Diss. ETH No. 23968

Access Control with Formal Security Guarantees

A thesis submitted to attain the degree of

Doctor of Sciences of ETH ZURICH

presented by

PETAR TSANKOV

Master of Science in Computer Science,
ETH Zurich

born on 05.05.1984

citizen of Bulgaria and the USA

accepted on the recommendation of

Prof. Dr. David Basin, examiner
Prof. Dr. Lujo Bauer, co-examiner
Prof. Dr. Srdjan Capkun, co-examiner
Prof. Dr. Michael Huth, co-examiner
Dr. Mohammad Torabi Dashti, co-examiner

2016

Abstract

Access-control systems are essential for protecting sensitive resources, be it digital or physical assets. Modern access-control systems are, however, complex software artifacts. They consist of multiple interconnected components that are subject to communication and component failures. These often influence access decisions in surprising and unintended ways. Furthermore, these components collectively enforce nontrivial decentralized policies, i.e. policies issued and managed by multiple principals, which are difficult to get right. Constructing *correct* access-control systems, i.e. systems that grant and deny access in conformance to all access-control requirements, is therefore a challenging task.

In this thesis, we tackle the problem of constructing correct access-control systems. We present a novel access-control framework that can be used to (i) *specify* decentralized access-control policies, which are policies that require both authority delegation and policy composition, (ii) *verify* access-control systems against security requirements in the presence of an active attacker who can selectively cause communication and component failures, and (iii) *synthesize* access-control policies for distributed enforcement points. We also develop an efficient interpreter that supports our policy specification language and can be used to construct access-control systems that directly enforce (verified) policies.

Our first contribution is a novel logic-programming language, called BELLOG, which extends stratified Datalog over the four truth values from Belnap's logic. We show that BELLOG can formalize access-control policies with authority delegation and policy composition, such as those based on the recent XACML v3.0 standard. Furthermore, we show that the verification of BELLOG policies against access-control requirements is readily reduced to BELLOG's core decision problems.

Our second contribution is a comprehensive and systematic investigation of the role of failure handling in access-control systems. We show that communication and component failures often affect access decisions in nontrivial and unintended ways, resulting in insecure systems. The verification of fail-security requirements, which define how access-control systems must handle failures, is therefore imperative in practice. We define a realistic attacker model tailored to failure scenarios and we demonstrate how the BELLOG access-control framework is used to verify access-control systems against fail-security requirements in the presence of our attacker.

Our third contribution is a synthesis approach for automatically constructing local access-control policies enforced by multiple, distributed enforcement points, such as the locks in a building. We focus on access-control policies for physical spaces, like office building and airports, where the requirements are often formulated in terms of system-wide requirements. For example, "there is an authorized path to exit the building from every room." We illustrate the effectiveness of our synthesis approach in three real-world case studies where we synthesize the policies for a university building, a corporate building, and an airport terminal.

To sum up, our contributions make up a comprehensive framework that can be used to solve key problems in access control that go beyond the specification and analysis of individual policies. Namely, it is the first framework that can be used to verify the correctness of failure handlers in access-control systems. Furthermore, it is the first framework that leverages program synthesis techniques to solve the problem of automatic construction of access-control policies from system-wide requirements.

Zusammenfassung

Zutrittskontrollsysteme sind wichtig für den Schutz sensibler Ressourcen, seien es digitale oder physikalische Vermögenswerte. Moderne Zutrittskontrollsysteme sind jedoch komplexe Softwareartefakte. Sie bestehen aus mehreren miteinander verbundenen Komponenten, die Kommunikations- und Komponentenfehlern unterliegen. Diese beeinflussen oft überraschende und unbeabsichtigte Zugriffsentscheidungen. Darüber hinaus erzwingen diese Komponenten kollektiv nicht-triviale dezentralisierte Richtlinien, die von mehreren Richtlinienverwaltern ausgegeben und gepflegt werden. Es ist schwierig diese Richtlinien korrekt zu definieren. Der Aufbau korrekter Zugangskontrollsysteme, d.h. Systeme, die den Zugang in Übereinstimmung mit allen Zugangsanforderungen gewähren und verweigern, ist daher eine anspruchsvolle Aufgabe.

In dieser Arbeit befassen wir uns mit dem Problem der Konstruktion korrekter Zugangskontrollsysteme. Wir präsentieren ein neuartiges Zugangskontroll-Framework, das verwendet werden kann, um (i) dezentralisierte Zugangskontrollrichtlinien festzulegen, bei denen es sich um Richtlinien handelt, die sowohl Bevollmächtigungen als auch Komposition erlauben, (ii) die Kontrolle der Zugangskontrollsysteme vor den Sicherheitsanforderungen in Anwesenheit eines aktiven Angreifers, der selektiv Kommunikations- und Komponentenfehler verursachen kann, zu schützen und (iii) die Synthese von Zugriffskontrollrichtlinien für verteilte Durchsetzungspunkte ermöglicht. Weiterhin entwickeln wir einen effizienten Interpreter, der unsere Policy-Spezifikationssprache unterstützt und für die Konstruktion von Zutrittskontrollsystemen verwendet werden kann, die direkt (verifizierte) Richtlinien umsetzen.

Unser erster Beitrag ist eine neuartige Logik-Programmiersprache namens BelLog, die stratifiziertes Datalog mit den vier Wahrheitswerten der Logik von Belnap erweitert. Wir zeigen, dass wir Zugriffskontrollrichtlinien mit Bevollmächtigung und Komposition in BelLog formalisieren können, wie beispielsweise basierend auf dem aktuellen XACML-Standard. Darüber hinaus zeigen wir, dass die Überprüfung der BelLog-Richtlinien hinsichtlich von Zugangsregelungsanforderungen leicht auf die Kernentscheidungsprobleme von BelLog reduziert werden können.

Unser zweiter Beitrag ist eine umfassende und systematische Untersuchung der Rolle des Ausfallmanagements in Zutrittskontrollsystemen. Wir zeigen, dass Kommunikations- und Komponentenfehler häufig Zugriffsentscheidungen auf nichttriviale und unbeabsichtigte Weise beeinflussen,

was zu unsicheren Systemen führt. Die Überprüfung der Fail-Security-Anforderungen, die definieren, wie Zutrittskontrollsysteme mit Fehlern umgehen müssen, ist daher in der Praxis zwingend erforderlich. Wir definieren ein realistisches Angreifermodell, das auf Fehlerszenarien zugeschnitten ist, und wir zeigen, wie das BelLog-Zugriffssteuerungs-Framework verwendet wird, um in Gegenwart unseres Angreifers Zugriffskontrollsysteme gegen Fehlersicherheitsanforderungen zu verifizieren.

Unser dritter Beitrag ist ein Syntheseansatz für die automatische Erstellung lokaler Zugangskontrollstrategien, die durch mehrere, verteilte Durchsetzungspunkte, wie die Schlösser in einem Gebäude, gewährleistet werden. Wir konzentrieren uns auf Zugangskontrollmassnahmen für physische Räume wie Bürogebäude und Flughäfen, in denen die Anforderungen oft systemweit formuliert werden, wie zum Beispiel: "Gibt es einen autorisierten Weg, um das Gebäude aus jedem Raum zu verlassen."Wir veranschaulichen die Wirksamkeit unseres Syntheseansatzes in drei realen Fallstudien, in denen wir die Richtlinien für jeweils ein Hochschulgebäude, ein Firmengebäude und ein Flughafenterminal erstellen.

Zusammenfassend bilden unsere Beiträge einen umfassenden Rahmen für die Lösung von Schlüsselproblemen in der Zugangskontrolle, die über die Spezifikation und Analyse einzelner Richtlinien hinausgehen. Insbesondere ist es das erste Framework, das verwendet werden kann, um die Korrektheit von Fehlerbehandlungsroutinen in Zugriffssteuerungssystemen zu überprüfen. Darüber hinaus ist es das erste Framework, das Programmsynthesetechniken verwendet, um das Problem der automatischen Erstellung von Zugangskontrollrichtlinien ausgehend von systemweiten Anforderungen zu lösen.

Acknowledgments

I would like to acknowledge my advisor, David Basin, who always provided prompt, insightful feedback to any of my ideas and writings. His guidance in formulating abstract, clean problem statements and his advices on technical writing were truly helpful in learning how to effectively communicate research ideas on paper. I would also like to acknowledge Mohammad Torabi Dashti, who always attentively listened to all my ideas and provided thoughtful, critical feedback on all technical pieces (including proofs) of this thesis. I want to also thank Srdjan Marinovic for his guidance during the early years of my studies and for all his creative ideas, which are the source from where research begins.

I would also like to thank the additional reviewers of my thesis — Lujo Bauer, Srdjan Capkun, and Michael Huth — for taking time out of their busy schedules to read my thesis and allow me to defend it. Much of the work developed in my thesis is inspired by their research.

Many friends and colleagues have helped in developing the structure and presentation of this thesis. I thank Marco Guarneri for the helpful comments and discussions on the Introduction section. I also thank all colleagues who have provided valuable feedback on paper drafts and slides related to this thesis: Carlos Cotrini, Jannik Dreier, Erwin Fang, Michèle Feltz, Arthur Gervais, Marco Guarnieri, Kari Kostiainen, Andreas Lochbihler, Ognjen Maric, Sasa Radomirovic, Joel Reardon, Hubert Ritzdorf, Ralf Sasse, Lara Schmidt, Christoph Sprenger, Der-Yeuan Yu, and Thilo Weghorn. I also thank Arthur Gervais and Thilo Weghorn for translating the English abstract into German.

I am also very thankful to Andreas Häberli and Paul Studerus for all discussions about access control from the industry's perspective. These meetings greatly helped to scope projects that are both practical and technically challenging from a research point of view.

Furthermore, I would like to thank all my friends at ETH for all the skiing and hiking around the Swiss alps, after-work movies and outings, and fun discussions during coffee breaks and lunches. All these were essential for surviving around stressful paper deadlines and have made the last few years very exciting and memorable.

Finally, I want to thank my parents Iliyan and Yordanka, and my two brothers Todor and Dimitar. Their support and encouragement, especially during the first months in Zurich, were extremely helpful. Last but not least, I would like to thank Ivelina for all her support throughout these years and for planning all sorts of exotic trips around the world.

Contents

1 Introduction **1**
- 1.1 The Access-Control Setting . . . 2
- 1.2 Challenges and Gaps . . . 4
- 1.3 Contributions . . . 6
- 1.4 Thesis Overview . . . 8

2 System Model **9**

3 The BelLog Language **13**
- 3.1 Syntax . . . 13
- 3.2 Semantics . . . 14
- 3.3 Decision Problems . . . 18
- 3.4 Syntactic Extensions . . . 19
- 3.5 Proofs . . . 21

4 BELLOG Access Control Framework **37**
- 4.1 Grid System Example . . . 37
- 4.2 Policy Specification . . . 39
- 4.3 Policy Verification . . . 46
- 4.4 Policy Enforcement . . . 51
- 4.5 Empirical Evaluation . . . 55
- 4.6 Related Work . . . 62
- 4.7 Technical Details and Proofs . . . 63

5 Fail-Security **71**
- 5.1 Motivating Examples . . . 72
- 5.2 Attacker Model . . . 78
- 5.3 Specifying PDPs with Failure Handling . . . 79
- 5.4 Verifying PDPs against Fail-Security Requirements . . . 85
- 5.5 Related Work . . . 92

6 Access Control Synthesis for Physical Spaces **93**
- 6.1 Overview . . . 96
- 6.2 Physical Access Control . . . 100
- 6.3 Specifying Requirements . . . 104
- 6.4 Policy Synthesis Problem . . . 111
- 6.5 Policy Synthesis Algorithm . . . 116
- 6.6 Implementation and Evaluation . . . 122
- 6.7 Related Work . . . 125
- 6.8 Proofs . . . 127

7 Conclusion **141**

Bibliography **144**

Resume **153**

List of Figures

1.1 Layout of a very small airport that consists of a terminal and an air traffic control room. Access to these subspaces is controlled by gates. 2
1.2 Access to the system's resources is restricted by an access-control system. The gray box on the left depicts the relevant components that define which resources can subjects access. Security engineers must verify that subjects access the resources in compliance with security requirements. 4

2.1 System model 9

3.1 BELLOG's truth space. 15
3.2 Truth tables of BELLOG's operators. 15
3.3 Derived connectives for combining composite rule bodies. Here p, q, and c denote rule bodies and $v \in \mathcal{D}$. 21

4.1 The enforcement model for our running example. 38
4.2 Conditional and override policy composition operators. . . 43
4.3 Shorthands for writing containment conditions. The symbols a, a_1, and a_2 denote BELLOG atoms; c_1 and c_2 denote containment conditions. 49
4.4 Design of our BELLOG PDP. 55
4.5 The figure shows which subjects have access according to the delegation chains policy. All subjects in the left-most row are researchers. Arrows represent delegations. A subject S has access if S is a researcher or S has a delegation chain rooted at a researcher. The parameters used to generate the attributes for this example are $N = 16$, $l = 3$, and $p = 0.25$. 56
4.6 The figure shows which subjects have access according to the delegation group with conflict resolution policy. The top-most subject is the only researcher. Solid arrows represent delegations and dashed arrows revocations. Subjects that are in the whitelist are depicted in a white box. The parameters used to generate this example are $N = 8$, $p_g = 0.1$, $p_d = 0.07$, $p_r = 0.125$, and $p_w = 0.5$. 58
4.7 Average PDP response times for the three policies. 61

4.8 Translating a policy containment condition *cond* to a set of BELLOG rules. 65

4.9 The BELLOG domain (left) and the Datalog domain (right). By Lemma 10, the result of applying BELLOG operator T_P on I is identical to the result of applying $T^D_{\mathcal{R}(P)}$ on $\delta(I)$. 67

5.1 PDP module for evaluating XACML v3.0 policy sets. The methods pol.evaluate(req) and authorize(pol, req) throw an exception if the PDP fails to execute them. 73

5.2 A PDP module for the web app example. 75

5.3 The figure shows which subjects in the depicted scenario have access according to **FR2** 77

5.4 A PDP module for the grid example. 78

5.5 Extended system model with an attacker that can cause communication and component failures. 79

5.6 A PDP module that meets **FR1**. 88

6.1 Access control synthesis for physical spaces 94

6.2 Floor plan and global requirements of our running example. 97

6.3 Synthesizing the local policies for our running example . . . 99

6.4 The double-lined edges denote the PEPs that deny the access request $q = \{\mathsf{role} \mapsto \mathsf{visitor}, \mathsf{time} \mapsto 10, \mathsf{correct\text{-}pin} \mapsto \bot\}$, given the configuration c from Figure 6.3. The resource structure $S_{c,q}$ is obtained by removing the double-lined edges and nodes. 103

6.5 Syntactic shorthands: $a \in \mathcal{A}$ is an attribute, $a_{\mathsf{num}} \in \mathcal{A}_{\mathsf{num}}$ is a numeric attribute, $a_{\mathsf{bool}} \in \mathcal{A}_{\mathsf{bool}}$ is a boolean attribute, $n, n' \in \mathbb{N}$ are natural numbers. 105

6.6 The relation $\models$ between a resource structure $S = (\mathcal{R}, E, r_e, L)$, a resource $r_0 \in \mathcal{R}$, and an access constraints φ. 106

6.7 BELLOG Patterns: The entry resource in the intuitive semantics is depicted using a gray rectangle. The arrows $\varphi \xrightarrow{\checkmark} \psi$ $\left(\varphi \xrightarrow{\times} \psi\right)$ indicate that there must (must not) exist a path from a φ-space to a ψ-space along which T-requests are granted. 108

6.8 Encoding the satisfaction of a requirement $T \Rightarrow \varphi$ in a resource structure $S = (\mathcal{R}, E, r_\varepsilon, L)$, given a template C, into an SMT constraint. The rewrite rules τ reduce an access constraint φ and a resource r_0 to an SMT constraint. For a resource $r_0 \in \mathcal{R}$, we write $E(r_0)$ for $\{r_1 \in \mathcal{R} \mid (r_0, r_1) \in E\}$. The $\exists$ and $\forall$ quantifiers range over a finite domain. Therefore, the former can be expanded as a finite number of disjunctions, and the latter as a finite number of conjunctions. 119
6.9 Scaling the number of PEPs 125
6.10 Encoding exists-until and always-until access constraints using SMT constraints. 130

List of Tables

3.1 Complexity of BELLOG's decision problems. 19

5.1 Analyzed access-control systems and their failure-handling idioms. 80

6.1 Complexity metrics and policy synthesis times (in seconds) for the three cases studies 124

Chapter 1
Introduction

Access control is essential for protecting both physical and digital assets. Consider, for example, an airport. There, access to terminals, boarding gates, and other physical spaces, is restricted to protect the physical assets in the airport, such as aircraft, equipment, and passengers. Access to digital assets is also strictly controlled. Financial institutions, like banks, carefully control who can access their business-critical data. This is necessary not only to keep this data away from the hands of their competitors but also to comply with mandatory regulations.

To secure access to resources, be it physical or digital assets, security engineers deploy *access-control systems*, which restrict how subjects access resources. For example, gates and turnstiles are deployed to control access to the physical spaces in an airport. Simply deploying an access-control system, however, does not entail that access to the resources is appropriately secured. A turnstile, for example, will not protect any of the airport's assets if it simply lets anyone through. To *correctly* secure the resources, the access-control system must grant and deny access in compliance with *access-control requirements*, which impose constraints on how the subjects can access the resources. Such requirements are usually elicited by the resource owners. For example, a bank's security manager may require that only project managers may access business-critical data. Similarly, an airport's security officer may require that all passengers must pass through a security check to access the terminals.

Access-control systems must grant and deny access in compliance with all access-control requirements. Otherwise, attackers can abuse the protected resources. For example, attackers may exploit *unintended grant* decisions to gain access to a bank's business-critical data. Furthermore, attackers may exploit *unintended deny* decisions to cause harm as well. Suppose an attacker can block all the turnstiles deployed in a stadium immediately before a big football game. Then, the attacker can prevent anyone, including visitors with tickets, to access the stadium. Even worse, the attacker can trap all the people inside the stadium.

In this thesis, we tackle the problem of constructing correct access-control systems, i.e. systems that grant and deny access in compliance with all access-control requirements. Before stating our contributions, we describe how modern access-control systems restrict access to resources and discuss key challenges that engineers face when building such systems.

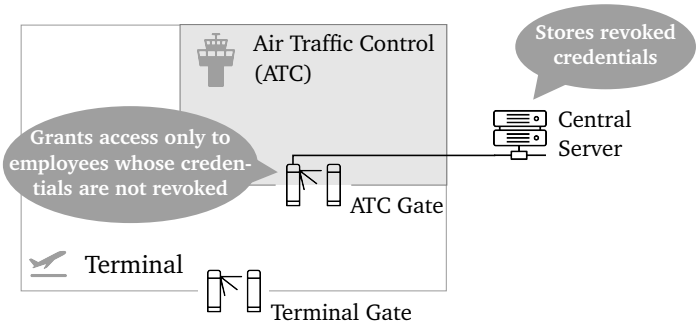


Figure 1.1: Layout of a very small airport that consists of a terminal and an air traffic control room. Access to these subspaces is controlled by gates.

1.1 The Access-Control Setting

We now describe the key components of modern access-control systems and explain how they affect which resources subjects can access.

