\[ [\text{print}] \triangleq \text{print} \]

Figure 4.4: Translation of ODRL Agreements
\[ [\text{true}]^x_{I, \text{priv}_u, a} \triangleq \text{true} \]

\[ [\text{prin}]^x_{I, \text{priv}_u, a} \triangleq [\text{prin}]^x \]

\[ [\text{forEachMember}[\text{prin}; \text{cons}_1, \ldots, \text{cons}_m]]^x_{I, \text{priv}_u, a} \triangleq \bigwedge_{(\text{prin}', i) \in P_m} [\text{cons}_i]^x_{I, \text{priv}_u, a} \]

where \( P_m = \text{principals}(\text{prin}) \times \{1, \ldots, m\} \)

\[ [\text{count}[n]]^x_{I, \text{priv}_u, a} \triangleq (\sum_{(s, id) \in I \times \text{subjects}(\text{prin}_u)} \text{count}(s, id)) < n \]

\[ [\text{prin}([\text{count}[n]])]^x_{I, \text{priv}_u, a} \triangleq (\sum_{(s, id) \in I \times \text{subjects}(\text{prin})} \text{count}(s, id)) < n \]

\[ [\text{req}]^x_{I, \text{priv}_u, a} \triangleq [\text{req}]^x_{0, \infty} \]

where \[ [\text{prePay}[r]]^t_{t', t''} \triangleq \exists t'' (t \leq t'' < t' \wedge \text{Paid}(r, I, t'')) \]

\[ [\text{attribution}[s]]^t_{t', t''} \triangleq \exists t'' (t \leq t'' < t' \wedge \text{Attributed}(s, t'')) \]

\[ [\text{inSeq}[\text{req}_1, \ldots, \text{req}_k]]^t_{t_1, t_k+1} \triangleq \begin{cases} [\text{req}_1]^t_{t_1, t_{k+1}} & \text{if } k = 1 \\ \exists t_2 \ldots \exists t_k (t_1 < \cdots < t_{k+1} \wedge \bigwedge_{i=1}^k [\text{req}_i]^t_{t_i, t_{i+1}}) & \text{if } k \geq 2 \end{cases} \]

\[ [\text{anySeq}[\text{req}_1, \ldots, \text{req}_k]]^t_{t', t''} \triangleq [\bigwedge_{i=1}^k [\text{req}_i]^t_{t_i, t'_{i+1}}]_{t', t''} \]

\[ [\text{not}[\text{ps}]]^x_{I, \text{priv}_u, a} \triangleq \neg ([\text{ps}]^x_{\text{priv}_u, a}) \]

\[ [\text{not}[\text{cons}]]^x_{I, \text{priv}_u, a} \triangleq \neg [\text{cons}]^x_{I, \text{priv}_u} \]

\[ [\text{and}[\text{prq}_1, \ldots, \text{prq}_m]]^x_{I, \text{priv}_u, a} \triangleq \bigwedge_{i=1}^m [\text{prq}_i]^x_{I, \text{priv}_u} \]

\[ [\text{or}[\text{prq}_1, \ldots, \text{prq}_m]]^x_{I, \text{priv}_u, a} \triangleq \bigvee_{i=1}^m [\text{prq}_i]^x_{I, \text{priv}_u} \]

\[ [\text{xor}[\text{prq}_1, \ldots, \text{prq}_m]]^x_{I, \text{priv}_u, a} \triangleq \bigvee_{i=1}^m ([\text{prq}_i]^x_{I, \text{priv}_u} \wedge (\bigwedge_{j=1, j \neq i}^m \neg [\text{prq}_j]^x_{I, \text{priv}_u})) \]

Figure 4.5: Translation of ODRL Prerequisites
\([\text{prin}]_x\) is true if and only if the subject denoted by the variable \(x\) is in the principal \(\text{prin}\).

There are two translations for policies: a positive translation, where the permissions described by a policy are granted, and a negative translation, where they are forbidden. The positive translation of a policy \(p\) is denoted \([p]^{+,\text{prin}_u,a}_x\), where \(\text{prin}_u\) is the user of the agreement, \(a\) is the asset, and \(x\) is a variable that ranges over the subjects. A policy of the form \(prq \implies act\) translates to an implication: if the prerequisite holds, then the subject represented by \(x\) is permitted to perform the action \(act\) on the asset \(a\). The negative translation of a policy \(p\) is a formula \([p]^{-,a}_x\), where \(a\) is the asset, and \(x\) is the variable that ranges over the subjects. If \(p\) is \(prq \implies act\), then the translation says that \(x\) is forbidden to do \(act\) to \(a\), regardless of whether \(prq\) holds. The positive and negative translations of policies are defined in terms of the translation of actions, which is simply the constant corresponding to the action. As with policy sets, conjunctions of policies translate to conjunctions of the corresponding formulas.

The translation of a prerequisite \(prq\) is a formula \([prq]^{I,\text{prin},a}_x\), where \(I\) is a set of policy identifiers, \(\text{prin}\) is a principal, \(a\) is an asset, and \(x\) is a variable of sort \(\text{Subjects}\). Intuitively, \(I\) includes (the identifier of) the policies that are implied by the prerequisites and \(\text{prin}\) is the principal to which the prerequisites apply (the agreement’s user, unless overridden within a \textsf{forEachMember} constraint). A Boolean combination of prerequisites translates to the Boolean combination of the formulas obtained by translating each prerequisite in turn. A user constraint \(\text{prin}\) translates to a formula that is true if the current subject \(x\) is a member of \(\text{prin}\). The translation of the other constraints is more complicated. A \textsf{forEachMember} constraint translates to a formula that is true if, intuitively, each constraint in
forEachMember is met by each subject mentioned in the constraint (i.e., each member). A constraint count[n] translates to a formula that is true if the subjects mentioned in prin_a have invoked the policies identified in I a total of i times where i is less than n. Similarly, a prin(count[n]) constraint translates to a formula that is true if the total number of times that a subject in prin has invoked a policy whose identifier is in I is less than n.

Requirements have a significantly different translation than other prerequisites because of their dependence on time (e.g., inSeq[prePay[r], attribution[s]] holds if r is paid before s is acknowledged). To handle time correctly, we translate \([req]_t,prin,a\) to \([req]_{t,\infty}\), where \([req]_{t,t'}\) is an auxiliary translation that returns a formula that is true if the events specified by requirement req occur within the interval of time between t and t'. If req is a primitive requirement (i.e., a payment or attribution), then we translate \([req]_{t,t'}\) to a formula that is true if the relevant payment or attribution occurred at some time between t and t'. An inSeq requirement is satisfied if there exists appropriate successive times between t and t' at which each subrequirement is satisfied. Similarly, an anySeq requirement is satisfied if the subrequirements are satisfied in some order (possibly simultaneously) between times t and t'.

Conditions are translated by negating the translation of either the policy set or the constraint specified as the argument. Recall that, in ODRL, we can capture statements such as “If Alice is not permitted to print the report, then she is permitted to display it”. We can also write “If Alice is permitted to print the report, then she is permitted to display it”, since xor[true, not[ps]] is equivalent to ps. It follows from our semantics that the first statement alone gives Alice the display permission if she is explicitly forbidden to print the report; the two
statements together imply that Alice may display the report, regardless of which print permissions are granted or denied.

Another subtlety arises in the interpretation of sequence requirements, particularly nested sequence requirements. To illustrate the issue, consider the nested requirement $\text{anySeq}[\text{inSeq}[req_1, req_2], req_3]$. What are the allowed sequences of requirements $req_1$, $req_2$, and $req_3$? One possibility, the one we adopt, is that $\text{inSeq}[req_1, req_2]$ is satisfied if $req_1$ happens before $req_2$. Thus, the following sequences are allowed: $\langle req_1 \text{ req}_2 \text{ req}_3 \rangle$, $\langle req_1 \text{ req}_3 \text{ req}_2 \rangle$, and $\langle req_3 \text{ req}_1 \text{ req}_2 \rangle$. Alternatively, one could say that $\text{inSeq}[req_1, req_2]$ is satisfied if $req_1$ and $req_2$ happen consecutively. Under this interpretation, only the following sequences are allowed: $\langle req_1 \text{ req}_2 \text{ req}_3 \rangle$ and $\langle req_3 \text{ req}_1 \text{ req}_2 \rangle$. We can capture this last interpretation by taking:

$$[\text{anySeq}[req_1, \ldots, req_k]]^I_{t_1, t_{k+1}} \triangleq$$

$$\begin{cases} [[req_1]]^I_{t_1, t_{k+1}} & \text{if } k = 1 \\ \exists t_2 \ldots \exists t_k (t_1 < \cdots < t_{k+1} \land \bigvee_{\pi \in S_k} \bigwedge_{i=1}^k [[req_{\pi(i)}]]^I_{t_i, t_{i+1}}) & \text{if } k \geq 2, \end{cases}$$

where $S_k$ is the set of all permutations of sets of $k$ elements.

Our translation is admittedly complex; however, it is not clear that a simpler translation is possible, due to the distributed nature of agreements (e.g., a count constraint can implicitly refer to policy identifiers that occur throughout the enclosing policy set). To conclude this section, we translate Examples 4.1.5 and 4.1.6 from Section 4.1.

**Example 4.2.1.** Recall the agreement in Example 4.1.5, which says that Alice and Bob may each display the asset ebook up to five times, and they may each print it once.
agreement
  for \{Alice, Bob\}
  about ebook
  with count[10] →

  \(\text{and[forEachMember[}\{Alice, Bob\}; count[5]] \rightarrow_{id_1} \text{display},\)

  \(\text{forEachMember[}\{Alice, Bob\}; count[1]\] \rightarrow_{id_2} \text{print}]\)

translates to the formula

\[\forall x ((x = Alice \lor x = Bob) \Rightarrow \]
\[\text{count}(Alice, id_1) + \text{count}(Alice, id_2) + \]
\[\text{count}(Bob, id_1) + \text{count}(Bob, id_2) \leq 10 \Rightarrow \]
\[((\text{count}(Alice, id_1) < 5 \land \text{count}(Bob, id_1) < 5) \Rightarrow \]
\[\text{Permitted}(x, display, ebook)) \land \]
\[((\text{count}(Alice, id_2) < 1 \land \text{count}(Bob, id_2) < 1) \Rightarrow \]
\[\text{Permitted}(x, print, ebook)).\]

**Example 4.2.2.** Recall the agreement in Example 4.1.6, which says that after paying five dollars and then acknowledging Charlie, Alice is permitted to play the asset \(\text{latestJingle}\) up to ten times. Moreover, any subject that is neither Alice nor Bob is forbidden from playing \(\text{latestJingle}\).

agreement
  for \{Alice, Bob\}
  about \(\text{latestJingle}\)
  with inSeq[prePay[5.00], attribution[Charlie]] →

  \((Alice(count[10]) \rightarrow_{id} \text{play})\)
translates to the formula

\[ \forall x ((x = Alice \lor x = Bob) \Rightarrow \\
\exists t_1 \exists t_2 (t_1 < t_2 \land \text{Paid}(5.00, t_1) \land \text{Attributed}(Charlie, t_2)) \Rightarrow \\
(x = Alice \land \text{count}(Alice, id) < 10 \Rightarrow \\
\text{Permitted}(x, \text{play}, \text{latestJingle})) \land \\
(\neg (x = Alice \lor x = Bob) \Rightarrow \neg \text{Permitted}(x, \text{play}, \text{latestJingle}))). \]

These examples illustrate that, despite the complexity of the translation, the structure of formulas obtained from the translation follows closely that of the agreements.

4.3 Queries

Our formal semantics provides a foundation for reasoning about agreements in a rigorous way. Because of their obvious usefulness, we focus on queries of the form “may subject s do action act to asset a”. In this section, we formally define such queries; then we examine the complexity of answering them.

4.3.1 Formal Definition

Whether a permission (or prohibition) follows from a set of agreements can depend on certain facts about the application. For our fragment of ODRL, the relevant facts are which payments have been made, which acknowledgments have been given, and the number of times each policy has been used to justify an action. We encode this information in an environment, which is a conjunction of positive ground literals, each of the form \text{Attributed}(s, t) or \text{Paid}(s, I, t), and equalities of the form \text{count}(s, id) = n. Based on the type of information stored in
the environment (both for our fragment and for all of ODRL), it seems reasonable to make a form of closed-world assumption: we assume all environment facts are known. That is, if a positive Permitted-free ground literal is not a conjunct of the environment then we assume it does not hold, with two exceptions. First, if there is a subject $s$ and policy identifier $id$ such that no conjunct of $E$ has the form $\text{count}(s, id) = n$, then we assume $\text{count}(s, id) = 0$. Second, if the environment together with the standard interpretation of $+, \times, =, <$, and $\leq$ imply that a positive literal holds, then we assume that it does. For example, if $s$ and $s'$ are subjects, $id$ and $id'$ are policy identifiers, and no conjunct of $E$ has the form $\text{count}(s, id) = n$ or $\text{count}(s', id') = n$, then we assume $\text{count}(s, id) = 0$, $\text{count}(s', id') = 0$, and $\text{count}(s, id) = \text{count}(s', id')$.

Suppose that we are interested in determining whether a set $A$ of agreements implies that a subject $s$ may do action $act$ to asset $a$ in environment $E$. We represent such a query as a tuple $(A, s, act, a, E)$. Answering the query corresponds to establishing the validity of a formula with respect to a particular class of models, which are identified using the following definitions. Recall that a Herbrand model is a model whose domain consists of the closed terms in the language. Recall that a model $m$ interprets the symbols $+, \times, =, <$, and $\leq$ in the standard way if it satisfies the axioms of real closed fields [Tar51] over the sorts Reals and Times—in the latter case, the axioms are modified in the standard way to deal with $\infty$. For all environments $E$, let $\mathcal{F}(E)$ be the set of formulas made up of $E$ itself, the real closed fields axioms (extended to deal with $\infty$), and formulas $\text{count}(s, id) = 0$ for every subject $s$ and policy identifier $id$ such that $\text{count}(s, id)$ is not a conjunct of $E$. Finally, for all queries $q = (A, s, act, a, E)$, define a model $M$ to be $E$-relevant if:
(1) \( M \) is a Herbrand model;

(2) \( M \) satisfies every formula in \( \mathcal{F}(E) \);

(3) \( M \) satisfies a positive \textbf{Permitted}-free ground literal \( \ell \) only if \( M' \) does not satisfy \( \mathcal{F}(E) \), where \( M' \) is the model that is identical to \( M \) except that \( M' \) does not satisfy \( \ell \).

By restricting our attention to \( E \)-relevant models, we limit our focus to exactly those Herbrand models that satisfy the environment \( E \) under the closed world assumption. We believe that these are the only models that should be considered during query evaluation.

Because an environment consists only of positive facts, an environment \( E \) is inconsistent if and only if \( E \) has two conjuncts \( \text{count}(s, id) = n_1 \) and \( \text{count}(s, id) = n_2 \) with \( n_1 \neq n_2 \). Thus, an environment \( E \) is consistent if and only if there exists an \( E \)-relevant model. When evaluating a query \( q = (A, s, act, a, E) \), we consider only those models that are \( E \)-relevant. A formula is \( E \)-valid if it holds in every \( E \)-relevant model.

We now have the necessary foundation to give an answer to a query \( q = (A, s, act, a, E) \). Define the formulas:

\[
\begin{align*}
    f_q^+ & \triangleq \bigwedge_{agr \in A} \lbrack agr \rbrack \Rightarrow \textbf{Permitted}(s, act, a) \\
    f_q^- & \triangleq \bigwedge_{agr \in A} \lbrack agr \rbrack \Rightarrow \neg \textbf{Permitted}(s, act, a).
\end{align*}
\]

The answer to the query depends on the \( E \)-validity of \( f_q^+ \) and \( f_q^- \).

- If both \( f_q^+ \) and \( f_q^- \) are \( E \)-valid, then either the environment is inconsistent, in which case all formulas are \( E \)-valid, or the agreements are inconsistent in
the environment. Either way, an appropriate answer to the query seems to be “Query inconsistent”.

- If $f_q^+$ is $E$-valid and $f_q^-$ is not, the answer is “Permission granted” because, roughly speaking, the permission necessarily follows from the agreements in the given environment.

- Similarly, if $f_q^-$ is $E$-valid and $f_q^+$ is not, then the answer is “Permission denied”.

- Finally, if neither $f_q^+$ nor $f_q^-$ is valid, then the agreements in the given environment do not imply that the permission is granted, nor do they imply that the permission is denied. So the answer is “Permission unregulated”.

4.3.2 Complexity

We now consider the computational complexity of answering queries. It turns out that we can design an algorithm that takes a query and returns the correct answer; however, it seems unlikely that any algorithm will run efficiently on all input.

**Theorem 4.3.1.** The problem of deciding for a query $q = (A, s, act, a, E)$ whether $f_q^+$ is $E$-valid is decidable and NP-hard. Similarly, the problem of deciding for a query $q = (A, s, act, a, E)$ whether $f_q^-$ is $E$-valid is decidable.

Since answering a query $q$ amounts to determining the $E$-validity of $f_q^+$ and $f_q^-$, the first of which cannot be done efficiently, answering a query cannot be done efficiently.

The proof of Theorem 4.3.1 suggests that the intractability result holds, at least in part, because ODRL includes conditions of the form $\text{not}[ps]$, where $ps$ is a
policy set. It might be possible to modify our translation of $\text{not}[ps]$ in such a way that the revised semantics matches the specification and answering queries in the revised language is tractable (i.e., solvable in polynomial time). This is because, as discussed in Section 4.1, the description of $\text{not}[ps]$ in the ODRL specification is open to interpretation. In addition, we could identify a large tractable fragment of ODRL, as interpreted here. However, we believe that neither of these endeavors is particularly interesting because, having discovered that a component of the language is not clearly specified and a natural interpretation leads to intractability, it seems likely that the meaning of that component will be revised. In fact, based on our discussions with the ODRL Working Group, we suspect that conditions of the form $\text{not}[ps]$ will not be included in the next version of ODRL. So, for the rest of this discussion, we restrict our attention to the fragment of ODRL that does not include these conditions.

Let $Q_1$ be the set of queries $(A, s, act, a, E)$ such that no agreement in $A$ mentions a prerequisite of the form $\text{not}[ps]$. We now show that we can answer a query $q = (A, s, act, a, E)$ in $Q_1$ efficiently. As a first step, we consider the special case in which the set of agreements is a singleton. For any expression $e$ (either in our ODRL syntax or in first-order logic), let $|e|$ be the length of $e$ when viewed as a string of symbols. For a set $A$ of agreements, let $|A|$ be $\Sigma_{agr \in A} |agr|$.

**Lemma 4.3.2.** There are algorithms that, given a query $q = (\{agr\}, s, act, a, E)$ in $Q_1$:

(a) determine whether $f_q^+$ is $E$-valid in time $O(|E||agr|^6)$, and

(b) determine whether $f_q^-$ is $E$-valid in time $O(|E| + |agr|)$.

It follows from Lemma 4.3.2 that $Q_1$ is tractable, provided that a permission
(or prohibition) follows from a set of agreements if and only if it follows from a single agreement in the set. Unfortunately, this is not necessarily true.

**Example 4.3.3.** Let $A = \{agr, agr'\}$, where $agr$ is

\begin{center}
agreement for Alice about file with print
\end{center}

and $agr'$ is

\begin{center}
agreement for Bob about file with true $\mapsto$ print.
\end{center}

Observe that $agr$ gives Alice permission to print the file and $agr'$ forbids Alice from printing it, since the agreement gives Bob the right exclusively. Because the agreements contradict each other, $f_q^+$ and $f_q^-$ are $E$-valid for all queries $q = (A, s, act, a, E)$. So the answer to the query $(A, Charlie, print, file, E)$ is “Query inconsistent”, whereas the answer to the query $(\{agr\}, Charlie, print, file, E)$ and to the query $(\{agr'\}, Charlie, print, file, E)$ is “Permission unregulated”.

If we consider only those queries in $Q_1$ for which the set of agreements holds in at least one relevant model, then we get the desired results.

**Lemma 4.3.4.** Suppose that $q = (A, s, act, a, E)$ is a query in $Q_1$ such that $\bigwedge_{agr \in A}[agr]$ is satisfied in at least one $E$-relevant model. For every $agr \in A$, let $q_{agr}$ be the query $(\{agr\}, s, act, a, E)$. Then

(a) $f_q^+$ is $E$-valid if and only if $f_{q_{agr}}^+$ is $E$-valid for some $agr \in A$ and

(b) $f_q^-$ is $E$-valid if and only if $f_{q_{agr}}^-$ is $E$-valid for some $agr \in A$.

It follows from Lemma 4.3.2 and 4.3.4 together that answering a query $q = (A, s, act, a, E)$ $Q_1$ can be done efficiently, provided that $\bigwedge_{agr \in A}[agr]$ is satisfied
in at least one $E$-relevant model. Moreover, if this is not the case, then the query can be answered immediately. If $\bigwedge_{agr \in A}[agr]$ does not hold in any $E$-relevant model then both $f^+_q$ and $f^-_q$ are $E$-valid, so the answer to $q$ is “Query inconsistent”. Therefore, we can answer queries in $Q_1$ efficiently provided we can quickly determine whether the agreements are satisfied in at least one relevant model.

**Lemma 4.3.5.** There is an algorithm that, given a query $q = (A, s, act, a, E)$ in $Q_1$, determines whether $\bigwedge_{agr \in A}[agr]$ is satisfied in at least one $E$-relevant model in time $O(|E||A|^8)$.

Putting all of these results together, we can conclude that answering queries in $Q_1$ is tractable.

**Theorem 4.3.6.** There is an algorithm that, given a query $q = (A, s, act, a, E)$ in $Q_1$, computes the answer to $q$ in time $O(|E||A|^8)$.

We conclude this section with a few observations. We suspect that many queries of practical interest have certain properties that could be used to improve the efficiency of our algorithms. For example, it seems unlikely that a set of agreements will give one principal an exclusive right and give someone else that same right (possibly under certain conditions). That is, if $A$ is a set of agreements such that an agreement in $A$ gives a principal $prin$ the exclusive-right to do an action $act$ to an asset $a$ and another agreement in $A$ gives a principal $prin'$ the right to do $act$ to $a$ if certain prerequisites hold, then we expect that $\text{subjects}(prin') \subseteq \text{subjects}(prin)$. A straightforward syntactic check can be used to determine whether this is the case for a particular query; if it is, then our proof of Lemma 4.3.5 can be easily modified to show that the consistency check can be done in time $O(|E|)$; that is,
in time $O(|E|)$, we can determine whether there is an $E$-relevant model satisfying
\[ \land_{agr \in A} [agr], \] where $A$ is the set of agreements mentioned in the query.

We conjecture that answering a query $(A, s, act, a, E)$ in ODRL can be done efficiently, provided that, if an agreement in $A$ mentions a prerequisite of the form $\text{xor}[prq_1, \ldots, prq_n]$, then $prq_i$ does not mention a prerequisite of the form $\text{not}[ps]$, where $ps$ is a policy set, for $i = 1, \ldots, n$. We believe that we can use ideas discussed in [HW03] to prove this result, however, we have not checked the details because, as previously discussed, it is not clear that such a result is of practical interest.

4.4 Discussion: Improving ODRL

The process of working through the ODRL specification to derive the formal semantics highlighted a number of potential weaknesses in the design of ODRL. In addition to not having formal semantics, the ODRL specification does not discuss which agreements should be enforced, how conflicts should be resolved, how agreements can be revoked, and how the environment can be maintained. We examine these issues in turn.

The ODRL specification does not say which agreements should be used when evaluating requests. The developers seem to assume that only a legitimate agent will be able to create a particular agreement; however, it is not clear which agents should be recognized as legitimate. Are there ODRL agreements that give subjects the right to create agreements? If so, who is allowed to write those agreements? A natural approach is simply to assume that everyone can write agreements; it is up to the enforcing system to determine which are legitimate. A problem with this design is that an agreement might be meaningless on some systems and quite significant on others. For example, suppose that Bob stores his diary on his home
machine, which assumes all agreements are legitimate, and on his work machine, which assumes an agreement is legitimate only if written by a manager of the company. If Bob’s sister Alice, who is not a manager of the company, writes an agreement that gives her permission to see Bob’s diary, then the home machine will permit the access while the work machine will not.

A more satisfying approach is to define the circumstances under which an agreement is legitimate and require only legitimate agreements to be considered during query evaluation. A definition for legitimacy might say that some agreements are legitimate by fiat (e.g., any agreement about an asset \( a \) issued by its owner), while others are legitimate because there is some proof of legitimacy (e.g., an agreement about an asset \( a \) issued by subject \( s \) is legitimate, because the owner of \( a \) has written an agreement that gives \( s \) permission to regulate access to \( a \)). This is essentially the approach adopted for XrML [Con01].

The ODRL specification does not discuss how conflicts should be resolved. For example, suppose that Alice gives Bob the exclusive right to distribute her movie and she gives Charlie the right to distribute it as well. Is Charlie allowed to distribute the movie? By the definition given in Section 4.3, the answer is “Query inconsistent” because the agreements are inconsistent in the environment (regardless of what the environment is). We can avoid this situation in at least two ways. We can store each agreement with the relevant asset; that way, conflicts can be detected, and hopefully resolved, as soon as a conflicting agreement is associated with the asset. Observe that this approach still allows conflicts and requires intervention, however conflicts are detected at a more opportune time, when agreements are made rather than when queries are asked. An alternative is to remove exclusive policy sets from the language so that conflicts cannot occur.
because no agreement can forbid an action. Finally, it is worth noting that, in languages such as XACML [Mos05] and FAF [JSSS01], conflicts are handled by requiring users to write overriding policies, such as “If an action is both permitted and forbidden, then it is forbidden”. Unfortunately, applying this solution to ODRL seems to require a substantial change to the language.