To illustrate the main concepts, we present a simple example taken from the physical access-control domain. In Figure 1.1, we show the layout of a very small airport, which consists of two subspaces — a terminal and an air traffic control (ATC) room. There are two gates that restrict who can access these two subspaces: one gate that controls access to the terminal and another one that controls access to the ATC room. To access a particular subspace, a subject presents credentials (e.g., stored on a phone or a smart card) to the gate that controls access to that subspace. Based on the provided credentials, the gate either opens and lets the subject in or remains closed. The ATC gate is connected to a central server, which stores a list of credentials that have been revoked.

Access-control Policies and Components. Modern access-control systems are configured with policies, which define a mapping from access requests to access decisions. Each policy is deployed at a component called a Policy Decision Point (PDP), which evaluates its policy and returns access decisions. In our airport example, each gate is equipped with a PDP. The ATC gate is configured with the policy that evaluates to grant for all requests made by non-revoked employees and to deny for all other requests. This policy is depicted in a gray dialog box in Figure 1.1. To give a subject access to the ATC room, the administrator issues a designated “employee” credential and gives it to the subject. Later on, the administrator may decide to revoke an employee credential, e.g. because the subject to which it belongs has been

fired. To revoke a credential, the administrator stores it at a central server that maintains a list of all revoked credentials; see Figure 1.1. To enforce its policy, the ATC gate's PDP checks whether the subject has submitted an employee credential and queries the central server to check whether the provided credential has been revoked. Clearly, all access decisions made by the PDP depend on the policy it enforces.

Communication and Component Failures. One may think that the policy deployed at a PDP fully determines its access decisions. This is, however, false. Communication and component failures may prevent the PDP from evaluating its policy, and thereby influence how it makes access decisions. To illustrate this point, consider our airport example. To check whether a credential has been revoked, the ATC gate's PDP queries the central server. However, what happens if the PDP is unable to reach the central server? In such a scenario, the PDP cannot fetch the list of revoked credentials and therefore it cannot evaluate its policy. To understand whether the PDP would return a grant or deny decision, we need to know how the PDP handles such failures. It may, for example, evaluate another policy (e.g., a designated *fallback* policy) or it may conservatively return a deny decision. This example shows that whenever failures affect the availability of information needed to evaluate a policy, the PDP's access decisions cannot be understood without considering its failure handlers as well. That is, access decisions also depend on the PDP's failure handlers.

Distributed Components. Access-control systems usually consist of multiple PDPs that enforce different access-control policies. The airport access-control system shown in Figure 1.1, for example, has two PDPs — one at the ATC gate and another one at the terminal gate. To understand how subjects access a particular resource, we cannot simply inspect the behavior of the PDP that protects that resource. For example, we cannot understand which subjects can access the ATC room by inspecting the policy of the ATC gate alone. The ATC gate's policy cannot guarantee that non-revoked employees can access the ATC room: If non-revoked employees cannot access the terminal that leads to the ATC room, then this policy is useless. Neither can it guarantee that revoked employees cannot access the ATC room: The ATC room may have another gate that simply lets anyone through. Here, to check system-wide requirements like "Employees who are not revoked can access the ATC room", we must consider the layout of the airport, which defines how subjects can navigate through its subspaces, along with all the policies deployed at the PDPs.

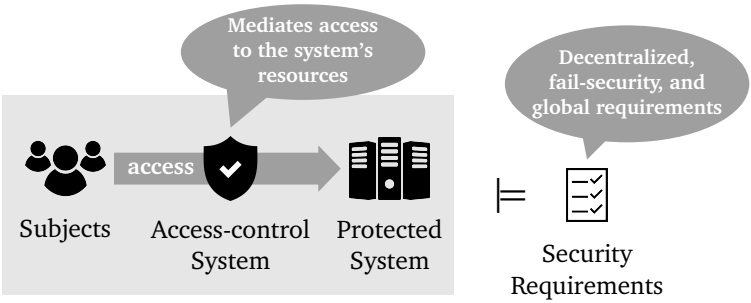


Figure 1.2: Access to the system's resources is restricted by an access-control system. The gray box on the left depicts the relevant components that define which resources can subjects access. Security engineers must verify that subjects access the resources in compliance with security requirements.

1.2 Challenges and Gaps

Constructing access-control systems that grant and deny access in compliance with *all* access-control requirements is difficult [28, 34, 35, 45, 47]. Security engineers must account for all possible ways subjects can access the resources, including unexpected scenarios, such as subjects who deliberately jam wireless channels to disrupt the PDP's communication to credential servers and subjects who hide their credentials. The security engineers must then ensure that the access-control requirements are always satisfied.

The task of formally verifying that a system satisfies a set of requirements is known as program verification. In the context of access control, to verify the correctness of an access-control system with respect to access-control requirements, security engineers must: (i) construct a formal specification that defines how subjects can access the system's resources, (ii) formalize all access-control requirements, and (iii) verify that the formal specification satisfies the requirements. We have abstractly depicted the access-control verification task in Figure 1.2.

The access-control community has developed numerous frameworks that can be used to verify the correctness of access-control systems. Examples include frameworks for formally specifying and reasoning about access-control policies [28, 34, 35], SAT-based and model-checking techniques for reasoning about physical access-control systems [45, 47], and others. Existing access-control frameworks, however, do not fully support recent standards for writing access-control policies, such as XACML v3.0 [94],

which is the latest access-control standard approved by OASIS committee [2]. Neither do they support reasoning about access-control systems that are subject to communication and component failures. These limitations have serious negative consequences in practice: security engineers cannot obtain security guarantees about real-world access-control systems.

In the following, we present three major problems that existing access-control frameworks do not address.

Decentralized Requirements Many access-control systems today, such as those deployed at electronic health record management systems [13], grid resource sharing systems [81], large physical access-control systems [60], are managed in a decentralized manner. That is, there are multiple resource owners who usually delegate their authority over resources to other principals. In corporate settings, for example, a security officer usually delegates authority over project files to project leaders. Since all principals who have authority over a given resource can issue policies, their policies must be composed to define how all access decisions are combined into one decision, namely the access decision that the access-control system must enforce. Policies with authority delegation and policy composition are common today, and the OASIS committee has recently extended the XACML access-control standard to support these policy idioms.

Unfortunately, existing formal access-control frameworks do not support policies with authority delegation and policy composition. It is therefore currently impossible to derive security guarantees about systems that enforce decentralized policies, e.g. those specified in XACML v3.0.

> **Problem 1:** Security engineers cannot specify policies with both authority delegation and policy composition and verify them against decentralized requirements.

Fail-Security Requirements Verifying the policy alone is insufficient in practice, as we have illustrated with our airport example. Virtually all access-control systems are distributed and therefore subject to communication and component failures. These often affect access decisions in surprising and unintended ways, resulting in insecure systems. Therefore, access-control frameworks must account for the PDP's failure handlers. Furthermore, they must support the specification and verification of *fail-security* requirements, which are access-control requirements that stipulate how the PDP must handle failures. Only then can security guarantees be derived for the PDP's access decisions, both in the presence and absence of failures.

Existing access-control frameworks are inadequate for this task. In more detail, they lack (i) a system and attacker model tailored for failure scenarios, (ii) idioms for specifying a PDP's failure handlers, and (iii) tools and algorithms for verifying fail-security requirements.

> **Problem 2:** Security engineers cannot formalize PDPs with failure handlers and verify them against fail-security requirements.

Global Requirements In access-control systems with multiple PDPs, such as physical access-control systems, global requirements express constraints on access paths through multiple PDPs. For example, "Employees who are not revoked can access the ATC room from the main entrance." To enforce such requirements, security engineers write local policies, one policy for each PDP, such that the PDP's local policies collectively enforce *all* global requirements. Writing these local policies is, however, challenging because (i) physical spaces constrain the ways subjects navigate through their subspaces, and (ii) global requirements often have interdependencies.

Writing the local policies manually is a tedious and error-prone task that scales poorly. None of the existing frameworks can be used to helps developers in writing a correct set of local policies.

> **Problem 3:** Security engineers lack tools for automatically constructing local policies that collectively enforce global requirements.

1.3 Contributions

In this thesis, we present a formal access-control framework that helps developers construct access-control systems and verify their correctness against access-control requirements – decentralized, fail-security, and global requirements. The main contributions of this thesis are:

- BELLOG **Access-control Framework.** We present a novel access-control framework that can be used to (i) specify policies with authority delegation and policy composition, (ii) verify the formalized policies against decentralized requirements, and (iii) construct access-control systems that directly enforce the verified access-control policies. At the heart of our access-control framework is a novel logic-programming language, called BELLOG. The BELLOG language subsumes (i) all decentralized policy languages based on Datalog [30] and (ii) all existing composite policy languages based on policy algebras [28, 36]. We illustrate how BELLOG can be used to specify

policies with authority delegation and policy composition. We demonstrate that practical decentralized requirements can be formalized as policy analysis questions, which are readily reduced to BELLOG's core decision problems. Finally, we present an efficient PDP that supports BELLOG policies and can be used to construct access-control systems that directly enforce BELLOG policies.

- **Specifying and Verifying Fail-Security Requirements.** We systematically analyze the role of failure handling in real-world access-control systems. We investigate different kinds of security flaws due to incorrect failure handling. Our findings demonstrate that such flaws can affect access decisions in subtle and unintended ways.

  We demonstrate how the BELLOG access-control framework can be used to verify a PDP against fail-security requirements. In particular: First, we demonstrate how the PDP, including its failure handlers, can be formalized in BELLOG. Second, we define an attacker model tailored to analyzing the effect of failures on the PDP's access decisions. Finally, we show how fail-security requirements are formalized, and we demonstrate the verification of PDPs against fail-security requirements with respect to our attacker model.

- **Synthesizing Local Policies from Global Requirements.** We develop an efficient algorithm for automatically constructing local policies that collectively enforce a set of global requirements for a given system. Security engineers formalize global requirements in a declarative language and give them as input to our synthesis algorithm. Our algorithm then outputs local access-control policies that enforce the requirements. We show that the synthesis problem can be efficiently solved for practically relevant requirements and setups. We demonstrate the effectiveness of our synthesis algorithm using real-world case studies: We construct the local policies for an airport terminal, a corporate building, and a university building.

Together these contributions enable security engineers to formalize PDPs that enforce decentralized policies, along with their failure-handlers, and to verify the formalized PDPs against decentralized and fail-security requirements. Furthermore, they enable the automatic construction of correct access-control policies enforced by multiple distributed PDPs from a set of global requirements, thereby automating this challenging task for security engineers. Together, our contributions address all three problems described in Section 1.2.

1.4 Thesis Overview

This thesis is organized into 7 chapters.

In Chapter 2, we introduce our access-control system model and introduce the terminology used in this thesis.

In Chapter 3, we define the BELLOG language. We also define the main decision problems for BELLOG programs and identify their complexity. We deliberately present the BELLOG language in a separate section, as it is a general-purpose logic-programming language that has applications beyond access control.

In Chapter 4, we illustrate the specification of policies with authority delegation and policy composition in BELLOG. We show that the verification of BELLOG policies against decentralized requirements can be reduced to BELLOG's decision problems. We also present a PDP that supports BELLOG policies and report on experiments that demonstrate the PDP's efficiency.

In Chapter 5, we introduce the concept of fail-secure access control. We give examples of PDP failure handlers and fail-security requirements for access-control systems. We define an attacker model for access-control systems prone to failures. We show the formalization of real-world access-control systems with failure handling in BELLOG and their verification against fail-security requirements using the BELLOG access-control framework.

In Chapter 6, we define a declarative language for formalizing global requirements. We formally define the policy synthesis problem and investigate its complexity. We present an efficient policy synthesis algorithm and report on several real-world case studies.

In Chapter 7, we conclude the thesis and discuss interesting directions for future work.

Publications The content presented in this thesis is based on the following publications:

- Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David Basin, *"Decentralized Composite Access Control"*, in **POST**, 2014
- Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David Basin, *"Fail-Secure Access Control"*, in **CCS**, 2014
- Petar Tsankov, Mohammad Torabi Dashti, and David Basin, *"Access Control Synthesis for Physical Spaces"*, in **CSF**, 2016

Chapter 2
System Model

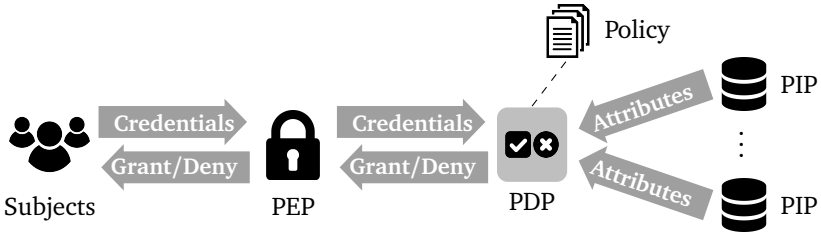


Figure 2.1: System model

In this chapter, we present our system model. To illustrate the key components of our system model, we will refer to our airport example depicted in Figure 1.1.

Components We consider distributed access-control systems that consist of multiple Policy Enforcement Points (PEPs), Policy Decision Points (PDPs), and Policy Information Points (PIP). Access to each resource is controlled by one or more PEPs, and a PEP may control access to multiple resources. To access a resource, a *subject* provides his credentials to one of the PEPs that control access to the resource. Each PEP is associated with a PDP, which is configured with a policy and issues access decisions. Upon receiving a subject's credentials, the PEP forwards them to the PDP. The PDP, in turn, queries PIPs if needed to fetch attributes such as the current time, evaluates its policy, and forwards the access decision — either grant or deny — to the PEP. The PEP then enforces the PDP's decision. We depict these components in Figure 2.1.

As an illustration, consider our airport example from Figure 1.1. The airport's access-control system has two PEPs — one deployed at the ATC gate and another one at the terminal gate. Each PEP controls when the respective gate should open. Subjects, such as passengers and airport employees, submit their credentials to the gate's PEP through a reader. Each of these PEPs is associated with a PDP (not depicted in Figure 1.1) that is configured with a policy. For example, the ATC gate's PDP is configured with the policy that grants access only to employees whose credentials are

not revoked. The airport access-control system has one PIP, the central server, which stores information about revoked credentials.

Attributes and Credentials The access requests and policies we consider are attribute based and may reference three kinds of attributes. A *subject attribute* represents information about a subject. For example, Alice's *organizational role* and *clearance level* are her subject attributes. A *contextual attribute* represents information about the security context provided by a PIP, such as the list of revoked credentials and the current time. We also introduce *resource attributes*, which represent information about resources. These are issued by a system architect. For example, the attributes *room-number* and *floor* may represent room number and the floor of a physical space.

We consider a decentralized access-control model where any subject may issue subject attributes. Each attribute is signed by its issuer, and we refer to signed attributes as *credentials*. Such credentials can be exchanged between the subjects. For example, the airport's manager Alice may issue an employee credential to Bob and she can give this credential to Bob. Bob can then provide his employee credential to the PEP, which can check the credential's authenticity by verifying Alice's digital signature. In our decentralized access-control model, a subject may delegate authority over attributes to other subjects. For example, Alice may delegate authority over the *employee* attribute to Bob, which allows Bob to issue employee credentials.

Access-control Policy The PDP is configured with an access-control policy, which maps attributes to access decisions. The PDP receives attributes from subjects, who may submit credentials along with their requests. The PDP may obtain additional attributes relevant for making access decisions from PIPs. The PDP may also locally store credentials. Attributes that are not explicitly communicated to the PDP are assumed not to have been issued, as is the case in other decentralized systems [25]. Recall that in our airport example the central server stores revoked credentials. If the central server does not store Bob's employee credential, then Bob's employee credential is assumed to be valid, i.e. not revoked.

The PDP's policy is defined by policy rules. Multiple subjects may issue policy rules and store them at the PDP for evaluation. The PDP has one designated subject, the administrator, who has the authority over all access requests and his policy rules are always evaluated. The PDP takes

other rules into account only if the administrator has delegated to their issuers, either directly or transitively, authority over the given request. For example, the administrator of our airport access-control system may delegate authority over all access requests to the airport's security officer. The PDP would then take into account all policy rules issued by the airport's security office.

Failures In our model, we assume that PDPs and PEPs do not fail, whereas PIPs can fail. We also assume that the communication channels between the PDP and the PIPs can fail, while all other channels (e.g. PEP-to-PDP) are reliable. In our airport example, this means that the central server may fail, and the communication channel between the ATC gate's PDP and the central server may fail. All other components and communication channels do not fail.

We assume that communication delays are bounded and failures are determined either by timeouts or by receiving corrupted messages. After the PDP queries a PIP for an attribute, it therefore receives one of two responses: (1) the attribute's value; or (2) *error*, indicating a communication failure. For example, whenever the ATC gate's PDP quires the central server, it either receives a yes/no answer that indicates whether the credential is revoked or it receives an error. Note that in our model, PIP failures are indistinguishable from communication failures.

Chapter 3
The BelLog Language

In this chapter, we define the syntax and semantics of BELLOG and study the time complexity of its decision problems. BELLOG builds upon the syntax and semantics of stratified Datalog [30], and extends it over a four-valued truth space. We see BELLOG as a foundation for constructing high-level access-control languages, and we therefore present BELLOG as a generic many-valued logic-programming language. In later chapters, we will use BELLOG for specifying and reasoning about access-control. Namely, in Chapter 4, we will illustrate how BELLOG can be used to specify access-control policies, and how its decision problems can be used to verify BELLOG policies against security requirements. Afterwards, in Chapter 5, we will use BELLOG to formalize PDPs with failure handling and verify them against fail-security requirements.

Organization We define BELLOG's syntax in Section 3.1 and its semantics in Section 3.2. We define BELLOG's core decision problems in Section 3.3. To simplify the writing of BELLOG programs, we present syntactic extension in Section 3.4. The proofs of all theorems stated in this chapter are given in Section 3.5.

3.1 Syntax

We fix a finite set $\mathcal{P}$ of predicate symbols, where $\mathcal{D}_4 = \{\mathsf{f}_4, \bot_4, \top_4, \mathsf{t}_4\} \subseteq \mathcal{P}$, along with a countably infinite set $\mathcal{C}$ of constants, and a countably infinite set $\mathcal{V}$ of variables. The sets $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{V}$ are pairwise disjoint. Each predicate symbol $p \in \mathcal{P}$ is associated with an arity and we may write p^n to emphasize that p's arity is n. The predicate symbols in $\mathcal{D}_4$ have zero arity. As a convention, we write P to denote a BELLOG program and use the remaining uppercase letters to denote variables. Predicate and constant symbols are written using lowercase *italic* and sans font respectively.

A *domain* Σ is a nonempty finite set of constants. We associate a domain Σ with a set of *atoms*

$$\mathcal{A}_{\Sigma(\mathcal{V})} = \{p^n(t_1, \cdots, t_n) \mid p^n \in \mathcal{P}, \{t_1, \cdots, t_n\} \subseteq \Sigma \cup \mathcal{V}\}.$$

A *literal* is either a, $\neg a$, or $\sim a$, for $a \in \mathcal{A}_{\Sigma(\mathcal{V})}$, and $\mathcal{L}_{\Sigma(\mathcal{V})}$ denotes the set of literals over Σ. We refer to $\neg a$ as *negative literals* and to a and $\sim a$ as

non-negative literals. The function $vars : \mathcal{A}_{\Sigma(\mathcal{V})} \mapsto \mathcal{P}(\mathcal{V})$ maps atoms to the set of variables appearing in them. An atom a is *ground* iff $vars(a) = \emptyset$, and $\mathcal{A}_{\Sigma(\emptyset)}$ denotes the set of ground atoms. We extend *vars* to literals in the standard way.