The ODRL specification discusses revocation, but does not give a mechanism for revoking agreements or for checking whether an agreement has been revoked. Revocation can be added in a fairly straightforward way. In particular, policies governing revocation could be part of agreements or built-in to ODRL; agreements revoked by someone with permission could be added to a revocation list stored in the environment; and only agreements not on the list could be considered during query evaluation. It is not clear, however, that these changes are necessary. Even without revocation, ODRL is still useful in practice because policies can “expire” (e.g., only apply during a specific time interval or for a fixed number of uses).

Finally, the specification does not discuss how the environment is updated while the system is running. Holzer, Katzenbeisser, and Schallhart [HKS04] propose a solution to this problem. They associate with every ODRL agreement an automaton that transitions whenever the user of an agreement performs an action. Thus, to recast their work using our terminology, the states of the automaton corresponding to an agreement are what we call environments. Holzer et al. do not describe how to compute which actions are allowed in any given environment, but they do describe how to update the environment. In contrast, we do not describe how to update environments, but our semantics describes how to compute which actions are permitted in any given environment. In this sense, the two works solve complementary problems.
4.5 Conclusion

ODRL is a popular rights language with features that we have not found in other approaches. However, the usefulness of ODRL is limited, in part, because the language does not have formal semantics. To address this deficiency, we have proposed a formal semantics for ODRL. In the process of creating this semantics, we discovered aspects of the specification that should be clarified and have discussed our findings with the language developers. They are currently working on the next version of the language, which has formal semantics as one of its seven design requirements.

In addition to giving the language formal semantics, we have considered the practical problem of determining whether a set of ODRL statements imply a permission or prohibition. Using our semantics, we have formally defined the problem and shown that it is, in general, NP-hard. After removing a component of ODRL whose meaning seems to be somewhat unclear, even to the developers, we can are left with a tractable fragment of the language. To prove that the fragment is tractable, we described a polynomial-time algorithm to determine whether a set of ODRL statements imply a permission (or prohibition). To the best of our knowledge, this is the first algorithm for answering such queries in ODRL.

Despite these successes, the work is far from done. We are currently collaborating with the language developers on the next version of ODRL. We are also interested in examining other types of queries, such as what, if anything, a subject can do to get a desired permission.
Chapter 5

Conclusion

In this dissertation, we defined Lithium, a fragment of first-order logic that seems well-suited to reasoning about policies. Using the insights gained from our work with Lithium, we proposed the first formal semantics for XrML and ODRL. The process of giving the languages semantics revealed significant problems with both languages. We worked with the language developers to correct the problems in the next versions of the languages.

We examined XrML and ODRL because, when we began our investigations, they were the two languages that seemed most favored by industry. Since then, a third language has come to share the limelight. It is called XACML. We give a brief overview of XACML in the next section and then conclude with our plans for future work.

5.1 XACML

XACML is an XML-based language that does not have formal semantics. Several companies worked together to develop the language including Entrust, IBM, and Sun Microsystems. On February 1, 2005, XACML was ratified as an OASIS Open Standard. More recently, Fedora (the Flexible Extensible Digital Object and Repository Architecture) has chosen to implement XACML as part of its security system. In this section, we describe XACML at the level needed to compare it to Lithium, XrML, and ODRL in a meaningful way. Giving formal semantics to XACML is left as future work.
An XACML policy can be represented as a closed formula

$$\forall x_1 \ldots \forall x_n (f \Rightarrow (\neg)\text{Permitted}(t_1, t_2, t_3)),$$

where $f$ is a quantifier-free $\text{Permitted}$-free formula and $(\neg)\text{Permitted}(t_1, t_2, t_3)$ means principal $t_1$ is permitted (or forbidden) to perform action $t_2$ on resource $t_3$. Observe that $f$ can mention negation and disjunction without restriction. It is not clear from the specification whether $f$ can mention variables that are not mentioned in the conclusion of the policy (i.e., in $t_1$, $t_2$, or $t_3$).

Policies in XACML are examined in a basic environment; that is, the environment can be represented as a conjunction of $\text{Permitted}$-free ground literals. A specific permission $\text{Permitted}(c_1, c_2, c_3)$ follows from a policy $\forall x_1 \ldots \forall x_n (f \Rightarrow \text{Permitted}(x_1, x_2, x_3))$ in an environment $E$ if and only if there is a substitution $\sigma$ such that $\text{Permitted}(x_1, x_2, x_3)\sigma = \text{Permitted}(c_1, c_2, c_3)$ and $E$ “implies” $f\sigma$. In XACML, $E$ implies $f\sigma$ if and only if the formula $f'$ is valid, where $f'$ is obtained from $f\sigma$ by replacing every instance of a literal $\ell$ by $true$ if $\ell$ is a conjunct of $E$, by $false$ if the negation of $\ell$ is a conjunct of $E$, and by a fresh ground literal otherwise. The intuition is that, whether the conditions of a policy hold under a particular substitution depends only on $E$; any literal not mentioned in $E$ is, roughly speaking, irrelevant.

**Example 5.1.1.** Suppose that we want to know whether Alice may access the company’s server given the following statements: employees are permitted to access the server; non-employees are permitted access if they sign a waiver; Alice has signed a waiver; and we do not know whether Alice is an employee. As discussed in Example 3.6.1, Alice should be allowed access because she is either an employee, in which case access is permitted, or she is not an employee, in which case access is again permitted because she signed a waiver.
The environment $E$ can be represented as the literal $\text{SignedWaiver}(\text{Alice})$, so the environment is basic and consistent. We can write a single policy $p$ to capture both statements about permissions, namely

$$\forall x ((\text{Employee}(x) \lor (\neg\text{Employee}(x) \land \text{SignedWaiver}(x))) \implies \text{Permitted}(x, \text{access}, \text{server}))$$

The permission follows from $p$ in $E$ if and only if, according to XACML, $E$ implies $f = \text{Employee}(\text{Alice}) \lor (\neg\text{Employee}(\text{Alice}) \land \text{SignedWaiver}(\text{Alice}))$. Since neither $\text{Employee}(\text{Alice})$ nor $\neg\text{Employee}(\text{Alice})$ are conjuncts of $E$, $E$ implies $f$ if and only if $\ell \lor (\ell' \land \text{true})$ is valid, where $\ell$ and $\ell'$ are distinct ground literals.

Since $\ell \lor (\ell' \land \text{true})$ is not true in a model that satisfies $\neg\ell \land \neg\ell'$, the formula $\ell \lor (\ell' \land \text{true})$ is not valid and, thus, $p$ does not imply that Alice may access the server.

Observe that $p$ does imply the permission in the environment $\text{Employee}(\text{Alice})$, because the formula $\text{true} \lor (\text{false} \land \ell)$ is valid for all ground literals $\ell$. \[1\]

A similar definition is used to determine whether a specific prohibition, such as $\neg\text{Permitted}(\text{Alice}, \text{access}, \text{server})$, follows from a certain policy in a particular environment.

The policy writer defines when a permission is granted or denied by a set $P$ of policies in an environment $E$ based on whether each policy in $P$ grants or denies the permission in $E$. For example, the policy writer could decide that a permission $\text{Permitted}(c_1, c_2, c_3)$ follows from a set $P$ of policies in an environment $E$ if and only if some policy $p \in P$ implies $\text{Permitted}(c_1, c_2, c_3)$ in $E$. Alternatively, a policy writer could say that $\text{Permitted}(c_1, c_2, c_3)$ follows from $P$ in $E$ if and only
if some policy $p \in P$ implies $\text{Permitted}(c_1, c_2, c_3)$ in $E$ and no policy $p' \in P$ implies $\neg\text{Permitted}(c_1, c_2, c_3)$ in $E$.

5.2 Future Work

The conclusions of Chapters 2, 3, and 4 give some ways to continue the work on Lithium, XrML, and ODRL, respectively. In addition, we would like to address two research questions. To motivate the first question, note that XrML, ODRL, and XACML are the languages that currently have the most industry support. We have shown that these languages have problems due to ambiguities and inconsistencies; the problems became readily apparent when we tried to give the languages formal semantics. This suggests that any reasonably expressive policy language is likely to have problems unless that language has formal semantics.

How do we ensure that the next generation of policy languages will have formal semantics? One answer is to provide semantics for each new language. For example, the next version of ODRL will have formal semantics because we are part of the working group and have the task of providing it. We might also give formal semantics for XACML. A problem with this solution is that it is fairly inefficient. To understand the intentions of the language developers at the level necessary to provide semantics requires a substantial amount of time and energy, both from the developers trying to clarify their specifications and from those trying to learn the language. Another problem is that, if we do not learn about a language until fairly late in its development, then the semantics have to be “retrofitted”, which is likely to lead to semantics that are not particularly intuitive.

A better solution is to convince language developers to provide their own formal semantics. As a step in that direction, we want to define a simple extensible policy
language $\mathcal{L}_p$ that is accessible to developers and has formal semantics. Because $\mathcal{L}_p$ is accessible to developers, a developer can provide a partial translation from her language to $\mathcal{L}_p$; that is, the developer can provide a translation for the fragment of her language that can be written in $\mathcal{L}_p$. Experts in formal methods can then complete the translation by extending $\mathcal{L}_p$ as needed. Observe that the translation provides formal semantics to the developer’s language because $\mathcal{L}_p$ has formal semantics and this approach is more efficient than having the experts provide the complete translation since the experts have to understand only the non-standard features of the language. In fact, because the same extension can be used for multiple languages, a new extension is needed only for features that are not included in any language that has been translated previously. Of course, translating several policy languages to $\mathcal{L}_p$ (with suitable extensions) does more than provide semantics; it facilitates comparisons between the languages and interoperability as well.

In addition, we want a better understanding of which features of a policy language are of practical interest. Lithium, XrML, ODRL, and XACML all have distinct features. For example, using Lithium, we can discover inconsistencies in a policy set, thereby helping policy writers to find mistakes. XrML has Said conditions and a mechanism for determining which policies have been issued by an appropriate authority (i.e., which are legitimate). ODRL makes the closed world assumption. Finally, in XACML, policies are considered individually and the policy writer can choose how queries are evaluated based on the results of each consideration. Given these differences, it seems reasonable to ask whether the features of one language can be incorporated into another. For example, it is easy to modify Lithium, XrML, and XACML to be suitable for applications that make the closed
world assumption. It is not at all clear whether we can add an error-detection mechanism to XACML that would in some sense match the Lithium capability. The problem with considering these types of questions is that each answer is of practical interest only if the corresponding feature is of practical interest. Continuing the example, if most applications do not make the closed world assumption, then modifying Lithium, XrML, and XACML to use the assumption is arguably a waste of time. So our first step in this direction is to survey many policy sets from real applications to gain an understanding of which features are of practical interest.
Appendix A

Proofs for Chapter 2

The following lemma is the key to proving Theorems 2.2.1 and 2.2.3.

Lemma A.0.1. Let \( L'_0 \) be a set of closed formulas with no constant symbols whose only predicate symbol is \( \text{Permitted} \). Let \( L''_0 \) be the set of closed formulas of the form

\[
(f \Rightarrow \text{Permitted}(c, c')) \Rightarrow \text{Permitted}(c, c'),
\]

where \( c \) and \( c' \) are constants of the appropriate sorts and \( f \in L'_0 \). If the validity problem for \( L'_0 \) is undecidable, then the validity problem for \( L''_0 \) is undecidable.

Proof: We reduce the validity problem for \( L'_0 \) to the validity problem for \( L''_0 \).

Standard manipulations show that \( (f \Rightarrow \text{Permitted}(c, c')) \Rightarrow \text{Permitted}(c, c') \) is equivalent to \( f \lor \text{Permitted}(c, c') \). Clearly, if \( f \lor \text{Permitted}(c, c') \) is not valid, then \( f \) is not valid. Suppose that \( f \lor \text{Permitted}(c, c') \) is valid. Since \( f \) does not mention a constant symbol, \( c \) and \( c' \) do not appear in \( f \), so \( f \lor \forall x \forall y \text{Permitted}(x, y) \) is valid. It follows that \( f \) is valid iff \( f \) is true in all models \( m \) that satisfy \( \forall x \forall y \text{Permitted}(x, y) \). To determine whether \( f \) is true in \( m \), let \( f' \) be the result of replacing all occurrences of \( \text{Permitted}(x, y) \) by \( \text{true} \). Clearly \( f \) is true in \( m \) iff \( f' \) is true in \( m \). Since \( f' \) has no nonlogical symbols, \( f' \) is true in \( m \) iff \( f' \) is valid. Moreover, the validity of \( f' \) is easy to determine. \( \blacksquare \)

Theorem 2.2.1: Let \( L_0 \) be the set of closed formulas of the form

\[
(f \Rightarrow \text{Permitted}(c, c')) \Rightarrow \text{Permitted}(c, c'),
\]

where \( c \) and \( c' \) are constants of the appropriate sorts, \( f \) has a single alternation
of quantifiers, and the only nonlogical symbol in $f$ is $\text{Permitted}$. The validity question for $L_0$ is undecidable.

**Proof:** Let $L_0^A$ be the set of closed formulas that have a single alternation of quantifiers and whose only nonlogical symbol is $\text{Permitted}$. The proof follows from Lemma A.0.1, where we take $L'_0$ to be $L_0^A$, because the validity problem for $L_0^A$ is undecidable [BGG97].

**Theorem 2.2.3:** Let $L_1$ be the set of closed formulas of the form

$$\forall x_1 \forall x_2 (f \Rightarrow \text{Permitted}(c, c')) \Rightarrow \text{Permitted}(c, c'),$$

where $c$ and $c'$ are constants of the appropriate sort and $f$ is a quantifier-free formula whose only nonlogical symbols are $\text{Permitted}$ and a unary function. The validity problem for $L_1$ is undecidable.

**Proof:** Let $L_1^A$ be the set of closed formulas of the form $\exists x_1 \exists x_2 f$, where $f$ is a quantifier-free formula whose only nonlogical symbols are $\text{Permitted}$ and a unary function. Because the validity problem for $L_1^A$ is undecidable [BGG97], it follows from Lemma A.0.1 that the validity problem for the set of formulas of the form

$$(\exists x_1 \exists x_2 f \Rightarrow \text{Permitted}(c, c')) \Rightarrow \text{Permitted}(c, c') \quad (A.1)$$

is undecidable. Standard manipulations show that a formula of the form (A.1) is equivalent to

$$\forall x_1 \forall x_2 (f \Rightarrow \text{Permitted}(c, c')) \Rightarrow \text{Permitted}(c, c').$$

It follows that the validity problem for $L_1$ is undecidable.

**Theorem 2.2.4:** Let $\Phi$ be a vocabulary that contains $\text{Permitted}$, constants $c$ and $c'$ of sorts Subjects and Actions, respectively, and possibly other predicate
and constant symbols (but no function symbols). Assume that there is a bound on
the arity of the predicate symbols in Φ (that is, there exists some N such that all
predicate symbols in Φ have arity at most N). Finally, let L_2 be the set of all
closed formulas in L^{fo}(Φ) of the form E ∧ p_1 ∧ ... ∧ p_n ⇒ \text{Permitted}(c, c') such
that E is a conjunction of quantifier-free and universal formulas and each policy
p_1, ..., p_n has the form \forall x_1 ... \forall x_m(f ⇒ \text{Permitted}(t_1, t_2)), where t_1 and t_2 are
terms of the appropriate sort and f is quantifier-free.

(a) The validity problem for L_2 is in \Pi^P_2.

(b) If L_3 is the set of formulas in L_2 in which every policy’s antecedent is a
    conjunction of literals, then the validity problem for L_3 is \Pi^P_2 hard.

(c) If L_4 is the set of L_2 formulas in which E is quantifier-free, then the validity
    problem for L_4 is both NP-hard and co-NP hard.

Proof: For part (a), straightforward manipulations show that each formula h in
L_2 is equivalent to a closed formula of the form g = \exists x_1 ... \exists x_k g', where g' is
a quantifier-free formula in L^{fo}(Φ). Moreover, |g| is polynomial in |h|. Suppose
that g mentions n distinct constant symbols. Let \mathcal{M} be the class of models whose
domain size is at most max(n, 1). We claim that (1) g is valid iff it is true in
every model in \mathcal{M}, and (2) the problem of determining if g is true in every model
m ∈ \mathcal{M} is in \Pi^P_2.

For part (1), the “only if” direction is trivial. To prove the “if” direction,
suppose by way of contradiction that g is true in every model in \mathcal{M} and g is
not true in a model m with domain D and interpretation I. Let D' = \{I(c) | c is a constant in g\} if g mentions at least one constant, and D' = \{d\} for some
fixed element $d \in D$ if $g$ does not mention any constants. Let $I'$ be the interpretation such that $I'(c) = I(c)$ if $I(c) \in D'$, $I'(c) = d'$ for some fixed $d \in D'$ if $I(c) \notin D'$, and $I'(R) = D'^k \cap I(R)$ for each $k$-ary predicate $R$ in $\Phi$. Let $m'$ be the model with domain $D'$ and interpretation $I'$. Notice that $m'$ is in $\mathcal{M}$. By assumption, $m'$ satisfies $g$, so there are domain elements $d_1, \ldots, d_k$ in $D'$ such that by interpreting $x_i$ as $d_i$ for $i = 1, \ldots, k$, $m'$ satisfies $g'$. Under the same interpretation of $x_1, \ldots, x_k$, $m$ satisfies $g'$. Therefore $m$ satisfies $g$, and we have the desired contradiction.

For part (2), first note that $g$ is true in all models in $\mathcal{M}$ iff it is true in all models with domain $\{1, \ldots, m\}$ for each $m \leq \max(n, 1)$, since every model in $\mathcal{M}$ is isomorphic to one with domain $\{1, \ldots, m\}$ for $m \leq \max(n, 1)$. The truth of $g$ in such a model depends only on the interpretation of the constant and predicate symbols that actually appear in $g$. Let a restricted interpretation be one that interprets only the symbols that appear in $g$. Because there are $m^k$ interpretations of a $k$-ary predicate, and the arity of predicates in $g$ is bounded in a domain of size $m$, the number of restricted interpretations is polynomial in $|g|$. It clearly can be determined in time polynomial in $|g|$ if the formula $g'$ is true under a given restricted interpretation in a model with domain $\{1, \ldots, m\}$. Thus, determining if $g$ is true in such a model is in NP (since it involves guessing an interpretation of $x_1, \ldots, x_k$). It follows that the problem of determining if $g$ is true in every model of $\mathcal{M}$ is in $\Pi^P_2$.

For part (b), let QBF$_2$ consist of all Quantified Boolean Formulas (QBFs) of the form
\[ \forall Q_1 \ldots \forall Q_m \exists P_1 \ldots \exists P_n \varphi, \]
where $\varphi$ is quantifier-free. It is well known that the problem of checking whether
a formula in QBF$_2$ is true is \( \Pi^p_2 \)-complete [Sto77]. We now show how to reduce this problem to the validity problem for \( L_2 \).

Let \( q = \forall Q_1 \ldots \forall Q_m \exists P_1 \ldots \exists P_n \varphi \) be an arbitrary formula in QBF$_2$. Let \( \varphi' \) be \( \varphi \) with \( Q_j \) replaced by the ground literal \( Q_j(c) \) and \( P_k \) replaced by the literal \( P_k(x_k) \), for \( j = 1, \ldots, m \) and \( k = 1, \ldots, n \). It is not hard to see that \( q \) is true iff

\[
q' = P_1(c) \land \ldots \land P_n(c) \land \neg P_1(c') \land \ldots \land \neg P_n(c') \Rightarrow \exists x_1 \ldots \exists x_n \varphi'
\]

is valid. This follows from two observations. First, note that \( q \) is true iff, for every assignment of values to \( Q_1, \ldots, Q_m \), there is an assignment of truth values to \( P_1, \ldots, P_n \) such that \( \varphi \) is true. Second, \( q' \) is valid iff, for every interpretation of \( Q_1(c), \ldots, Q_m(c) \), there is an assignment of domain elements to \( x_1, \ldots, x_n \) such that \( \varphi' \) is true. The antecedent \( P_1(c) \land \ldots \land P_n(c) \land \neg P_1(c') \land \ldots \land \neg P_n(c') \) of \( q' \) ensures that assigning \( x_i \) to \( c \) makes \( P_i(x_i) \) true, while assigning \( x_i \) to \( c' \) makes \( P(x_i) \) false. Thus, the assignment of values to the variables \( x_1, \ldots, x_n \) acts essentially like a truth assignment to \( P_1, \ldots, P_n \).

Straightforward manipulations show that \( A \Rightarrow B \) is valid iff \( A \land \neg B \Rightarrow \text{false} \) is valid, and \( A \land \neg B \Rightarrow \text{false} \) is valid iff \( (\neg A \Rightarrow C) \land \neg B \Rightarrow C \) is valid, provided that none of the nonlogical symbols in \( C \) appear in \( A \) or \( B \). Taking \( A \) to be \( P_1(c) \land \ldots \land P_n(c) \land \neg P_1(c') \land \ldots \land \neg P_n(c') \), \( B \) to be \( \exists x_1 \ldots \exists x_n \varphi' \), and \( C \) to be \( \neg \text{Permitted}(d, d') \), where \( d \) and \( d' \) are distinct from \( c \) and \( c' \), it follows that \( q \) is true iff

\[
\forall x_1 \ldots \forall x_n \neg \varphi' \land (\neg A \Rightarrow \text{Permitted}(d, d')) \Rightarrow \text{Permitted}(d, d') \quad (A.2)
\]

is valid. The formula \( \neg A \Rightarrow \text{Permitted}(d, d') \) is equivalent to

\[
\bigwedge_{i=1}^{n} \neg P_i(c) \Rightarrow \text{Permitted}(d, d') \land \bigwedge_{i=1}^{n} P_i(c') \Rightarrow \text{Permitted}(d, d')).
\]
Replacing $\neg A \Rightarrow \text{Permitted}(d, d')$ in (A.2) by the latter formula gives us a formula in $\mathcal{L}_2$. Thus, we have reduced the truth of a QBF formula to the validity of a formula in $\mathcal{L}_2$, as desired.

For part (c), we prove the NP hardness result by reducing the Hamiltonian path problem to the validity problem for $\mathcal{L}_4$. Let $G$ be an undirected graph, where $V = \{v_1, \ldots, v_n\}$ is the set of nodes and $E$ is the set of edges. Let $\Phi$ be a vocabulary that includes the constants $v_1, \ldots, v_n$, a binary predicate $\text{Edge}$, and $\text{Permitted}$. Finally, let $E = \bigwedge_{(v_i, v_j) \in E} \text{Edge}(v_i, v_j)$, and let

$$p = \forall x_1 \ldots \forall x_n \left( \bigwedge_{i,j \leq n; i \neq j} (x_i \neq x_j) \land \bigwedge_{i < n} \text{Edge}(x_i, x_{i+1}) \right) \Rightarrow \text{Permitted}(c, c').$$

It is not hard to show that $E \land p \Rightarrow \text{Permitted}(c, c')$ is valid iff there is a Hamiltonian path in $G$. The key observations are (1) there is a Hamiltonian path iff there is an assignment of distinct domain elements to $x_1, \ldots, x_n$ such that there is an edge between $x_i$ and $x_{i+1}$ for $i < n$, and (2) there is such an assignment iff $E \land p \Rightarrow \text{Permitted}(c, c')$ is valid.

We prove the co-NP hardness result by reducing the validity problem for propositional logic to the validity problem for $\mathcal{L}_4$. Let $g$ be a propositional formula, let $v_1, \ldots, v_n$ be the propositions in $g$, and let $g'$ be the first-order formula obtained by replacing the proposition $v_i$ in $g$ with the ground literal $R(c_i)$ for $i = 1, \ldots, n$. It is easy to see that $g$ is valid iff $g'$ is valid. Because $g'$ does not include $\text{Permitted}$, $g'$ is valid iff $g' \lor \text{Permitted}(c, c')$ is valid. Standard manipulations show that $g' \lor \text{Permitted}(c, c')$ is equivalent to the $\mathcal{L}_4$ formula $(g' \Rightarrow \text{Permitted}(c, c')) \Rightarrow \text{Permitted}(c, c').$

To prove the theorems in Section 2.3, we need to extend resolution slightly using techniques of paramodulation [RW83]. Note that if $c$ is the clause $\forall x_1 \ldots x_n (c' \lor t_c =
$t', d$ is the clause $\forall y_1 \ldots \forall y_m d'$, $t_d$ is a term in $d'$, and $\sigma$ is a substitution such that $\sigma(t_c) = \sigma(t_d)$, then the following formula is valid:

$$c \land d \Rightarrow \forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_m (c' \lor d'[t_d/t'_c])\sigma. \tag{A.3}$$

A set of clauses is said to be closed under paramodulation if it contains the right-hand side of (A.3) whenever it contains the clauses on the left-hand side. Let $R^P(f)$ be the set of clauses obtained by closing $f$ under resolution and paramodulation. In other words, $R^P(f)$ is the smallest set of clauses that includes the conjuncts of $f$ (when $f$ is in CNF) and, if we can infer a clause $d$ from two clauses $c$ and $c'$ in $R^P(f)$ by using either resolution or paramodulation, then $d$ is in $R^P(f)$.

**Theorem A.0.2.** [Bra75] If $f$ is a formula in CNF one of whose conjuncts is $\forall x(x = x)$, then $f$ is satisfiable if and only if $R^P(f)$ does not include false.

We remark that the clause $\forall(x = x)$ is needed here, although it is valid. For example, it is easy to check that $R^P(\forall(x \neq x))$ does not include false, even though $\forall(x \neq x)$ is not satisfiable. On the other hand, $R^P(\forall(x \neq x) \land \forall(x = x))$ clearly includes false.

**Corollary A.0.3.** Let $f$ be a CNF formula, none of whose clauses mentions a disjunct of the form $t = t'$. Then $f$ is satisfiable iff $R(f \land \forall x(x = x))$ does not include false.

**Proof:** Clearly $f$ is satisfiable iff $f \land \forall x(x = x)$ is satisfiable. Let $g = f \land \forall x(x = x)$. By Theorem A.0.2, it suffices to show that $R^P(g) = R(g)$. Clearly, $R(g) \subseteq R^P(g)$. To show that $R^P(g) \subseteq R(g)$, it suffices to show that $R(g)$ is closed under paramodulation. It is not hard to see that, because no clause in $f$ mentions a disjunct of the form $t = t'$, no clause in $R(g) - \{\forall x(x = x)\}$ mentions a disjunct
of the form \( t = t' \). Therefore, applying paramodulation does not lead to any new clauses. ■

The next four lemmas relate the closures of various formulas and give bounds on the complexity of computing the closure. In these proofs, it is convenient to associate a clause \( c \) with its set of disjuncts, which we denote as \( S(c) \). For example, if \( \ell_1, \ldots, \ell_k \) are literals, then \( S(\ell_1 \lor \cdots \lor \ell_k) = \{\ell_1, \ldots, \ell_k\} \). For the next four lemmas, let \( S = \{s \neq s \mid s \text{ is a term}\} \).

**Lemma A.0.4.** Let \( c \) be a clause with no bipolar literals and let \( f \) be a conjunction of ground literals. If a clause \( c' \) is in \( R(c \land f \land \forall x(x = x)) \), then \( c' \) is in \( R(f \land \forall x(x = x)) \) or \( S(c') \subseteq S(c\sigma) \subseteq S(c') \cup S(\neg f) \cup S \) for some substitution \( \sigma \).

**Proof:** Let \( R'(c \land f) \) consist of the clauses in \( R(f \land \forall x(x = x)) \) and all clauses \( c' \) such that, for some substitution \( \sigma \), \( S(c') \subseteq S(c\sigma) \subseteq S(c') \cup S(\neg f) \cup S \). We want to show that \( R(c \land f \land \forall x(x = x)) \subseteq R'(c \land f \land \forall x(x = x)) \). Because every conjunct of \( c \land f \land \forall x(x = x) \) is in \( R'(c \land f \land \forall x(x = x)) \), it suffices to show that \( R'(c \land f \land \forall x(x = x)) \) is closed under resolution. To do this, suppose that \( c_1 \) and \( c_2 \) are clauses in \( R'(c \land f \land \forall x(x = x)) \) that resolve on a literal \( \ell \) to create the resolvent \( c_3 \). We want to show that \( c_3 \in R'(c \land f \land \forall x(x = x)) \).

If both \( c_1 \) and \( c_2 \) are in \( R(f \land \forall x(x = x)) \), then \( c_3 \) is in \( R(f \land \forall x(x = x)) \), so \( c_3 \in R'(c \land f \land \forall x(x = x)) \). If exactly one of the clauses is in \( R(f \land \forall x(x = x)) \), then assume without loss of generality that it is \( c_1 \). Because \( f \land \forall x(x = x) \) is a conjunction of literals, every clause in \( R(f \land \forall x(x = x)) \) is either a conjunct of \( f \land \forall x(x = x) \) or \textit{false}; \( c_1 \) is the parent of a resolvent, so it is a conjunct of \( f \land \forall x(x = x) \). Since \( c_2 \in R'(c \land f \land \forall x(x = x)) \), there is a substitution \( \sigma \) such that \( S(c_2) \subseteq S(c\sigma) \subseteq S(c_2) \cup S(\neg f) \cup S \). Since \( c_1 \) and \( c_2 \) are
the parents of the resolvent \( c_3 \) and \( c_1 \) is a conjunct of \( f \land \forall x(x = x) \), there is a substitution \( \sigma' \) such that \( c_2 \sigma' = c_3 \lor \sim c_1 \), where \( \sim c_1 \) is the negation of a conjunct of \( f \) or has the form \( s \neq s \). Because \( S(c_2) \subseteq S(c\sigma) \), it follows that \( S(c_3) \subseteq S(c\sigma\sigma') \). Moreover,

\[
S(c\sigma\sigma') \subseteq S(c_2\sigma') \cup S(\neg f\sigma') \cup S = S(c_3) \cup \{\sim c_1\} \cup S(\neg f) \cup S
\]

since \( \{(s \neq s)\sigma'|s \text{ is a term}\} \subseteq S \), \( \neg f\sigma' = \neg f \) (because \( f \) mentions no variables), and \( \sim c_1 \) is either a conjunct of \( \neg f \) or a literal in \( S \). So \( c_3 \in R'(c \land f \land \forall x(x = x)) \).

Finally, if neither \( c_1 \) nor \( c_2 \) is in \( R(f \land \forall x(x = x)) \), then it is not hard to see that there are substitutions \( \sigma \) and \( \sigma' \) such that \( \ell \) is a disjunct of \( c\sigma \) and \( \sim \ell \) is a disjunct of \( c\sigma' \), contradicting the assumption that \( c \) has no bipolar literals.

For the next three lemmas, let \( f = E_0 \land \forall x(x = x) \land \neg \text{Permitted}(t, t') \) and let \( f' = E_1 \land P \), where \( t \) and \( t' \) are closed terms.

**Lemma A.0.5.** \( R(f' \land f) = \bigcup_{c \in R(f')} R(c \land f) \).

**Proof:** Let \( c \) be a clause in \( R(f') \). Because every conjunct of \( c \land f \) is in \( R(f' \land f) \) and \( R(f' \land f) \) is closed under resolution, \( R(c \land f) \subseteq R(f' \land f) \). It follows that

\[
\bigcup_{c \in R(f')} R(c \land f) \subseteq R(f' \land f).
\]

For the opposite inclusion, observe that every conjunct of \( R(f' \land f) \) is in \( \bigcup_{c \in R(f')} R(c \land f) \). So, it suffices to show that \( \bigcup_{c \in R(f')} R(c \land f) \) is closed under resolution. To do this, suppose that \( c_1, c_2 \in R(f') \) and that \( e \) is a resolvent with parents \( d_1 \in R(c_1 \land f) \) and \( d_2 \in R(c_2 \land f) \). It suffices to show that \( e \in \bigcup_{c \in R(f')} R(c \land f) \).

If \( d_1 \in R(f) \), then clearly \( d_1 \in R(c_2 \land f) \), so \( e \in R(c_2 \land f) \) and we are done.
Similarly, if \( d_2 \in R(f) \), then \( e \in R(c_1 \land f) \). Suppose that neither \( d_1 \) nor \( d_2 \) is in \( R(f) \). Then it follows from Lemma A.0.4 that there are substitutions \( \sigma_1 \) and \( \sigma_2 \) such that \( c_1 \sigma_1 = d_1 \lor d'_1 \) and \( c_2 \sigma_2 = d_2 \lor d'_2 \), where \( S(d'_1) \subseteq S(\neg f) \cup S \) and \( S(d'_2) \subseteq S(\neg f) \cup S \). Because \( d_1 \) and \( d_2 \) are the parents of \( e \), there are substitutions \( \sigma'_1 \) and \( \sigma'_2 \), clauses \( d''_1 \) and \( d''_2 \), and a literal \( \ell \) such that \( e = d''_1 \lor d''_2 \), \( d_1 \sigma'_1 = d''_1 \lor \ell \), and \( d_2 \sigma'_2 = d''_2 \lor \neg \ell \). Putting the pieces together,

\[
c_1 \sigma_1 \sigma'_1 = d'_1 \lor \ell \lor d'_1 \sigma'_1 \quad \text{and} \quad c_2 \sigma_2 \sigma'_2 = d'_2 \lor \neg \ell \lor d'_2 \sigma'_2.
\]

(Note that \( S(d'_1) \subseteq S(\neg f) \cup S \) and \( S(d'_2) \subseteq S(\neg f) \cup S \), because the only variables that appear in \( d'_1 \) or \( d'_2 \) are in disjuncts of the form \( t \neq t' \).) Clearly \( c_1 \) and \( c_2 \) resolve to create a resolvent \( e' \in R(f') \). Moreover, \( e' \sigma_1 \sigma_2 = e \lor d'_1 \sigma'_1 \lor d'_2 \sigma'_2 \), so \( e \in R(e' \land f) \). 

**Lemma A.0.6.** If every clause in \( f' \) has at most one literal that is bipolar in \( f' \), then \( R(f') \) has \( O(|f'|^2) \) clauses, each of length at most \( 2L_f L'_{f'} \), and \( R(f') \) can computed in time \( O(|f'|^2) \).

**Proof:** Note that the resolvent \( e \) of two clauses in \( f' \) has no bipolars, because every clause in \( f' \) has at most one bipolar. It follows that \( e \) is not a parent of a resolvent in \( R(f') \). So,

\[
R(f') = \{ c \mid c \text{ is in } S(f') \text{ or is the resolvent of two clauses in } S(f') \}.
\]

Thus, \( R(f') \) has \( O(|f'|^2) \) clauses and each clause has length less than \( 2L_f L'_{f'} \). To find \( R(f') \), we simply check each pair of clauses \( c \) and \( c' \) in \( f' \) to see if there is a literal on which they resolve; if so, we resolve them. The check can be done in time \( O(|c| |c'|) \); the resolution can be done in time \( O(|c| + |c'|) \). Since, by assumption, each clause contains at most one instance of a bipolar literal, there will be at most
one resolvent for each pair of clauses. It easily follows that \( R(f') \) can be computed in time \( O(|f'|^2) \).

If \( C \) is a set of clauses, let \( \|C\| = \sum_{c \in C} |c| \). For all predicate symbols \( Q \), a variable \( v \) is \( Q\text{-constrained} \) in a clause \( c \) if \( v \) appears as an argument to \( Q \) in \( c \). Note that a constrained variable, as defined in Section 2.3.1, is \textbf{Permitted}-constrained.

\textbf{Lemma A.0.7.} Suppose that \( f \) mentions \( m \) terms and \( C \) is a non-empty set of clauses such that, for every \( c \in C \), no literal in \( c \) is bipolar in \( c \). Then

\( (a) \ false \in \bigcup_{c \in C} R(c \land f) \) iff (i) \( false \in R(f) \) or (ii) there is a clause \( c \in C \) and a substitution \( \sigma \) such that \( S(c \sigma) \subseteq S(\neg f) \cup S \);

\( (b) \) we can determine whether (i) holds in time \( O(|E_0| \log |E_0|) \);

\( (c) \) we can determine whether (ii) holds in time \( O((R\|C\||\textbf{Permitted}(t, t')| + |E_0|) \log |E_0|)) \), where \( R = m \) if every literal in every clause \( c \) in \( C \) mentions at most one variable that is not constrained in \( c \); otherwise \( R = m^k \), where every clause \( c \) in \( C \) has at most \( k \) variables that are not constrained in \( c \).

\textbf{Proof:} For part (a), the “only if” direction follows immediately from Lemma A.0.4. For the “if” direction, it is easy to see that \( R(f) \subseteq R(c \land f) \) for every clause \( c \in C \). So, if \( false \in R(f) \), then \( false \in \bigcup_{c \in C} R(c \land f) \). Also, if there is a clause \( c \in C \) and a substitution \( \sigma \) such that \( S(c \sigma) \subseteq S(\neg f) \cup S \), then it readily follows from the definition of resolution that \( false \in R(c \land f) \), so \( false \in \bigcup_{c \in C} R(c \land f) \).

For part (b), because \( f \) is a conjunction of literals, it is easy to see that \( false \in R(f) \) iff (1) \( E_0 \) includes a literal of the form \( t \neq t \) or (2) \( E_0 \) includes a literal and its negation. Clearly, we can check whether (1) holds in time \( O(|E_0|) \). To check whether (2) holds, we use a splay tree [ST83], a form of binary search tree
for which, starting with an empty tree, $K$ insertions and $S$ searches take time $O((K + S) \log K)$. Specifically, we insert every negative literal in $E_0$ into the empty splay tree $T$. Then, for every positive literal $\ell$ in $E_0$, we search $T$ for $\neg \ell$. Since at most $|E_0|$ insertion and $|E_0|$ search operations are involved, time $O(|E_0| \log |E_0|)$ is required.

For part (c), recall that $f = E_0 \land \forall x(x = x) \land \neg \text{Permitted}(t, t')$. For any clause $c$, let $c_E$ and $c_P$ be clauses such that $c = c_E \lor c_P$, $c_E$ is $\text{Permitted}$-free, and every disjunct in $c_P$ mentions $\text{Permitted}$. Because $E_0$ is $\text{Permitted}$-free, $S(c\sigma) \subseteq S(\neg f) \cup S$ iff $S(c_E\sigma) \subseteq S(\neg E_0) \cup S$ and $S(c_P\sigma) \subseteq \{\text{Permitted}(t, t')\}$. It follows that we can find a substitution $\sigma$ such that $S(c\sigma) \subseteq S(\neg f) \cup S$, if one exists, by finding substitutions $\sigma'$ and $\sigma''$ such that $S(c_P\sigma') \subseteq \{\text{Permitted}(t, t')\}$ and $S(c_E\sigma'\sigma'') \subseteq S(\neg E_0) \cup S$, and taking $\sigma = \sigma' \circ \sigma''$. We can assume without loss of generality that $\sigma(x) = x$ for every variable $x$ that does not appear in $c$. Thus, we can clearly check if an appropriate substitution $\sigma$ exists in time $O(|c||\text{Permitted}(t, t')|)$, by pattern-matching each occurrence of $\text{Permitted}$ in $c_P$ with $\text{Permitted}(t, t')$. Moreover, if $\sigma$ exists, then $|c\sigma| \leq |c||\text{Permitted}(t, t')|$, since $\sigma$ substitutes terms in $\text{Permitted}(t, t')$ for variables in $c_P$.

Let

$$D = \{d : \text{there is a clause } c \in C \text{ and a substitution } \sigma \text{ such that } \sigma(x) = x \text{ if } x \text{ does not appear in } c, S(c_P\sigma) \subseteq \{\text{Permitted}(t, t')\}, \text{ and } d = c_E\sigma\}.$$  

We can clearly construct $D$ in time $O(||C|||\text{Permitted}(t, t')|)$, by considering the clauses in $C$ one at a time, and $||D|| < ||C|||\text{Permitted}(t, t')|$). Thus, to complete the proof of part (c), it suffices to show that we can determine whether there is a $d \in D$ and a substitution $\sigma$ such that $S(d\sigma) \subseteq S(\neg E_0) \cup S$ in time $O(||E_0| + R||D||) \log |E_0|)$, where $R$ is as defined in the lemma. We can do this by a brute-
force search. In more detail, we insert every literal in $E_0$ into an empty splay tree $T$; then, for each clause $d \in D$ and each possible assignment $\sigma$ of terms in $E_0$ to variables in $d$, we check whether every literal in $d\sigma$ is the negation of a literal in $T$ or is of the form $t \neq t$. Suppose every clause $c$ in $C$ has at most $k$ variables that are not constrained in $c$. Then $d$ has at most $k$ variables. Since $E_0$ mentions at most $m$ terms, it follows that there are at most $m^k$ ways of assigning terms in $E_0$ to variables in $d$. As we have observed, the $O(|E_0|)$ insertions and $O(m^k|d|)$ searches can be done in time $O((|E_0| + m^k|d|) \log |E_0|)$. For each literal $\ell$ in $S(d\sigma) - S(\neg E)$, we can determine whether $\ell$ is of the form $t = t$ in time $O(|\ell|)$. Thus, the time needed to check every clause $d \in D$ is $O((|E_0| + m^k|D|) \log |E_0|)$.

We may be able to do better if every literal in every clause $c$ in $C$ has at most one variable that is not constrained in $c$. In this case, every literal in every clause $d$ in $D$ has at most one variable. It follows that, given a clause $d \in D$, we can partition the literals in $d$ into sets according to their variable. That is, in time $O(|d|)$, we can write $d$ as $d_1 \lor \ldots \lor d_k$, where two literals $\ell$ and $\ell'$ mention the same variable iff $\ell$ and $\ell'$ both appear in $d_i$ for $i = 1, \ldots, k$. Clearly, there is a substitution $\sigma$ such that $S(d\sigma) \subseteq S(\neg E_0) \cup S$ iff there are substitutions $\sigma_1, \ldots, \sigma_k$ such that $S(d_i\sigma_i) \subseteq S(\neg E_0) \cup S$, for $i = 1, \ldots, k$. For a particular $d_i$, there are at most $m$ possible substitutions of terms in $E_0$ to the variable in $d_i$. So, given a splay tree $T$ whose entries are the conjuncts in $E_0$, we can determine if there is an appropriate $\sigma_i$ in time $O(m|d_i| \log |E_0|)$. Thus, given $T$, we can determine if there are appropriate substitutions $\sigma_1, \ldots, \sigma_k$ in time $O(m|d| \log |E_0|)$. Since we can construct $T$ in time $O(|E_0| \log |E_0|)$, the total time needed for a particular clause $d \in D$ is $O((|E_0| + m|d|) \log |E_0|)$, and the time needed to check every $d \in D$ is $O((|E_0| + m|D|) \log |E_0|)$. \qed
Proposition 2.3.2: Suppose that \( E \land P \Rightarrow \text{Permitted}(t, t') \) is a standard query in which \( E \) is basic, the equality symbol is not mentioned in \( E \land P \), and there are no bipolars in \( P \). Then \( E \land P \Rightarrow \text{Permitted}(t, t') \) is valid iff there is a conjunct \( p \) of \( P \) such that \( E \land p \Rightarrow \text{Permitted}(t, t') \) is valid.

Proof: Here and elsewhere, let \( E^+ \) be an abbreviation for \( \forall x(x = x) \land E \). By Corollary A.0.3, it suffices to show that \( R(E^+ \land P \land \neg \text{Permitted}(t, t')) \) includes false iff \( R(E^+ \land p \land \neg \text{Permitted}(t, t')) \) includes false for some conjunct \( p \) of \( P \).

It follows from Lemma A.0.5, where we take \( f \) to be \( E^+ \land \neg \text{Permitted}(t, t') \) and \( f' \) to be \( P \), that \( R(E^+ \land P \land \neg \text{Permitted}(t, t')) \) includes false iff \( R(E^+ \land c \land \neg \text{Permitted}(t, t')) \) includes false for some \( c \) in \( R(P) \). Since there are no bipolar literals in \( P \), \( R(P) \) is just the set of conjuncts in \( P \), so we are done. 

Theorem 2.3.1: Let \( \mathcal{L}_5 \) consist of all standard queries of the form \( E \land P \Rightarrow \text{Permitted}(t, t') \) such that

(1) \( E \) is basic (i.e., \( E \) is a conjunction of ground literals),

(2) there are no bipolar literals in \( P \),

(3) equality is not mentioned in \( E \land P \), and

(4) every variable appearing in a conjunct \( p \) of \( P \) is constrained in \( p \).

We can determine the validity of formulas in \( \mathcal{L}_5 \) in time \( O((|P||\text{Permitted}(t, t')| + |E|) \log |E|) \), where \(|\varphi| \) denotes the length of \( \varphi \), when viewed as a string of symbols.

Proof: Let \( S_p \) be the set of conjuncts of \( P \). By Proposition 2.3.2, \( E \land P \Rightarrow \text{Permitted}(t, t') \) is valid iff \( E \land p \Rightarrow \text{Permitted}(t, t') \) is valid, for some conjunct \( p \) of \( P \). By Corollary A.0.3, the latter statement holds iff \( false \in \bigcup_{p \in S_p} R(E^+ \land p \land \neg \text{Permitted}(t, t')) \). It follows from Lemma A.0.7(a), where we take \( C = S_p \) and
\[ f = E^+ \land \neg \text{Permitted}(t, t'), \text{ that } false \in \bigcup_{p \in S_p} R(E^+ \land p \land \neg \text{Permitted}(t, t')) \]

iff (a) \( false \) is in \( R(E^+ \land \neg \text{Permitted}(t, t')) \) or (b) there is a clause \( p \in S_p \) and a substitution \( \sigma \) such that \( \mathcal{S}(p\sigma) \subseteq \mathcal{S}(\neg E^+ \lor \text{Permitted}(t, t')) \cup \{ s \neq s \mid s \text{ is a term} \} \). By Lemma A.0.7(b), we can determine whether (a) holds in time \( O(|E| \log |E|) \). It follows from Lemma A.0.7(c), where \( f = E^+ \land \neg \text{Permitted}(t, t') \), \( C = S_p \), and \( k = 0 \), that we can determine whether (b) holds in time \( O((|E| + |P||\text{Permitted}(t, t')|) \log |E|) \).

Rather than just proving Theorem 2.3.5, we prove a slightly stronger result, from which we will also be able to prove Theorem 2.4.1. Note that part (b) of the following theorem is equivalent to Theorem 2.3.5.

**Theorem A.0.8.** Suppose that \( E \) is a standard environment, \( P \) is a conjunction of pure permitting policies, and \( D \) is a conjunction of (not necessarily pure) denying policies such that, for every resolvent \( f \) created by resolving a conjunct of \( P \) and a conjunct of \( D \) on a literal that mentions \text{Permitted}, either \( E \Rightarrow f \) is valid or \( q \Rightarrow f \) is valid for some conjunct \( q \) of \( P \land D \). Then

(a) \( E \land P \) is consistent iff \( E \land P \land D \) is consistent

(b) \( E \land P \land \neg \text{Permitted}(t, t') \) is consistent iff \( E \land P \land D \land \neg \text{Permitted}(t, t') \) is consistent, where \( t \) and \( t' \) are terms of the appropriate sort.

**Proof:** We prove part (a) here; the proof of part (b) is identical.

Suppose that the hypotheses of the theorem hold. Since \( g = E \land \bigwedge_{p \in P} p \) is consistent, it has a Herbrand model, that is, a model whose domain consists of all the variable-free terms in the language. Of the Herbrand models for \( g \), let \( m \) be a \textit{minimally permissive} one, that is, one for which the extension of the \text{Permitted} predicate is minimal. We claim that in fact \( m \models E \land \bigwedge_{p \in P} p \land \bigwedge_{d \in D} d \). For suppose
not. Then there is a denying policy \( \forall x_1 \ldots \forall x_n d \) in \( D \) and a variable substitution \( \sigma_d \) such that:

1. \( m \models \neg d \sigma_d \) and

2. for all denying policies \( \forall x_1 \ldots \forall x_n e \in D \) and variable substitutions \( \sigma_e \) such that \( m \models \neg e \sigma_e \), the number of negative literals in \( e \sigma_e \) mentioning \textit{Permitted} is at least the number of negative literals in \( d \sigma_d \) mentioning \textit{Permitted}.

(Note that we are assuming all policies are in CNF, so that the number of negative literals in a policy is well-defined.)

Since \( d \) is a denying policy, \( d \sigma_d \) has at least one negated \textit{Permitted} formula among its clauses; that is \( d \sigma_d = d' \lor \neg \text{Permitted}(s_d, s'_d) \) for some terms \( s_d \) and \( s'_d \). Since \( m \models \neg d \sigma_d \), we have that \( m \models \neg d' \land \text{Permitted}(s_d, s'_d) \). Since \( m \) is minimally permissive, there must be a pure permitting policy \( \forall x_1 \ldots \forall x_n p \in P \) and a variable substitution \( \sigma_p \) such that \( p \sigma_p = p' \lor \text{Permitted}(s_d, s'_d) \) and \( m \models \neg p' \).

(Otherwise, consider the model obtained by removing \((s_d, s'_d)\) from the extension of \textit{Permitted}; it must also satisfy \( g \), and is less permissive than \( m \).) Let \( f = p' \lor d' \) be the formula created by resolving \( p \sigma_p \) and \( d \sigma_d \) on \textit{Permitted}(\(s_d, s'_d\)). Note that, by choice of \( p' \) and \( d' \), \( m \models \neg f \). It follows from the definition of resolution and the fact that \( p \) is a \textit{pure} permitting policy that the number of negative literals in \( f \) that mention \textit{Permitted} is less than the number of such literals in \( d \sigma_d \). Moreover, by hypothesis, either \( E \Rightarrow f \) is valid or \( q \Rightarrow f \) is valid for some \( q \in P \cup D \). Since \( m \models E \land \neg f \), \( E \Rightarrow f \) is not valid, so \( q \Rightarrow f \) is valid for some \( q \in P \cup D \). Since \( m \models \neg f, m \models \neg q \); and, since \( m \models \bigwedge_{p \in P} p \) by assumption, \( q \in D \). Therefore, there is a denying policy \( \forall x_1 \ldots \forall x_n e \in D \) such that \( \forall x_1 \ldots \forall x_n e \Rightarrow f \) is valid. It is not hard to show that \( \forall x_1 \ldots \forall x_n e \Rightarrow f \) is valid iff there is a variable substitution \( \sigma_e \).
such that $e\sigma_e = f$. Thus, $\forall x_1 \ldots \forall x_n e$ is a denying policy in $D$ and $\sigma_e$ is a variable substitution such that $m \models \neg e\sigma_e$ (because $m \models \neg f$). The number of negative literals in $e\sigma_e$ that mention $\text{Permitted}$ (which is the number of negative literals in $f$ that mention $\text{Permitted}$) is less than the number of such literals in $d\sigma_d$. Thus, we have a contradiction.

**Proposition 2.3.6:** If $q$ is an equality-safe standard query, then there is a standard query $q'$ of the form $E'_0 \land E'_1 \land P' \Rightarrow \text{Permitted}(t, t')$ such that (a) $q$ is valid iff $q'$ is valid, (b) $q'$ is equation-free, and (c) $|q'| = O(|q| |L_q|)$, where $L_q'$ is the length of the longest term in $q$. Moreover, we can find such a $q'$ in time $O(|q|)$.