A BELLOG program, defined over the domain Σ, is a finite set of *rules* of the form:

$$a \leftarrow l_1, \ldots, l_n \ ,$$

where $n > 0$, $a \in \mathcal{A}_{\Sigma(\mathcal{V})}$, $\{l_1, \cdots, l_n\} \subseteq \mathcal{L}_{\Sigma(\mathcal{V})}$, and $vars(a) \subseteq \bigcup_{1 \leq i \leq n} vars(l_i)$. We refer to a as the rule's head and to $l_1, \ldots, l_n$ as the rule's body.

The predicate symbols in a BELLOG program P are partitioned into intensionally defined predicates, denoted idb_P, and extensionally defined predicates, denoted edb_P. The set idb_P contains all predicate symbols that appear in the heads of P's rules, and the set edb_P contains the remaining predicate symbols. We write $\mathcal{A}^{\mathsf{edb}_P}_{\Sigma(\mathcal{V})}$ ($\mathcal{L}^{\mathsf{edb}_P}_{\Sigma(\mathcal{V})}$) and $\mathcal{A}^{\mathsf{idb}_P}_{\Sigma(\mathcal{V})}$ ($\mathcal{L}^{\mathsf{idb}_P}_{\Sigma(\mathcal{V})}$) to denote the sets of atoms (literals) constructed from predicate symbols in edb_P and idb_P respectively.

A rule $a \leftarrow l_1, \cdots, l_n$ is *ground* iff all the literals in its body are ground. The *grounding* of a BELLOG program P is the finite set of ground rules, denoted by $P^{\downarrow}$, obtained by substituting all variables in P's rules with constants from Σ in all possible ways.

A BELLOG program P is *stratified* iff the rules in P can be partitioned into sets $P_0, \cdots, P_n$ called strata, such that:

1. for every predicate symbol p, all rules with p in their heads are in one stratum P_i;
2. if a predicate symbol p occurs as a non-negative literal in a rule of P_i, then all rules with p in their heads are in a stratum P_j with $j \leq i$;
3. if a predicate symbol p occurs as a negative literal in a rule's body in P_i, then all rules with p in their heads are in a stratum P_j with $j < i$.

The given definition of stratified BELLOG extends with non-negative literals that of stratified Datalog [10].

3.2 Semantics

The truth space of BELLOG is the lattice $(\mathcal{D}, \preceq, \wedge, \vee)$, where $\mathcal{D} = \{\mathsf{f}, \bot, \top, \mathsf{t}\}$, $\preceq$ is the partial truth ordering on $\mathcal{D}$, and $\wedge$ and $\vee$ are the meet and join operators. Figure 3.1 shows the lattice's Hasse diagram, where $\preceq$ is depicted upwards. We adopt the meaning of the non-classical truth values $\bot$ and $\top$ from Belnap's four-valued logic [22]: $\bot$ denotes *missing information* and $\top$ denotes *conflicting information*. We define the partial knowledge

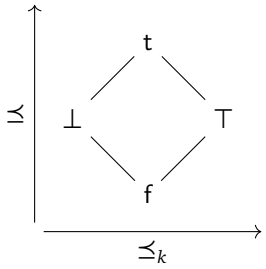


Figure 3.1: BELLOG's truth space.

	¬	∼
f	t	f
⊥	⊥	⊤
⊤	⊤	⊥
t	f	t

∧	f	⊥	⊤	t
f	f	f	f	f
⊥	f	⊥	f	⊥
⊤	f	f	⊤	⊤
t	f	⊥	⊤	t

∨	f	⊥	⊤	t
f	f	⊥	⊤	t
⊥	⊥	⊥	t	t
⊤	⊤	t	⊤	t
t	t	t	t	t

Figure 3.2: Truth tables of BELLOG's operators.

ordering on $\mathcal{D}$, denoted with $\preceq_k$, and depict it in Figure 3.1 rightwards. We denote the meet and join operators on the lattice $(\mathcal{D}, \preceq_k)$ by $\otimes$ and $\oplus$, respectively. The truth tables of the unary operators $\neg$ and $\sim$ are given in Figure 3.2, where we also depict the truth tables for the operators $\wedge$ and $\vee$ for convenience.

An *interpretation* I, over a domain Σ, is a function $I : \mathcal{A}_{\Sigma(\emptyset)} \to \mathcal{D}$, mapping ground atoms to truth values, where $I(\mathsf{f}_4) = \mathsf{f}$, $I(\bot_4) = \bot$, $I(\top_4) = \top$, and $I(\mathsf{t}_4) = \mathsf{t}$. Fix a domain Σ, and let $\mathcal{I}$ be the set of all interpretations over Σ. We define a partial ordering $\sqsubseteq$ on interpretations: given $I_1, I_2 \in \mathcal{I}$, $I_1 \sqsubseteq I_2$ iff $\forall a \in \mathcal{A}_{\Sigma(\emptyset)}.\ I_1(a) \preceq I_2(a)$. We define the meet $\sqcap$ and join $\sqcup$ operators on $\mathcal{I}$ as: $I_1 \sqcap I_2 = \lambda a.\ I_1(a) \wedge I_2(a)$ and $I_1 \sqcup I_2 = \lambda a.\ I_1(a) \vee I_2(a)$. The structure $(\mathcal{I}, \sqsubseteq, \sqcap, \sqcup, I_\mathsf{f}, I_\mathsf{t})$ is a complete lattice where $I_\mathsf{f} = \lambda a.\mathsf{f}$ is the least element and $I_\mathsf{t} = \lambda a.\mathsf{t}$ is the greatest element. Given a continuous function $\Phi : \mathcal{I} \to \mathcal{I}$, we write $\lceil \Phi \rceil$ for the least fixed point of Φ. The interpretation $\lceil \Phi \rceil$ is calculated, using the Kleene fixed point theorem, as M^ω where $M^0 = I_\mathsf{f}$, and $M^{i+1} = \Phi(M^i)$ for $i \geq 0$.

We extend interpretations over the operators $\neg$ and $\sim$ as $I(\neg a) = \neg I(a)$ and $I(\sim a) = \sim I(a)$, respectively, where $a \in \mathcal{A}_{\Sigma(\emptyset)}$. We also extend interpretations over vectors of literals as $I(\vec{l}) = I(l_1) \wedge \cdots \wedge I(l_n)$ where $\vec{l} = l_1, \ldots, l_n$ and $\{l_1, \ldots, l_n\} \subseteq \mathcal{L}_{\Sigma(\emptyset)}$. We write $\bigvee\{v_1, \ldots, v_n\}$ for $v_1 \vee \cdots \vee v_n$. For the empty set we put $\bigvee\{\} = \mathsf{f}$.

An interpretation I is a *model* of a given program P iff

$$\forall (a \leftarrow \vec{l}) \in P^{\downarrow}.\ I(a) \succeq I(\vec{l}).$$

A model therefore, for every rule, assigns to the head a truth value no smaller, in $\preceq$, than the truth value assigned to the body. A model I is *supported* iff

$$\forall a \in \mathcal{A}_{\Sigma(\emptyset)}.\ I(a) = \bigvee \{I(\vec{l}) \mid (a \leftarrow \vec{l}) \in P^{\downarrow}\}.$$

Note that the definition of supported models for BELLOG programs extends that of stratified Datalog. Intuitively, a model I is supported if it does not over-assign truth values to head atoms. In contrast to stratified Datalog, BELLOG's truth values are not totally ordered; therefore, a supported model I of a BELLOG program P does not guarantee that for an atom a there is a rule $(a \leftarrow \vec{l}) \in P^{\downarrow}$ such that $I(a) = I(\vec{l})$. For example, for the program $P = \{a \leftarrow \top_4, a \leftarrow \bot_4\}$ the interpretation $I = \{a \mapsto \mathrm{t}\}$ is a supported model; note that $\{a \mapsto \bot\}$ and $\{a \mapsto \top\}$ are not models of P.

We associate a BELLOG program P with the operator $T_P : \mathcal{I} \mapsto \mathcal{I}$:

$$T_P(J)(a) = \bigvee \{J(\vec{l}) \mid (a \leftarrow \vec{l}\,) \in P^{\downarrow}\}$$

Lemma 1. *Given a* BELLOG *program P, an interpretation I is a supported model iff $T_P(I) = I$.*

The proof follows immediately from the definition of T_P.

In general, a program P may have multiple supported models. For instance, any interpretation is a supported model for the program $\{a \leftarrow a\}$. For BELLOG's semantics we choose a minimal supported model: a supported model I is *minimal* iff there does not exist another supported model I' such that $I' \sqsubset I$. For any a program P that contains only non-negative literals, T_P is monotone (see Theorem 7 in Section 3.5.1), hence continuous due to the finiteness of $\mathcal{I}$, and has a unique minimal supported model. In contrast, if a program P contains negative literals in its rules, then the operator T_P is not monotone, and there could be multiple minimal supported models. For example, the program $P = \{a \leftarrow \neg b\}$ has more than one minimal supported models, e.g. $\{a \mapsto \mathrm{f}, b \mapsto \mathrm{t}\}$ and $\{a \mapsto \mathrm{t}, b \mapsto \mathrm{f}\}$.

For a stratified BELLOG program P, we construct one minimal supported model by computing, for each strata of P, the minimal supported model that contains the model of the previous stratum. This construction is analogous to that of stratified Datalog given in [5]. To define the model construction, we introduce the following notation. We write $(P^{\downarrow}) \triangleleft I$ for the program

obtained by replacing all literals in $P^{\downarrow}$ constructed with edb_P predicate symbols with their truth values according to I. Formally,

$$(P^{\downarrow}) \triangleleft I \;=\; \{a \leftarrow l'_1, \cdots, l'_n \mid (a \leftarrow l_1, \cdots, l_n) \in P^{\downarrow}, \\ l'_i = I(l_i) \text{ if } l_i \in \mathcal{L}^{\mathsf{edb}_P}_{\Sigma(\emptyset)}, \text{otherwise } l'_i = l_i\} \;.$$

Note that all negative literals in a stratum P_i of a stratified BELLOG program are constructed with predicate symbols in edb_{P_i}. Given an interpretation I, the program $P_i^{\downarrow} \triangleleft I$ therefore contains only non-negative literals, and the operator $T_{P_i^{\downarrow} \triangleleft I}$ is monotone.

We now define the model semantics of a stratified BELLOG program:

Definition 1. *Given a stratified* BELLOG *program P, with strata $P_0, \cdots, P_n$, the model of P, denoted $[\![P]\!]$, is the interpretation M_n, where $M_{-1} = I_\mathfrak{f}$, and $M_i = \lceil T_{P_i^{\downarrow} \triangleleft M_{i-1}} \rceil \sqcup M_{i-1}$ for $0 \leq i \leq n$.*

Each M_i, for $0 \leq i \leq n$, is well-defined because the operators $T_{P_i^{\downarrow} \triangleleft M_{i-1}}$ are monotone, and therefore continuous because the lattice $(\mathcal{I}, \sqsubseteq, \sqcap, \sqcup)$ is finite.

Theorem 1. *Given a stratified* BELLOG *program P, $[\![P]\!]$ is a minimal supported model.*

We prove Theorem 1 in Section 3.5.1.

For the previous example $P = \{a \leftarrow \neg b\}$, the given construction results in $[\![P]\!] = \{a \mapsto \mathsf{t}, b \mapsto \mathsf{f}\}$. We choose a minimal supported model semantics for BELLOG because it does not over-assign truth values to head atoms and it assumes that least amount of truth for atoms which are not explicitly assigned a truth value. We justify, in terms of access-control decisions, our choice of semantics in Section 4.2.

We remark that a BELLOG program P that does not use the predicates $\top_4$, $\bot_4$, and the operator $\sim$ in its rules is a syntactically valid stratified Datalog program. Furthermore, stratified BELLOG subsumes stratified Datalog; we prove this in Section 3.5.2. In particular, this means that BELLOG can express all policy languages based on stratified Datalog.

The *input* to a BELLOG program P is an interpretation $I \in \mathcal{I}$, where all atoms from $\mathcal{A}^{\mathsf{idb}_P}_{\Sigma(\emptyset)}$ are mapped to f. For a program P and the input I, we write $[\![P]\!]_I$ as a shorthand for $[\![P \cup P_I]\!]$, where $P_I = \{a \leftarrow v_4 \mid I(a) = v\}$ and $v \in \mathcal{D}$.

From the definition of stratification, it is immediate that given a stratified program P with strata $P_0, \cdots, P_n$, and an input I, the program $P \cup P'$ can be stratified into strata $P_I, P_0, \cdots, P_n$.

We finally remark that the semantics of a BELLOG program is independent of the given stratification. We state and prove this theorem in Section 3.5.3.

3.3 Decision Problems

We define BELLOG's decision problems. In Section 4.3, we reduce the decision problems within our policy analysis framework to BELLOG's decision problems.

Similarly to the *data* complexity of Datalog [90], we study the complexity of the given decision problems when the maximum arity of predicates in P and the set of variables that appear in P are fixed. The input size for BELLOG's decision problems is thus determined by the number of predicate symbols in $\mathcal{P}$, the number of rules in P, and the number of constants in the domain Σ.

Let P be a stratified BELLOG program, Σ be a domain of constants, and a be a ground atom. For a given input I, the *query entailment* decision problem, denoted $P \models^{I}_{\Sigma} a$, asks whether $[\![P]\!]_I(a) = \mathrm{t}$. The general case of $[\![P]\!]_I(a) = v$, with $v \in \mathcal{D}$, is immediately reducible to the query entailment problem.

Theorem 2. *The query entailment problem for stratified* BELLOG *programs belongs to the complexity class* PTIME.

The *query validity* decision problem, denoted $P \models_{\Sigma} a$, asks whether for all inputs I defined over Σ, $P \models^{I}_{\Sigma} a$.

Theorem 3. *The query validity problem for stratified* BELLOG *programs belongs to the complexity class* CO-NP-COMPLETE.

We next consider a generalization of the query validity problem. Let Σ_P denote the set of constants that appear in P. The *all-domains query validity* decision problem, denoted $P \models a$, asks whether $P \models_{\Sigma'} a$ for all domains $\Sigma' \subseteq \mathcal{C}$ that contain Σ_P and the constants in a; recall that $\mathcal{C}$ is the infinite set of constants. The problem of all-domains query validity is in general undecidable for BELLOG programs, because the problem of query validity in Datalog, which is undecidable [80], can be reduced to this problem. We show, however, that all-domains query validity is decidable for any stratified BELLOG program P that has only unary predicate symbols in edb_P. We call those *unary-edb programs*. We show in Chapter 4 that the unary-edb BELLOG programs capture a useful class of policies. Namely, those policies where the set of principals is finite.

Decision problem	Complexity
Entailment	PTIME
Domain containment	CO-NP-COMPLETE
All-domains containment	UNDECIDABLE
All-domains containment for unary-edb programs	CO-NEXP

Table 3.1: Complexity of BELLOG's decision problems.

Theorem 4. *The all-domains query validity problem for a unary-edb* BELLOG *program belongs to* CO-NEXP.

Note that the input for the all-domains query validity problem is determined only by the number of predicate symbols in $\mathcal{P}$ and the number of rules in the program P. We summarize the complexity of BELLOG's decision problems in Table 3.1.

3.4 Syntactic Extensions

We now present a set of syntactic extension to BELLOG to ease the specification of complex rules. In Chapter4, we use these extensions for writing decentralized composite policies. The proofs of all theorems given in this section are in Section 3.5.4.

We extend the syntax for writing policy rules to

$$\begin{aligned} \mathit{rule} &::= a \leftarrow \mathit{body} \\ \mathit{body} &::= l_1, \cdots, l_n \mid \neg \mathit{body} \mid {\sim}\mathit{body} \mid \mathit{body} \wedge \mathit{body}\ , \end{aligned}$$

where $n > 0$, $a \in \mathcal{A}_{\Sigma(\mathcal{V})}$, and $\{l_1, \cdots, l_n\} \subseteq \mathcal{L}_{\Sigma(\mathcal{V})}$. We call the rules of the form $a \leftarrow l_1, \cdots, l_n$ *basic rules* and the remaining rules *composite rules*. Similarly to basic rules, we require that for any composite rule $a \leftarrow \mathit{body}$, $\mathit{vars}(a) \subseteq \mathit{vars}(\mathit{body})$.

We define the translation function $\mathcal{T}$ that maps a basic rule r to the set $\{r\}$:

$$\mathcal{T}(a \leftarrow l_1, \cdots, l_n) = \{a \leftarrow l_1, \cdots, l_n\}\ ,$$

and maps a composite rule $a \leftarrow \mathit{body}$ to a set of basic rules:

$$\mathcal{T}(a \leftarrow \neg body) = \{a \leftarrow \neg p_{\mathsf{fresh}}(\vec{X})\} \cup \mathcal{T}(p_{\mathsf{fresh}}(\vec{X}) \leftarrow body)$$
$$\mathcal{T}(a \leftarrow \sim body) = \{a \leftarrow \sim p_{\mathsf{fresh}}(\vec{X})\} \cup \mathcal{T}(p_{\mathsf{fresh}}(\vec{X}) \leftarrow body)$$
$$\mathcal{T}(a \leftarrow body_1 \wedge body_2) = \{a \leftarrow p_{\mathsf{fresh1}}(\vec{X}_1), p_{\mathsf{fresh2}}(\vec{X}_2)\} \cup \mathcal{T}(p_{\mathsf{fresh1}}(\vec{X}_1) \leftarrow body_1) \cup \mathcal{T}(p_{\mathsf{fresh2}}(\vec{X}_2) \leftarrow body_2)$$

In these rules p_{fresh}, p_{fresh1}, p_{fresh2} are predicate symbols that do not appear in $\mathcal{P}$, $\vec{X} = vars(body)$ and $\vec{X}_i = vars(body_i)$ for $i \in \{1,2\}$. In Section 3.5.4 we prove that the recursive function $\mathcal{T}$ terminates for any composite rule and it yields a set of basic rules. The size of the set of basic rules is linear in the number of nested *body* elements in the composite rule.

The meaning of a BelLog program P with composite rules is that of the BelLog program $P' = \bigcup_{r \in P}(\mathcal{T}(r))$. For example, consider the composite rule:

$$p(X) \leftarrow \neg\sim q(X,Y)\ .$$

The function $\mathcal{T}$ translates this composite rule into a set of basic rules:

$$\{ \quad \begin{array}{lll} p(X) & \leftarrow & \neg p_{\mathsf{fresh}}(X,Y) \\ p_{\mathsf{fresh}}(X,Y) & \leftarrow & \sim q(X,Y) \end{array} \quad \}\ .$$

A BelLog program P with composite rules is *well-formed* iff its rules can be partitioned into sets $P_0, \cdots, P_n$ such that: (1) for every predicate symbol p, all rules with p in their heads are in one stratum P_i; (2) if a predicate symbol p occurs as a non-negative literal in a basic body in P_i, then all rules with p in their heads are in a stratum P_j with $j \leq i$; and (3) if a predicate symbol p occurs in the body of a composite rule in P_i or as a negative literal in a basic rule in P_i, then all rules with p in their heads are in a stratum P_j with $j < i$. Note that well-formed BelLog extends stratified BelLog with the condition that if a predicate symbol p occurs in the body of a composite rule in P_i, then all rules with p in their heads are in a stratum P_j with $j < i$. This is a sufficient but not necessary condition that any composite rule of a well-formed program is translated into a stratified set of basic rules.

Theorem 5. *The translation of a well-formed* BelLog *program with composite rules is a stratified* BelLog *program.*

In Figure 3.3, we derive additional connectives using syntactic combinations of $\neg$, $\sim$, and $\wedge$. The binary connective _ $\vee$ _ corresponds to the join

$$
\begin{array}{rcl}
p \vee q & := & \neg(\neg p \wedge \neg q) \\
p \otimes q & := & (p \wedge \bot) \vee (q \wedge \bot) \vee (p \wedge q) \\
p \oplus q & := & (p \wedge \top) \vee (q \wedge \top) \vee (p \wedge q) \\
p = \mathsf{t} & := & p \wedge {\sim} p \\
p = \mathsf{f} & := & \neg(p \vee {\sim} p) \\
p = \bot & := & (p \neq \mathsf{f}) \wedge (p \neq \mathsf{t}) \wedge ((p \vee \top) = \mathsf{t}) \\
p = \top & := & (p \neq \mathsf{f}) \wedge (p \neq \mathsf{t}) \wedge ((p \vee \bot) = \mathsf{t}) \\
p \neq v & := & \neg(p = v)
\end{array}
$$

Figure 3.3: Derived connectives for combining composite rule bodies. Here p, q, and c denote rule bodies and $v \in \mathcal{D}$.

operator on the lattice $(\mathcal{D}, \preceq)$, and the binary connectives $\_ \otimes \_$ and $\_ \oplus \_$ correspond to the meet and join operators on the lattice $(\mathcal{D}, \preceq_k)$, respectively; for details see [22]. The unary connective $\_ = v$, where $v \in \mathcal{D}$, indicates whether the truth value assigned to the atom is v. The result of $a = v$ is t if a's result is v, and f otherwise. The composition $a \neq v$ returns t only if a's result is not v, otherwise it returns f. Furthermore, we formally establish that BELLOG can represent any n-ary operator $D^n \to D$:

Theorem 6. *Given an operator $g : D^n \to D$ and a list of n rule bodies $b_1, \cdots, b_n$, there exists a* body *expression ϕ for a* BELLOG *composite rule $a \leftarrow \phi$ such that*

$$[\![P]\!]_I(a) = g([\![P]\!]_I(b_1), \ldots, [\![P]\!]_I(b_n)) \,,$$

for all inputs I, and programs P where $\{a \leftarrow \phi\} \subseteq P$ and a is not the head of any other rule.

3.5 Proofs

In this section, we prove all theorems stated in this chapter.

3.5.1 BELLOG Semantics

Below, we prove the model $[\![P]\!]$ of any BELLOG program P is a minimal supported model. First, we show that the consequence operator T_P is monotone for programs with non-negative literals. Then, we state and prove several useful lemmas pertaining to the consequence operator T_P. We conclude with the proof of Theorem 1.

Theorem 7. *For a* BELLOG *program P, defined over a domain Σ, where P has only non-negative literals in its rules, the operator T_P is monotone.*

Proof. Let $I_1 \sqsubseteq I_2$ for some $I_1, I_2 \in \mathcal{I}$, where $\mathcal{I}$ is the set of all interpretations defined over the domain Σ. We show that $T_P(I_1) \sqsubseteq T_P(I_2)$.

To prove the claim we need to show that for an arbitrary atom $a \in \mathcal{A}_{\Sigma(\emptyset)}$, $T_P(I_1)(a) \preceq T_P(I_2)(a)$. By definition of the T_P operator,

$$T_P(I_i)(a) = \bigvee \{I_i(\vec{l}) \mid (a \leftarrow \vec{l}) \in P^{\downarrow}\},$$

for $i \in \{1, 2\}$.

- If the sets $\{I_i(\vec{l}) \mid (a \leftarrow \vec{l}) \in P^{\downarrow}\}$ are the empty set, then $T_P(I_1)(a) = T_P(I_2)(a) = \bigvee\{} = \mathsf{f}$.
- Otherwise, there is at least one rule in $P^{\downarrow}$ with a in its head. Note that the operator $\sim$ is monotone, because for any $v_1, v_2 \in \mathcal{D}$, if $v_1 \preceq v_2$ then $\sim v_1 \preceq \sim v_2$. Furthermore, P's rules have only non-negative literals and the operator $\wedge$ is monotone. Therefore for any rule body $\vec{l}$ we have $I_1(\vec{l}) \preceq I_2(\vec{l})$, simply because $I_1 \sqsubseteq I_2$. By definition of T_P, all rule bodies with a in their heads are combined with the $\vee$ operator. Since $\vee$ is monotone it follows that $T_P(I_1)(a) \preceq T_P(I_2)(a)$.

This concludes our proof. □

We proceed with three lemmas, pertaining to the T_P operator, which we use throughout the remaining proofs in this section. To avoid clutter in the following proofs, we use the following terminology. For a program P defined over a domain Σ, we say that an atom a is an edb *atom of* P if $a \in \mathcal{A}^{\mathsf{edb}_P}_{\Sigma(\emptyset)}$. Similarly we say that an atom a is an idb *atom of* P if $a \in \mathcal{A}^{\mathsf{idb}_P}_{\Sigma(\emptyset)}$. When the program P is clear from the context, we may write edb *atom* instead of edb *atom of* P. We refer to the set of atoms that appear in the bodies of P's rules as the *body atoms of* P.

Lemma 2. *Given two programs P and P' and an interpretation I, $T_{P \cup P'}(I) = T_P(I) \sqcup T_{P'}(I)$.*

Proof. By definition T_P computes each rule independently and then combines their result using the meet $\vee$ operator. As the operator $\vee$ is associative and symmetric, we get $T_{P \cup P'}(I) = T_P(I) \sqcup T_{P'}(I)$. □

Lemma 3. *Given a program P, and interpretations I_1, I_2, if $I_1(a) \preceq I_2(a)$ for any body atom a of P, then $T_P(I_1 \sqcup I_2) = T_P(I_2)$.*

Proof. Since for any body atom a we have $I_1(a) \preceq I_2(a)$, $T_P(I_1 \sqcup I_2)$ computes the body atoms' truth values according to I_2 because $(I_1 \sqcup I_2)(a) = I_2(a)$. Therefore $T_P(I_1 \sqcup I_2) = T_P(I_2)$. □

Lemma 4. *Given a program P, and interpretations I_1, I_2, if for any* edb *atom a it holds that $I_1(a) \preceq I_2(a)$ and for any* idb *atom b it holds that $I_2(b) \preceq I_1(b)$, then $T_P(I_1 \sqcup I_2) = T_{P^{\downarrow} \triangleleft I_2}(I_1)$.*

Proof. By definition of T_P we have $T_P(I_1 \sqcup I_2) = T_{P^{\downarrow}}(I_1 \sqcup I_2)$.

Recall that $P^{\downarrow} \triangleleft I_2$ replaces the edb atoms in P's rules by their truth values according to I_2. Since for any edb atom a we have $I_1(a) \preceq I_2(a)$, it follows that $(I_1 \sqcup I_2)(a) = I_2(a)$. Therefore the computation of $T_{P^{\downarrow}}(I_1 \sqcup I_2)$ always computes the edb atoms' truth values according to I_2, and therefore $T_{P^{\downarrow}}(I_1 \sqcup I_2) = T_{P^{\downarrow} \triangleleft I_2}(I_1 \sqcup I_2)$.

Finally, note that the body atoms of $P^{\downarrow} \triangleleft I_2$ are the idb atoms of P. Because for any idb atom b of P, we have $I_2(b) \preceq I_1(b)$, for any body atom b of $P^{\downarrow} \triangleleft I_2$ we have $I_2(b) \preceq I_1(b)$. By Lemma 3 it follows that $T_{P^{\downarrow} \triangleleft I_2}(I_1 \sqcup I_2) = T_{P^{\downarrow} \triangleleft I_2}(I_1)$. □

Recall that $[\![P]\!] = M_n$ where $M_{-1} = I_{\mathsf{f}}$ and $M_i = \lceil T_{P_i^{\downarrow} \triangleleft M_{i-1}} \rceil \sqcup M_{i-1}$ for $0 \leq i \leq n$. Here, P_i are the strata of P, with $0 \leq i \leq n$. Note that the fixed points $\lceil T_{P_i^{\downarrow} \triangleleft M_{i-1}} \rceil$ are well-defined due to Theorem 7.

Lemma 5. *Given a stratified* BELLOG *program P, the interpretation $[\![P]\!]$ is a supported model of P.*

Proof. By Lemma 1, the interpretation $[\![P]\!]$ is a supported model of P iff $[\![P]\!]$ is a fixed point of T_P.

To show that $[\![P]\!]$ is a fixed point of T_P, we use induction to prove that $T_{P_k \cup \cdots \cup P_0}(M_k) = M_k$ holds for $0 \leq k \leq n$. Note that $T_P = T_{P_n \cup \cdots \cup P_0}$.

Base Case For the base case, $k = 0$, we have $M_0 = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil \sqcup I_{\mathsf{f}}$. Since no edb of P_0 is the head of a rule in $P_0^{\downarrow} \triangleleft I_{\mathsf{f}}$, any edb atom a of P_0 is mapped to f in $\lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil$, thus $\lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil(a) \preceq I_{\mathsf{f}}(a)$. Also, for any idb atom b of P_0, $I_{\mathsf{f}}(b) \preceq \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil(b)$. By Lemma 4, it follows that

$$T_{P_0}(M_0) = T_{P_0}(\lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil \sqcup I_{\mathsf{f}}) = T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}}(\lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil) = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil \tag{3.1}$$

Since $M_0 = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil \sqcup I_{\mathsf{f}} = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil$, we conclude that $T_{P_0}(M_0) = M_0$.

Inductive Step Assume that for a given $0 \leq k < n$, $T_{P_k \cup \cdots \cup P_0}(M_k) = M_k$. We prove that $T_{P_{k+1} \cup \cdots \cup P_0}(M_{k+1}) = M_{k+1}$.

By Lemma 2, we can now rewrite $T_{P_{k+1} \cup \cdots \cup P_0}(M_{k+1})$ to

$$T_{P_{k+1}}(M_{k+1}) \sqcup T_{P_k \cup \cdots \cup P_0}(M_{k+1}) \tag{3.2}$$

Recall that $M_{k+1} = \lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil \sqcup M_k$. We first simplify $T_{P_{k+1}}(M_{k+1})$. Since no edb atom of P_{k+1} is the head of a rule in $P^{\downarrow}_{k+1} \triangleleft M_k$, any edb atom a of P_{k+1} is mapped to f in $\lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil$, and thus $\lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil(a) \preceq M_k(a)$. Also, for any idb atom b of P_{k+1} we have $M_k(b) = \mathsf{f} \preceq \lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil(b)$. By Lemma 4,

$$\begin{aligned} T_{P_{k+1}}(M_{k+1}) &= T_{P_{k+1}}(\lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil \sqcup M_k) = \\ &= T_{P^{\downarrow}_{k+1} \triangleleft M_k}(\lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil) = \lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil \end{aligned} \tag{3.3}$$

We second simplify $T_{P_k \cup \cdots \cup P_0}(M_{k+1})$. Due to stratification, any body atom b of $P_k \cup \cdots \cup P_0$ is not the head of a rule in P_{k+1} and therefore b is mapped to f in $\lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil$; thus $\lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil(b) \preceq M_k(b)$ for any body atom b of $P_k \cup \cdots \cup P_0$. Now, by Lemma 3, and the induction hypothesis, we get:

$$\begin{aligned} T_{P_k \cup \cdots P_0}(M_{k+1}) &= T_{P_k \cup \cdots \cup P_0}(\lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil \sqcup M_k) = \\ &= T_{P_k \cup \cdots \cup P_0}(M_k) = M_k \end{aligned} \tag{3.4}$$

From (3.2), (3.3), and (3.4) it follows that $T_{P_{k+1} \cup \cdots \cup P_0}(M_{k+1}) = \lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil \sqcup M_k$, and therefore $T_{P_{k+1} \cup \cdots \cup P_0}(M_{k+1}) = M_{k+1}$. □

Theorem 1 *Given a stratified* BELLOG *program P, $[\![P]\!]$ is a minimal supported model of P.*

Proof. $[\![P]\!]$ is a supported model of P by Lemma 5. We claim that $[\![P]\!]$ is minimal. We use induction to show that for any interpretation I, if $I \sqsubseteq M_k$ and $T_{P_0 \cup \cdots \cup P_k}(I) = I$ then $I = M_k$ for $0 \leq k \leq n$. Note that the case $k = n$ proves the claim.

Base Case For the base case, assume that $I \sqsubseteq M_0$ and $T_{P_0}(I) = I$ for some interpretation I. We prove that $I = M_0$. Since no edb atom of P_0 appears in the head of a rule in P_0, for any edb atom a of P_0 we have $I(a) = T_{P_0}(I)(a) = \mathsf{f}$. That is, $I(a) = \mathsf{f} \preceq I_{\mathsf{f}}(a)$ for any edb atom a of P_0. For any *idb* atom b of P_0 we have $I_{\mathsf{f}}(b) = \mathsf{f} \preceq I(b)$. Now, by Lemma 4 we get $T_{P_0}(I) = T_{P_0}(I \sqcup I_{\mathsf{f}}) = T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}}(I) = I$. Hence, I is a fixed point of $T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}}$. From $M_0 = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil \sqcup I_{\mathsf{f}} = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil$, it follows that M_0 is the least fixed point of $T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}}$. Thus, $M_0 \sqsubseteq I$. From the assumption $I \sqsubseteq M_0$, it then follows that $I = M_0$.

Inductive Step Assume that for a given $0 \leq k < n$ and any interpretation J, if $J \sqsubseteq M_k$ and $T_{P_0 \cup \cdots \cup P_k}(J) = J$, then $J = M_k$. We prove that $I = M_{k+1}$ for any interpretation I where $I \sqsubseteq M_{k+1}$ and $T_{P_0 \cup \cdots \cup P_{k+1}}(I) = I$.

It is immediate that I can be uniquely decomposed into $I = I_k \sqcup I_{k+1}$ such that I_k maps all *idb* atoms of P_{k+1} to f and I_{k+1} maps all *edb* atoms of P_{k+1} to f. By Lemma 2:

$$T_{P_0 \cup \cdots \cup P_k}(I_k \sqcup I_{k+1}) \sqcup T_{P_{k+1}}(I_k \sqcup I_{k+1}) = I_k \sqcup I_{k+1} \tag{3.5}$$

Note that $T_{P_0 \cup \cdots \cup P_k}(I_k \sqcup I_{k+1})$ maps all *idb* atoms of P_{k+1} to f and $T_{P_{k+1}}(I_k \sqcup I_{k+1})$ maps all *edb* atoms of P_{k+1} to f. Therefore $T_{P_0 \cup \cdots \cup P_k}(I_k \sqcup I_{k+1}) = I_k$ and $T_{P_{k+1}}(I_k \sqcup I_{k+1}) = I_{k+1}$, by the uniqueness of the decomposition.

In the following, we show that **(a)** $I_k = M_k$ and **(b)** $I_{k+1} = \lceil T_{P_{k+1} \triangleleft M_k} \rceil$. These two entail $I = M_{k+1}$, thus completing the proof.

Part (a). For any *edb* atom a of P_{k+1} we have $I_{k+1}(a) = \mathsf{f} \preceq I_k(a)$, simply because only *edb* atoms of P_{k+1} can appear in the rule bodies of $P_0 \cup \cdots \cup P_k$. Now by Lemma 3 we get $T_{P_0 \cup \cdots \cup P_k}(I_k \sqcup I_{k+1}) = T_{P_0 \cup \cdots \cup P_k}(I_k)$. That is, I_k is a fixed point of $T_{P_0 \cup \cdots \cup P_k}$:

$$T_{P_0 \cup \cdots \cup P_k}(I_k) = I_k \tag{3.6}$$

Recall that $I = I_k \sqcup I_{k+1} \sqsubseteq M_{k+1} = M_k \sqcup \lceil T_{P_{k+1} \triangleleft M_k} \rceil$, by the assumption. If a is an *edb* atom of P_{k+1}, then $(I_k \sqcup I_{k+1})(a) = I_k(a) \preceq (M_k \sqcup \lceil T_{P_{k+1} \triangleleft M_k} \rceil)(a) = M_k(a)$; otherwise a is an *idb* atom of P_{k+1} and we have $I_k(a) = M_k(a) = \mathsf{f}$. Therefore,

$$I_k \sqsubseteq M_k. \tag{3.7}$$

From 3.6, 3.7, and the induction hypothesis, it follows that $I_k = M_k$.

Part (b). With an argument similar to Part (a), it follows that $I_{k+1} \sqsubseteq \lceil T_{P_{k+1} \triangleleft M_k} \rceil$. Then, by replacing I_k with M_k in $T_{P_{k+1}}(I_k \sqcup I_{k+1}) = I_{k+1}$ we get $T_{P_{k+1}}(M_k \sqcup I_{k+1}) = I_{k+1}$. For any *edb* atom a of P_{k+1} we have $I_{k+1}(a) = \mathsf{f} \preceq M_k(a)$, and $M_k(b) = \mathsf{f} \preceq I_{k+1}(b)$ for any *idb* atom b of P_{k+1}. Applying Lemma 4 we get $T_{P_{k+1}}(M_k \sqcup I_{k+1}) = T_{P_{k+1} \triangleleft M_k}(I_{k+1}) = I_{k+1}$. That is, I_{k+1} is a fixed point of $T_{P_{k+1} \triangleleft M_k}$. Since $\lceil T_{P_{k+1} \triangleleft M_k} \rceil$ is the least fixed point of $T_{P_{k+1} \triangleleft M_k}$ and $I_{k+1} \sqsubseteq \lceil T_{P_{k+1} \triangleleft M_k} \rceil$, we have $I_{k+1} = \lceil T_{P_{k+1} \triangleleft M_k} \rceil$.

This conclude our proof. □

3.5.2 Semantic Link between Datalog and BELLOG

We first define Datalog's syntax and semantics before proceeding with the proof of the theorem.

Syntax of Stratified Datalog We define the syntax of stratified Datalog as a syntactic restriction of BELLOG: A *stratified Datalog program* is any stratified BELLOG program P where the predicates $\bot, \top$, and the operator $\sim$ do not appear in P's rules.

In the following we fix a stratified Datalog program P, with strata $P_0, \cdots, P_n$, defined over a domain Σ.

Semantics of Stratified Datalog We adopt the semantics of stratified Datalog programs from [10]. The set of Datalog interpretations is $\mathcal{J} = \mathcal{P}(\mathcal{A}_{\Sigma(\emptyset)})$. Unlike BELLOG interpretations, which maps ground atoms to BELLOG's truth values, a Datalog interpretations is a set of ground atoms. The structure $(\mathcal{J}, \subseteq, \cup, \cap, \emptyset, \mathcal{A}_{\Sigma(\emptyset)})$ is a complete lattice. Define $T_P^D : \mathcal{J} \mapsto \mathcal{J}$ as

$$T_P^D(I) = \{a \in \mathcal{A}_{\Sigma(\emptyset)} \mid \exists (a \leftarrow l_1, \cdots, l_n) \in P^{\downarrow}.\ \forall l \in \{l_1, \cdots, l_n\}.\ I \models_D l_i\}$$

where $I \models_D l$ iff
(1) l is an atom a and $a \in I$, or
(2) l is a negative literal $\neg a$ and $a \notin I$.
The powers of the operator T_P^D are defined as:

$$\begin{aligned} T_P^D \uparrow^0 (I) &= I \\ T_P^D \uparrow^{i+1} (I) &= T_P^D(T_P^D \uparrow^i (I)) \cup T_P^D \uparrow^i (I), \qquad \text{for } i > 0 \end{aligned}$$

The model of P, denoted with $[\![P]\!]_D$ is M_n^D, where $M_{-1}^D = \emptyset$ and $M_i^D = T_{P_i}^D \uparrow^{\omega} (M_{i-1}^D)$, for $0 \leq i \leq n$.