**Proof:** Suppose that $q$ has the form $F_0 \land F_1 \land E_1 \land P \Rightarrow \text{Permitted}(s, s')$, where $F_0$ is the conjunction of the equality statements, while $F_1$ consists of the remaining conjuncts in $E_0$. To create $q'$, we partition the set of terms in $E_0$ into equivalence classes; terms $t_e$ and $t'_e$ are in the same class if the equality formulas in $E_0$ imply $t_e = t'_e$. The equivalence classes can be found in linear time.\(^1\) Since $q$ is equality-safe, each equivalence class has at most one term that is not a constant. If an equivalence class has a term that is not a constant, then we choose that term to represent the class; otherwise, we select a representative arbitrarily. Let $q' = F'_1 \land E'_1 \land P' \Rightarrow \text{Permitted}(t, t')$, where $F'_1$, $E'_1$, $P'$, $t$, and $t'$ are the result of replacing each closed term in $F_1$, $E_1$, $P$, $s$, and $s'$, respectively, that also appears in $F_0$ by its representative. It suffices to show that $q$ is valid if and only if $q'$ is valid, because the other statements in the conclusion of Proposition 2.3.6 follow immediately from

\(^1\)In general, the problem of constructing equivalence classes is harder than linear time. For example, if $c_1 = c_2$, then $f(c_1) = f(c_2)$. However, we do not have to worry about drawing such inferences—if $E_0 \Rightarrow (c_1 = c_2)$ is valid, then it cannot be the case that $f(c_1)$ and $f(c_2)$ are both terms in $E_0$, for then $E_0 \Rightarrow (f(c_1) = f(c_2))$ is valid, and $q$ would not be equality-safe.
the construction of \( q' \). Let \( q'' = F_0 \land F_1' \land E_1' \land P' \Rightarrow \text{Permitted}(t, t') \). It is easy to see that \( q \) is equivalent to \( q'' \); substituting a term by its representative in the equivalence class is justified in the presence of \( F_0 \). Thus, it suffices to show that \( q'' \) is valid iff \( q' \) is valid. The “if” direction is trivial. For the “only if” direction, suppose by way of contradiction that \( q'' \) is valid and \( q' \) is not. It follows that there is a model \( m \) with interpretation \( I \) that does not satisfy \( q' \). Let \( m' \) be a model that is identical to \( m \) except that \( m' \) interprets a constant \( r \) as \( I(r') \) if \( r \) and \( r' \) are in the same equivalence class and \( r' \) is the class representative. Clearly, \( m' \) satisfies \( F_0 \). Moreover, because \( m \) does not satisfy \( q' \) and the only difference between \( m \) and \( m' \) is the interpretation of constants that are not mentioned in \( q' \), \( m' \) does not satisfy \( q' \). This contradicts the validity of \( q' \). \[ \blacksquare \]

The following example illustrates the procedure for creating \( q' \) from \( q \).

**Example A.0.9.** Consider the query “may Bob nap”, given that Alice is Bob’s wife, Alice may nap, and any individual may nap if his wife may nap. We can write the query as \( q = e \land p_1 \land p_2 \Rightarrow \text{Permitted}(\text{Bob, nap}) \), where

\[
e = (\text{Alice} = \text{wifeOf}(\text{Bob})),
\]
\[
p_1 = \text{Permitted}(\text{Alice, nap}), \quad \text{and}
\]
\[
p_2 = \forall x (\text{Permitted}(\text{wifeOf}(x), \text{nap}) \Rightarrow \text{Permitted}(x, \text{nap})).
\]

The query \( q' \) is the result of removing the conjunct \( e \) from \( q \) and replacing every occurrence of \( \text{Alice} \) by \( \text{wifeOf}(\text{Bob}) \). Thus, \( q' = p_1' \land p_2' \Rightarrow \text{Permitted}(\text{Bob, nap}) \), where

\[
p_1' = \text{Permitted}(\text{wifeOf}(\text{Bob}), \text{nap}) \quad \text{and}
\]
\[
p_2' = \forall x (\text{Permitted}(\text{wifeOf}(x), \text{nap}) \Rightarrow \text{Permitted}(x, \text{nap})).
\]

Note that we replace \( \text{Alice} \) by \( \text{wifeOf}(\text{Bob}) \) because the two terms are in the same equivalence class and, since \( \text{wifeOf}(\text{Bob}) \) mentions a function symbol, it is
the class representative. Also note that if we replace \( \text{wifeOf}(\text{Bob}) \) by \( \text{Alice} \), then the resulting query is not valid, even though \( q \) is. In general, we do not preserve validity if we replace a term that includes a function symbol. That is why we restrict to equality-safe queries in Proposition 2.3.6.

**Theorem 2.3.7:** The validity of an equation-free Lithium query \( q = E_0 \land E_1 \land P \Rightarrow \text{Permitted}(t, t') \) with \( m \) terms in \( E_0 \) can be determined in time \( O((|E_0| + T|E_1 \land P|^2) \log |E_0|) \), where \( T = mL_{E_1 \land P}L_{E_1 \land P}'|\text{Permitted}(t, t')| \) if every literal in every conjunct \( c \) of \( E_1 \land P \) mentions at most one variable that is not constrained in \( c \) relative to \( q \); otherwise, \( T = m^{2k}L_{E_1 \land P}L_{E_1 \land P}'|\text{Permitted}(t, t')| \), where every conjunct \( c \) of \( E_1 \land P \) has at most \( k \) variables that are not constrained in \( c \) relative to \( q \).

**Proof:** By Corollary A.0.3, the query \( q \) is valid iff the set \( R(E_0 \land \forall x(x = x) \land E_1 \land P \land \neg \text{Permitted}(t, t')) \) includes \( \text{false} \). Let \( E_0^+ \) be \( E_0 \land \forall x(x = x) \) and let \( q^+ \) be the result of replacing \( E_0 \) in the antecedent of \( q \) by \( E_0^+ \). By Lemma A.0.5, \( \text{false} \in R(\neg q^+) \) iff there is a clause \( c \in R(E_1 \land P) \) such that \( \text{false} \in R(c \land E_0^+ \land \neg \text{Permitted}(t, t')) \). By Lemma A.0.7(a), the latter statement holds iff (1) \( \text{false} \in R(E_0^+ \land \neg \text{Permitted}(t, t')) \) or (2) there is a clause \( c \in R(E_1 \land P) \) and a substitution \( \sigma \) such that \( S(c\sigma) \subseteq S(\neg E_0^+ \lor \text{Permitted}(t, t')) \cup \{s \neq s \mid s \text{ is a term}\} \). By Lemma A.0.7(b), we can check whether (1) holds in time \( O(|E_0| \log |E_0|) \). To determine whether (2) holds, we first note that, by Lemma A.0.6, we can compute \( R(E_1 \land P) \) in time \( O(|E_1 \land P|^2) \). Once we have \( R(E_1 \land P) \), it follows from Lemma A.0.7(c), where we take \( C = R(E_1 \land P) \), that we can determine whether (2) holds in time \( O((|E_0| + m^{k'}|R(E_1 \land P)||\text{Permitted}(t, t')|) \log |E_0|) \) if every clause \( c \in R(E_1 \land P) \) has at most \( k' \) variables that are not constrained in \( c \). It follows from Lemma A.0.6 that \( |R(E_1 \land P)| \) is \( O(|E_1 \land P|^2L_{E_1 \land P}L_{E_1 \land P}') \); it follows from
the way resolution is defined that $k' \leq 2k$. So, we can determine whether (2) holds in time $O((|E_0| + m^{2k}|E_1 \land P|^2L_{E_1 \land P}L'_{E_1 \land P}|\text{Permitted}(t, t'))| \log |E_0|).

Suppose that every literal in every conjunct $c$ of $E_1 \land P$ mentions at most one variable that is not constrained in $c$ relative to $q$. Then it follows from the definition of resolution that every literal $\ell$ in every clause $c$ in $R(E_1 \land P)$ mentions at most one variable that is not constrained in $c$. It follows from Lemma A.0.7(c), where we again take $C = R(E_1 \land P)$, that we can determine whether (2) holds in this case in time $O((|E_0| + m|R(E_1 \land P)||\text{Permitted}(t, t'))| \log |E_0|)$, once we have computed $R(E_1 \land P)$. By Lemma A.0.6, we can compute $R(E_1 \land P)$ in time $O(|E_1 \land P|^2)$ and $|R(E_1 \land P)|$ is $O(|E_1 \land P|^2L_{E_1 \land P}L'_{E_1 \land P})$. So, the total time needed to determine whether (2) holds is $O((|E_0| + m|E_1 \land P|^2L_{E_1 \land P}L'_{E_1 \land P}|\text{Permitted}(t, t'))| \log |E_0|)$.

Theorem 2.4.1: Suppose that $E$ is an environment, $P$ is a conjunction of pure permitting policies, and $D$ is a conjunction of (not necessarily pure) denying policies such that the antecedent of Theorem 2.3.5 holds. Then $E \land P \land D$ is satisfiable iff $E$ is satisfiable.

Proof: If $E$ is satisfiable, then $E \land P$ satisfiable. (For any model $m$ that satisfies $E$ there is a model $m'$ that is identical to $m$, except $m'$ satisfies $\text{Permitted}(s, s')$ for all terms $s$ and $s'$ of the appropriate sort; $m'$ satisfies $E \land P$.) The result is now immediate from Theorem A.0.8(a).
Appendix B

Proofs for Chapter 3

Proposition 3.2.7: For all closed conditions $d$ and sets $L$ of licenses,

(a) every execution of $\text{Holds}(d, L)$ that terminates returns the same output,

(b) every execution of $\text{Holds2}(d, L, \emptyset)$ that terminates returns the same output,

(c) if an execution of $\text{Holds}(d, L)$ terminates by returning the truth value $t$, then an execution of $\text{Holds2}(d, L, \emptyset)$ terminates by returning $t$.

Proof:

Parts (a) and (b) are immediate from the description of the $\text{Holds}$ and $\text{Holds2}$. To prove part (c), say that a call tree for $\text{Holds}(d, L)$ is non-repeating if it is not the case that there exists a path $p$ in the call tree and two nodes $n_1$ and $n_2$ on the path such that both nodes are labeled by the same call to $\text{Holds}$. If $\text{Holds}(d, L)$ terminates, then it has a finite call tree. Moreover, it is easy to see that if there is a finite call tree for $\text{Holds}(d, L)$, then there is a nonrepeating call tree: If there is a call to $\text{Holds}(d', L')$ at two nodes on a path, we simply replace the subtree below the first call to $\text{Holds}(d', L')$ by the subtree below the last call to $\text{Holds}(d', L')$. A non-repeating call tree for $\text{Holds}(d, L)$ is essentially a call tree for $\text{Holds2}(d, L, \emptyset)$; the same calls are made at every step (the third component has to change appropriately).

For the proofs of Proposition 3.2.8 and Lemma B.0.20, we rely on the observation that, if $T$ is the call tree for an execution of $\text{Holds2}(d, L, S)$, then $T$ can be viewed as an and-or tree, where a node labeled $\text{Holds2}(d', L, S')$ is an and node if...
$d'$ is a conjunction with at least two conjuncts, an or node if $d'$ is a Said condition and $\text{Holds2}(d', L, S')$ makes at least one recursive call, and a leaf if $d'$ is true or if $d'$ is a Said condition and $\text{Holds2}(d', L, S')$ makes no recursive calls. For future reference, note that each node in $T$ can be assigned a truth value in an obvious way. An and node is assigned “true” if all its children are; an or node is assigned “true” if at least one child is; a leaf labeled $\text{Holds2}(\text{true}, L, S')$ is assigned “true”; and a leaf labeled $\text{Holds2}(\text{Said}(p, e), L, S')$ is assigned “false”.

**Proposition 3.2.8:** If $d$ is a closed condition, $L$ is a hierarchical set of licenses, $S$ is a set of closed Said conditions, and $T$ is the call tree of an execution of $\text{Holds2}(d, L, S)$, then the height of $T$ is at most $2\#(L) + 1$.

**Proof:** Because $L$ is hierarchical, there exists a strict partial order $\prec$ on licenses such that, if $\ell$ and $\ell'$ are licenses in $L$ and $\ell$ affects $\ell'$, then $\ell \prec \ell'$. A node $v$ in $T$ is a non-and node if $v$ is an or node or a leaf. It follows from the description of $\text{Holds2}$ that every and node has at least two children and every child of an and node is a non-and node. So, if a path in $T$ from the root to a leaf has $n$ non-and nodes, then that path has at most $2n$ total nodes; thus, it suffices to show that every path in $T$ has at most $\#(L) + 1$ non-and nodes. If $L = \emptyset$, then it is immediate from the description of $\text{Holds2}$ that $T$ has height at most 1. Suppose that $L \neq \emptyset$. Then, for every path $t$ in $T$, either $t$ includes at most 2 non-and nodes, in which case $t$ mentions at most $\#(L) + 1$ non-and nodes, or $t$ includes 2 non-and nodes $v_i$ and $v_j$ such that an or node precedes $v_i$, which precedes $v_j$, and no or node is between $v_i$ and $v_j$. If $v_i$ has a label of the form $\text{Holds2}(d_i, L, S_i)$ and $v_j$ has a label of the form $\text{Holds2}(d_j, L, S_j)$, then it follows from the description of $\text{Holds2}$ that there are licenses $(p_i, g_i)$ and $(p_j, g_j)$ in $L$ and closed substitutions $\sigma_i$ and $\sigma_j$ such that the antecedent of $g_i$ under $\sigma_i$ mentions
the antecedent of $g_j$ under $\sigma_j$ mentions $d_j$; and $(p_j, g_j)$ affects $(p_i, g_i)$. Thus, $(p_j, g_j) \prec (p_i, g_i)$. It follows that $t$ has at most $\#(L) + 1$ non- and nodes.

**Definition B.0.10.** Let $(e, L, R, E)$ be a query, let $X$ be some execution of Query2$(e, L, R, E)$, and let $A = A(e, L, R, E, X)$.

\[
E^*(e, L, R) = \{ \text{Permitted}(p, \text{issue}, g) \mid (p, g) \in L \} \cup \{ e \}.
\]

\[
S^*(e, L, R, E, X) = \{ \text{Said}(p, \text{Pr}(p')) \mid p, p' \in P \text{ and } \text{Pr} \in \text{primitiveProp} \} \cup \{ \text{Said}(p, \text{Permitted}(p', \text{issue}, g)) \mid p, p' \in P \text{ and } g \in A \}.
\]

\[\]

**Theorem 3.3.1:** For all strings $s$ in the language and all finite sets $L$ of licenses, $A$ of closed resources, $S$ of closed Said conditions, and $E$ of closed conclusions, $s^{L,A,S,E}$ is well defined.

**Proof:** Let $S_L$ be the set of Said conditions that are mentioned in issued grants; that is, $\text{Said}(p, e) \in S_L$ iff there is a license $(p', g) \in L$ such that $g$ mentions $\text{Said}(p, e)$. Let $S_s$ be the set of Said conditions mentioned in $s$. Finally, let $S_{L,s} = S_L \cup S_s$. We define a lexicographic order on the tuples $(s, S)$ such that $(s, S) < (s', S')$ iff either (a) $\#(S_{L,s} - S) < \#(S_{L,s} - S')$ or (b) $\#(S_{L,s} - S) = \#(S_{L,s} - S')$ and $|s| < |s'|$. The proof is by induction on this ordering. If $\#(S_{L,s} - S) = 0$ and $|s| = 1$, then $s^{L,A,S,E} = s$, so the translation is well defined. The inductive step is trivial except when $s = \text{Said}(p, e)$ and $s \not\in S$.

Suppose that $s$ is of the form $\text{Said}(p, e)$ and $s \not\in S$. Recall that

\[
\text{Said}(p, e)^{L,A,S,E} = \text{Val}(\bigwedge_{g \in R_p} g^{L,A,S',\emptyset} \Rightarrow e^{L,A,S',\emptyset}),
\]

where $R_p = \{ g \mid (p', g) \in L \text{ for a } p' \in p \}$ and $S' = S \cup \{ \text{Said}(p, e) \}$. Because $L$ is a finite set, $R_p$ is a finite set and because $e$ is a conclusion, $e^{L,A,S',\emptyset}$ is well
defined. So, to prove that $\text{Said}(p, e)^{p, A, S, E}$ is well defined, it suffices to show that $g^{L, A, S', \emptyset}$ is well defined for all $g \in R_p$. Suppose that $s \not\in S_L$. Then $\#(S_L, g) - S' = \#(S_L - S')$ since $S_L, g = S_L$; $\#(S_L - S') = \#(S_L - S)$ since $s \not\in S_L$; $\#(S_L - S) < \#(S_L - S \cup \{s\})$ since $s \not\in S_L$; and $\#(S_L - S \cup \{s\}) = \#(S_L, s) - S)$ since $s \not\in S$. So, putting the pieces together, $\#(S_L, g) - S'$ is well defined for all $g \in R_p$.

We next prove Theorem 3.3.2. We actually prove a stronger result, given as Theorem B.0.18; Theorem B.0.18(c) is Theorem 3.3.2. The next five lemmas and definition are used in the proof of Theorem B.0.18.

Lemma B.0.11. Suppose that $(e, L, R, E)$ is a query. Then during an execution $X$ of $\text{Query2}(e, L, R, E)$

(a) every call made to $\text{Query2}$, $\text{Auth2}$, and $\text{Holds2}$ takes $L$ as its second argument;

(b) every call made to $\text{Query2}$ and $\text{Auth2}$ takes $R$ as its third argument;

(c) if $\text{Query2}(e', L, R, E')$ is called, then $e' \in E^*(e, L, R)$;

(d) if $\text{Auth2}(e', L, R, E')$ is called, then $e' \in E^*(e, L, R)$; and

(e) if $\text{Holds3}(d, L, S)$ is called, then every conjunct of $d$ is in $S^*(e, L, R, E, X) \cup \{\text{true}\}$. 
Proof: Parts (a) through (d) follow immediately from the descriptions of \textbf{Query2}, \textbf{Auth2}, and \textbf{Holds2}. For part (e), suppose that \textbf{Holds3}(d, L, S) is called. Because \(d\) is a closed condition, every conjunct of \(d\) is either \textit{true} or of the form \textit{Said}(p, e'), where \(p\) is a closed principal and \(e'\) is a closed conclusion. If \(e'\) is of the form \textit{Pr}(p'), then \textit{Said}(p, e') is clearly in \(S^*(e, L, R, E, X)\). Otherwise, \(e'\) is of the form \textit{Permitted}(p', \textit{issue}, g). Because \(e'\) is an input to a call made during \(X\) and \(g\) is mentioned in \(e'\), \(g \in A(e, L, R, E, X)\).

**Lemma B.0.12.** Suppose that \((e, L, R, E)\) is a query such that \(e \in E\), \(A\) is a set of closed resources, and \(S\) is a set of closed \textit{Said} conditions. Then \(\bigwedge_{\ell \in L} \ell^{L,A,S,E} \land \bigwedge_{g \in R} g^{L,A,S,E} \Rightarrow e^{L,A,S,E}\) is not acceptably valid (and hence not valid).

Proof: Let \(m\) be an acceptable model that satisfies \(e^{L,A,S,E}\) iff \(e' \neq e\). Recall that, for a grant \(g = \forall x_1 \ldots \forall x_n(d_g \rightarrow e_g)\), \(g^{L,A,S,E}\) is a conjunction of formulas of the form

\[
(\bigwedge_{e \in E} \neg \text{Val}(e^{L,A,S,E} \iff (e_g \sigma)^{L,A,S,E}) \land (d_g \sigma)^{L,A,S,E}) \Rightarrow (e_g \sigma)^{L,A,S,E},
\]

where \(\sigma\) is a closed substitution. If \(e \in E\), then \(m\) satisfies \(g^{L,A,S,E}\) because, for all substitutions \(\sigma\), either \((e_g \sigma)^{L,A,S,E} \neq e^{L,A,S,E}\), in which case \(m\) satisfies \((e_g \sigma)^{L,A,S,E}\), or \((e_g \sigma)^{L,A,S,E} = e^{L,A,S,E}\), in which case \(\bigwedge_{e \in E} \neg \text{Val}(e^{L,A,S,E} \iff (e_g \sigma)^{L,A,S,E})\) is equivalent to \textit{false}. Since \(m\) satisfies every grant, \(m\) satisfies \(\bigwedge_{\ell \in L} \ell^{L,A,S,E} \land \bigwedge_{g \in R} g^{L,A,S,E}\). By construction, \(m\) does not satisfy \(e^{L,A,S,E}\), so \(m\) does not satisfy \(\bigwedge_{\ell \in L} \ell^{L,A,S,E} \land \bigwedge_{g \in R} g^{L,A,S,E} \Rightarrow e^{L,A,S,E}\). \(\blacksquare\)

**Lemma B.0.13.** Suppose that \((e, L, R, E)\) is a query, \(A\) is a set of closed resources, and \(S\) is a set of closed \textit{Said} conditions. Then (a) \(e^{L,A,S,E} = e^{L,A,S,(E \cup \{e\})}\) for every closed conclusion \(e'\) in the language, (b) \(g^{L,A,S,E} \Rightarrow g^{L,A,S,(E \cup \{e\})}\) is valid for
every grant \( g \) in the language, and (c) \( \ell^{L,A,S,E} \Rightarrow \ell^{L,A,S,(E \cup \{e\})} \) is valid for every license \( \ell \) in the language.

**Proof:** Part (a) follows immediately from the translation.

For part (b), let \( g = \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \). It is easy to see that \( g^{L,A,S,E} \Rightarrow g^{L,A,S,(E \cup \{e\})} \) is valid if, for all closed substitutions \( \sigma \), \( d_g \sigma^{L,A,S,(E \cup \{e\})} \Rightarrow d_g \sigma^{L,A,S,E} \) is valid. The latter statement holds because the translation of a condition does not depend on the final input argument (i.e., the set of conditions), so \( d_g \sigma^{L,A,S,E} = d_g \sigma^{L,A,S,E} \).

For part (c), let \( \ell = (p,h) \). If \( \text{Permitted}(p,\text{issue},h) \in E \cup \{e\} \) or \( (p,h) \not\in L \), then \( \ell^{L,A,S,(E \cup \{e\})} = \text{true} \), so the formula \( \ell^{L,A,S,E} \Rightarrow \ell^{L,A,S,(E \cup \{e\})} \) is valid. If \( \text{Permitted}(p,\text{issue},h) \not\in E \cup \{e\} \) and \( (p,h) \in L \), then

\[
\ell^{L,A,S,E} = \text{Permitted}(p,\text{issue},c_h) \Rightarrow h^{L,A,S,E} \text{ and } \\
\ell^{L,A,S,(E \cup \{e\})} = \text{Permitted}(p,\text{issue},c_h) \Rightarrow h^{L,A,S,(E \cup \{e\})}.
\]

It follows that \( \ell^{L,A,S,E} \Rightarrow \ell^{L,A,S,(E \cup \{e\})} \) is valid if \( h^{L,A,S,E} \Rightarrow h^{L,A,S,(E \cup \{e\})} \) is valid. The latter formula is valid by part (b). \( \blacksquare \)

**Definition B.0.14.** For a set \( A \) of closed resources, an \( A \)-closed substitution \( \sigma \) is a closed substitution such that, for all variables \( x \) of sort \( Rsrc \), \( \sigma(x) \in A \).

**Lemma B.0.15.** Suppose that \( G \) is a set of grants, \( L \) is a set of licenses, \( A \) is a set of closed resources, \( S \) is a set of closed Said conditions, \( E \) is a set of grants, and \( e \) is a closed conclusion. Then \( \bigwedge_{g \in G} g^{L,A,S,E} \Rightarrow e^{L,A,S,E} \) is acceptably valid iff \( g^{L,A,S,E} \Rightarrow e^{L,A,S,E} \) is acceptably valid for some \( g \in G \). Moreover, for any grant \( g \), \( g^{L,A,S,E} \Rightarrow e^{L,A,S,E} \) is acceptably valid iff \( e \not\in E \) and, for some \( A \)-closed substitution \( \sigma \), the formula \( d_g \sigma^{L,A,S,E} \) is acceptably valid and \( e_g \sigma = e \).
Proof: We first show that $\bigwedge_{g \in G} g^{L,A,S,E} \Rightarrow e^{L,A,S,E}$ is acceptably valid iff $g^{L,A,S,E} \Rightarrow e^{L,A,S,E}$ is acceptably valid for some $g \in G$. The “if” direction is trivial. For the “only if” direction, suppose by way of contradiction that $\bigwedge_{g \in G} g^{L,A,S,E} \Rightarrow e^{L,A,S,E}$ is acceptably valid and $g^{L,A,S,E} \Rightarrow e^{L,A,S,E}$ is not acceptably valid for all $g \in G$. Let $m$ be an acceptable model such that, for all closed conclusions $e'$, $m$ satisfies $e'^{L,A,S,E}$ iff $e' \neq e$. Since $\bigwedge_{g \in G} g^{L,A,S,E} \Rightarrow e^{L,A,S,E}$ is acceptably valid, there is a $g = \forall x_1 \ldots \forall x_n(d_g \rightarrow e_g) \in G$ such that $m$ does not satisfy $g^{L,A,S,E}$. By the translation, it follows that there is an $A$-closed substitution $\sigma$ such that $e_g^\sigma \not\in E$, $d_g^\sigma^{L,A,S,E}$ holds in $m$, and $e_g^\sigma \neq e$. Because, for all conditions $d'$, $d'^{L,A,S,E}$ can be written as $\text{Val}(\varphi)$ for an appropriate formula $\varphi$, $d_g^\sigma^{L,A,S,E}$ is acceptably valid since it holds in an acceptable model. It follows that $g^{L,A,S,E} \Rightarrow e^{L,A,S,E}$ is acceptably valid, which contradicts the assumption.