We link Datalog interpretations to BELLOG interpretations with the function $\alpha : \mathcal{J} \mapsto \mathcal{I}$, defined as $\alpha(J)(a) = \mathsf{t}$ if $a \in J$, and $\alpha(J)(a) = \mathsf{f}$ otherwise.

Theorem 8. *Given a stratified Datalog program P, $\alpha([\![P]\!]_D) = [\![P]\!]$.*

Proof. We prove using induction that $\alpha(M_k^D) = M_k$ for $-1 \leq k \leq n$.

Base Case For the base case, we have $\alpha(M_{-1}^D) = \alpha(\emptyset) = I_{\mathsf{f}} = M_{-1}$.

Inductive Step Assume that $\alpha(M_k^D) = M_k$, for some k where $0 \leq k < n$. The definition of the operators $\vee$, $\wedge$, and $\neg$, if the predicates $\bot, \top$ and the operator $\sim$ do not appear in P_{k+1}'s rules then the truth values $\bot$ and $\top$ do not appear in $T_{P_{k+1}}(I)$ for any interpretation I. Therefore to show that $\alpha(M_{k+1}^D) = M_{k+1}$, it is sufficient to prove that $a \in M_{k+1}^D$ iff $M_{k+1}(a) = \mathsf{t}$, for any atom a.

We proceed by case distinction on atoms.

- Assume that a is an edb atom of P_{k+1}. Then, for any set of atoms from Datalog's domain $J \in \mathcal{J}$, $a \notin T^D_{P_{k+1}}(J)$ because a does not appear in the head of any rule in P_{k+1}. Therefore $a \in M^D_{k+1}$ iff $a \in M^D_k$. Similarly, for any interpretation $J \in \mathcal{I}$, $T_{P^{\downarrow}_{k+1} \triangleleft M_k}(J)(a) = \mathsf{f}$, and therefore $M_{k+1}(a) = \mathsf{t}$ iff $M_k(a) = \mathsf{t}$. From the induction hypothesis, we conclude that $a \in M^D_{k+1}$ iff $M_{k+1}(a) = \mathsf{t}$.

- Assume that a is an idb atom of P_{k+1}. For any idb atom a, $a \notin M^D_k$ and $M_k(a) = \mathsf{f}$. Therefore, $a \in M^D_{k+1}$ iff a is derived in some iteration of $T^D_{P_{k+1}} \uparrow^i (M^D_k)$. Similarly, $M_{k+1}(a) = \mathsf{t}$ iff $\lceil T_{P^{\downarrow}_{k+1} \triangleleft M_k} \rceil(a) = \mathsf{t}$. By the definition of the operators $T^D_{P_{k+1}}$ and $T_{P^{\downarrow}_{k+1} \triangleleft M_k}$, $a \in T^D_{P_{k+1}}(I)$ iff $T_{P^{\downarrow}_{k+1} \triangleleft M_k}(\alpha(I))(a) = \mathsf{t}$, for any $I \in \mathcal{J}$. From the induction hypothesis $M_k = \alpha(M^D_k)$, and because at every iteration the operators $T^D_{P_{k+1}}$ and $T_{P^{\downarrow}_{k+1} \triangleleft M_k}$ derive the same idb atoms, we conclude that $a \in M^D_{k+1}$ iff $M_{k+1}(a) = \mathsf{t}$.

This concludes our proof. □

3.5.3 Independence of Stratification

We prove that given two different stratifications of a program P, the iterative fixed point construction defined in §3.1 results in the same minimal supported model for P.

Given a stratification $P_0, \cdots, P_n$ of a program P, we write M_{P_i} for the model of $P_0 \cup \cdots \cup P_i$ obtained using the iterative fixed point construction; see §3.1. A predicate symbol p is *defined* in P_i if all rules with p in their heads are in P_i. Given a program P, a predicate symbol p *refers-to* q iff there is a rule r in P such that p appears in r's head and q appears in r's body. Let p *depends-on* q be the transitive closure of the *refers-to* relation. A stratum P_i is *minimal* iff for any two predicate symbols $p, q \in \mathcal{P}$ defined in P_i, p *depends-on* q iff q *depends-on* p. A stratification $P_0, \cdots, P_n$ is *refined* iff all P_i are minimal, with $0 \leq i \leq n$. It is straightforward to see that given two different refined stratifications $P_0, \cdots, P_n$ and $P'_0, \cdots, P'_m$, $n = m$ and for any stratum P_i, there is a stratum P'_j such that $P_i = P'_j$, for $0 \leq i \leq n$ and $0 \leq j \leq m$, and vice versa.

The proof proceeds as follows. We will show that any stratification $P_0, \cdots, P_n$ can be transformed into a refined stratification $P'_0, \cdots, P'_m$ such that $M_{P_n} = M_{P'_m}$. Then we will prove that for any two refined stratifications the iterative fixed point construction results in the same model. These two points establish that the computed model for P is independent to how the

rules are partitioned into strata. We start with the following lemma which allows us to partition the set of rules of a non-minimal stratum:

Lemma 6. *Given a program P where all negative literals in P are constructed from predicate symbols in* edb_P*, an input I, and a stratification P_1, P_2 of P, we have $M = M_2$ where $M = \lceil T_{P^{\downarrow \lhd I}} \rceil \sqcup I$, $M_1 = \lceil T_{P_1^{\downarrow \lhd I}} \rceil \sqcup I$, $M_2 = \lceil T_{P_2^{\downarrow \lhd M_1}} \rceil \sqcup M_1$.*

Proof. We proceed by case distinction on the atoms a.

- Case a is an edb atom of P. Because $M(a) = I(a)$, and $M_2(a) = M_1(a) = I(a)$, it is immediate that $M(a) = M_2(a)$.
- Case a is an idb atom of P_1. Due to the stratification requirements, all rules with a in their heads are contained in P_1. It follows that $M_2(a) = M_1(a) = \lceil T_{P_1^{\downarrow \lhd I}} \rceil(a)$. Since no atoms defined in P_2 appear in the rule bodies in P_1, we get $M(a) = \lceil T_{P^{\downarrow \lhd I}} \rceil(a) = \lceil T_{P_1^{\downarrow \lhd I}} \rceil(a)$. Therefore $M(a) = M_2(a)$.
- Case a is an idb atom of P_2. For $M(a)$ we have $M(a) = \lceil T_{P^{\downarrow \lhd I}} \rceil(a)$, and for $M_2(a)$ we have $M_2(a) = \lceil T_{P_2^{\downarrow \lhd M_1}} \rceil(a)$. Any idb atom of P_1 has the same truth value in M_1 and $\lceil T_{P^{\downarrow \lhd I}} \rceil(a)$; see previous case. We can thus subtract the rules of P_1 from P and replace the truth values of idb atoms of P_1 according to M_1, i.e. we get $\lceil T_{P^{\downarrow \lhd I}} \rceil(a) = \lceil T_{P_2^{\downarrow \lhd M_1}} \rceil(a)$.

This concludes our proof. □

We now prove that any two refined stratifications result in the same model for P.

Theorem 9. *Given two refined stratifications $P_0, \cdots, P_n$ and $P'_0, \cdots, P'_n$, we have $M_{P_n} = M_{P'_n}$.*

Proof. We use induction to prove that for any atom a, if a is defined in $P_0 \cup \cdots \cup P_i$ and $P'_0 \cup \cdots \cup P'_j$ then $M_{P_i}(a) = M_{P'_j}(a)$, for $0 \leq i \leq n$ and $0 \leq j \leq n$. Note that the case for $i = j = n$ completes our proof.

Base Case For the base case, let a is defined in P_0 and P'_0; otherwise the claim obviously holds. It is immediate that $M_{P_0}(a) = M_{P'_0}(a)$ because $P_0 = P'_0$.

Inductive Step Assume that for a given $0 \leq i < n$ and $0 \leq j < n$, if a is defined in $P_0 \cup \cdots \cup P_i$ and $P'_0 \cup \cdots \cup P'_j$ then $M_{P_i}(a) = M_{P'_j}(a)$. We claim that for any atom a, if a is defined in $P_0 \cup \cdots \cup P_{i+1}$ and $P'_0 \cup \cdots \cup P'_j$, then $M_{P_{i+1}}(a) = M_{P'_j}(a)$. The inductive step for $j+1$ is symmetric.

Consider an atom a. Let a be defined in $P'_0 \cup \cdots \cup P'_j$. Note that otherwise the claim obviously holds.

Assume a is defined in $P_0 \cup \cdots \cup P_i$, then $M_{P_{i+1}}(a) = M_{P_i}(a)$ because no rules with a in the head appear in P_{i+1}, The claim holds by the induction hypothesis.

Assume a is not defined in $P_0 \cup \cdots \cup P_i$. Let a be defined in $P_0 \cup \cdots \cup P_{i+1}$. Note that otherwise the claim obviously holds. By the stratification requirements, a is defined in exactly one stratum. Let P'_k, with $0 \leq k \leq j$, be the stratum where a is defined in $P'_0 \cup \cdots \cup P'_j$. Since the stratifications are refined, it follows that $P_{i+1} = P'_k$. Due to the stratification requirements, all edb atoms of P_{i+1} and P'_k are defined in previous strata, and by the induction hypothesis they are mapped to the same truth values according to M_{P_i} and $M_{P'_{k-1}}$. Therefore $M_{P_{i+1}}(a) = M_{P'_k}(a)$. □

We show that any stratification can be transformed into a refined stratification. Take a stratification $P_0, \cdots, P_n$ and a stratum P_i that is not-minimal, with $0 \leq i \leq n$. Let $P_i = P_i^1 \cup P_i^2$ such that P_i^1, P_i^2 is a stratification of P_i. The iterative fixed point construction applied on $P_0, \cdots, P_{i-1}, P_i^1, P_i^2, P_{i+1}, \cdots, P_n$ results in the same model for P, because $M_{P_i^2} = M_{P_i}$ due to Lemma 6. We successively partition the non-minimal strata to obtain a refined stratification with the same model as M_{P_n}.

It follows that any stratification can be transformed into a refined one. Now, by Lemma 9 the following theorem is immediate.

Theorem 10. *Given two stratifications $P_0, \cdots, P_n$ and $P'_0, \cdots, P'_m$, of a stratified program P, $M_{P_n} = M_{P'_m}$.*

3.5.4 BELLOG Extensions

In this Section, we prove all theorems pertaining to BELLOG's syntactic extensions.

Termination To prove that the translation function $\mathcal{T}$, which maps a composite rule to a set of basic rules, terminates, we associate a BELLOG

rule r with the measure $\mu(r)$, where μ is inductively defined as:

$$\begin{aligned}\mu(a \leftarrow body) &= \mu(body)\\ \mu(l_1, \cdots, l_n) &= 1\\ \mu(\neg body) &= 1 + \mu(body)\\ \mu(\sim body) &= 1 + \mu(body)\\ \mu(body_1 \wedge body_2) &= 1 + \mu(body_1) + \mu(body_2)\end{aligned}$$

Recall that given a BELLOG program P with composite rules, the program P is translated into a program $P' = \bigcup_{r \in P} \mathcal{T}(r)$ with basic rules, where $\mathcal{T}$ is the recursive function that maps rules to sets of basic rules. To show that this translation terminates, we state and prove the following Lemma.

Theorem 11. *Given a rule r, the recursive function $\mathcal{T}(r)$ terminates.*

Proof. The proof proceeds by showing that given a rule r, $\forall r' \in \mathcal{T}(r).\, (\mu(r) = \mu(r') = 1) \vee (\mu(r') < \mu(r))$. By definition of μ, for any rule r, $\mu(r) \geq 1$.

Assume $\mu(r) = 1$. By definition of μ, r must be a basic rule $a \leftarrow l_1, \cdots, l_n$. $\mathcal{T}(r)$ terminates simply because $\mathcal{T}(a \leftarrow l_1, \cdots, l_n) = \{a \leftarrow l_1, \cdots, l_n\}$.

Assume $\mu(r) > 1$. By definition of μ, r must be a composite rule. By definition of $\mathcal{T}$, the intermediate step of $\mathcal{T}(r)$ is a set of rules that contains one basic rule and one or two fresh rules, and then $\mathcal{T}$ is recursively applied on the fresh rules. We show that $\mu(r') < \mu(r)$, where r' is a fresh rule generated by $\mathcal{T}$. We proceed by case distinction on r:

- Case $r = a \leftarrow \neg body$. $\mathcal{T}$ generates one fresh rule $r' = a_{\mathsf{fresh}} \leftarrow body$. By definition of μ we have $\mu(r) = 1 + \mu(body)$ and $\mu(r') = \mu(body)$, thus $\mu(r') < \mu(r)$.
- Case $r = a \leftarrow \sim body$. Similarly to the case $r = a \leftarrow \neg body$, $\mathcal{T}$ generates one fresh rule $r' = p_{\mathsf{fresh}} \leftarrow body$, and we get $\mu(r') < \mu(r)$.
- Case $r = a \leftarrow body_1 \wedge body_2$. $\mathcal{T}$ generates two fresh rules $r_1 = p_{\mathsf{fresh1}} \leftarrow body_1$ and $r_2 = p_{\mathsf{fresh2}} \leftarrow body_2$. Because $\mu(r) = 1 + \mu(body_1) + \mu(body_2)$, $\mu(r_1) = \mu(body_1)$, and $\mu(r_2) = \mu(body_2)$, we get $\mu(r_1) < \mu(r)$ and $\mu(r_2) < \mu(r)$.

This completes our proof. □

Stratification of Well-formed BELLOG Programs We now prove that well-formed BELLOG programs are translated into stratified BELLOG programs.

Theorem 5 *Given a well-formed* BELLOG *program P with composite rules, the translated program $P' = \bigcup_{r \in P} \mathcal{T}(r)$ is stratified.*

Proof. The definition of a well-formed program extends the conditions of a stratified program. Therefore, any well-formed program P that contains only basic rules is stratified.

Let $r \in P$ be a rule of a well-formed program P, and $P_0, \cdots, P_n$ are the partitions that satisfy the conditions of a well-formed program. Assume $r \in P_i$ for some $0 \leq i \leq n$. By definition of $\mathcal{T}$, the intermediate result of applying $\mathcal{T}$ on r is a set of rules R containing one basic rule and one or two fresh rules. We claim that $(P \setminus \{r\}) \cup R$ is well-formed. Since $\mathcal{T}$ is applied on P's rules to obtain a program P' with basic rules, the claim implies that P' is well-formed, thus stratified, which completes our proof.

We prove that $(P \setminus \{r\}) \cup R$ is well-formed by case distinction on the rule r.

- Case $r = a \leftarrow l_1, \cdots, l_n$. $R = \{a \leftarrow l_1, \cdots, l_n\}$, and clearly the partitions $P_0, \cdots, P_{i-1}, (P_i \setminus \{r\}) \cup \{a \leftarrow l_1, \cdots, l_n\}, P_{i+1}, \cdots, P_n$ satisfy the conditions of a well-formed program, because $P_i = (P_i \setminus \{r\}) \cup \{a \leftarrow l_1, \cdots, l_n\}$.

- Case $r = a \leftarrow \neg body$. $R = \{a \leftarrow \neg p_{\mathsf{fresh}}, p_{\mathsf{fresh}} \leftarrow body\}$, and the partitions

  $$P_0, \cdots, P_{i-1}, \{p_{\mathsf{fresh}} \leftarrow body\}, (P_i \setminus \{r\}) \cup \{a \leftarrow \neg p_{\mathsf{fresh}}\}, P_{i+1}, \cdots, P_n$$

  satisfy the conditions of a well-formed program, because all rules with a's predicate symbol in the heads are contained in $(P_i \setminus \{r\}) \cup \{a \leftarrow \neg p_{\mathsf{fresh}}\}$, and all predicate symbols that appear in $body$ can only appear in the heads of the rules contained in $P_0 \cup \cdots \cup P_{i-1}$.

- Case $r = a \leftarrow \sim body$. This case is analogous to the case $r = a \leftarrow \neg body$.

- Case $r = a \leftarrow body_1 \wedge body_2$. $R = \{(a \leftarrow p_{\mathsf{fresh1}}, p_{\mathsf{fresh2}}), (p_{\mathsf{fresh1}} \leftarrow body_1), (p_{\mathsf{fresh2}} \leftarrow body_2)\}$. The partitions

  $$P_0, \cdots, P_{i-1}, \{(p_{\mathsf{fresh1}} \leftarrow body_1), (p_{\mathsf{fresh2}} \leftarrow body_2)\},$$
  $$(P_i \setminus \{r\}) \cup \{a \leftarrow p_{\mathsf{fresh1}}, p_{\mathsf{fresh2}}\}, P_{i+1}, \cdots, P_n$$

  satisfy the conditions of a well-formed program, because all rules with a's predicate symbol in the heads are contained in $(P_i \setminus \{r\}) \cup$

$\{a \leftarrow p_{\mathsf{fresh1}}, p_{\mathsf{fresh2}}\}$, and all predicate symbols that appear in $body_1$ and $body_2$ can only appear in the heads of the rules contained in $P_0 \cup \cdots \cup P_{i-1}$.

□

Theorem 6 *Given an operator $g : D^n \to D$ and a list of n rule bodies $b_1, \cdots, b_n$, there exists a* body *expression ϕ for a* BELLOG *composite rule $a \leftarrow \phi$ such that*

$$[\![P]\!]_I(a) = g([\![P]\!]_I(b_1), \ldots, [\![P]\!]_I(b_n)) \,,$$

for all inputs I, and programs P where $\{a \leftarrow \phi\} \subseteq P$ and a is not the head of any other rule.

Proof. Fix an arbitrary $g : \mathcal{D}^n \to \mathcal{D}$, for some $n > 0$, and let $b_1, \cdots, b_n$ be the list of rule bodies.

For each $(d_1, \cdots, d_n) \in \mathcal{D}^n$, we construct the composite body

$$\phi_{d_1,\cdots,d_n} := (b_1 = d_1 \wedge \cdots \wedge b_n = d_n) \overset{\mathsf{t}}{\mapsto} g(d_1, \cdots, d_n)$$

Let the body ϕ of the rule $a \leftarrow \phi$ be the disjunction of composite bodies $\phi_{d_1,\ldots,d_n}$ for all possible $(d_1, \cdots, d_n) \in \mathcal{D}^n$. That is,

$$\phi = \bigvee \{\phi_{d_1,\cdots,d_n} \mid (d_1, \cdots, d_n) \in \mathcal{D}^n\}$$

By construction, given an input I, exactly one $\phi_{d_1,\ldots,d_n}$, namely the one where $[\![P]\!]_I(b_i) = d_i$ for $1 \leq i \leq n$, evaluates to t; all others evaluate to f. The body ϕ thus evaluates to $g(d_1, \cdots, d_n)$.

Finally, we remark that for any well-formed program P where a does not appear in the head of any rule in P, the program $P \cup \{a \leftarrow \phi\}$ is well-formed. □

3.5.5 Complexities of Decision Problems

In this section we show the complexities of BELLOG's decision problems. Given a program P, the maximum arity of predicates in P and the set of variables that appear in P are fixed. The input size for BELLOG's decision problems is thus determined by the number of predicate symbols in $\mathcal{P}$, the number of rules in the program P, and the number of constants in the domain Σ.

Lemma 7. *Given a set P of ground rules with non-negative literals, the complexity of computing the least fixed point of T_P belongs to the complexity class* PTIME.

Proof. Following Kleene's fixed point theorem, we can compute the least fixed point $\lceil T_P \rceil$ as T^ω where $T_0 = I_\mathsf{f}$ and $T^{i+1} = T_P(T^i)$ for $i \geq 0$; recall that T_P is monotone by Theorem 7, and due to the finiteness of the lattice of interpretations monotonicity of T_P entails its continuity.

We claim that the operator T_P needs to be iteratively applied to I_f at most $3 \times |\mathcal{A}_{\Sigma(\emptyset)}|$ times (to compute the least fixed point $\lceil T_P \rceil$). This is because in each application of T_P at least one ground atom changes its truth value to a value strictly higher in the lattice $(\mathcal{D}, \preceq)$; otherwise, a fixed point has been reached. Since the height of the lattice $(\mathcal{D}, \preceq)$ is 3, the number of iterated applications of T_P is bound by $3\times$ the number of ground atoms in $\mathcal{A}_{\Sigma(\emptyset)}$. This proves the aforementioned claim.

The number of ground atoms in $\mathcal{A}_{\Sigma(\emptyset)}$ is at most $|\mathcal{P}| \times |\Sigma|^c$, where c is the fixed maximum arity of the predicate symbols in $\mathcal{P}$. We conclude that the number of iterated applications of T_P is at most $3 \times |\mathcal{P}| \times |\Sigma|^c$.

Finally, the number of steps taken when computing $T_P(I)$, for any interpretation I, is linear in the number of (ground) rules in P. Consequently, the complexity of computing the least fixed point $\lceil T_P \rceil$ (under the assumption that the maximum arity of the predicates in $\mathcal{P}$ is fixed) is polynomial in the number of predicate symbols in $\mathcal{P}$, the number of constants in Σ, and the number of rules in P. □

Theorem 2 *The query entailment problem for stratified* BELLOG *programs belongs to the complexity class* PTIME.

Proof. The query entailment problem $P \models^I_\Sigma a$ can be decided by constructing P's model $[\![P]\!]$ and then checking whether, or not, $[\![P]\!](a) = \mathsf{t}$ holds.

To compute the model $[\![P]\!]$ of P, we must compute the interpretation M_i associated to each stratum P_i. Consider a stratum P_i. To compute $M_i = \lceil T_{P_i^\downarrow \triangleleft M_{i-1}} \rceil \sqcup M_{i-1}$, we need to compute the least fixed point of $T_{P_i^\downarrow \triangleleft M_{i-1}}$; recall that this operator is continuous.

The number of rules in $P_i^\downarrow \triangleleft M_{i-1}$ is bounded by $|P_i| \times |\Sigma|^k$, where $|P_i|$ is the number of (non-ground) rules in P_i, and k is the fixed number of variables that appear in P_i's rules. By Lemma 7, M_i can be computed in PTIME. Since the number of strata of P is no larger than the number of rules in P, we conclude that the complexity of computing the model $[\![P]\!]$, and in turn the complexity of deciding query entailment, is in PTIME. □

Theorem 3 *The query validity problem for stratified* BELLOG *programs belongs to the complexity class* CO-NP-COMPLETE.

Proof. First, we show that the query validity problem is in CO-NP. The complement of $P \models_\Sigma a$, namely $P \not\models_\Sigma a$, can be decided by non-deterministically choosing an input I such that $P \not\models^I_\Sigma a$. By Theorem 2, the complexity of deciding $P \models^I_\Sigma a$ belongs to PTIME, and therefore the complexity of deciding $P \not\models_\Sigma a$ belongs to the complexity class NP. Therefore, the complexity of deciding $P \models_\Sigma a$ belongs to CO-NP.

Second, we reduce the proposition validity decision problem, which belongs to CO-NP-COMPLETE, to query validity. Take an instance of propositional validity ϕ, where ϕ is a propositional formula constructed with propositions, $\wedge$, and $\vee$. Let $P = \{a \leftarrow \phi\}$ be a BELLOG program, where a does not appear in ϕ. Clearly P is well-formed. It is immediate that $P \models_\Sigma a$ iff ϕ is valid in any interpretation. □

Complexity of all-domains query validity In the following we prove that the all-domains query validity decision problem is decidable for unary-edb BELLOG programs.

We fix a stratified program P with strata $P_0, \cdots, P_n$, and with unary predicate symbols in edb_P. We also fix a query a. In the following, we assume, without loss of generality, that the constants appearing in the query a also appear in P. Let Σ_P be the set of constants that appear in P. A domain $\Sigma \subseteq \mathcal{C}$ is *suitable* for P iff $\Sigma_P \subseteq \Sigma$, where $\mathcal{C}$ is the infinite countable set of constant symbols. Let $\mathcal{I}$ be the set of all interpretations over all suitable domains for P. Each interpretation $I \in \mathcal{I}$ is associated with a domain Σ over which I is defined. We write $dom(I)$ to denote I's domain.

We define a *constant type* as a four-way partitioning $(t_\mathsf{f}, t_\bot, t_\top, t_\mathsf{t})$ of the predicate symbols in edb_P. Let $\mathcal{T}$ be the finite set of all possible constant types. Given an interpretation $I \in \mathcal{I}$ with $dom(I) = \Sigma$, a constant $c \in \Sigma$ is of type $(t_\mathsf{f}, t_\bot, t_\top, t_\mathsf{t})$ iff $\forall v \in \mathcal{D}.\ \forall p \in t_v.\ I(p(c)) = v$. We write $\tau(c, I)$ to denote the type of the constant c according to I. For $c, c' \in dom(I)$, write $c \equiv c'$ iff $\tau(c, I) = \tau(c', I)$. It is straightforward that the equivalence $\equiv$ is a congruence, $c \equiv c' \implies T_P(I)p(\cdots, c, \cdots) = T_P(I)p(\cdots, c', \cdots)$, for any $p \in \mathcal{P}$ and any input I.

Let $I \in \mathcal{I}$ and define $\Sigma_I = \Sigma_P \cup \{[c]_\equiv \mid c \in dom(I) \setminus \Sigma_P\}$. Now, for any interpretation J defined over Σ_I, we say I and J *agree* iff $\forall c \in \Sigma_P.\ \tau(c, I) = \tau(c, J)$ and $\forall c \notin \Sigma_P.\ \tau(c, I) = \tau([c]_\equiv, J)$. We claim $[\![P]\!]_I(a) = [\![P]\!]_J(a)$.

Lemma 8. $[\![P]\!]_I(a) = [\![P]\!]_J(a)$.

Proof. The proof is immediate by induction on the minimal fixed points of the strata of P. The only non-trivial observation pertains to that any $c \in dom(I) \setminus \Sigma_P$ and the corresponding $[c]_{\equiv} \in \Sigma_I$ (recall that $dom(J) = \Sigma_I$) have the same constant types. □

Note that for any $I \in \mathcal{I}$, the set Σ_I can have finitely many elements. This is because Σ_P is finite and there are finitely many constant types. Therefore, there are finitely many interpretations J that agree with the infinitely many interpretations of $\mathcal{I}$. The proof of decidability therefore is immediate now: one needs to answer finitely many problems of the form $P \models^J_{dom(J)} a$ to answer $P \models a$. These problems are decidable, due to Theorem 3. The proof of the following theorem is now immediate.

Theorem 12. *The all-domains query validity problem for unary-edb* BELLOG *programs is decidable.*

Theorem 4 *The all-domains query validity problem for unary-edb* BELLOG *programs belongs to* CO-NEXP.

Proof. The complement of $P \models a$ can be decided by non-deterministically choosing an input I such that $P \not\models^I_{dom(I)} a$. Due to Lemma 8, instead of checking $P \not\models^I_{dom(I)} a$ we can check $P \not\models^J_{\Sigma_I} a$ for some J where I and J agree. The size of Σ_I is bounded by $4^{|\mathsf{edb}_P|} + |\Sigma_P|$, because there are at most $4^{|\mathsf{edb}_P|}$ constant types. Therefore, by Theorem 2, the complexity $P \not\models a$ is in NEXP. The complexity of $P \models a$ is thus CO-NEXP. □

Chapter 4
BELLOG Access Control Framework

In this chapter, we illustrate how the BELLOG language can be used to specify and verify decentralized composite policies, which are policies that require both authority delegation and policy composition. To illustrate these two concepts and the tight coupling between them, we present a simple grid system example, which we use throughout this chapter to show how we use BELLOG to specify and verify decentralized composite policies. We present syntactic extensions of BELLOG that ease the specification of common policy composition and authority delegation idioms. Examples include permit-override, only-one-applicable, agreement, hand-off trust application, transitive delegation, etc. We present a policy analysis framework for verifying policies written in BELLOG, and demonstrate how different policy analysis questions are used to reason about a policy's behavior in some or all system configurations. We also show how generic access-control requirements, such as "the policy must not deny all requests" and "all conflicts must be handled", can be encoded within our policy analysis framework. Finally, we present a PDP for BELLOG policies and evaluate its performance and scalability. Our results indicate that BELLOG policies can be enforced efficiently.

Organization We present our grid example in Section 4.1. We show how BELLOG is used to specify decentralized composite policies in Section 4.2. We illustrate the verification of BELLOG policies in Section 4.3. In Section 4.4, we present our BELLOG PDP and the experiments we conducted with it.

4.1 Grid System Example

Consider a grid system that stores files for multiple research projects. Each project has one or more project leaders. The grid system has one PDP that decides access for all files. We depict the example in Figure 4.1. The access-control requirements, inspired by policies in the Swedish Grid Initiative (SweGrid) system [81], are:

R1: A project leader controls access to the project's files and folders, and can delegate these rights.

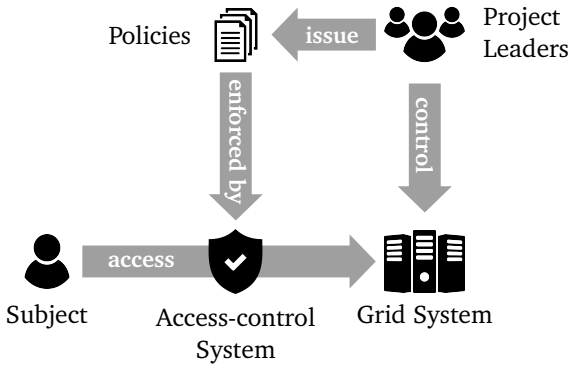


Figure 4.1: The enforcement model for our running example.

R2: If there is a conflicting decision among the project leaders for a given request, then grant access only to requests made by the project leaders.

R3: If no policy applies to a given request, then grant the request if its target is a public project folder, otherwise deny it.

R4: Access rights are recursively extended to sub-folders.

These requirements demonstrate how modern access-control systems require both authority delegation and policy composition features. In particular, requirements **R1** and **R4** require authority delegation, and **R2-3** require policy composition. Existing composition-only policy language, e.g. such as XACML v2.0, and delegation-only policy languages, such as KeyNote 2 [25], cannot be used to specify the policy for this example.

The example also exemplifies the tight coupling between the use of delegation and composition in decentralized composite policies. The PDP must first compute the delegations for each folder according to **R1**, then compose the access rights for each folder according to **R2** and **R3**, and finally extend the policy decisions to sub-folders according to **R4**. Note that **R4** can be encoded as delegation from a parent folder to its children. Such couplings of delegation and composition idioms prevent the decentralized composite policies from being split into and evaluated as two independent, delegation and composition, parts.

We remark that such decentralized composite access-control systems are commonplace; for example, they are also found in electronic health record management systems [13], highly distributed Web services [11], physical

access-control systems [61], and so forth. To cater for such decentralized composite access-control systems, the industry has recently released the XACML v3.0 standard.

4.2 Policy Specification

We first introduce the basic building blocks, namely attributes and delegations, and then we demonstrate how to encode decentralized composite policies in BELLOG, including the policy of our running example. We conclude with a discussion of BELLOG's more intricate features for policy specifications.

We assume that the PDP's domain database contains all constants that appear in the policies, attributes, and access requests, as well as any other additional constants which may denote roles, file names, etc.

4.2.1 Attributes and Delegations

Attributes We represent attributes with predicate symbols. We take the first argument of an attribute as the issuing principal's identifier. For example, $hr(\mathsf{ann}, \mathsf{fred})$ denotes that, according to Ann, Fred works in the Human Resources department. To highlight the attribute's issuer, we may write $\mathsf{ann}:hr(\mathsf{fred})$ instead of $hr(\mathsf{ann}, \mathsf{fred})$. For brevity, we omit prepending admin to the attributes issued by the administrator.

The truth value of an attribute a is t if it is present at the PDP. Recall that the PDP obtains attributes from subjects, PIPs, and its local storage, as described in our system model in Chapter 2. An attribute's truth value is f if it is not present at the PDP. In short, the attributes are by default assumed not to exist if they are not present. For some policies it may however be more appropriate to assume that a given attribute (e.g. an attribute that is provided by the subject) is missing ($\bot$) rather than non-existent (f). BELLOG can accommodate for such policies too. For example, given an attribute a, we can define its *assume-missing* counterpart $a_\bot$ with the rule $a_\bot \leftarrow a \vee \bot$.

Delegations Subjects can delegate authority over attributes to other subjects. Attribute delegations are specified with BELLOG rules where the rule's head is the delegated attribute and the rule body is the delegation condition. For example, with the rule

$$\mathsf{ann}:researcher(S) \leftarrow \mathsf{ann}:hr(S'), S':lab\text{-}card(S)\ ,$$

Ann asserts that a subject S is a researcher if a subject S' with the attribute *hr* asserts that S is a researcher. That is, Ann delegates the attribute *researcher* to subjects who have the attribute *hr*. For example, if Fred has the attribute *hr* and issues $\mathsf{fred}:\mathit{lab\text{-}card}(\mathsf{dave})$, then the PDP derives $\mathsf{ann}:\mathit{researcher}(\mathsf{dave})$.

Delegations may require non-monotonic operators. Imagine that Ann stores at the PDP a list of revoked subjects, and she will not accept delegations of the attribute *researcher* for revoked subjects. We extend her delegation rule as

$$\mathsf{ann}:\mathit{researcher}(S) \leftarrow \mathsf{ann}:\mathit{hr}(S'), S':\mathit{lab\text{-}card}(S), \neg\mathsf{ann}:\mathit{revoked}(S)\ .$$

Non-monotonic operators must be used with caution when applied to the attributes that subjects supply. This is because a subject may gain access if she can withhold the attribute *revoked* from the PDP; cf. [36]. In Section 4.3, we return to this issue and show how one can verify whether a policy is monotone with respect to the attributes provided by the subject.

BELLOG's composite rules can be used to express more complex delegation conditions. In our grid example, the administrator may for instance require two project leaders — Ann and Fred — to agree on the *pub* file attribute, denoting that a file is public. This is written as

$$\mathit{pub\text{-}agree}(F) \leftarrow \mathsf{ann}:\mathit{pub}(F) \oplus \mathsf{fred}:\mathit{pub}(F)\ ,$$

where $\oplus$ is the maximal agreement operator. Note that the administrator derives a conflict if the principals disagree whether a file is public, because $\mathsf{f} \oplus \mathsf{t} = \top$. This conflict can be resolved using the *conflict-override* composition operator, which we define shortly.

As illustrated, BELLOG can specify standard attribute delegations, as well as non-monotonic delegation idioms which cannot be captured in existing Datalog-based languages. There are other delegation idioms that BELLOG can express, but we omit their presentation. For example, the hand-off idiom [4], where a principal delegates authority over all attributes, can be expressed in BELLOG by representing attributes with a predicate *says* where one of the arguments denotes an attribute name.

4.2.2 Policies

We now explain how we specify decentralized composite policies using BELLOG rules and access decisions are represented using BELLOG's truth values.

Policy Decisions We take the $\mathsf{t}, \mathsf{f}, \bot$, and $\top$ elements as, respectively, *grant*, *deny*, *gap*, and *conflict* policy decisions. The *gap* decision indicates that a policy neither grants nor denies a request, and *conflict* indicates that a policy can both grant and deny a request. The partial ordering $\preceq$ in Figure 3.1 defines the *permissiveness* of policy decisions. The meet $\wedge$ and join $\vee$ operators on the lattice $(\mathcal{D}, \preceq)$ correspond to the standard *deny-override* and *permit-override* operators for composing policy decisions. The meet $\otimes$ and join $\oplus$ operators on the lattice $(\mathcal{D}, \preceq_k)$ correspond to the *maximal agreement* and *minimal agreement* composition operators; see [44].

Policies Policies are specified using BELLOG rules. A policy's decisions are represented by a designated predicate symbol. In our grid scenario, we use the predicate symbol *pol* to represent policy decisions: the atom $\mathit{Prin} : \mathit{pol}(S, F)$ denotes the decision of the policy issued by *Prin*, for the subject S accessing the file F. The project leader Piet may, for example, issue the policy

$$\mathsf{piet} : \mathit{pol}(S, F) \leftarrow \mathsf{piet} : \mathit{researcher}(S), \mathsf{piet} : \mathit{prj\text{-}file}(F) \ ,$$

which grants his researchers S access to any project files F.

An access request for a policy P is a pair (I, q), where I is an input for P and q is a ground policy atom. The input I defines which attributes are present at the PDP. Recall that attributes can be pushed to the PDP in the form of credentials, stored locally at the PDP, and fetched from PIPs; see Chapter 2. Furthermore, the input I is defined over a domain Σ, where the constants in Σ identify subjects, resources, and other relevant elements. The policy atom q identifies which subject requests access to which resource. For example, $\mathit{pol}(\mathsf{fred}, \mathsf{foo.txt})$ represents that Fred requests access to the file foo.txt.

Definition 2. *Given a policy P and an access request (I, q), P* grants *(I, q) if $P \models^{I}_{\Sigma} q$.*

When the PDP derives t for the atom $\mathsf{piet} : \mathit{pol}(\mathsf{fred}, \mathsf{foo.txt})$, the PDP interprets this as "Piet's policy grants Fred access to the file foo.txt". Similarly, Ann, who is a project leader, may issue the policy

$$\begin{aligned} \mathsf{ann} : \mathit{pol}(\mathsf{ann}, F) &\leftarrow \mathsf{ann} : \mathit{prj\text{-}file}(F) \\ \mathsf{ann} : \mathit{pol}(S, F) &\leftarrow \mathsf{ann} : \mathit{pol}(S', F), S' : \mathit{give\text{-}access}(S, F) \ , \end{aligned}$$

where the first rule grants Ann access to any project file F, and the second rule states that any subject S' with access to F may delegate this access to any subject S by issuing a *give-access* attribute. Then, Ann may provide access to Fred by issuing $\mathsf{ann}:\mathit{give\text{-}access}(\mathsf{fred},\mathsf{foo.txt})$; Fred too may issue $\mathsf{fred}:\mathit{give\text{-}access}(\mathsf{dave},\mathsf{foo.txt})$ to further delegate to Dave access to foo.txt.

A principal can issue multiple policies for different subjects and resources; we insist however that each principal has one designated root policy. A root policy combines all of the principal's sub-policies and possibly other principals' policies. We fix the atom $\mathit{Prin} : \mathit{pol}(\mathit{Sub},\mathit{File})$ to denote *Prin*'s root policy. Principals may choose any other predicate symbols to denote decisions of their sub-policies.