It remains to show that $g^{L,A,S,E} \Rightarrow e^{L,A,S,E}$ is acceptably valid for a grant $g = \forall x_1 \ldots \forall x_n(d_g \rightarrow e_g)$ iff $e \not\in E$ and, for some $A$-closed substitution $\sigma$, the formula $d_g^\sigma^{L,A,S,E}$ is acceptably valid and $e_g^\sigma = e$. The “if” direction is immediate from the translation. For the “only if” direction, suppose by way of contradiction that $g^{L,A,S,E} \Rightarrow e^{L,A,S,E}$ is acceptably valid and either $e \in E$ or, for each $A$-closed substitution $\sigma$, either $d_g^\sigma^{L,A,S,E}$ is not valid or $e_g^\sigma \neq e$. Let $m$ be the acceptable model defined above; that is, for all conclusions $e'$, $m$ satisfies $e'^{L,A,S,E}$ iff $e' \neq e$. We can get a contradiction by showing that $m$ satisfies $g^{L,A,S,E}$. If $e \in E$, then $m$ satisfies $g^{L,A,S,E}$ since either $e_g^\sigma \in E$ (because $e_g^\sigma = e$), or $e_g^\sigma$ holds in $m$ (because $e_g^\sigma \neq e$). Otherwise, by assumption, either $d_g^\sigma^{L,A,S,E}$ is not acceptably valid or $e_g^\sigma \neq e$, for each $A$-closed substitution $\sigma$. Note that, because $d_g^\sigma^{L,A,S,E}$ (like every formula of the form $d^{L,A,S,E}$ for some condition $d$) is equivalent to a formula of the form $\text{Val}(\varphi)$, then if it is not acceptably valid, it is not true in any acceptable
model and, in particular, not in $m$. It then easily follows from the translation that $m$ satisfies $g^{L,A,S,E}$. This gives us the desired contradiction.

**Definition B.0.16.** Let $(e, L, R, E)$ be a query, let $X$ be a terminating execution of Query2$(e, L, R, E)$, and let $A = A(e, L, R, E, X)$. Then

$$G(e, L, R, E, X) =$$

$$R \cup \{h \mid \text{for some principal } p, (p, h) \in L \text{ and }$$

$$(\bigwedge_{\ell \in L} \ell^{L,A,\emptyset,(E\cup\{e\})}) \land (\bigwedge_{g \in R} g^{L,A,\emptyset,(E\cup\{e\})})$$

$$\Rightarrow \text{Permitted}(p, \text{issue}, c_h) \text{ is acceptably valid}\}.$$

**Lemma B.0.17.** Suppose that $(e, L, R, E)$ is a query, $X$ is a terminating execution of Query2$(e, L, R, E)$, and $A = A(e, L, R, E, X)$. Then $\bigwedge_{\ell \in L} \ell^{L,A,\emptyset,E} \land \bigwedge_{g \in R} g^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E}$ is acceptably valid iff there is a grant $h \in G(e, L, R, E, X)$ such that $h^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E}$ is acceptably valid.

**Proof:** For the “if” direction, suppose that $h$ is a grant in $G(e, L, R, E, X)$ such that $h^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E}$ is acceptably valid. If $h \in R$, then $\bigwedge_{\ell \in L} \ell^{L,A,\emptyset,E} \land \bigwedge_{g \in R} g^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E}$ is acceptably valid. If $h \in G(e, L, R, E, X) - R$, then there is a principal $p$ such that

(1a) $(p, h) \in L$,

(1b) $\text{Permitted}(p, \text{issue}, h) \not\in E$, and

(1c) $\bigwedge_{\ell \in L} \ell^{L,A,\emptyset,(E\cup\{e\})} \land \bigwedge_{g \in R} g^{L,A,\emptyset,(E\cup\{e\})} \Rightarrow \text{Permitted}(p, \text{issue}, c_h)$ is acceptably valid.
Let \( \varphi = \bigwedge_{\ell \in L} \ell^{L,A,\emptyset,E} \land \bigwedge_{g \in R} g^{L,A,\emptyset,E} \). It follows from (1a) that \( \varphi \Rightarrow (p, h)^{L,A,\emptyset,E} \) is acceptably valid. It follows from (1a), (1b), and the translation that \( \varphi \Rightarrow (\text{Permitted}(p, \text{issue}, c_h) \Rightarrow h^{L,A,\emptyset,E}) \) is acceptably valid. It follows from (1c) and Lemma B.0.13 that \( \varphi \Rightarrow \text{Permitted}(p, \text{issue}, c_h) \Rightarrow h^{L,A,\emptyset,E} \) is acceptably valid. By assumption \( h^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E} \) is acceptably valid, so \( \varphi \Rightarrow e^{L,A,\emptyset,E} \) is acceptably valid.

For the “only if” direction, suppose that there is no grant \( g \in G(e, L, R, E, X) \) such that \( g^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E} \) is acceptably valid. Let \( m \) be an acceptable model that does not satisfy \( e^{L,A,\emptyset,E} \) and the formulas in \( \{ \text{Permitted}(p, \text{issue}, h)^{L,A,\emptyset,E} \mid (p, h) \in L, \text{Permitted}(p, \text{issue}, h) \not\in E, \text{ and } h \not\in G(e, L, R, E, X) \} \). Because \( m \) does not satisfy \( e^{L,A,\emptyset,E} \), it suffices to show that \( m \) satisfies \( \bigwedge_{\ell \in L} \ell^{L,A,\emptyset,E} \land \bigwedge_{g \in R} g^{L,A,\emptyset,E} \). We do this by showing that (1) \( m \) satisfies \( (p, h)^{L,A,\emptyset,E} \) for every license \( (p, h) \) such that \( h \not\in G(e, L, R, E, X) \), and (2) \( m \) satisfies \( g^{L,A,\emptyset,E} \) for every grant \( g \in G(e, L, R, E, X) \).

For part (1), observe that if \( \text{Permitted}(p, \text{issue}, h) \in E \) or \( (p, h) \not\in L \), then \( (p, h)^{L,A,\emptyset,E} = \text{true} \), so \( (p, h)^{L,A,\emptyset,E} \) holds in \( m \). If \( \text{Permitted}(p, \text{issue}, h) \not\in E \) and \( (p, h) \in L \), then \( (p, h)^{L,A,\emptyset,E} = \text{Permitted}(p, \text{issue}, c_h) \Rightarrow h^{L,A,\emptyset,E} \) and, by construction, \( m \) does not satisfy \( \text{Permitted}(p, \text{issue}, c_h) \); so \( (p, h)^{L,A,\emptyset,E} \) is again true in \( m \).

For part (2), let \( g = \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in G(e, L, R, E, X) \), and recall that \( g^{L,A,\emptyset,E} \) is the conjunction of formulas of the form

\[
\bigwedge_{e \in E} \neg \text{Val}(e^{L,A,\emptyset,E} \iff (e_g \sigma)^{L,A,\emptyset,E} \land (d_g \sigma)^{L,A,\emptyset,E} \Rightarrow (e_g \sigma)^{L,A,\emptyset,E}),
\]

where \( \sigma \) is an \( A \)-closed substitution. Clearly, \( m \) satisfies \( g^{L,A,\emptyset,E} \) iff, for every \( A \)-closed substitution \( \sigma \), \( m \) satisfies \((\bigwedge_{e' \in E} \neg \text{Val}(e'^{L,A,\emptyset,E} \iff (e_g \sigma)^{L,A,\emptyset,E} \land (d_g \sigma)^{L,A,\emptyset,E} \Rightarrow (e_g \sigma)^{L,A,\emptyset,E}) \). It is easy to see that the latter statement holds if,
for all $A$-closed substitutions $\sigma$, either $e_\sigma \in E$, $(d_\sigma)^{L,A,\emptyset,E}$ is not true in $m$, or $(e_\sigma)^{L,A,\emptyset,E}$ is true in $m$. We claim that this is indeed the case. To prove the claim, suppose by way of contradiction that $e_\sigma \not\in E$, $(d_\sigma)^{L,A,\emptyset,E}$ is true in $m$, and $(e_\sigma)^{L,A,\emptyset,E}$ is not true in $m$. Since $(e_\sigma)^{L,A,\emptyset,E}$ is not true in $m$, either $e_\sigma = e$ or $e_\sigma \in \{\text{Permitted}(p,\text{issue}, h) | (p, h) \in L, \text{Permitted}(p,\text{issue}, h) \not\in E, \text{ and } h \not\in G(e, L, R, E, X)\}$.

If $e_\sigma = e$, then we claim that $g^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E}$ is acceptably valid. To see this note that $g^{L,A,\emptyset,E} \Rightarrow (\bigwedge_{e' \in E} \neg \text{Val}(e') \iff (e_\sigma)^{L,A,\emptyset,E} \wedge (d_\sigma)^{L,A,\emptyset,E} \Rightarrow (e_\sigma)^{L,A,\emptyset,E})$ is acceptably valid. Because $e_\sigma \not\in E$, $\bigwedge_{e' \in E} \neg \text{Val}(e') \iff (e_\sigma)^{L,A,\emptyset,E} \wedge (d_\sigma)^{L,A,\emptyset,E} \Rightarrow (e_\sigma)^{L,A,\emptyset,E}$ is equivalent to true; so, $g^{L,A,\emptyset,E} \Rightarrow ((d_\sigma)^{L,A,\emptyset,E} \Rightarrow (e_\sigma)^{L,A,\emptyset,E})$ is acceptably valid. Since $(d_\sigma)^{L,A,\emptyset,E}$ is true in $m$ by assumption, and, as we have observed, every formula of the form $d^{L,A,\emptyset,E}$ is equivalent to $\text{Val}(\varphi)$ for some formula $\varphi$, $(d_\sigma)^{L,A,\emptyset,E}$ is acceptably valid and, as a result, $g^{L,A,\emptyset,E} \Rightarrow (e_\sigma)^{L,A,\emptyset,E}$ is acceptably valid. By assumption, $e_\sigma = e$, so $g^{L,A,\emptyset,E} \Rightarrow e^{L,A,\emptyset,E}$ is acceptably valid. Since $g \in G(e, L, R, E, X)$ and, by assumption, none of the grants in $G(e, L, R, E, X)$ imply $e^{L,A,\emptyset,E}$, we have a contradiction.

Finally, suppose that $e_\sigma \neq e$ and $e_\sigma = \text{Permitted}(p,\text{issue}, h)$, where $(p, h) \in L, \text{Permitted}(p,\text{issue}, h) \not\in E, \text{ and } h \not\in G(e, L, R, E, X)$. We now prove that $g^{L,A,\emptyset,(E\cup\{e\})} \Rightarrow \text{Permitted}(p,\text{issue}, c_h)$ is acceptably valid, so $h \in G(e, L, R, E, X)$, which contradicts the assumptions. We begin by noting that

$$g^{L,A,\emptyset,(E\cup\{e\})}$$

$$\Rightarrow ((\bigwedge_{e' \in E\cup\{e\}} \neg \text{Val}(e') \iff (e_\sigma)^{L,A,\emptyset,(E\cup\{e\})} \wedge (d_\sigma)^{L,A,\emptyset,(E\cup\{e\})})$$

$$\Rightarrow (e_\sigma)^{L,A,\emptyset,(E\cup\{e\})}$$

is acceptably valid. By assumption, $e_\sigma \not\in E \cup \{e\}$, therefore $g^{L,A,\emptyset,(E\cup\{e\})} \Rightarrow$
\[(d_g \sigma)^{L,A,\emptyset,(E\cup\{e\})} \Rightarrow (e_g \sigma)^{L,A,\emptyset,(E\cup\{e\})}\]
is acceptably valid. Since the conclusions \(e_g \sigma = \text{Permitted}(p, \text{issue}, h)\) and \(e_g \sigma^{L,A,\emptyset,(E\cup\{e\})} = \text{Permitted}(p, \text{issue}, c_h)\), the formula \(g^{L,A,\emptyset,(E\cup\{e\})} \Rightarrow ((d_g \sigma)^{L,A,\emptyset,(E\cup\{e\})} \Rightarrow \text{Permitted}(p, \text{issue}, c_h))\) is acceptably valid. It remains to be shown that \((d_g \sigma)^{L,A,\emptyset,(E\cup\{e\})}\) is acceptably valid. Because the translation of a condition does not depend on the set of conclusions, it suffices to show that \(d_g \sigma^{L,A,\emptyset,E}\) is acceptably valid. But, as we observed above, this follows immediately from the assumption that \(d_g \sigma^{L,A,\emptyset,E}\) is true in \(m\).

**Theorem B.0.18.** Suppose that \((e, L, R, E)\) is a query, \(X\) is a terminating execution of \(\text{Query2}(e, L, R, E)\), and \(A = A(e, L, R, E, X)\). Then for all calls of the form \(\text{Holds2}(d, L, S)\), \(\text{Auth2}(e', L, R, E')\), or \(\text{Query2}(e', L, R, E')\) made during execution \(X\), including the initial call,

(a) \(\text{Holds2}(d, L, S)\) returns true iff \(d^{L,A,S,E'}\) is acceptably valid, where \(E'\) is an (arbitrary) set of closed conclusions;

(b) \(\text{Auth2}(e', L, R, E')\) returns the set \(D\) of closed conditions, where \(D = \{d \mid e' \notin E'\text{ and, for some grant } \forall x_1 \ldots \forall x_n(d_g \rightarrow e_g) \in G(e', L, R, E', X)\text{ and closed substitution } \sigma, \ d_g \sigma = d \text{ and } e_g \sigma = e'\}\); and

(c) \(\text{Query2}(e', L, R, E')\) returns true iff \(\bigwedge_{\ell \in L} f^{L,A,\emptyset,E'} \wedge \bigwedge_{g \in R} g^{L,A,\emptyset,E'} \Rightarrow e'^{L,A,\emptyset,E'}\) is acceptably valid.

**Proof:** We prove part (a) by induction on \(#(S^*(e, L, R, E, X) - S)\), with a subinduction on the structure of \(d\). Suppose that \(#(S^*(e, L, R, E, X) - S) = 0\). If \(d = \text{true}\), then \(\text{Holds2}(d, L, S) = \text{true}\) and \(d^{L,A,S,E'} = \text{true}\). Suppose that \(d\) is of the form \(\text{Said}(p, e')\). Then, by Lemma B.0.11, \(d \in S^*(e, L, R, E)\). By assumption, \(#(S^*(e, L, R, E) - S) = 0\), so \(d \in S\). It follows that \(\text{Holds2}(d, L, S) = \text{false}\) and
Finally, if \( d \) is a conjunction, then the result is immediate from the induction hypothesis. For the induction step, the argument used for the base case applies if \( d = \text{true} \) or if \( d \) is a conjunction of conditions. Suppose that \( d \) has the form \( \text{Said}(p, e') \). If \( d \in S \), then \( \text{Holds}_2(d, L, S) = \text{false} \) and \( d^{L,A,S,E'} = \text{false} \).

If \( d \notin S \) then, by the description of \( \text{Holds}_2 \), \( \text{Holds}_2(d, L, S) = \text{true} \) iff there is a grant \( g = \forall x_1 \ldots \forall x_n(d_g \rightarrow e_g) \in R_p \) and an \( A \)-closed substitution \( \sigma \) such that \( \text{Holds}_2(d_g\sigma, L, S \cup \{d\}) = \text{true} \) and \( e_g\sigma = e' \). By the induction hypothesis,

\[
\text{Holds}_2(d_g\sigma, L, S \cup \{d\}) = \text{true} \iff d_g\sigma^{L,A,(S\cup\{d\}),E'} \text{ is acceptably valid.}
\]

By the translation, the latter statement holds iff \( d_g\sigma^{L,A,(S\cup\{d\}),\emptyset} \text{ is acceptably valid.} \)

We prove parts (b) and (c) by simultaneous induction on \( \#(E^*(e, L, R) - E') \).

If \( \#(E^*(e, L, R) - E') = 0 \), then \( e' \in E^*(e, L, R) \) by Lemma B.0.11, so \( e' \in E' \).

Because \( e' \in E' \), \( \text{Auth}_2(e', L, R, E') = \emptyset \), so part (b) holds. For part (c), \( \text{Query}_2 \) begins by calling \( \text{Auth}_2(e', L, R, E') \), which returns the empty set, and then \( \text{Query}_2 \) returns \( \text{false} \). Since \( e' \in E' \), it follows from Lemma B.0.12 that

\[
\bigwedge_{\ell \in L} \ell^{L,A,\emptyset,E'} \land \bigwedge_{g \in R} g^{L,A,\emptyset,E'} \Rightarrow e'^{L,A,\emptyset,E'}
\]

is not acceptably valid, so the invariant holds.

Now consider the inductive step. For part (b), suppose that \( \text{Auth}_2(e', L, R, E') \) is called during the execution of \( \text{Query}_2(e, L, R, E) \). If \( e' \in E' \), then part (b) holds by the same argument as in the base case. If \( e' \notin E' \), then \( \text{Auth}_2 \) returns a set \( D \) of closed conditions such that \( d \in D \) iff there is a grant \( \forall x_1 \ldots \forall x_n(d_h \rightarrow e_h) \in S_L \).
and a closed substitution $\sigma$ such that $d_h \sigma = d$ and $e_h \sigma = e$, where

$$S_L = R \cup \{ h \mid \text{for some principal } p, \ (p, h) \in L \text{ and, during execution } X,$$

$$\text{Query2(Permitted}(p, \text{issue}, h), L, R, (E' \cup \{ e' \})) \text{ returns } true \}.$$ 

It clearly suffices to show that $S_L = G(e', L, R, E', X)$. By Lemma B.0.11, $e' \in E^*(e, L, R)$ and, by assumption, $e \not\in E'$. So it follows from the induction hypothesis that

$$S_L = R \cup \{ h \mid \text{for some principal } p, \ (p, h) \in L \text{ and }$$

$$\text{for all } \ell \in L, g \in R \text{ such that } g \in G(e', L, R, E', X),$$

$$\Rightarrow \text{Permitted}(p, \text{issue}, c_h) \text{ is acceptably valid},$$

which is $G(e', L, R, E', X)$.

For part (c), observe that if $e' \in E'$ then we can use the same reasoning as in the base case to show that the invariant holds. If $e' \not\in E'$ then, during execution $X$, $\text{Query2}(e', L, R, E')$ returns $true$ iff there is a closed condition $d$ in the output of $\text{Auth2}(e', L, R, E')$ such that $\text{Query2}$ calls $\text{Holds2}(d, L, \emptyset)$, which returns $true$. By part (b), $\text{Auth2}(e', L, R, E')$ returns a set of conditions that includes $d$ iff there is a grant $g = \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in G(e', L, R, E', X)$ and a closed substitution $\sigma$ such that $d_h \sigma = d$ and $e_h \sigma = e'$. Moreover, since $\text{Holds2}(d, L, \emptyset)$ is called during execution $X$ of $\text{Query2}(e, L, R, E)$, $\sigma$ is $A$-closed. By part (a), $\text{Holds2}(d, L, \emptyset) = true$ iff $d^{L, A, \emptyset, E'}$ is acceptably valid. So $\text{Query2}(e', L, R, E')$ returns $true$ iff there is a grant $g = \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in G(e', L, R, E', X)$ and an $A$-closed substitution $\sigma$ such that $d^{L, A, \emptyset, E'}$ is acceptably valid and $e_g \sigma = e'$. By assumption, $e' \not\in E'$; so, by Lemma B.0.15, $\text{Query2}(e', L, R, E') = true$ iff $g^{L, A, \emptyset, E'} \Rightarrow e'^{L, A, \emptyset, E'}$ is acceptably valid for some $g \in G(e', L, R, E', X)$. It follows from Lemma B.0.17 that the latter statement holds iff $\bigwedge_{\ell \in L} \ell^{L, A, \emptyset, E'} \land \bigwedge_{g \in R} g^{L, A, \emptyset, E'} \Rightarrow e'^{L, A, \emptyset, E'}$ is acceptably valid. ■
Theorem 3.4.1: Determining whether some execution of Query2\((e, L, R, E)\) returns true is undecidable for the set of queries \((e, L, R, E)\) such that at most one grant in \(R \cup L\) is not restrained.

Proof: We reduce the Post correspondence problem (PCP) \cite{Pos46} to the problem of determining whether some execution of Query2\((e, L, R, \emptyset)\) returns true for a query \((e, L, R, \emptyset)\), where all but one grant in \(R \cup L\) is restrained. Let \(\Sigma\) be an alphabet; let \(s_1, \ldots, s_n\) and \(t_1, \ldots, t_n\) be strings over \(\Sigma\); and, for all strings \(s\) and \(s'\), let \(s \cdot s'\) be the concatenation of \(s\) and \(s'\). We want to determine if there are integers \(i_1, \ldots, i_k \in \{1, \ldots, n\}\) such that \(s_{i_1} \cdot \ldots \cdot s_{i_k} = t_{i_1} \cdot \ldots \cdot t_{i_k}\).

To encode the problem as a query, assume that the language includes the primitive principal \(p_\sigma\) for each symbol \(\sigma \in \Sigma\), the primitive principal \(p\), and the property \(Pr\). For every string \(s\) over \(\Sigma\), define a function \(G_s\) from grants to grants by induction on the length of \(s\). If \(s\) has length one \((s \in \Sigma)\), then \(G_s(g) = \text{Permitted}(p_\sigma, \text{issue}, g)\). If \(s = \sigma s'\), then \(G_s = G_\sigma \circ G_{s'}\). For all grants \(g_1\) and \(g_2\), define \(G(g_1, g_2)\) to be the grant \(\text{Said}(p, \text{Permitted}(p, \text{issue}, g_1)) \rightarrow \text{Permitted}(p, \text{issue}, g_2)\).

We claim that there are integers \(i_1, \ldots, i_k \in \{1, \ldots, n\}\) such that \(s_{i_1} \cdot \ldots \cdot s_{i_k} = t_{i_1} \cdot \ldots \cdot t_{i_k}\) iff an execution of Query2\((Pr(p), L, R, \emptyset)\) returns true, where

\[
L = \{(p, \text{Permitted}(p, \text{issue}, G(G_{s_1}(\text{Pr}(p)), G_{t_1}(\text{Pr}(p)))) | i = 1, \ldots, n) \cup \\
(p, \forall x_1 \forall x_2(\text{Said}(p, \text{Permitted}(p, \text{issue}, G(x_1, x_2))) \rightarrow \\
\text{Permitted}(p, \text{issue}, G(G_{s_1}(x_1), G_{t_1}(x_2)))) | i = 1, \ldots, n) \cup \\
(p, \forall x(\text{Said}(p, \text{Permitted}(p, \text{issue}, G(x, x)) \rightarrow \text{Pr}(p))) \}
\]

and \(R = \{\forall x(\text{Said}(p, \text{Permitted}(p, \text{issue}, G(x, x)) \rightarrow \text{Pr}(p))) \}
\]

Recall that an execution of Query2\((e, L, R, \emptyset)\) returns true iff an execution of Auth2\((e, L, R, \emptyset)\) returns a set \(D\) of conditions such that an execution of Holds2\((d, L, \emptyset)\) returns true for some condition \(d \in D\). It is
easy to see that every execution of \textbf{Auth2}(e, L, R, \emptyset) returns the set \( D = \{ \text{Said}(p, \text{Permitted}(p, \text{issue}, G(g, g))) \mid \text{g is a closed grant} \} \). Moreover, if \( d \) is of the form \( \text{Said}(p, \text{Permitted}(p, \text{issue}, G(g, g))) \), where \( g \) is a closed grant, then it is not hard to see that an execution of \textbf{Holds2}(d, L, \emptyset) returns true iff there are integers \( i_1, \ldots, i_k \in \{1, \ldots, n\} \) such that \( g = G_{s_1 i_1} (G_{s_2 i_2} (\ldots G_{s_k i_k} (\text{Pr}(p)) \ldots)) \) and \( g = G_{t_1 i_1} (G_{t_2 i_2} (\ldots G_{t_k i_k} (\text{Pr}(p)) \ldots)) \). The latter statements holds iff there are integers \( i_1, \ldots, i_k \in \{1, \ldots, n\} \) such that \( s_{i_1} \cdot \ldots \cdot s_{i_k} = t_{i_1} \cdot \ldots \cdot t_{i_k} \).

\[\text{Theorem 3.4.2:} \quad \text{The problem of deciding whether at least one execution of} \quad \text{Query2}(e, L, R, E) \quad \text{returns true for} \quad (e, L, R, E) \in \mathcal{L}_0 \cap \mathcal{L} \cap \mathcal{L}' \quad \text{is NP-hard for} \quad \mathcal{L}, \mathcal{L}' \in \{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}.\]

\[\text{Proof:} \quad \text{For the NP hardness results, it suffices to show that the problem of deciding whether} \quad \text{Query2}(e, L, R, E) = \text{true} \quad \text{is NP-hard if (a)} \quad (e, L, R, E) \in \mathcal{L}_0 \cap \mathcal{L}_2 \cap \mathcal{L}_3, \quad \text{(b)} \quad (e, L, R, E) \in \mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_3, \quad \text{and (c)} \quad (e, L, R, E) \in \mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2.\]

For part (a), we show that we can reduce the Hamiltonian path problem to the problem of determining whether \( \text{Query2}(e, L, R, E) = \text{true} \), for some \( (e, L, R, E) \in \mathcal{L}_0 \cap \mathcal{L}_2 \cap \mathcal{L}_3 \). Given a graph \( G(V, E) \), where \( V = \{v_1, \ldots, v_n\} \), we take \( v_1, \ldots, v_n \) to be primitive principles. We also assume that the language has primitive properties \textbf{Node}, \textbf{Edge}, and \textbf{Path}. For each node \( v \in V \), let \( g_v \) be the grant \textbf{Node}(v) (recall that this is an abbreviation for \( \text{true} \rightarrow \textbf{Node}(v) \)). For each edge \( e = (v, v') \in E \), let \( g_{(v,v')} \) be the grant \textbf{Edge}(\{v, v'\}) (recall that \( \{v, v'\} \) is an abbreviation for \( \{v\} \cup \{v'\} \)). Finally, let \( g \) be the grant \( \forall x_1 \ldots \forall x_n (d_1 \land d_2 \rightarrow \textbf{Path} (\{x_1, \ldots, x_n\})) \), where

\[d_1 = \bigwedge_{1 \leq i \leq n} \text{Said}(\text{Alice, Node}(x_i)) \quad \text{and} \]

\[d_2 = \bigwedge_{1 \leq i \leq n-1} \text{Said}(\text{Alice, Edge}(\{x_i, x_{i+1}\})).\]
Let

\[ L = \{(Alice, g_v) \mid v \in V\} \cup \{(Alice, g_e) \mid e \in E\} \]

and \( R = \{g\} \).