Composite Policies A policy can also combine the decisions of a set of sub-policies; we call these *composite* policies. A composite policy specified with a basic BELLOG rule, for example, implicitly combines the sub-policies' decisions using the deny-override $\wedge$ operator. Composite policies that combine their sub-policies' decisions with more complex composition operators, such as the gap- and conflict-override operators, are specified with BELLOG composite rules.

In addition to $\wedge$, BELLOG's operators $\neg$, $\sim$, $\vee$, $\otimes$, $\oplus$ can also be employed as composition operators. To complement these operators, in Figure 4.2 we define further conditional and override operators for composing policies. The ternary operator $\_ \triangleleft \_ \triangleright \_$ is the *if-then-else* operator. The result of the composition $p \triangleleft c \triangleright q$ is p's decision only if c's result is t, otherwise q's decision is taken.

The binary operator $\_ \overset{v}{\mapsto} \_$, where $v \in \mathcal{D}$, is the *v-override operator*. The result of the composition $p \overset{v}{\mapsto} q$ is q if p's decision is v, otherwise it results in p's decision. The operators $\overset{\bot}{\mapsto}$ and $\overset{\top}{\mapsto}$ correspond to the *gap-override* and *conflict-override* operators, respectively. Given a list of policies $p_1, \cdots, p_n$, we encode the operator *first-applicable* as $p_1 \overset{\bot}{\mapsto} (p_2 \overset{\bot}{\mapsto} (\cdots \overset{\bot}{\mapsto} p_n))$, i.e. the composition takes the decision of the first policy in the list whose decision is not $\bot$.

The binary operator $\_ \bowtie \_$ is the *only-one-applicable* operator, i.e. the composition $p \bowtie q$ results in $\bot$ if both policy decisions are not $\bot$ or both decisions are $\bot$, otherwise the result is the policy decision that is not $\bot$.

The binary operator $\_ \succcurlyeq \_$ is the *on-permit-apply-second*[1] operator. The composition $p \succcurlyeq q$ returns q only if the decision of p is t, otherwise it

[1] The on-permit-apply-second operator has been recently proposed as an additional operator for the XACML 3 standard. See [78] for full description.

$$
\begin{array}{rcll}
p \triangleleft c \triangleright q & := & ((c = \mathsf{t}) \wedge p) \vee ((c \neq \mathsf{t}) \wedge q) & (\textit{if-then-else}) \\
p \overset{v}{\mapsto} q & := & q \triangleleft (p = v) \triangleright p & (\textit{v-override}) \\
p \bowtie q & := & p \triangleleft (q = \bot) \triangleright (q \triangleleft (p = \bot) \triangleright \bot) & (\textit{only-one-applicable}) \\
p \gg q & := & q \triangleleft (p = \mathsf{t}) \triangleright \bot & (\textit{on-permit-apply-second})
\end{array}
$$

Figure 4.2: Conditional and override policy composition operators.

returns $\bot$. The operator $\gg$ is useful for specifying policies that either (1) grant or provide no decision, or (2) deny or provide no decision. For example, the policy $\mathit{researcher}(\mathit{Sub}) \gg \mathsf{t}$ grants access only if the subject *Sub* is a researcher; otherwise, the policy returns $\bot$. In contrast, the policy $\mathit{revoked}(\mathit{Sub}) \gg \mathsf{f}$ denies access if the subject *Sub* is revoked, and provides no decision otherwise. We also use the operator $\gg$ for specifying policies with policy targets, which define the requests that are applicable to a policy. Given a policy p and its target p_{target}, $p_{\mathsf{target}} \gg p$ results in $\bot$ if p_{target} does not evaluate to t, otherwise it results in p's decision.

We finally remark that BELLOG can express any four-valued policy composition language, such as PBel [28]. This is a corollary of Theorem 6 given in Section 3.4.

4.2.3 Grid Policy

We now exercise these operators to specify the policy of our running example. The administrator may compose the policies issued by the project leaders Piet and Ann with the maximal agreement operator:

$$\mathit{pol\text{-}leaders}(S, F) \leftarrow \mathsf{piet} : \mathit{pol}(S, F) \oplus \mathsf{ann} : \mathit{pol}(S, F) \ .$$

For brevity, we have not specified the policies of Piet and Ann. The composition of their policies may result in conflicts and gaps. According to requirements **R2** and **R3**, the administrator must resolve conflicts by granting requests made by project leaders, and resolve gaps by granting access only to public folders. The *pol-root* policy encodes these requirements:

$$\mathit{pol\text{-}root}(S, F) \leftarrow (\mathit{pol\text{-}leaders}(S, F) \overset{\top}{\mapsto} \mathit{prj\text{-}leader}(S)) \overset{\bot}{\mapsto} \mathit{pub}(F) \ .$$

The composite policy *pol-leaders* considers the decisions of Piet's and Ann's policies for all requests. The administrator may, however, want to consider the decisions of Piet's policy only for the files contained in the

folder prj1. This can be encoded by defining a policy with an explicit policy target:

$$pol\_piet(S,F) \leftarrow contains(\mathsf{prj1},F) \gg \mathsf{piet}:pol(S,F)\ ,$$

where the attribute $contains(F_1,F_2)$ indicates that the folder F_1 contains F_2. The attribute is transitively assigned to sub-folders:

$$contains(F_1,F_2) \leftarrow \mathsf{fs}:subfolder(F_1,F_2)\ ,$$
$$contains(F_1,F_3) \leftarrow contains(F_1,F_2), contains(F_2,F_3)\ ,$$

where the attribute $\mathsf{fs}:subfolder(F_1,F_2)$ is provided by the file system fs and indicates that F_1 is directly contained in F_2. Note that the policy *pol_piet* results in $\bot$ for any request to a file not contained in the folder prj1.

The administrator must also encode the requirement **R4**, which states that any access right to a folder is transitively extended to sub-folders. Namely

$$pol\text{-}root(S,F) \leftarrow contains(F',F), pol\text{-}root(S,F')\ .$$

Note that the policy decision for a folder is extended to sub-folders with the permit-override operator. This is because instantiating the variable F' results in multiple rules with the same head atom, which are combined with the operator $\vee$ according to BELLOG's semantics. To illustrate this, consider the folder f_3, where f_3 is contained in f_2, which in turn is contained in f_1. Instantiating the variable F' and simplifying the instantiated rules result in the following rule:

$$pol\text{-}root(S,f_3) \leftarrow pol\text{-}root(S,f_1) \vee pol\text{-}root(S,f_2)\ .$$

Alternatively, the administrator may want to combine the instantiated rule bodies with deny-override, maximal agreement, or minimal agreement. We show how this can be done with BELLOG's intensional operators, defined below.

4.2.4 Intensional Compositions

So far, we have presented *extensional* policy composition operators that compose a fixed, explicitly given list of sub-policies. For example, we used

$$pol\text{-}leaders(S,F) \leftarrow \mathsf{piet}:pol(S,F) \oplus \mathsf{ann}:pol(S,F)$$

to combine policies of two project leaders, one from Piet and one from Ann, with the maximal agreement operator. Such extensional encodings are tediously "static", because if new project leaders are added to or removed from the PDP, then the administrator must explicitly change the policy rule. Alternatively, the administrator may write a rule that composes the policies that are issued by any principal who is a project leader. One attempt to do this is:

$$\mathit{pol\text{-}leaders}(S,F) \leftarrow P : \mathit{pol}(S,F), \mathit{prj\text{-}leader}(P) \ ,$$

where the set of composed policies is *intensionally* defined as those issued by project leaders. This attempt however fails because the project leaders' policies are implicitly combined with the permit-override operator, instead of the maximal agreement operator $\oplus$. This is because BELLOG's semantics, much like other logic programs, uses the join operator $\vee$ when combining rule bodies with the same head atom.

We extend BELLOG's syntax with additional operators to account for intensional compositions:

$$\mathit{rule} ::= a \leftarrow [\ \bigvee \mid \bigwedge \mid \bigoplus \mid \bigotimes\ ]\ \mathit{body} \ ,$$

where $a \in \mathcal{A}_{\Sigma(V)}$, *body* is a composite rule body, as defined in Section 3.4, and $\mathit{vars}(a) \subseteq \mathit{vars}(\mathit{body})$. We refer to the operators written in front of *body* as *intensional* composition operators. Intuitively, the intensional operator $\bigoplus$ combines all grounded bodies of rules with the same head atom with the $\oplus$ operator. For example, grounding the simple rule $p(\mathsf{a}) \leftarrow \bigoplus q(X)$ over the domain $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ results in two grounded bodies, $q(\mathsf{a})$ and $q(\mathsf{b})$, with the same head atom $p(\mathsf{a})$. The grounded bodies are combined with $\oplus$; the meaning of $p(\mathsf{a}) \leftarrow \bigoplus q(X)$ is therefore $p(\mathsf{a}) \leftarrow q(\mathsf{a}) \oplus q(\mathsf{b})$. Other operators behave similarly with respect to their syntactic counterparts. We relegate the formal translation of the these intensional operators to Section 4.7.1. We remark that the intensional operators $\bigwedge$, $\bigoplus$, and $\bigotimes$ cannot have the head atom appear in the rule body because their encoding uses composite rules.

We can now specify the intensional composition of the project leaders' policies with the maximal agreement operator as

$$\mathit{pol\text{-}leaders}(S,F) \leftarrow \bigoplus (P : \mathit{pol}(S,F) \ \triangleleft\ \mathit{prj\text{-}leader}(P) \ \triangleright\ \bot) \ .$$

Note that the policies that are *not* issued by a project leader are replaced with $\bot$, and the composition "ignores" such policies, because $v \oplus \bot = v$ for any $v \in \mathcal{D}$.

Intensional compositions are also useful for specifying policies that propagate policy decisions over hierarchically structured data, such as file systems, role hierarchies, etc. To illustrate, we extend our grid example with Piet's policy that by default permits a subject S to access a folder F, unless Piet issues the attribute $deny(S,F)$. In contrast to the requirement **R4**, he uses the deny-override operator to propagate deny decisions over the sub-folders:

$$\mathsf{piet}:pol\text{-}fold(S,F) \leftarrow \neg\mathsf{piet}:deny(S,F)$$
$$\mathsf{piet}:pol(S,F) \leftarrow \bigwedge(\mathsf{piet}:pol\text{-}fold(S,F') \triangleleft contains(F',F) \triangleright \mathsf{t})\,.$$

The last rule replaces the policy decisions for folders F' that do not contain F with t, since for any $v \in \mathcal{D}$ we have $v \wedge \mathsf{t} = v$.

We summarize the key difference between intensional and extensional operators as follows. The intensional operators reflect changes in the domain (e.g. addition and removal of principals, files, etc.) through changes in the policy input. The extensional operators require explicit modification of the policy rules to reflect such changes.

4.3 Policy Verification

Writing a correct policy, i.e. one that grants and denies requests as intended by the access-control requirements, is often challenging in practice. This is both because requirements are often initially given informally and imprecisely and because the security engineer can err in their formalization. In particular, a security engineer must foresee all possible access requests, understand how the delegation rules, the sub-policies, and their compositions influence the policy's access decisions, and verify that the policy does not exhibit any unintended access decisions. As a first step towards verifying the policy, the security engineer specifies the access-control requirements as formal policy analysis questions. Second, a decision procedure is used to check, in an automated manner, whether the analysis questions are answered positively, or not.

Below we present our framework for analyzing policies written in BELLOG. We fix the atom $pol(S,R)$ to denote the decisions of the policy under analysis, for the subject S and the resource R.

4.3.1 Policy Entailment

Policy entailment answers whether a policy grants a given access request. Policy entailment analysis is akin to software testing in that the security

engineer checks the policy for unintended grants and denies. Although limited in its scope, since the security engineer must give a specific policy input, determining policy entailment scales with the size of the domain, unlike the policy containment problem which we define shortly.

To illustrate policy entailment, consider the following policy P:

$$\{\, pol(S,R) \leftarrow (pol\text{-}leaders(S,R) \overset{\top}{\mapsto} prj\text{-}leader(S)) \overset{\bot}{\mapsto} pub(R) \,\}\ .$$

For simplicity we do not specify the policy *pol-leaders*. One requirement for P, which is derived from the requirement **R2**, may be to deny access to subjects who are not project leaders whenever the policy *pol-leaders* returns a conflict. To check this property, we may ask whether the policy entails the policy atom $pol(\mathsf{fred}, \mathsf{foo.txt})$ in the input:

$$\begin{aligned} I = \{&pol\text{-}leaders(\mathsf{fred}, \mathsf{foo.txt}) \mapsto \top, \\ &prj\text{-}leader(\mathsf{fred}) \mapsto \mathsf{f}\}\ , \end{aligned}$$

where the remaining atoms are mapped to f. For this input the policy does not entail the policy atom, as expected.

Because the guarantees provided by entailment analysis are limited to the access request provided by the security engineer, the requirement may not hold for other access requests. For example, the given policy P violates its requirement for

$$\begin{aligned} I' = \{&pol\text{-}leaders(\mathsf{fred}, \mathsf{foo.txt}) \mapsto \top, \\ &prj\text{-}leader(\mathsf{fred}) \mapsto \bot, \\ &pub(\mathsf{foo.txt}) \mapsto \mathsf{t}\}\ , \end{aligned}$$

because the policy entails $pol(\mathsf{fred}, \mathsf{foo.txt})$, although *pol-leaders* results in a conflict and the PDP does not know whether Fred is a project leader.

Deciding policy entailment is reducible to query entailment; see Section 3.3. Policy entailment can be therefore decided in time polynomial in the size of the context.

4.3.2 Policy Containment

Policy containment thoroughly analyzes a policy against all access requests. It can be used to answer questions such as: "*Do all access requests evaluate to a conclusive policy decision, i.e. grant or deny?*" Containment analysis is done either for a particular policy domain or for all possible policy domains. In more detail, the domain policy containment answers whether a policy P_1

is more permissive than another policy P_2 for all access request for *a given domain*. The all-domains policy containment answers whether a policy P_1 is more permissive than another policy P_2 for all access requests for *all possible domains*. Even though all-domains evaluations imply those for one domain, checking for all domains is decidable only for a fragment of BELLOG, as we later show.

To verify whether a policy satisfies a given access-control requirement, security engineers check whether the policy has the desired behavior for all access requests to which the requirement is applicable. For example, to verify that the policy P satisfies the requirement **R2**, the security engineer must check whether P denies all requests made by subjects who are not project leaders, for all access requests where the policy *pol-leaders* results in a conflict. We encode such analysis questions with a condition that constraints the access requests where the policies are compared. Formally, the syntax for writing containment questions is

$$cond \Rightarrow P_1 \preceq P_2 \ .$$

The symbols P_1 and P_2 are policies and *cond* is inductively defined as

$$\begin{aligned} cond &::= \forall X.cond \mid a \preceq v \mid v \preceq a \mid \neg cond \mid cond \wedge cond \mid \mathsf{t} \\ v &::= \bot \mid \top \ , \end{aligned}$$

where $X \in \mathcal{V}$, $a \in \mathcal{A}^{\mathsf{edb}_P}_{\Sigma(\mathcal{V})}$, i.e. a is an input attribute. Note that the attributes in a condition may contain variables. We write $fv(cond)$ for the set of variables in *cond* that are not in the scope of $\forall$. We fix the variables S and R to denote the subject and the resource in the policy atom $pol(S,R)$. A policy containment question $cond \Rightarrow P_1 \preceq P_2$ is well-formed iff $fv(cond) \subseteq \{S,R\}$.

We define the satisfaction relation $\Vdash_\Sigma$ between a policy input I, a condition *cond* of a well-formed policy containment question, and a policy domain Σ:

$$\begin{array}{lll} I \Vdash_\Sigma \mathsf{t} & & \\ I \Vdash_\Sigma a \preceq v & \textit{if} & I(a) \preceq v \\ I \Vdash_\Sigma v \preceq a & \textit{if} & v \preceq I(a) \\ I \Vdash_\Sigma \neg cond & \textit{if} & I \not\Vdash_\Sigma cond \\ I \Vdash_\Sigma cond_1 \wedge cond_2 & \textit{if} & I \Vdash_\Sigma cond_1 \textit{ and } I \Vdash_\Sigma cond_2 \\ I \Vdash_\Sigma \forall X.cond(X) & \textit{if} & \forall X \in \Sigma.\ I \Vdash_\Sigma cond(X) \end{array}$$

In Figure 4.3, we define syntactic shorthands to ease the writing of containment conditions.

$$
\begin{aligned}
a \neq v &:= \neg(a = v) \\
c_1 \vee c_2 &:= \neg(\neg c_1 \wedge \neg c_2) \\
a_1 = a_2 &:= (a_1 = \mathsf{f} \wedge a_2 = \mathsf{f}) \vee (a_1 = \bot \wedge a_2 = \bot) \\
&\quad \vee (a_1 = \top \wedge a_2 = \top) \vee (a_1 = \mathsf{t} \wedge a_2 = \mathsf{t})
\end{aligned}
$$

Figure 4.3: Shorthands for writing containment conditions. The symbols a, a_1, and a_2 denote BELLOG atoms; c_1 and c_2 denote containment conditions.

Definition 3. *(Domain Policy Containment) Given a question* $cond \Rightarrow P_1 \preceq P_2$*, and a domain* Σ*,* P_1 *is contained in* P_2 *for all policy inputs over* Σ *that satisfy cond, denoted by* $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$*, iff*

$$\forall I \in \mathcal{I}, \forall S, R \in \Sigma.\ (I \Vdash_\Sigma cond) \rightarrow (\llbracket P_1 \rrbracket_I(pol(S,R)) \preceq \llbracket P_2 \rrbracket_I(pol(S,R)))\ ,$$

where $\mathcal{I}$ *is the set of all policy inputs defined over the domain* Σ*.*

Note that we overload the relation $\Vdash_\Sigma$.

In practice, the policy domain may change over time, e.g. subjects and resources are added to and removed from the system. After changes to Σ, domain policy containment may no longer hold. As mentioned, a stronger policy containment guarantee is thus to verify that P_1 is contained in P_2 for *all* domains Σ'.

Definition 4. *(All-domains Policy Containment) Given a question* $cond \Rightarrow P_1 \preceq P_2$*,* P_1 *is contained in* P_2 *for all access requests in all policy domains, denoted* $\Vdash cond \Rightarrow P_1 \preceq P_2$*, iff* $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$ *holds for all domains* Σ*.*

Example As an example, we use policy containment to specify the requirement that the policy P denies access to subjects who are not project leaders whenever the policy *pol-leaders* results in a conflict:

$$(\textit{pol-leaders}(S,R) = \top) \wedge \neg(\textit{prj-leader}(S) = \mathsf{t}) \Rightarrow P \preceq P_\mathsf{f}\ ,$$

where P_f is the policy that denies all requests. This asks whether P denies access requests where the policy *pol-leaders* results in a conflict, i.e. ($\textit{pol-leaders}(S,R) = \top$), and the subject S is not a project leader, namely $\neg(\textit{prj-leader}(S) = \mathsf{t})$. Both domain and all-domains containment evaluations give negative answers; see the counterexample above. The policy,

however, satisfies the requirement if the attribute *prj-leader* is either t or f. We can easily encode this assumption as

$$(\mathit{pol\text{-}leaders}(S,R) = \top) \wedge (\mathit{prj\text{-}leader}(S) = \mathsf{f}) \Rightarrow P \preceq P_{\mathsf{f}} \,.$$

Domain and all-domains containment evaluations answer this question positively.