It is not hard to show that \( \text{Query2}(\text{Path}(\{v_1, \ldots, v_n\}), L, R, \emptyset) = \text{true} \) iff \( G \) has a Hamiltonian path. To see this, observe that \( \text{Auth2}(\text{Path}(\{v_1, \ldots, v_n\}), L, R, \emptyset) \) returns \( \{d_1\sigma \land d_2\sigma \mid \sigma(x_i) = v_\pi(i), i = 1, \ldots, n, \text{ where } \pi \text{ is some permutation of } \{1, \ldots, n\}\} \). The condition \( d_2\sigma \) holds iff there is a path \( x_1\sigma, \ldots, x_n\sigma \). Thus, \( \text{Query2}(\text{Path}(\{v_1, \ldots, v_n\}), L, R, \emptyset) = \text{true} \) iff there is a Hamiltonian path in \( G \).

Moreover, it is clear that \( (\text{Path}(\{v_1, \ldots, v_n\}), L, R, \emptyset) \in L_0 \cap L_1 \cap L_2 \) and it is not hard to see that \( (\text{Path}(\{v_1, \ldots, v_n\}), L, R, \emptyset) \in L_3 \), because the antecedent of every issued grant is \text{true}.

For part (b), we show that we can reduce the 3-satisfiability problem to the problem of determining whether \( \text{Query2}(e, L, R, E) = \text{true} \), for \( (e, L, R, E) \in L_0 \cap L_1 \cap L_2 \). Let \( f = c_1 \land \ldots \land c_n \) be a formula in propositional logic, where each \( c_i \) is a clause with three disjuncts. Let \( q_1, \ldots, q_m \) be the primitive propositions mentioned in \( f \). We want to determine if \( f \) is satisfiable.

To encode the problem as an XrML query, suppose that \( p_1, \ldots, p_n, p_t, p_f \) are distinct primitive principals, \( \text{Pr} \) is a property, and \( x_1, \ldots, x_m \) are distinct variables of sort \( \text{Princ} \). Let \( g_0 \) be a fixed closed grant. Given principals \( t_1, \ldots, t_m \), we define grants \( g_1(t_1), \ldots, g_m(t_1, \ldots, t_m) \) inductively as follows: \( g_1(t_1) \) is the grant \( \text{true} \rightarrow \text{Permitted}(t_i, \text{issue}, g_0) \) and, for \( i = 2, \ldots, m \), \( g_i \) is the grant \( \text{true} \rightarrow \text{Permitted}(t_i, \text{issue}, g_{i-1}(t_1, \ldots, t_{i-1})) \). Let \( e(t_1, \ldots, t_m) \) be the conclusion \( \text{Permitted}(t_m, \text{issue}, g_{m-1}(t_1, \ldots, t_{m-1})) \). For ease of exposition, let \( e' \) be
the conclusion $e(x_1, \ldots, x_m)$. Let

$$L = \{(p_i, \forall x_1 \ldots \forall x_m(e'[x_j/p_i])) | q_j \text{ is a disjunct of } c_i\} \cup \{(p_i, \forall x_1 \ldots \forall x_m(e'[x_j/p_f])) | \neg q_j \text{ is a disjunct of } c_i\}$$

and

$$R = \{\forall x_1 \ldots \forall x_m(\bigwedge_{i=1,\ldots,n} \text{Said}(p_i, e') \rightarrow \text{Pr}(p_t)\}.$$ 

We claim that $f$ is satisfiable iff $\text{Query2}(\text{Pr}(p_t), L, R, \emptyset) = true$. Note that $(\text{P}(p_t), L, R, \emptyset) \in \mathcal{L}_1 \cap \mathcal{L}_0 \cap \mathcal{L}_3^2$, since none of the grants mention a variable of sort $Rsrc$, the $\cup$ operator is not mentioned in the query, and the antecedent of every issued grant is $true$.

To prove the claim, first observe that $\text{Query2}(\text{Pr}(p_t), L, R, \emptyset) = true$ if and only if $\bigwedge_{i=1,\ldots,n} \text{Said}(p_i, e')\sigma$ holds for some substitution $\sigma$. It is not hard to see that if $\sigma$ exists, then $f$ is satisfied by the truth assignment that sets $q_i = true$ if $\sigma$ sets $x_i$ to $p_t$, and sets $q_i$ to $false$ otherwise. Similarly, if $f$ is satisfied by a truth assignment $A$, then $\bigwedge_{i=1,\ldots,n} \text{Said}(p_i, e')\sigma$ holds for the substitution $\sigma$ that replaces $x_i$ by $p_t$ if $A$ assigns $x_i$ to $true$, and replaces $x_i$ by $p_f$ otherwise.

For part (c), we show that we can reduce the 3-satisfiability problem to the problem of determining whether $\text{Query2}(e, L, R, E) = true$, for $(e, L, R, E) \in \mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2^0$. As in part (b), let $f$ be the 3-CNF formula $c_1 \land \ldots \land c_n$, whose primitive propositions are $q_1, \ldots, q_m$. Define the condition $e(t_1, \ldots, t_m)$ as in part (b); again, take $e'$ to be an abbreviation for $e(x_1, \ldots, x_m)$. Let $p'_1, \ldots, p'_m$ be fresh principals, distinct from $p_1, \ldots, p_n, p_f, p_t$. We claim that $f$ is satisfied iff $\text{Query2}(e(p'_1, \ldots, p'_m), L, R, \emptyset) = true$, where

$$L = \{(p_i, \forall x_1 \ldots \forall x_m(\text{Said}(p_{i+1}, e'[x_j/p_i]) \rightarrow e'[x_j/p])) | q_j \text{ is a disjunct of } c_i, p \neq p_f, i = 1, \ldots, n - 1\}.$$
\[ R = \{ \text{Said}(p_1, e(p_1', \ldots, p_m')) \rightarrow e(p_1', \ldots, p_m') \}. \]

If \( t_1, \ldots, t_m \) are variable-free principals, let \( A(t_1, \ldots, t_m) \) be the set of all truth assignments to \( q_1, \ldots, q_m \) such that \( q_i \) is assigned \textit{true} if \( t_i = p_i \) and \( q_i \) is assigned \textit{false} if \( t_i = p_f \), for \( i = 1, \ldots, m \). (If \( t_i \notin \{p_i, p_f\} \), then there are no constraints on \( q_i \).) Let \( A_i(t_1, \ldots, t_m) \) be the set of all truth assignments to \( q_1, \ldots, q_m \) under which \( c_i \land \ldots \land c_n \) is \textit{true}. We show by induction on \( n - i \) that \( A_i(t_1, \ldots, t_m) \) is nonempty iff \( \text{Said}(p_i, e(t_1, \ldots, t_m)) \) holds. If \( n - i = 0 \), then \( i = n \). It is easy to see that \( A_i(t_1, \ldots, t_m) \) is nonempty iff, for some \( j = 1, \ldots, m \), either \( q_j \) is a disjunct of \( c_n \) and \( t_j \neq p_f \), or \( \neg q_j \) is a disjunct of \( c_n \) and \( t_j \neq p_t \). For the inductive step, suppose that \( n - i > 0 \). Clearly, \( A_i(t_1, \ldots, t_m) \) is nonempty iff there is an assignment in \( A_{i-1}(t_1, \ldots, t_m) \) under which \( c_i \) is \textit{true}. If there is at least one such assignment, then \( A_{i-1}(t_1', \ldots, t_m') \) is nonempty, where \( t_1', \ldots, t_m' \) are variable-free principals such that, for some \( j \in \{1, \ldots, m\} \) and for all \( i \neq j \), \( t_i' = t_i \) and either \( q_j \) is a disjunct of \( c_i \), \( t_j \neq p_f \), and \( t_j' = p_t \), or \( \neg q_j \) is a disjunct of \( c_i \), \( t_j \neq p_t \), and \( t_j' = p_f \). It follows from the induction hypothesis that \( \text{Said}(p_i, e(t_1', \ldots, t_m')) \) holds and it follows from \( L \) that \( \text{Said}(p_i, e(t_1, \ldots, t_m)) \) holds as well. If there is no assignment in \( A_{i-1}(t_1, \ldots, t_m) \) under which \( c_i \) is \textit{true} then, for every disjunct \( q_j \) in \( c_i \), \( t_i = p_f \) and, for every disjunct \( \neg q_j \) in \( c_i \), \( t_j = p_t \). It follows that \( A_i(t_1, \ldots, t_m) = \emptyset \) and \( \text{Said}(p_i, e(t_1, \ldots, t_m)) \) does not hold.

The desired result now follows quickly. It is easy to see that \( \text{Query2}(e, L, R, \emptyset) \)
returns true iff $\text{Said}(p_1, e(p'_1, \ldots, p'_m))$ holds. Since none of $p'_1, \ldots, p'_m$ is $p_f$ or $p_t$, by definition, $A(p'_1, \ldots, p'_m)$ consists of all truth assignments. Thus, by the induction argument, it follows that $\text{Query2}(e, L, R, \emptyset) = \text{true}$ iff $f = c_1 \land \ldots \land c_n$ is satisfiable. Moreover, it is easy to see that $(e, L, R, \emptyset) \in \mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2^0$, because the query does not mention union and, for every variable $x$ mentioned in a grant $g$ that is in $R \cup L$, $x$ is mentioned in the conclusion of $g$. $lacksquare$

We next prove Theorem 3.4.3, which considers the complexity of determining whether $\text{Query2}(e, L, R, E)$ returns true for $(e, L, R, E) \in \mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2^n \cap \mathcal{L}_3^h$. In the statement of the theorem, we viewed $n$ and $h$ as constants. In our proof, we treat them as parameters, so as to bring out their role.

To prove the theorem we need three preliminary lemmas. The first uses the fact that, for every condition $d$, there is a dag (directed acyclic graph) $G_d$ such that $G_d$ represents $d$ and $G_d$ is no larger than $d$. To make this precise, recall that $|s|$ is the length of string $s$ when viewed as a string of symbols. For ease of exposition, we assume that each pair of parenthesis and set braces has length 2, and each comma has length 1. For a graph $G(V, E)$, let $|G| = \#(V) + \#(E)$. It is easy to see that a condition $d$ can be represented as a tree $T_d$, where $|T_d| \leq |d|$. For example, we can represent the condition $d = \text{Said}(\{\text{Alice, Bob}\}, \text{Smart(Amy)}) \land \text{Said}(\{\text{Alice, Bob}\}, \text{Pretty(Amy)})$ as the tree $T_d$ shown in Figure B.1. Note that $|d| = 27$ and, because the tree has 13 nodes and 12 edges, $|T_d| = 25$. By “merging” identical subtrees, we can create a dag representation of $d$ that can be substantially smaller than $|d|$. Continuing our example, the dag $D_d$ shown in Figure B.2 represents the condition $d$ and $|D_d| = 19$.

Lemma B.0.19. Suppose that $T$ is the call tree for an execution of $\text{Holds2}(d, L, \emptyset)$; every license in $L$ is restrained; the $\cup$ operator is not mentioned in $d$ or in a grant
Figure B.1: Tree Representation of $d$

Figure B.2: Dag Representation of $d$
in $L$; and $v$ is a node in $T$ with label $\textbf{Holds}_2(d', L, S)$. If $G_d$ is a dag representing $d$, then there exists a dag $G_{d'}$ representing $d'$ such that $|G_{d'}| \leq h|L| + |G_d|$, where $h$ is the height of $T$.

**Proof:** Because $v$ is a node in $T$, there is a path $v_0, \ldots, v_k$ in $T$ such that $v_0$ is the root of $T$ and $v_k = v$. We prove by induction on $k$ that there is a dag $G_{d'}$ representing $d'$ such that $|G_{d'}| \leq k|L| + |G_d|$. Since $k \leq h$ by assumption, it easily follows that $|G_{d'}| \leq h|L| + |G_d|$.

If $k = 0$, then $v$ is the root of $T$, so $d' = d$. If $k > 0$, then $v$ is the child of a node $v_{k-1}$. Let $\textbf{Holds}_2(d'', L, S')$ be the label of $v_{k-1}$. The proof is by cases on the structure of $d''$. It follows from the description of $\textbf{Holds}_2$ that $d''$ is not true because $d''$ is not a leaf in $T$. If $d''$ is a conjunction, then $d'$ is a conjunct of $d''$. So the space needed to represent $d'$ is less than the space needed to represent $d''$, thus the result follows easily from the induction hypothesis. Finally, if $d''$ has the form $\textbf{Said}(p, e)$, then it follows from the description of $\textbf{Holds}_2$ that there is a license $(p, g) \in L$, where $g = \forall x_1 \ldots \forall x_m(d_g \rightarrow e_g)$, and a closed substitution $\sigma$ such that $d' = d_g\sigma$ and $e_g\sigma = e$. A dag representing $d_g\sigma$ (i.e., $d'$) can be obtained by taking a dag representing $d_g$ and replacing every variable $x$ by a dag representing $\sigma(x)$. Because every grant in $L$ is restrained, $g$ is restrained, so $\sigma$ assigns every variable of sort $Rsrc$ mentioned in $d_g$ to a term in $e$. Since $\sigma(x)$ is a subterm of $e$ or a primitive principal, given a dag $G_{d_g}$ representing $d_g$ and a dag $G_e$ representing $e$, we can construct a dag $G_{d'}$ representing $d'$ such that $|G_{d'}| \leq |G_{d_g}| + |G_e|$. Since, for every condition $d$, there is a tree representation of $d$ whose size is at most $|d|$, there is a dag $G_{d_g}$ representing $d_g$ such that $|G_{d_g}| \leq |d_g|$. Because $d_g$ is the antecedent of a grant in $L$, $|d_g| < |L|$ so it follows that $|G_{d_g}| < L$. Because $e$ is a subterm of $d'' = \textbf{Said}(p, e)$, and by the induction hypothesis, there is a dag $G_{d''}$ representing
Lemma B.0.20. If \( \text{Holds}_2(d, L, \emptyset) \) is \( h \)-bounded, the \( \cup \) operator is not mentioned in \( d \) or in a grant in \( L \), \( L \) is both restrained and \( n \)-restricted, and \( G_d \) is a dag representing \( d \), then the output of \( \text{Holds}_2(d, L, \emptyset) \) can be determined in time

\[
O\left(\max(|G_d|, |L| |P_0|^n)\right)\left(|L| |P_0|^n\right)^{h-2}|L| \left|h \right| + |G_d| (h + |L|)
\]

Proof: Let \( T \) be the call tree for an execution of \( \text{Holds}_2(d, L, \emptyset) \). Our goal is to compute the truth value associated with the root of \( T \), since that truth value is the output of \( \text{Holds}_2(d, L, \emptyset) \).

It is clear that once we have written the call tree, computing the truth value of the root can be done in time linear in the number of nodes in the tree. The obvious way to construct the tree is to start at the root and, for each node \( v \), construct the successors of \( v \) (if there are any). In constructing the call tree, we assume that the condition \( d' \) and the elements of the set \( S \) in a node labeled \( \text{Holds}_2(d', L, S) \) are described using the dags of Lemma B.0.19. Consider a node \( v \) in \( T \) that is labeled \( \text{Holds}_2(d', L, S) \) and is neither the root nor a leaf. Since \( v \) is not a leaf, \( d' \neq \text{true} \). If \( d' \) is a conjunction, then a bound on the number of conjuncts (and hence on the successors of the node) is \( |L| \) since \( d' \) is of the form \( d_g \sigma \), where \( d_g \) is the antecedent of a grant \( g \) that is in \( L \), and \( \sigma \) is a closed substitution. It is easy to see that \( d_g \), and hence \( d_g \sigma \), has at most \( |L| \) conjuncts, and these can be computed in time \( O(|L|) \).

Suppose that \( d' \) is of the form \( \text{Said}(p, e) \). If \( d' \in S \), then \( v \) is a leaf. Since the height of \( T \) is at most \( h \), \( S \) has at most \( h \) elements. It follows from Lemma B.0.19
that each of these elements can be represented using a dag of size at most $h|L| + |G_d|$, so checking whether $\text{Said}(p, e) \in S$ can be done in time $O(h^2|L| + h|G_d|)$. If $d' \notin S$, then each child of $v$ has the form $d_g \sigma$, where $g = \forall x_1 \ldots \forall x_i (d_g \rightarrow e_g)$ is a grant in $L$ and $\sigma$ is a closed substitution such that $e_g \sigma = e$. Since every grant in $L$ is restrained and $n$-restricted, $d_g$ mentions at most $n$ variables that are not mentioned in $e_g$ and each of these variables is of sort $\text{Princ}$. Since $d$ and the grants in $L$ do not mention the $\cup$ operator and $\#(P_0) = |P_0|$, there are at most $|P_0|^n$ possible substitutions $\sigma$. Finding $\sigma(x)$ for all of the variables $x$ that are mentioned in $e_g$ takes time linear in the size of the dag representing $e$ (since $e_g \sigma = e$). Clearly the dag representing $e$ has size less than that representing $d' = \text{Said}(p, e)$. By Lemma B.0.19, the latter dag has size at most $h|L| + |G_d|$. Since $\#(L) \leq |L|$, there are at most $|L||P_0|^n$ children of $v$ and computing what they are takes time $O(|L||P_0|^n + (h|L| + |G_d|)(h + |L|))$.

Similarly, the root of $T$ has at most $\max(|G_d|, |L||P_0|^n)$ children since the root has zero children if $d = \text{true}$, less than $|G_d|$ children if $d$ is a conjunction, and at most $|L||P_0|^n$ children if $d$ is a $\text{Said}$ condition. The children of the root can be computed in time $O(|G_d|)$ if $d$ is a conjunction and in time $O(|L||P_0|^n + |G_d||L|)$ if $d$ is a $\text{Said}$ condition. This follows from the reasoning given for the case when the node is neither the root nor a leaf modified to account for the fact that $d \notin S$, since $S = \emptyset$, and there is a dag representation of $d$ that has length $|G_d|$.

To determine the number of non-leaf nodes of $T$, observe that, if the root of $T$ has $n$ children and each subtree of $T$ has at most $m$ non-leaf nodes, then $T$ has at most $1 + nm$ non-leaf nodes. It follows that $T$ has at most $1 + 2\max(|G_d|, |L||P_0|^n)(|L||P_0|^n)^{h-2}$ non-leaf nodes, since a tree with outdegree
at most $c$ and height $h$ has $c^h/(c-1) \leq 2c^{h-1}$ non-leaf nodes. Thus, it takes time

$$O\left(\max(|G_d|, |L||P_0|^n)(|L||P_0|^n)^{h-2}(|L||P_0|^n + (h|L| + |G_d|)(h + |L|))\right)$$

to compute the children of the $\max(|G_d|, |L||P_0|^n)(|L||P_0|^n)^{h-2}$ non-leaf nodes other than the root. Since this time dominates the time to compute the children of the root, it is also the time required to compute $T$.

Once $T$ is constructed, the truth value of its root can be computed in time linear in the number of nodes of $T$. Thus, $\text{Holds}^2(d, L, \emptyset)$ can be computed in time

$$O\left(\max(|G_d|, |L||P_0|^n)(|L||P_0|^n)^{h-2}(|L||P_0|^n + (h|L| + |G_d|)(h + |L|))\right).$$

\[ \]

**Lemma B.0.21.** Suppose that $(e, L, R, E)$ is a query in $L_0 \cap L_1 \cap L_2 \cap L_3$ such that $e \notin E$ and $D$ is the output of $\text{Auth}^2(e, L, R, E)$. Then

(a) $\#(D)$ is at most $\#(P_0)^n(\#(R) + \#(L))$;

(b) if $d$ is a closed condition in $D$, then there is a dag $G_d$ representing $d$ such that $|G_d| \leq |R| + |L| + |e|$; and

(c) $D$ can be computed in time $O(|L||E \cup \{e\}| + |L|^2 \log(|R| + 1) + |L|^2(|L||P_0|^n)^{h+1}h^2)$.

**Proof:** Let $X$ be an execution of $\text{Query}^2(e, L, R, E)$ and let $G = G(e, L, R, E, X)$.

For part (a), by Theorem B.0.18(b), if $e \notin E$, then

$$D = \{d \mid \text{for some grant } \forall x_1 \ldots \forall x_m (d_g \rightarrow e_g) \in G \text{ and closed substitution } \sigma, d_g\sigma = d \text{ and } e_g\sigma = e\}. \quad (B.1)$$
Since every grant in $G$ is either in $R$ or $L$, $\#(G) \leq \#(R) + \#(L)$. Moreover, because $(e, L, R, E) \in L_0 \cap L_2^n$, for every grant $g = \forall x_1 \ldots \forall x_m(d_g \rightarrow e_g) \in G$, there are at most $n$ variables mentioned in $d_g$ that are not mentioned in $e_g$, and each of these variables is of sort $Princ$. As in the proof of Lemma B.0.20, it follows that there are at most $\#(P_0)^n$ substitutions of variables in $g$ to closed terms such that $e_g \sigma = e$ because $(e, L, R, E) \in L_1$. Part (a) follows immediately.

For part (b), let $d$ be a closed condition in $D$. By (1), $d = d_g \sigma$, where $d_g$ is the antecedent of a grant $g \in G$ and $\sigma$ is a closed substitution. By the proof of part (a), $\sigma$ assigns every variable in $d_g$ to a term in $e$ or to a principal in $P_0$. Given dags $G_e$ and $G_{d_g}$ representing $e$ and $d_g$, respectively, we can obtain a dag $G_d$ representing $d$ by replacing every variable in $G_{d_g}$ by either a subgraph of $G_e$ or by some $p \in P_0$. So there is a dag $G_d$ representing $d$ such that $|G_d| \leq |G_{d_g}| + |G_e|$. Recall that, for every string $s$, there is a dag $G_s$ representing $s$ such that $|G_s| \leq |s|$. So there is a dag $G_d$ representing $d$ such that $|G_d| \leq |d_g| + |e|$. Since $d_g$ is the antecedent of a grant in $G$ and every grant in $G$ is a grant in $R$ or $L$, $|d_g| < |R| + |L|$, and we are done.

For part (c), by (B.1), we can compute $D$ by (i) checking whether $e \in E$; (ii) computing $G$; and (iii) for each grant $g = \forall x_1 \ldots \forall x_m(d_g \rightarrow e_g) \in G$, computing $D_g = \{d \mid$ for some closed substitution $\sigma, d_g \sigma = d$ and $e_g \sigma = e\}$. (Observe that these are the same steps taken in Auth2; however, our approach computes $G$ more efficiently.) Step (i) takes time $O(|E|)$. We show below that $G$ can be completed in time $O(|L|^h|P_0|^n(2^{h-1} + |L|^2|P_0|^{n(h-1)})(|P_0|^n + h^2 + h|L|) + |L|^2 \log(|R| + 1) + |L|(|E| + |e|))$. For step (iii), essentially the same arguments as those used in Lemma B.0.20 show that, given grant $g \in G$, $D_g$ can be computed in time $O(|e| + |e_g| + |P_0|^n|d_g|)$. So, $\{D_g \mid g \in G\}$ can be computed in time $O(|G|(|e| + |P_0|^n))$. 
Since $|G| \leq |R|+|L|$, the total time needed to compute $D$ is $O(|E|+|L|^h|P_0|^n(2^{h-1}+|L|^2|P_0|^{n(h-1)}(|P_0|^n+h^2+h|L|)+|L|^2 \log(|R|+1)+|L|(|E|+|e|)+|R|(|e|+|P_0|^n))$.

For step (ii), let $A = A(e, L, R, E, X)$. For all integers $k \geq 0$, define the set $G^*_k$ of grants inductively as follows: $G^*_0 = R$ and, for $i > 0$, $G^*_i = R \cup \{g \mid$ for some principal $p$, $(p, g) \in L$ and $\bigwedge_{g' \in G^*_{i-1}} g'^{L,A,\emptyset,(E \cup \{e\})} \Rightarrow \text{Permitted}(p, \text{issue}, c_g) \}$ is acceptably valid. We claim that $G^*_\#(L) = G$.

To show that $G^*_\#(L) \subseteq G$, we prove by induction that $G^*_i \subseteq G$ for all $i \geq 0$. The base case is immediate because $G^*_0 = R$. For the inductive step, it suffices to show that, if there is a license $(p, g) \in L$ and a subset $G' \subseteq G$ such that $\bigwedge_{g' \in G'} g'^{L,A,\emptyset,(E \cup \{e\})} \Rightarrow \text{Permitted}(p, \text{issue}, c_g)$ is acceptably valid, then $g \in G$. Let $\varphi = ((\bigwedge_{e \in L} g^{L,A,\emptyset,(E \cup \{e\})}) \land (\bigwedge_{g \in R} g^{L,A,\emptyset,(E \cup \{e\})}))$. Because $(p, g) \in L$, it is immediate from the definition of $G$ that $g \in G$ if $\varphi \Rightarrow \text{Permitted}(p, \text{issue}, c_g)$ is acceptably valid. Because $G' \subseteq G$, every grant $g' \in G'$ is either in $R$ or there is a principal $p'$ such that $(p', g') \in L$ and $\varphi \Rightarrow \text{Permitted}(p', \text{issue}, c_{g'})$ is acceptably valid. It follows that $\varphi \Rightarrow \bigwedge_{g' \in G'} g'^{L,A,\emptyset,(E \cup \{e\})}$ is acceptably valid. Since $\bigwedge_{g' \in G'} g'^{L,A,\emptyset,(E \cup \{e\})} \Rightarrow \text{Permitted}(p, \text{issue}, c_g)$ is acceptably valid, $\varphi \Rightarrow \text{Permitted}(p, \text{issue}, c_g)$ is acceptably valid.

To show that $G \subseteq G^*_\#(L)$, we first observe that, for all $i$, $G^*_i \subseteq G^*_{i+1}$ and, if $G^*_i = G^*_{i+1}$, then $G^*_i = G^*_{i+j}$ for all $j > 0$. Since $G^*_0 = R$ and $G^*_i \subseteq R \cup \{g \mid$ for some principal $p$, $(p, g) \in L$}, it follows that $G^*_\#(L) = G^*_{\#(L)+1}$. To show that $G \subseteq G^*_\#(L)$, it suffices to show that for all licenses $(p, g) \in L$ such that $\varphi \Rightarrow \text{Permitted}(p, \text{issue}, c_g)$ is acceptably valid, $g \in G^*_\#(L)$, suppose by way of contradiction that there is a license $(p, g) \in L$ such that $\varphi \Rightarrow \text{Permitted}(p, \text{issue}, c_g)$ is acceptably valid and $g \not\in G^*_\#(L)$. Let $\varphi' = \bigwedge_{g' \in G^*\#(L)} g'^{L,A,\emptyset,(E \cup \{e\})}$. Since $G^*\#(L) = G^*_{\#(L)+1}$, the grant $g \not\in G^*_{\#(L)+1}$ so, by the definition of $G^*_{\#(L)+1}$, the
formula $\varphi' \implies \text{Permitted}(p, \text{issue}, c_g)$ is not acceptably valid. It follows that there is an acceptable model $m$ that satisfies $\varphi' \land \neg \text{Permitted}(p, \text{issue}, c_g)$ and is “most forbidding” in the sense that, for all principals $p'$ and grants $g'$, either $m$ does not satisfy $\text{Permitted}(p', \text{issue}, c_{g'})$ or the model $m'$ that does not satisfy $\text{Permitted}(p', \text{issue}, c_{g'})$ and is otherwise identical to $m$ does not satisfy $\varphi'$. Since $m$ satisfies $\neg \text{Permitted}(p, \text{issue}, c_g)$ and $\varphi \implies \text{Permitted}(p, \text{issue}, c_g)$ is acceptably valid, $m$ does not satisfy $\varphi$. Because $R \subseteq G'_{\#(L)}$ and $m$ satisfies $\varphi'$, $m$ satisfies $\land_{g' \in R} g'^{L,A,\emptyset,(E \cup \{e\})}$. So, there is a license $(p', g') \in L$ such that $m$ does not satisfy $(p', g')^{L,A,\emptyset,(E \cup \{e\})}$. If $\text{Permitted}(p', \text{issue}, g') \in E \cup \{e\}$, then $(p', g')^{L,A,\emptyset,(E \cup \{e\})} = \text{true}$, so $m$ satisfies $(p', g')^{L,A,\emptyset,(E \cup \{e\})}$. Thus, $\text{Permitted}(p', \text{issue}, g') \notin E \cup \{e\}$. But then $(p', g')^{L,A,\emptyset,(E \cup \{e\})} = \text{Permitted}(p', \text{issue}, c_{g'}) \implies g'^{L,A,\emptyset,(E \cup \{e\})}$. Since $m$ does not satisfy this formula, $m$ satisfies $\text{Permitted}(p', \text{issue}, c_{g'})$. By the construction of $m$, the model $m'$ that does not satisfy $\text{Permitted}(p', \text{issue}, c_{g'})$ and is otherwise identical to $m$ does not satisfy $\varphi'$. So there is a grant $g'' = \forall x_1 \ldots \forall x_n(d_{g''} \rightarrow e_{g''}) \in G'_{\#(L)}$ such that $m'$ does not satisfy $g'^{L,A,\emptyset,(E \cup \{e\})}$. Because $m$ satisfies $g'^{L,A,\emptyset,(E \cup \{e\})}$ and the two models $m$ and $m'$ differ only in their interpretation of $\text{Permitted}(p', \text{issue}, c_{g'})$, it follows from the translation of $g''$ that there is a substitution $\sigma$ such that $e_{g''} \sigma = \text{Permitted}(p', \text{issue}, g')$, $e_{g''} \sigma \notin E \cup \{e\}$, and $d_{g''} \sigma^{L,A,\emptyset,(E \cup \{e\})}$ is valid. So $g''^{L,A,\emptyset,(E \cup \{e\})} \implies \text{Permitted}(p', \text{issue}, c_{g'})$ is acceptably valid. Since $g'' \in G'_{\#(L)}$, $\varphi' \implies \text{Permitted}(p', \text{issue}, c_{g'})$ is acceptably valid, $g' \in G'_{\#(L)+1}$. Because $G'_{\#(L)+1} = G'_{\#(L)}$, the grant $g' \in G'_{\#(L)}$, and, since $m$ satisfies $\varphi'$, $m$ satisfies $g'^{L,A,\emptyset,(E \cup \{e\})}$. So $m$ satisfies $(p', g')^{L,A,\emptyset,(E \cup \{e\})}$, which contradicts the assumptions.

We next consider the complexity of computing $G = G'_{\#(L)}$. Let $L' = \{(p, g) \in L \mid \text{Permitted}(p, \text{issue}, g) \notin E \cup \{e\}\}$. Clearly, we can compute $L'$ in time
For all $k > 1$, let $L_k = \{(p, g) \in L' \mid g \notin G_k'\}$ and let $G_k'' = G_k' - G_{k-1}'$. We plan to compute $G_k''$ inductively. It will be useful in the induction to represent the elements of $G_k'$ in a splay tree. (Recall that a splay tree is a form of binary search tree such that $k$ insertions and searches can be done in a tree with at most $n$ nodes in time $O(k \log n)$ [ST83].) If $G_k'$ is represented as a splay tree, then we can compute $L_k'$ in time $O(|L| \log (|L| + |R|))$ (since $G_k' \subseteq L \cup R$).

For $0 < k < \#(L)$,

$$G_{k+1}'' = \{g \mid \text{for some principal } p, (p, g) \in L_k' \text{ and }$$

$$\forall g' \in G_k'' g'^{L,A,0,(E \cup \{e\})} \Rightarrow \text{Permitted}(p, \text{issue}, c_g) \text{ is acceptably valid}\}.$$

By Lemma B.0.15,

$$G_{k+1}'' = \bigcup_{(p, g) \in L_k'} \bigcup_{g' \in G_k''} \{g \mid g'^{L,A,0,(E \cup \{e\})} \Rightarrow \text{Permitted}(p, \text{issue}, c_g) \text{ is acceptably valid}\}.$$

Moreover, it follows from Lemma B.0.15 that, for $(p, g) \in L'$, $g'^{L,A,0,(E \cup \{e\})} \Rightarrow \text{Permitted}(p, \text{issue}, c_g)$ is acceptably valid iff the formula $d_{g'} \sigma$ is valid for some $A$-closed substitution $\sigma$ such that $e_{g'} \sigma = \text{Permitted}(p, \text{issue}, c_g)$, where $g' = \forall x_1 \ldots \forall x_n(d_{g'} \Rightarrow e_{g'})$. Given $(p, g) \in L'$ with $g \notin G_k'$ and $g' \in G_k''$, we can clearly check in time $c_1(|e_{g'}| + |(p, g)|)$ if there exists an $A$-closed substitution $\sigma$ such that $e_{g'} \sigma = \text{Permitted}(p, \text{issue}, g)$, where $c_1$ is a constant independent of $k$. If so, as in part (a), there are at most $\#(P_0)^n$ distinct formulas of the form $d_{g'} \sigma$ (since there are at most $\#(P_0)^n$ possible substitutions for the free variables in $d_{g'}$). It follows from Theorem B.0.18(a) that $d_{g'} \sigma^{L,A,0,(E \cup \{e\})}$ is valid iff $\text{Holds2}(d_{g'} \sigma, L, \emptyset) = \text{true}$. We show shortly that there is an execution of $\text{Query2}(e, L, R, E)$ that calls $\text{Holds2}(d_{g'} \sigma, L, \emptyset)$, so $\text{Holds2}(d_{g'} \sigma, L, \emptyset)$ is $h$-bounded. It follows from Lemma B.0.20 that we can determine if $\text{Holds2}(d_{g'} \sigma, L, \emptyset) = \text{true}$ in time $c_2 \max(|G_{d_{g'} \sigma}|, |L||P_0|^n)(|L||P_0|^n)^{h-2}(|L||P_0|^n + (h|L| + |G_{d_{g'} \sigma}|)(h + |L|))$, where $c_2$
is a constant independent of $k$ and $G_{d'\sigma}$ is a dag representing $d'\sigma$. As in the proof of part (b), we can obtain $G_{d'\sigma}$ from a dag $G_{d'}$ representing $d'$ by replacing every variable with a principal in $P_0$ or a resource mentioned in $\text{Permitted}(p, \text{issue}, g)$. So there is a dag $G_{d'\sigma}$ representing $d'\sigma$ such that $|G_{d'\sigma}| < |d'| + |g|$. Repeating this process for each of the at most $|P_0|^n$ formulas $d'\sigma$, it follows that we can check if $g^{L,A,\theta,\{E\cup[e]\}} \Rightarrow \text{Permitted}(p, \text{issue}, c_g)$ is acceptably valid in time $c_2|P_0|^n \max(|d'| + |g|, |L||P_0|^n)(|L||P_0|^n)^{h-2}(|L||P_0|^n + (h|L| + |d'| + |g|)(h + |L|))$.

Assuming we have already computed $L'_k$ and $G''_k$, we can repeat the process above for all $g' \in G''_k$ and $(p, g) \in L'_k$. It is not hard to show that we can compute $G''_{k+1}$ in time

$$\sum_{g' \in G''_k} \sum_{(p, g) \in L'_k} c_1(|e_{g'}| + |(p, g)|) + c_2|P_0|^n \max(|d'| + |g|, |L||P_0|^n)(|L||P_0|^n)^{h-2}.$$ 

$$= 2c_1|G''_k||L| + c_2|P_0|^n(|L||P_0|^n)^{h-2}(h + |L|).$$ 

$$= \sum_{g' \in G''_k} \sum_{(p, g) \in L} (|d'| + |g| + |L||P_0|^n)(|L||P_0|^n + h|L| + |d'| + |g|)$$

$$\leq 2c_1|G''_k||L| + c_2|P_0|^n(|L||P_0|^n)^{h-2}(h + |L|)2|G''_k||L|2|P_0|^n(|L||P_0|^n + h|L| + |G''_k| + |L|)$$

$$\leq 2c_1|G''_k||L| + 2c_2|G''_k|(|L||P_0|^n)^{h}(h + |L|)(|L||P_0|^n + h|L| + |G''_k| + |L|)$$

$$\leq c_3|G''_k|(|L||P_0|^n)^{h}(h + |L|)(|L||P_0|^n + h|L| + |G''_k| + |L|)$$

for some constant $c_3$. We can then build the splay tree for $G'_{k+1}$ by inserting the grants in $G''_k$ into the splay tree for $G'_k$; this can be done in time $O(|G''_k| \log(|L| + |R|))$.

Since $\bigcup_{k=1}^{\lfloor L \rfloor} G''_k \subseteq L$, the total time to compute $G''_1, \ldots, G''_k$ (ignoring the time to compute the sets $L'$ and $L'_k$, and to build the splay trees for $G'_k$) is at most

$$c_4|L|^2(|L||P_0|^n)^{h+1}h^2$$
for some constant $c_4$; i.e., it is $O(|L|^2(|L||P_0|^n)^{h+1}h^2)$.

Now taking into account the complexity of computing $L'$ and $L'_k$ and to build the splay trees, and using the observation that $\log(a+b) \leq \log(a+1) + \log(b+1)$, we get that the complexity for computing $G$ is

$$O(|L||E \cup \{e\}| + |L|^2\log(|R| + 1) + |L|^2(|L||P_0|^n)^{h+1}h^2).$$

It remains to show that if $g' = \forall x_1 \ldots \forall x_n(d_{g'} \rightarrow e_{g'}) \in G'_k - G'_{k-1}$, $(p,g) \in L'$ with $g \notin G'_k$, and $e_{g'}\sigma = \text{Permitted}(p,\text{issue},g)$ with $A$-closed substitution $\sigma$, then there is an execution $X$ of $\text{Query2}(e,L,R,E)$ that calls $\text{Holds2}(d_{g'}\sigma,L,\emptyset)$. By assumption, $e \notin E$, so $\text{Query2}(e,L,R,E)$ calls $\text{Auth2}(e,L,R,E)$, which calls $\text{Query2}(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\})$, which calls $\text{Auth2}(\text{Permitted}(p,\text{issue},gR),L,R,E \cup \{e\})$.

Since $(p,g)$ is in $L'$, the conclusion $\text{Permitted}(p,\text{issue},g)$ is not in $E \cup \{e\}$. It follows that $\text{Auth2}(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\})$ computes $G(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\},X)$ and, if $g'$ is in $G(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\},X)$, then every execution of $\text{Auth2}(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\})$ returns a set $D$ that includes $d_{g'}\sigma$. After $\text{Auth2}(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\})$ returns $D$, it is easy to see that some execution of $\text{Query2}(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\})$ calls $\text{Holds2}(d_{g'}\sigma,L,\emptyset)$. So, in short, it suffices to show that $g' \in G(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\},X)$. The proof is by induction on $k$. If $k = 0$, then $g' \in R \subseteq G(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\},X)$. If $k > 0$ then, by the induction hypothesis, $G'_{k-1} \subseteq G(\text{Permitted}(p,\text{issue},g),L,R,E \cup \{e\},X)$, so

$$\bigwedge_{\ell \in L} e^{L,A,\emptyset,\emptyset \cup \{e\}}^{\text{Permitted}(p,\text{issue},g)} \land \bigwedge_{g'' \in R} g''^{L',A,\emptyset,\emptyset \cup \{e\}}^{\text{Permitted}(p,\text{issue},g)} \Rightarrow \\
\bigwedge g'' \in G'_{k-1} \bigwedge g''^{L',A,\emptyset,\emptyset \cup \{e\}}^{\text{Permitted}(p,\text{issue},g)}$$

is acceptably valid. Since $g' \in G'_{k_0} - G'_{k_0-1}$, there is a grant $g'' \in G'_{k_0-1}$ and a principal $p'$ such that $(p',g'') \in$
\[ L \text{ and } g^{\mu L,A,\emptyset,E\cup\{e\}} \Rightarrow \text{Permitted}(p',\text{issue},g') \text{ is acceptably valid. Because } g' \in G'_k \text{ and } g \notin G'_k, \text{ } g \neq g' \text{ and, thus, it follows from the translation that } g^{\mu L,A,\emptyset,E\cup\{e\}\cup\text{Permitted}(p,\text{issue},g)} \Rightarrow \text{Permitted}(p',\text{issue},g') \text{ is acceptably valid. Putting the pieces together, there is a principal } p' \text{ such that } (p',g') \in L \text{ and } \bigwedge_{t \in L} (L,\emptyset) \cup (e)\cup\text{Permitted}(p,\text{issue},g) \land \bigwedge_{g'' \in R} g^{\mu L,A,\emptyset,E\cup\{e\}\cup\text{Permitted}(p,\text{issue},g)} \Rightarrow \text{Permitted}(p',\text{issue},g') \text{ is acceptably valid, so } g' \in G(\text{Permitted}(p,\text{issue},g), L, R \cup \{e\}, X). \]

We are now ready to prove Theorem 3.4.3.

**Theorem 3.4.3:** For fixed \( n \) and \( h \), if \((e, L, R, E) \in \mathcal{L}_0 \cap \mathcal{L}_1 \cap \mathcal{L}_2 \cap \mathcal{L}_3^h\), then determining whether \( \text{Query2}(e, L, R, E) \) returns \( \text{true} \) takes time \( O(|L||E| + (|R| + |L|)(|L|^{h-1}(|L| + |R| + |e|)^2)) \).

**Proof:** Let \( D \) be the output of \( \text{Auth2}(e, L, R, E) \). It is immediate from the description of \( \text{Query2} \) that \( \text{Query2}(e, L, R, E) = \text{true} \) iff there is some condition \( d \in D \) such that \( \text{Holds2}(d, L, \emptyset) = \text{true} \). So the output of \( \text{Query2}(e, L, R, E) \) can be determined in time \( T + \#(D)T' \), where \( T \) is the time needed to compute \( D \) and \( T' \) is the time needed to determine the output of \( \text{Holds2}(d, L, \emptyset) \) for a condition \( d \in D \). By Lemma B.0.21(c), \( \text{Holds2}(d, L, \emptyset) \) for a condition \( d \in D \). By Lemma B.0.21(c), \( T = c_1(|L||E \cup \{e\}| + |L|^2 \log(|R| + 1) + |L|^2(|L||P_0|^n)^{h+1}h^2) \) for some constant \( c_1 \). If \( n \) and \( h \) are treated as constants, then \( T = c'_1(|L||E \cup \{e\}| + |L|^2 \log(|R| + 1) + |L|^{h+3}) \) for some constant \( c'_1 \); i.e., \( T \) is \( O(|L||E \cup \{e\}| + |L|^2|R| + |L|^{h+3}) \).

By Lemma B.0.21(a), \( \#(D) \leq \#(P_0)^n(\#(R) + \#(L)) \). By Lemma B.0.20, \( T' \) is at most \( c_2(|G_d| + |L||P_0|^{n}(|L||P_0|^{n})^{h-2}(|L||P_0|^{n} + (h||L| + |G_d|)(h + |L|)) \), for some constant \( c_2 \). If \( n \) and \( h \) are treated as constants, then there is a constant \( c'_2 \) such
that \( T' \) is at most
\[
    c'(|G_d| + |L|h^{-2}(|L| + (|L| + |G_d|)|L|)
\]
\[
    = c'|L|h^{-1}(|G_d| + |L|)(1 + (|L| + |G_d|))
\]
\[
    = c'|L|h^{-1}(|G_d| + |L|)(2(|G_d| + |L|))
\]
\[
    \leq 2c'|L|h^{-1}(|G_d| + |L|)^2.
\]

Since, by Lemma B.0.21(b), \(|G_d| \leq |R| + |L| + |e|\), it follows that \( T' \leq 2c'|L|h^{-1}(2|L| + |R| + |e|)^2 \), i.e., \( O(|L|h^{-1}(|L| + |R| + |e|)^2) \).

Since \( \#(D) \leq \#(P_0)^n(\#(R) + \#(L)) \leq |P_0|^n(|R| + |L|) \), a straightforward computation shows that \( T + \#(D)T' \), the time needed to determine whether \( \text{Query2}(e, L, R, E) \) returns \( true \), is \( O(|L||E| + (|R| + |L|)(|L|h^{-1}(|L| + |R| + |e|)^2)) \).

\[\Box\]

**Theorem 3.6.2:** Let \((e, L, R, E)\) be a tuple in \( L_0 \cap L_1 \cap L_2 \cap L_3 \) extended to include negated \textit{Said} conditions and negated conclusions. The problem of deciding whether
\[
    \bigwedge_{\ell \in L} \ell_{L,A,S,E} \wedge \bigwedge_{g \in R} g_{L,A,S,E} \Rightarrow e_{L,A,S,E}
\]
is valid is NP-hard. This result holds even if \( e \), all of the licenses in \( L \), and all of the conclusions in \( E \) are in XrML, all but one of the grants in \( R \) is in XrML, and the one grant that is in in XrML\(^-\) is of the form \( \forall x_1 \ldots \forall x_n(\neg e) \).

**Proof:** The proof is by reduction of the 3-satisfiability problem. The reduction is identical to the reduction given in the proof for the case of \( L_0 \cap L_1 \cap L_2 \) in Theorem 3.4.2, except that \( R = \{ \forall x_1 \ldots \forall x_m((\bigwedge_{i=1}^n \text{Said}(p_i, e')) \rightarrow e'), \neg e' \} \).

To show that \( \text{Query2}(e, L, R, \emptyset) = true \) if and only if \( f \) is valid, we observe that \( \text{Query2}(e, L, R, \emptyset) = true \) if and only if \( L \) and \( R \) imply \( false \), which occurs if and only if \( \bigwedge_{i=1}^n \text{Said}(p_i, e')\sigma \) holds for some substitution \( \sigma \). The rest of the argument proceeds as in the proof of Theorem 3.4.2. \[\Box\]
Appendix C

Proofs for Chapter 4

In the proofs below, we use the notation \(\#(S)\) for the cardinality of set \(S\). We also use the notation \(f[t/x]\) for the capture-avoiding substitution of term \(t\) for variable \(x\) in formula \(f\).

**Theorem 4.3.1:** The problem of deciding, for a query \(q = (A, s, \text{act}, a, E)\), whether \(f_q^+\) is \(E\)-valid is decidable and NP-hard. Similarly, the problem of deciding, for a query \(q = (A, s, \text{act}, a, E)\), whether \(f_q^-\) is \(E\)-valid is decidable.

**Proof.** To prove decidability, we present an algorithm to determine whether \(f_q^+\) is \(E\)-valid. The algorithm first checks if \(E\) is inconsistent, by simply scanning \(E\). (Recall that \(E\) is inconsistent if and only if \(E\) has two conjuncts of the form \(\text{count}(s, \text{id}) = n\) and \(\text{count}(s, \text{id}) = n'\) with \(n \neq n'\).) If \(E\) is inconsistent, then there are no \(E\)-relevant models, \(f_q^+\) is trivially \(E\)-valid, and the algorithm returns “Yes”.

If \(E\) is consistent, then the set of \(E\)-relevant models is not empty, and the algorithm proceeds as follows. Let \(g\) be the formula obtained from \(f_q^+\) by replacing every subformula of the form \(\forall x(h)\) by \(\bigwedge_{s \in S}(h[s/x])\) and every subformula of the form \(\exists x(h)\) by \(\bigvee_{s \in S}(h[s/x])\), where \(S\) is the set of variable-free terms mentioned in \(q\) that have the same sort as \(x\). We claim that \(f_q^+\) is \(E\)-valid if and only if \(g\) is \(E\)-valid. We prove this claim by constructing \(g\) in steps; during this process, we consider in some detail the subformulas of the form \(\forall x(h)\) and \(\exists x(h)\) that can appear in \(f_q^+\).

- Let \(g_0\) be the formula obtained from \(f_q^+\) by replacing every subformula of the
form

$$\forall x((x = s_1 \lor \ldots \lor x = s_n \land g') \Rightarrow (g'' \Rightarrow \textit{Permitted}(x, act', a')))$$

by

$$\bigwedge_{s \in S}((x = s_1 \lor \ldots \lor x = s_n \land g') \Rightarrow (g'' \Rightarrow \textit{Permitted}(x, act', a')))[s/x],$$

where $S$ is the set of variable-free terms of sort $ Subjects$ mentioned in $q$. Since $\{s_1, \ldots, s_n\} \subseteq S$, it is easy to see that $f_q^+$ is $E$-valid if and only if $g_0$ is $E$-valid.

- Let $\Sigma$ be the set of substitutions $\sigma$ such that, for all variables $t$ of sort $Times$ in $g_0$, $\sigma(t)$ is a variable-free term of sort $Times$ that appears in $q$ and, for all other variables $x$, $\sigma(x) = x$. Note that $\Sigma$ is finite. Let $g_1$ be the formula obtained from $g_0$ by replacing every formula of the form $\exists t_1 \ldots \exists t_n(h)$, where every free variable of $h$ is of sort $Times$, with $\bigvee_{\sigma \in \Sigma}(h\sigma)$. It follows from the translation that, if $t$ is a free variable in $h$, then $h$ is a conjunction of formulas and one of those conjuncts has either the form $\textit{Paid}(r, I, t)$ or the form $\textit{Attributed}(s, t)$. It follows from the closed-world assumption that $g_0$ is $E$-valid if and only if $g_1$ is $E$-valid.

- It follows from the translation that every variable remaining in $g_1$ is of sort $ Subjects$; $g_1$ includes a subformula of the form $\forall x(h)$ if and only if $h$ can be written as $x \neq s_1 \land \ldots \land x \neq s_n \Rightarrow \neg \textit{Permitted}(x, act', a')$, where $s_i$ is a variable-free term in $q$, for $i = 1, \ldots, n$. Let $g_2$ be the formula obtained from $g_1$ by replacing every subformula of the form $\forall x(h)$ by $\bigwedge_{s \in S} h[s/x]$, where $S$ is the set of variable-free terms of sort $ Subjects$ mentioned in $q$. Note that $g_2 = g$. So, it remains to show that $g_1$ is $E$-valid if and only if $g_2$ is $E$-valid.
The “if” direction is trivial. For the “only if” direction, suppose by way of contradiction that \( g_1 \) is \( E \)-valid and \( g_2 \) is not. Note that \( g_1 \) is of the form \( g'_1 \Rightarrow \text{Permitted}(s, \text{act}, a) \) and \( g_2 \) is of the form \( g'_2 \Rightarrow \text{Permitted}(s, \text{act}, a) \) for appropriate formulas \( g'_1 \) and \( g'_2 \). Since \( g_2 \) is not \( E \)-valid, there is an \( E \)-relevant model \( M \) that satisfies \( g'_2 \land \neg \text{Permitted}(s, \text{act}, a) \). Let \( M' \) be the \( E \)-relevant model that is identical to \( M \), except that the domain of \( M' \) is limited to the closed terms that are mentioned in \( q \). It is easy to see that \( g'_2 \) holds in \( M' \) since the formula holds in \( M \), is variable-free, and mentions only those terms that appear in \( q \). It follows from the construction of \( g_2 \) that, because \( g'_2 \) holds in \( M' \), \( g'_1 \) holds in \( M' \). Since, by construction, \( M' \) does not satisfy \( \text{Permitted}(s, \text{act}, a) \), \( M \) does not satisfy \( g_1 \), which gives us the desired contradiction.

Since \( g \) is variable-free, the algorithm proceeds by replacing every \( \text{Permitted} \)-free literal appearing in \( g \) by either \textit{true} or \textit{false} depending on \( E \) and the standard interpretations of =, < and \( \leq \). Let \( h \) be the formula obtained from \( g \) by doing this replacement. Clearly, \( g \) is \( E \)-valid if and only if \( h \) is \( E \)-valid. Moreover, since \( \text{Permitted} \) is the only predicate symbol appearing in \( h \), \( h \) is \( E \)-valid if and only if \( h \) is valid. The algorithm determines the validity of \( h \) by checking if \( h \) holds for all assignments of \textit{true} or \textit{false} to the \( \text{Permitted} \) literals in \( h \) (where a positive literal \( \ell \) is not given the same assignment as \( \neg \ell \)). Obviously, \( h \) is valid if it holds under every substitution and is not valid otherwise.

The same strategy can be used to derive an algorithm that determines the \( E \)-validity of \( f_q^- \).

We now reduce the 3-satisfiability problem to the problem of determining whether \( f_q^+ \) is \( E \)-valid for an appropriate query \( q \), thereby showing that the latter
problem is NP-hard. Let $\varphi = C_1 \land \ldots \land C_n$ be a formula in propositional logic, where each $C_i$ is a clause with three disjuncts. Without loss of generality, we assume that no conjunct $C_i$ is valid. Let $P_1, \ldots P_m$ be the primitive propositions mentioned in $\varphi$. We want to determine if $\varphi$ is satisfiable.

Let $s_0, \ldots, s_m$ be subjects and let $a$ be an asset. For each conjunct $C_i = L_1 \lor L_2 \lor L_3$ of $\varphi$, let $agr_i$ be the agreement

$$\text{agreement for } \{s_0, \ldots, s_m\} \text{ about } a \text{ with } \text{and}[prq_1, prq_2, prq_3] \Rightarrow \text{display},$$

where

$$prq_j \triangleq \begin{cases} & \text{and}[s_0, \text{not}[s_k \Rightarrow \text{print}]] \quad \text{if } L_j \text{ is } P_k \\ & \text{and}[s_0, \text{xor}[	ext{true, not}[s_k \Rightarrow \text{print}]]] \quad \text{if } L_j \text{ is } \neg P_k. \end{cases}$$

Let $q$ be the query $\{(agr_1, \ldots, agr_n), s_0, \text{display, a, E}\}$, where $E$ is the empty environment (i.e., true). We claim that $\varphi$ is satisfiable if and only if $f_q^+$ is not $E$-valid. For every assignment $A$ of truth values to $P_1, \ldots, P_m$, let $M_A$ be the $E$-relevant model that satisfies $\neg \text{Permitted}(s_i, \text{print}, a)$ if and only if $A$ assigns $P_i$ to false or $s_i = 0$. It is not hard to show that a truth assignment $A$ satisfies a conjunct $c_i$ of $\varphi$ if and only if $M_A$ satisfies $[[agr_i]]$. The key observation is that, for each conjunct $C_i = L_1 \lor L_2 \lor L_3$ of $\varphi$, we can write $[[agr_i]]$ as

$$f_{i,1} \land f_{i,2} \land f_{i,3} \Rightarrow \text{Permitted}(s_0, \text{display}, a),$$

where

$$f_{i,j} = \begin{cases} & \neg \text{Permitted}(s_k, \text{print}, a) \quad \text{if } L_j \text{ is } P_k \\ & \text{Permitted}(s_k, \text{print}, a) \quad \text{if } L_j \text{ is } \neg P_k. \end{cases}$$

So, if $\varphi$ is satisfiable, then there is a truth assignment $A$ that satisfies $\varphi$, the model $M_A$ satisfies $\bigwedge_{agr \in A}[agr] \land \neg \text{Permitted}(s_0, \text{display}, a)$, and, thus, $f_q^+$ is not $E$-valid. If $\varphi$ is not satisfiable then, for every truth assignment $A$, $M_A$ does not
satisfy some \([agr_i]\), so \(M_A\) satisfies \(f_q^+\). Let \(\mathcal{M}\) be the set of models \(M\) such that, for all truth assignments \(A\), \(M \neq M_A\). It is not hard to see that every model in \(\mathcal{M}\) satisfies \(\text{Permitted}(s_0, \text{display}, a)\), thereby satisfying \(f_q^+\). Since every \(E\)-relevant model satisfies \(f_q^+\), the formula is \(E\)-valid. 

The following result is used in Lemmas 4.3.2 and 4.3.4.

**Lemma C.0.22.** Suppose that \(f\) is a \(\text{Permitted}\)-free formula and \(E\) is an environment such that the set of \(E\)-relevant models is nonempty. Then \(f\) holds in at least one \(E\)-relevant model if and only if \(f\) is \(E\)-valid.

**Proof.** Follows immediately from the definitions. 

Given a policy set \(ps\), let \(S_{ps}^+\) be the set of tuples \((prq, I, prq', id, act')\) such that \(ps\) mentions the policy set \(prq \rightarrow p\) or \(prq \leftarrow p\), \(I\) is the set of policy identifiers appearing in \(p\), and \(p\) mentions the policy \(prq' \rightarrow_{id} act'\). Finally, let \(S_{ps}^-\) be the set of actions such that an action \(act'\) is in \(S_{ps}^-\) if and only if \(ps\) mentions an exclusive policy set that mentions a policy of the form \(prq \Rightarrow act'\).

**Lemma C.0.23.** Suppose \(agr\) is an agreement of the form 

\[
\text{agreement for prin}_u \text{ about } a \text{ with } ps.
\]

Then \([agr]\) holds in model \(M\) if and only if

(a) \(M\) satisfies \(\neg \text{Permitted}(s', act', a)\) for every \(act' \in S_{ps}^-\) and every \(s' \not\in \text{subjects}(\text{prin}_u)\), and

(b) for every \((prq, I, prq', id, act') \in S_{ps}^+\) and \(s' \in \text{subjects}(\text{prin}_u)\), either \(M\) satisfies \(\text{Permitted}(s', act', a)\) or \(M\) does not satisfy \([prq]_{s'}^{I, \text{prin}_u} \land [prq']_{s'}^{(id), \text{prin}_u}\).

**Proof.** Immediate by the definition of \(S_{ps}^+\) and \(S_{ps}^-\) and the translation \([\cdot]\).
Lemma 4.3.2: There are algorithms that, given a query \( q = (\{agr\}, s, act, a, E) \) in \( Q_1 \):

(a) determine whether \( f_q^+ \) is \( E \)-valid in time \( O(|E||agr|^6) \), and

(b) determine whether \( f_q^- \) is \( E \)-valid in time \( O(|E| + |agr|) \).

Proof. Let \( agr \) be an agreement agreement for \( prin_u \) about \( a' \) with \( ps \).

For part (a), we claim that \([agr] \Rightarrow \text{Permitted}(s, act, a)\) is \( E \)-valid if and only if the set of \( E \)-relevant models is empty, or all of the following conditions hold:

(i) \( s \in \text{subjects}(prin_u) \),

(ii) \( a' = a \), and

(iii) there is a tuple \((prq, I, prq', id, act) \in S^+_{ps}\) such that \([prq]^{I}_{s,\text{prim}_u} \land [prq']^{\{id\}_{s,\text{prim}_u}}\) is \( E \)-valid.

• For the “if” direction, if the set of \( E \)-relevant models of \( agr \) is empty, then the formula \([agr] \Rightarrow \text{Permitted}(s, act, a)\) is trivially \( E \)-valid. If (i), (ii), and (iii) hold, then it is immediate from the translation that \([agr] \Rightarrow \text{Permitted}(s, act, a)\) is \( E \)-valid.

• For the “only if” direction, suppose by way of contradiction that the formula \([agr] \Rightarrow \text{Permitted}(s, act, a)\) is \( E \)-valid, the set of \( E \)-relevant models of \( agr \) is not empty, and either (i), (ii), or (iii) does not hold. Because the set of \( E \)-relevant models is not empty, there is a model \( M \) that is \( E \)-relevant and that satisfies \( \text{Permitted}(t_1, t_2, t_3) \) if and only if \( t_1 \in \text{subjects}(prin_u) \), \( t_3 = a' \), and \( \text{Permitted}(t_1, t_2, t_3) \neq \text{Permitted}(s, act, a) \), for all closed terms \( t_1, t_2, \) and \( t_3 \) of the appropriate sorts. We claim that \( M \) satisfies \([agr] \), thus
contradicting the assumption that \([agr] \Rightarrow \textbf{Permitted}(s, act, a)\) is \(E\)-valid.

By Lemma C.0.23, it suffices to show that C.0.23(a) and C.0.23(b) hold. C.0.23(a) follows from the construction of \(M\). If (i) or (ii) does not hold, then \(M\) satisfies \(\textbf{Permitted}(s', act', a')\), for every tuple \((prq, I, prq', I', act') \in S^+_ps\), so C.0.23(b) holds. Suppose that (iii) does not hold. Then, for each tuple \((prq, I, prq', I', act') \in S^+_ps\) and subject \(s' \in \text{subjects}(prin_u)\), either \(s' \neq s\), in which case \(M\) satisfies \(\textbf{Permitted}(s, act, a')\); \(act' \neq act\), in which case \(M\) satisfies \(\textbf{Permitted}(s', act', a'); or \(s' = s, act' = act, and f = \{prq\}^{I, prin_u}_s \land \{prq'\}^{\{id\}, prin_u}_s\) is not \(E\)-valid. It follows from Lemma C.0.22 that \(f\) does not hold in \(M\) because it is \(\textbf{Permitted}\)-free (neither \(prq\) nor \(prq'\) mention a policy set), so C.0.23(b) holds again.

It follows that we can determine the \(E\)-validity of \([agr] \Rightarrow \textbf{Permitted}(s, act, a)\) by running the following algorithm: determine whether the set of \(E\)-relevant models is empty; if so, return “Yes”, otherwise check conditions (i), (ii), and (iii); if all hold, then return “Yes”, else return “No”. The set of \(E\)-relevant models is non-empty if and only if \(E\) is inconsistent, which can be checked in time \(O(|E|)\). We can check whether (i) and (ii) hold in time \(O(|agr|)\). We can also compute \(S^+_ps\) in time \(O(|agr|)\). Finally, the cardinality of \(S^+_ps\) is less than \(|agr|\). We show that, for each tuple \((prq, I, prq', id, act) \in S^+_ps\), we can determine whether \([prq]^{I, prin_u}_s \land [prq']^{\{id\}, prin_u}_s\) is \(E\)-valid in time \(O(|E||agr|)\), so the total running time of the algorithm is \(O(|E||agr|^5)\).

Using the translation as a guide, we can construct an algorithm for determining whether \([prq]^{I, prin_u}_s\) (or \([prq']^{\{id\}, prin_u}_s\)) is \(E\)-valid. The first step is to rewrite the prerequisites \(prq\) and \(prq'\) so that they do not contain nested \textbf{ForEachMember}
constraints. Examining the translation, it is clear that the constraint

\[ \text{forEachMember}[^{prin}; \text{forEachMember}[^{prin'}; cons'], cons] \]

translates to a formula that is logically equivalent to the translation of

\[ \text{and}[^{\text{forEachMember}[^{prin}; cons], \text{forEachMember}[^{prin'}; cons']}. \]

Generalizing this idea, we can rewrite, in time \( O(|prq|) \), the prerequisite \( prq \) to an equivalent prerequisite \( prq_0 \) of size \( O(|prq|) \) that does not contain nested \( \text{forEachMember} \) constraints, and similarly rewrite \( prq' \) to an equivalent prerequisite \( prq'_0 \). We then determine whether \( [prq_0]_s^{I, prin_u} \) and \( [prq'_0]_s^{I, prin_u} \) are \( E \)-valid.

Checking whether a prerequisite is \( E \)-valid is straightforward. The key observation is that, if \( prq_0 \) is a constraint or a condition, then \( [prq_0]_s^{I, prin_u} \) is a variable-free formula in which each literal is equivalent to \( true \) or \( false \) based on \( E \); the formula is \( E \)-valid iff it is equivalent to \( true \). If \( prq_0 \) is a requirement, then determining the \( E \)-validity of \( [prq_0]_s^{I, prin_u} \) is straightforward, unless \( prq_0 \) is of the form \( \text{inSeq}[req_1, \ldots, req_m] \); in this case, we determine the \( E \)-validity of \( [prq_0]_s^{I, prin_u} \) by considering the events (payments made and attributions given) in the order in which those events occurred as described in \( E \). It is not hard to write a naive recursive algorithm that determines the \( E \)-validity of \( [prq_0]_s^{I, prin_u} \) and \( [prq'_0]_s^{I, prin_u} \), and that runs in time \( O(|E||agr|^5) \). (The assumption that there are no nested \( \text{forEachMember} \) in \( prq \) is crucial to ensure this running time; without this assumption, running time may be exponential in the size of the prerequisite.) We leave the straightforward details to the reader.

For part (b), we claim that \( [agr] \Rightarrow \neg \text{Permitted}(s, act, a) \) is \( E \)-valid if and only if the set of \( E \)-relevant models is empty or all of the following conditions hold:

(i) \( s \notin \text{subjects}(prin_u) \),
(ii) \( a' = a \), and

(iii) \( agr \) includes an exclusive policy set that mentions a policy of the form \( prq \Rightarrow act \).

- For the “if” direction, if the set of \( E \)-relevant models is empty, then the formula \([agr] \Rightarrow \neg \text{Permitted}(s, act, a)\) is trivially \( E \)-valid. If (i), (ii), and (iii) hold then \([agr]\) can be written as a conjunction of formulas, one of which says that every subject who is not mentioned in prin is forbidden to do \( act \) to \( a \), so \([agr]\) \( \Rightarrow \neg \text{Permitted}(s, act, a)\) is again \( E \)-valid.

- For the “only if” direction, suppose that the set of \( E \)-relevant models is non-empty. It follows that there is an \( E \)-relevant model \( M \) such that, for all closed terms \( t_1, t_2, \) and \( t_3 \) of the appropriate sorts, \( M \) satisfies \( \neg \text{Permitted}(t_1, t_2, t_3) \) if and only if \( t_1 \not\in \text{subjects}(prin) \), \( t_3 = a' \), and \( \neg \text{Permitted}(t_1, t_2, t_3) \neq \neg \text{Permitted}(s, act, a) \). We claim that, if at least one of the conditions ((i), (ii), or (iii)) does not hold, then \([agr]\) holds in \( M \) and, thus, \([agr] \Rightarrow \neg \text{Permitted}(s, act, a)\) is not \( E \)-valid. To prove the claim, observe that, by Lemma C.0.23, it suffices to show that C.0.23(a) and C.0.23(b) hold. Since \( M \) satisfies \( \text{Permitted}(t_1, t_2, t_3) \) for all closed terms such that \( t_1 \in \text{subjects}(prin) \), C.0.23(b) holds. If either (i) or (ii) does not hold, then \( M \) satisfies \( \neg \text{Permitted}(t_1, t_2, t_3) \) if and only if \( t_1 \not\in \text{subjects}(prin) \) and \( t_3 = a' \). It follows that, for all subjects \( s' \not\in \text{subjects}(prin) \) and all actions \( act'' \in S^{-}_{ps} \), \( M \) satisfies \( \neg \text{Permitted}(s', act'', a') \); so C.0.23(a) holds. If (iii) does not hold, then \( act \not\in S^{-}_{ps} \). It follows from the construction of \( M \) that, for each action \( act'' \neq act \) and each subject \( s' \not\in \text{subjects}(prin) \), \( M \) satisfies \( \neg \text{Permitted}(s', act'', a') \), so C.0.23(b) holds.
Thus, we can determine the $E$-validity of $\neg \text{Permitted}(s, act, a)$ by running the following algorithm: determine whether the set of $E$-relevant models is empty; if so, return “Yes”, otherwise check conditions (i), (ii), and (iii); if all hold, then return “Yes”, else return “No”. Checking that the set of $E$-relevant models is empty can be done in time $O(|E|)$. Checking conditions (i), (ii), and (iii) can be done in time $O(|A|)$. \qed

Lemma 4.3.4: Suppose that $q = (A, s, act, a, E)$ is a query in $Q_1$ such that $\bigwedge_{agr \in A}[agr]$ is satisfied in at least one $E$-relevant model. For every $agr \in A$, let $q_{agr}$ be the query ($\{agr\}, s, act, a, E$). Then

(a) $f_q^+$ is $E$-valid if and only if $f_{q_{agr}}^+$ is $E$-valid for some $agr \in A$, and

(b) $f_q^-$ is $E$-valid if and only if $f_{q_{agr}}^-$ is $E$-valid for some $agr \in A$.

Proof. For part (a), the “if” direction is trivial. For the “only if” direction, suppose by way of contradiction that $\bigwedge_{agr \in A}[agr] \Rightarrow \text{Permitted}(s, act, a)$ is $E$-valid and $[agr] \Rightarrow \text{Permitted}(s, act, a)$ is not $E$-valid for every $agr \in A$. By assumption, there is an $E$-relevant model $M$ that satisfies $\bigwedge_{agr \in A}[agr]$. Let $M'$ be the model that is identical to $M$ except that $M'$ satisfies $\neg \text{Permitted}(s, act, a)$. Because $M$ is $E$-relevant and $M'$ differs from $M$ only on the interpretation of $\text{Permitted}$, $M'$ is $E$-relevant. Since $M'$ satisfies $\neg \text{Permitted}(s, act, a)$ and, by assumption, $\bigwedge_{agr \in A}[agr] \Rightarrow \text{Permitted}(s, act, a)$ is $E$-valid, there is an agreement $agr$ in $A$ such that $M'$ does not satisfy $[agr]$. We now show that $[agr]$ implies $\text{Permitted}(s, act, a)$, which contradicts the assumptions. Because no agreement in $A$ mentions a condition of the form $\text{not}[ps]$, it follows from the translation that we can write $[agr]$ as $\forall x(f_1) \land \cdots \land \forall x(f_n)$, where each $f_i$ is of the form $g \Rightarrow$
(¬)\textbf{Permitted}(x, act', a'), \ g \ \text{is Permitted-free, and both} \ act' \ \text{and} \ a' \ \text{are closed terms of the appropriate sorts. Because} \ [agr] \ \text{holds in} \ M \ \text{and does not hold in} \ M', \ \text{there exists integer} \ i \ \text{such that} \ f_i = g \Rightarrow \textbf{Permitted}(x, act, a) \ \text{and} \ g[s/x] \ \text{is satisfied in} \ M'. \ \text{Since} \ g[s/x] \ \text{is Permitted-free and is satisfied in a} \ E\text{-relevant model, it follows from Lemma C.0.22 that} \ g[s/x] \ \text{is} \ E\text{-valid. Putting the pieces together, we can write} \ [agr] \ \text{as} \ \forall x(h \wedge (g \Rightarrow \textbf{Permitted}(x, act, a))), \ \text{for an appropriate formula} \ h, \ \text{and} \ g[s/x] \ \text{is} \ E\text{-valid. It readily follows that} \ [agr] \Rightarrow \textbf{Permitted}(s, act, a) \ \text{is} \ E\text{-valid.}

The proof for part (b) is nearly identical to the proof for part (a); in fact, the former can be obtained from the latter by replacing every occurrence of \textbf{Permitted} by \textbf{¬Permitted} and vice versa.

\textbf{Lemma 4.3.5:} \ There is an algorithm that, given a query \ \( q = (A, s, act, a, E) \) in \ \( Q_1 \), determines whether \( \bigwedge_{agr \in A}[agr] \) is satisfied in at least one \( E\text{-relevant model} \) in time \( O(|E||A|^8) \).

\textbf{Proof.} We claim that \( \bigwedge_{agr \in A}[agr] \) holds in an \( E\text{-relevant model} \) if and only if

(i) the set of \( E\text{-relevant models} \) is not empty, and

(ii) for every pair of agreements

\begin{align*}
\textbf{agreement for} \ prin_u \ \text{about} \ a \ \text{with} \ ps, \ \text{and} \\
\textbf{agreement for} \ prin'_u \ \text{about} \ a' \ \text{with} \ ps'
\end{align*}

in \( A \), either

(a) \( a \neq a' \), or
(b) for all actions \( \text{act} \in S^{-}_{ps} \), tuples \((prq, I, prq', id, act) \in S^{+}_{ps'}\), and subjects 
\( s \in \text{subjects}(\text{prin'u}) \setminus \text{subjects}(\text{prin}_u) \), 
\([\text{prq}]^{I}_{s, \text{prin'}_u} \land [\text{prq'}]^{(id)}_{s, \text{prin'}_u}\) is not 
\( E\)-valid.

For the “if” direction, observe that if (i) holds, then there is an \( E\)-relevant model 
\( M \) such that, for all closed terms \( t_1, t_2 \), and \( t_3 \) of the appropriate sort, \( M \) satis-
flies \( \neg\text{Permitted}(t_1, t_2, t_3) \) if and only if there is an agreement \( agr \) of the form

**agreement for prin\(_u\) about a with ps in A** such that \( t_1 \notin \text{subjects}(\text{prin}_u) \), \( ps \) includes an exclusive policy set that mentions a policy of the form \( prq \Rightarrow t_2 \), and 
\( t_3 = a \). It is not hard to see that, if (ii) holds, then \( M \) satisfies \( \bigwedge_{agr \in A}[agr] \) and we are done. For the “only if” direction, observe that, if (i) does not hold, 
then \( \bigwedge_{agr \in A}[agr] \) clearly does not hold in an \( E\)-relevant model. If (ii) does not
hold, then there is a subject \( s \), action \( \text{act} \), and asset \( a \) such that, for an agree-
ment \( agr \in A, [agr] \Rightarrow \text{Permitted}(s, \text{act}, a) \) is \( E\)-valid and, for an agreement
\( agr' \in A, [agr] \Rightarrow \neg \text{Permitted}(s, \text{act}, a) \) is \( E\)-valid. Since no model can sat-
sify both \( \text{Permitted}(s, \text{act}, a) \) and \( \neg \text{Permitted}(s, \text{act}, a) \), no \( E\)-relevant model

can satisfy both \([agr]\) and \([agr']\), so \( \bigwedge_{agr \in A}[agr] \) does not hold in any \( E\)-relevant
model.

We can determine whether (i) holds in time \( O(|E|) \), since (i) holds if and only
if \( E \) is consistent. To check whether (ii) holds, we first construct the sets \( S^{+}_{ps} \) 
and \( S^{-}_{ps} \), which takes time \( O(|A|) \); then we compare all \(|A|^2 \) pairs of agreements.
For every agreement \( agr \) in every pair of agreements, we determine whether cer-
tain prerequisites hold; this takes time \( O(|E||agr|^6) \), because there are at most
\(|agr| \) prerequisites per agreement \( agr \) and evaluating each requirement takes time
\( O(|E||agr|^5) \), as shown in the proof of Theorem 4.3.2. Since \(|agr| \leq |A| \) for every
agreement \( agr \in A \), we get a total running time of \( O(|E||A|^8) \). \( \square \)
Theorem 4.3.6: There is an algorithm that, given a query $q = (A, s, act, a, E)$ in $Q_1$, computes the answer to $q$ in time $O(|E||A|^8)$.

Proof. First, run the algorithm of Lemma 4.3.5 to determine if $\bigwedge_{agr \in A} [agr]$ is satisfied in at least one $E$-relevant model. This can be done in time $O(|E||A|^8)$. If the result is “No”, then return “Query inconsistent”. If the result is “Yes”, then use the algorithms of Lemma 4.3.2 to check whether $f^+_{q_{agr}}$ and $f^-_{q_{agr}}$ are $E$-valid for each query $q = (\{agr\}, s, act, a, E)$ such that $agr \in A$. This can be done in time $O(|E||A|^7)$: there are less than $|A|$ agreements in $A$, and for every $agr \in A$, $|agr| \leq |A|$. By Lemma 4.3.4, $f^+_q$ is $E$-valid if and only if $f^-_{q_{agr}}$ is $E$-valid for an $agr \in A$, and similarly for $f^-_q$. Thus, if $f^+_{q_{agr}}$ is $E$-valid for an $agr \in A$, and $f^-_{q_{agr}}$ is not $E$-valid for all $agr \in A$, then return “Permission granted”. Similarly, if $f^-_{q_{agr}}$ is $E$-valid for an $agr \in A$, and $f^+_{q_{agr}}$ is not $E$-valid for all $agr \in A$, then return “Permission denied”. Otherwise, return “Permission unregulated”. ☐
BIBLIOGRAPHY