Policy Conclusiveness To illustrate how containment questions are specified and used to verify policies against generic requirements, we formalize the requirement: "*All access requests evaluate to a conclusive policy decision*". To specify this requirement as a containment question for the policy P, we construct a policy P' by first renaming the predicate symbol *pol* in P to pol' and then adding the rule

$$\mathit{pol}(S,R) \leftarrow (\mathit{pol}'(S,R) \overset{\top}{\mapsto} \mathsf{f}) \overset{\bot}{\mapsto} \mathsf{f} \,.$$

By construction, the policy P' denies all requests that are evaluated to gap or conflict by the policy P. Therefore, $\models_{\Sigma} \mathsf{t} \Rightarrow P \preceq P'$ holds iff the policy P is conclusive. We set the condition to t because we must this requirement applies to all access requests.

Push-Monotonicity Policy containment is also useful for comparing the policy for one access requests with another access requests that defines a different policy input. Consider a scenario where a subject can push some attributes to the PDP. An important property for the policy is that a subject cannot influence the policy to grant a request by withholding attributes. We refer to such policy as *push-monotonic*: whenever a subject provides fewer attributes to the PDP, the policy results in a less permissive decision. Consider the policy P:

$$\begin{aligned} \{\ & \mathit{pol}(S,R) \leftarrow \mathit{researcher}(S), \mathit{prj\text{-}file}(R) \\ & \mathit{researcher}(S) \leftarrow \mathit{hr}(S'), \mathit{lab\text{-}card}(S',S), \neg \mathit{revoked}(S) \ \} \end{aligned}$$

To check whether this policy is push-monotonic, the security engineer may formulate the question: "*Is the policy more restrictive when the subject provides fewer (pushed) attributes?*" To answer this question, one must compare the policy to itself in all access requests that are identical except for the attributes pushed by the subject. To encode this question, we first construct a policy P' by renaming every predicate symbol p that appears in edb_P

to p', where $\mathsf{edb}_P = \{revoked(\cdot), lab\text{-}card(\cdot,\cdot), hr(\cdot), revoked(\cdot), prj\text{-}file(\cdot)\}$. Suppose the attribute *revoked* is locally stored at the PDP and the remaining attributes are pushed by the subject. The analysis question is encoded as

$$\left(\begin{array}{l} \forall X.\,(revoked(X) = revoked'(X)) \\ \wedge \forall X, Y.\,(lab\text{-}card(X,Y) \preceq lab\text{-}card'(X,Y)) \\ \wedge \forall X.\,(hr(X) \preceq hr'(X)) \\ \wedge \forall X.\,(prj\text{-}file(X) \preceq prj\text{-}file'(X)) \end{array}\right) \Rightarrow P \preceq P' \,.$$

This analysis problem asks whether P is less permissive than P' in all access requests that are identical for the stored attribute and all pushed attributes to P are also pushed to P'. The question indeed holds for the policy P.

Deciding Policy Containment The problems of deciding domain and all-domains policy containment are reducible to domain and all-domains query validity, respectively.

Theorem 13. *Policy containment is polynomially reducible to query validity.*

Corollary 1. *The problem of domain policy containment belongs to the complexity class* CO-NP-COMPLETE. *The problem of all-domains policy containment for unary-edb policies belongs to the complexity class* CO-NEXP.

If a policy has attributes associated to a single user, group, resource, etc., and there are finitely many principals, then the policy can be written in the unary-edb fragment. This is because all attributes have the form *attr-name*(*Issuer*,*Resource*) can be re-encoded as $attr\text{-}name_{Issuer}(Resource)$ since there are finitely many principals.

4.4 Policy Enforcement

In this section, we present a PDP for policies written in BELLOG, and we evaluate its performance and scalability. The key idea underpinning the design of our BELLOG PDP is that the policy entailment decision problem in BELLOG can be reduced to the query entailment problem in stratified Datalog (which is in PTIME). In the following, we first define a translation procedure that takes as input a stratified BELLOG program and outputs a stratified Datalog program. Second, we describe the design of our BELLOG PDP. Finally, we empirically evaluate the performance and scalability of the BELLOG PDP.

4.4.1 Translating BELLOG to Stratified Datalog

The first insight that powers our translation is that we can encode the truth value of every BELLOG atom through truth values of two Datalog atoms. Namely, for any BELLOG atom a, we introduce two Datalog atoms $a_\bot$ and $a_\top$ that encode whether a's truth value is greater than or equal to $\bot$ and, respectively, $\top$. This is sufficient to encode a's truth value. For example, if both Datalog atoms $a_\bot$ and $a_\top$ are derived, then the truth value of a must be t. As another example, if $a_\bot$ is derived but not $a_\top$, then a must be $\bot$.

The second insight is that BELLOG's operators can be encoded through Datalog operators. This allows us to precisely encode the derivation of the introduced Datalog atoms $a_\bot$ and $a_\top$, which we introduce for each BELLOG atom a.

Translation We now formalize the translation from stratified BELLOG to stratified Datalog. For the syntax and semantics of stratified Datalog we refer the reader to Section 3.5.2.

Let P be a stratified BELLOG program defined over the set $\mathcal{P}$ of predicate symbols, the set $\mathcal{V}$ of variables, and the domain $\Sigma \subseteq \mathcal{C}$ of constants. We translate P into a stratified Datalog program, D, defined over the set

$$\mathcal{P}_D = \{p_\bot, p_\top \mid p \in \mathcal{P}\}$$

of predicate symbols, the set of $\mathcal{V}$ of variables, and the set Σ of constants.

Let $\mathcal{A}^D_{\Sigma(\mathcal{V})}$ denote the set of Datalog atoms. We define the function $\rho : \mathcal{A}_{\Sigma(\mathcal{V})} \times \{\bot, \top\} \to \mathcal{A}^D_{\Sigma(\mathcal{V})}$ as

$$\begin{array}{lcl} \rho(p^n(t_1, \cdots, t_n), \bot) & = & p_\bot(t_1, \cdots, t_n) \\ \rho(p^n(t_1, \cdots, t_n), \top) & = & p_\top(t_1, \cdots, t_n)\,. \end{array}$$

The function ρ maps a BELLOG atom and a truth value from $\{\bot, \top\}$ to a Datalog atom. We overload ρ over BELLOG literals as follows:

$$\begin{array}{lcllcl} \rho(\neg a, \bot) & = & \neg\rho(a, \top) & \rho(\sim a, \bot) & = & \rho(a, \top) \\ \rho(\neg a, \top) & = & \neg\rho(a, \bot) & \rho(\sim a, \top) & = & \rho(a, \bot)\,. \end{array}$$

Note that ρ maps negative BELLOG literals ($\neg a$) to negative Datalog literals, and non-negative BELLOG literals (a and $\sim a$) to positive Datalog literals. Given a BELLOG literal l, the Datalog literals $\rho(l, \bot)$ and $\rho(l, \top)$ encode that the truth value of l is greater than or equal to $\bot$ and $\top$, respectively.

We now define the function $\mathcal{R}$ that maps BELLOG rules to sets containing two Datalog rules

$$\mathcal{R}(a \leftarrow l_1, \cdots, l_n) = \{ \quad \rho(a, \bot) \leftarrow \rho(l_1, \bot), \cdots, \rho(l_n, \bot); \\ \rho(a, \top) \leftarrow \rho(l_1, \top), \cdots, \rho(l_n, \top) \} .$$

The Datalog translation of a BELLOG program P into stratified Datalog, denoted by $\mathcal{R}(P)$, is

$$\mathcal{R}(P) = \bigcup_{r \in P} \mathcal{R}(r) .$$

Note that if P is a stratified BELLOG program, then $\mathcal{R}(P)$ is a stratified Datalog program because ρ maps only negative BELLOG literals to negative Datalog literals. Furthermore, for any BELLOG program that consists of n basic BELLOG rules, the above translation procedure generates a stratified Datalog program with $2n$ rules and twice as many predicate symbols. In particular, the predicates' arities do not increase.

Correctness To state the correctness of our translation, we link BELLOG interpretations to Datalog interpretations. Let $\mathcal{I}$ and $\mathcal{J}$ denote the set of all BELLOG interpretations and the set of all Datalog interpretations, respectively. We link these sets with the functions $\delta : \mathcal{I} \to \mathcal{J}$ and $\bar{\delta} : \mathcal{J} \to \mathcal{I}$:

$$\delta(I) = \begin{array}{c} \{\rho(a, \top) \mid \top \preceq I(a)\} \\ \cup \\ \{\rho(a, \bot) \mid \bot \preceq I(a)\} \end{array}$$

and

$$\bar{\delta}(J)(a) = \begin{cases} \mathsf{t} & \text{if} \quad \rho(a, \bot) \in J \text{ and } \rho(a, \top) \in J \\ \top & \text{if} \quad \rho(a, \bot) \notin J \text{ and } \rho(a, \top) \in J \\ \bot & \text{if} \quad \rho(a, \bot) \in J \text{ and } \rho(a, \top) \notin J \\ \mathsf{f} & \text{if} \quad \rho(a, \bot) \notin J \text{ and } \rho(a, \top) \notin J \end{cases}$$

Here, $a \in \mathcal{A}_{\Sigma(\emptyset)}$ is a ground BELLOG atom, $I \in \mathcal{I}$ is BELLOG interpretation, and $J \in \mathcal{J}$ is a Datalog interpretation.

Theorem 14. *Given a stratified* BELLOG *program P and an input I for P, we have $[\![P]\!]_I = \bar{\delta}([\![\mathcal{R}(P) \cup \delta(I)]\!]_D)$.*

The above theorem establishes that we can compute the model of a BELLOG program P by (i) computing the model of P's Datalog translation $\mathcal{R}(\mathcal{P})$ for the input $\delta(I)$ and then (ii) mapping back the computed Datalog model to a BELLOG model with the function $\bar{\delta}$. We prove this theorem in Section 4.7.3. In the following we give an example, which illustrates how we compute $[\![P]\!]_I$ using the above translation procedure.

Example Consider the following stratified BELLOG program:

$$P = \{p(X) \leftarrow q(X), \neg r(X), {\sim}s(X)\} \ ,$$

and the input:

$$\begin{array}{rllllll} I = \{ & p(a) & \mapsto & \mathsf{f}, & \mathsf{t}_4 & \mapsto & \mathsf{t} \\ & q(a) & \mapsto & \mathsf{t}, & \mathsf{f}_4 & \mapsto & \mathsf{f} \\ & r(a) & \mapsto & \mathsf{f}, & \bot_4 & \mapsto & \bot \\ & s(a) & \mapsto & \bot, & \top_4 & \mapsto & \top \quad \} \ . \end{array}$$

The input I assigns truth values to the ground atoms in $\mathcal{A}_{\Sigma(\emptyset)}$ as well as to the nullary predicate symbols $\mathsf{t}_4, \mathsf{f}_4, \bot_4$, and $\top_4$. The model $[\![P]\!]_I$ is

$$\begin{array}{rllllll} [\![P]\!]_I = \{ & p(a) & \mapsto & \top, & \mathsf{t}_4 & \mapsto & \mathsf{t}, \\ & q(a) & \mapsto & \mathsf{t}, & \mathsf{f}_4 & \mapsto & \mathsf{f}, \\ & r(a) & \mapsto & \mathsf{f}, & \bot_4 & \mapsto & \bot, \\ & s(a) & \mapsto & \bot, & \top_4 & \mapsto & \top \quad \} \ . \end{array}$$

We now show how we can compute $[\![P]\!]_I$ using P's Datalog translation.

Step 1: Translation We first translate the BELLOG program P to a stratified Datalog program $\mathcal{R}(P)$:

$$\begin{array}{rlll} \mathcal{R}(P) = \{ & p_\bot(X) & \leftarrow & q_\bot(X), \neg r_\top(X), s_\top(X) \\ & p_\top(X) & \leftarrow & q_\top(X), \neg r_\bot(X), s_\bot(X) \quad \} \ . \end{array}$$

Step 2: Map Input to Datalog We translate the BELLOG input I to a Datalog interpretation J:

$$J = \delta(I) = \{q_\bot(a), q_\top(a), s_\bot(a), \mathsf{t}_{4\bot}, \mathsf{t}_{4\bot}, \bot_{4\bot}, \top_{4\top}\} \ .$$

Step 3: Compute Datalog Model We compute the Datalog model of $\mathcal{R}(P) \cup J$:

$$[\![\mathcal{R}(P) \cup J]\!]_D = \{p_\top(a)\} \cup J \ .$$

Step 4: Map Datalog Model to BELLOG Interpretation Finally, we map the computed Datalog model to a BELLOG interpretation:

$$\begin{array}{rllllll} \bar{\delta}([\![\mathcal{R}(P) \cup J]\!]_D) = \{ & p(a) & \mapsto & \top, & \mathsf{t}_4 & \mapsto & \mathsf{t}, \\ & q(a) & \mapsto & \mathsf{t}, & \mathsf{f}_4 & \mapsto & \mathsf{f}, \\ & r(a) & \mapsto & \mathsf{f}, & \bot_4 & \mapsto & \bot, \\ & s(a) & \mapsto & \bot, & \top_4 & \mapsto & \top \quad \} \ . \end{array}$$

Note that $\bar{\delta}([\![\mathcal{R}(P) \cup J]\!]_D) = [\![P]\!]_I$.

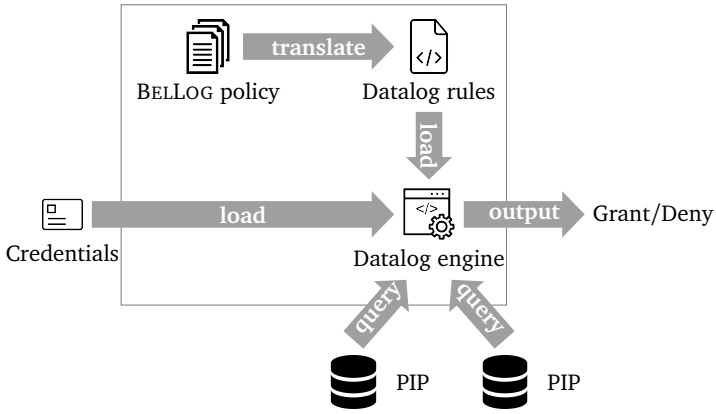


Figure 4.4: Design of our BELLOG PDP.

4.4.2 BELLOG PDP

Our BELLOG PDP has two main components: (i) the *translator* component which translates BELLOG policies and credentials into Datalog rules, and (ii) the *Datalog interpreter* which evaluates the resulting rules. The PDP queries PIPs using the standard SQL interface. Before any requests are evaluated, the security engineer must load a BELLOG policy into the PDP. The policy is translated into Datalog rules and loaded into the interpreter. Figure 4.4 depicts the design of our PDP.

The input to the PDP is an access request, i.e. a policy input and a policy atom. The PDP first loads the policy input into the Datalog interpreter. The PDP then queries the Datalog interpreter with the policy atom. It outputs *grant* if the interpreter derives this policy atom; otherwise, it outputs *deny*.

We implemented the PDP using XSB [3] as the Datalog interpreter component. The translator component is implemented in Python and has 460 lines of code. To connect to a database, the PDP uses XSB's database module and translates BELLOG remote queries into XSB prepared queries. Our implementation is publicly available at `http://bellog.org`.

4.5 Empirical Evaluation

In this section, we measure the performance of our BELLOG PDP. Our main goal here is to investigate whether the BELLOG PDP is practical and can be used to construct real-world access-control systems. In the following, we first describe our experimental setup and then we report on our results.

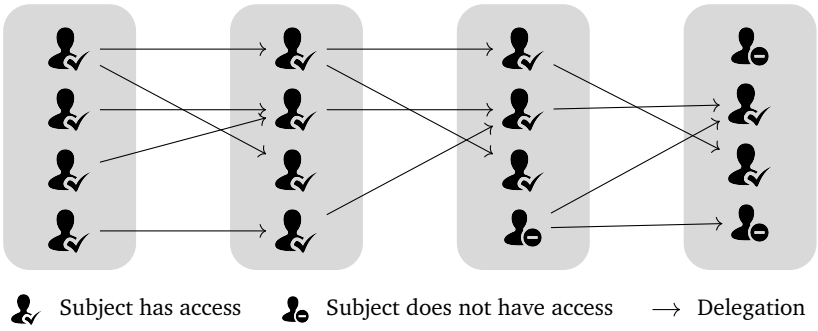


Figure 4.5: The figure shows which subjects have access according to the delegation chains policy. All subjects in the left-most row are researchers. Arrows represent delegations. A subject S has access if S is a researcher or S has a delegation chain rooted at a researcher. The parameters used to generate the attributes for this example are $N = 16$, $l = 3$, and $p = 0.25$.

4.5.1 Experimental Setup

For our experiments, we use the BELLOG policies and algorithms presented in [27]. To keep the thesis self-contained, we describe the policies used in our experiments and the algorithms used to generate attributes and access requests for these policies.

Policy 1: Delegation Chains Authority delegation is a key idiom in decentralized access-control policies. To investigate how the length of the delegation chains and the number of subjects affect the PDP's response time, we use the following policy:

$$\begin{aligned} pol(S) &\leftarrow researcher(S) \\ pol(S) &\leftarrow pol(S'), S' : give\text{-}access(S) \end{aligned}$$

The predicate $pol(S)$ denotes the policy decision for the subject S. The administrator issues the credential $researcher(S)$ to all researchers. The first rule formalizes that researchers are granted access. The second rule formalizes that a subjects S' who has access may delegate access to another subject S by issuing the credential $S' : give\text{-}access(S)$.

The input attributes for this policy are *researcher* and *give-access*. The algorithm for generating these attributes is parameterized by three parameters:

- N, the number of subjects,

- l, the length of the longest delegation chain,
- p, the probability of issuing a *give-access* credential.

The principals are partitioned into $l+1$ partitions of equal size. For each subject S in the first partition, an attribute *researcher*(S) is generated. The first partition thus contains all researchers and the remaining partitions contain the subjects who are not researchers. For each pair of subjects (S', S) where S' is a subject from a partition i, with $0 \leq i \leq l$, and S is a subject from partition $i+1$, we generate an attribute S':*give-access*(S) with probability p. The ith partition thus consists of subjects who may have multiple delegations of length $i-1$. Note that the longest delegation length is thus l. In Figure 4.5 we depict an access-control scenario that is generated using parameters $N = 16$, $l = 3$, and $p = 0.25$.

We generate access requests for the subjects contained in partition $l+1$ (the right-most partition in Figure 4.5). This allows us to focus on the impact of the delegation length on the PDP's response time.

Policy 2: Delegation Group with Conflict Resolution The second policy features both authority delegation and policy compositions. This policy extends the delegation chains policy as follows:

$$
\begin{array}{lcll}
\mathit{pol}(S) & \leftarrow & (\mathit{grant}(S) \oplus \neg \mathit{deny}(S)) \overset{\top}{\mapsto} \mathit{whitelist}(S) & (R1) \\
\mathit{grant}(S) & \leftarrow & \mathit{researcher}(S) & (R2) \\
\mathit{grant}(S) & \leftarrow & \mathit{grant}(S'), S' : \mathit{give\text{-}access}(S) & (R3) \\
\mathit{deny}(S) & \leftarrow & \mathit{grant}(S'), S' : \mathit{deny\text{-}access}(S) & (R4)
\end{array}
$$

Rules *(R2-3)* are identical to those used in the delegation chains policy. They formalize that researchers are granted access and may further delegate their access to other subjects. Rule *(R4)* states that a subject S' that is granted access may also decide to deny access to another subject S by issuing the credential S':*deny-access*(S). Finally, rule *(R1)* combines the truth values assigned to the attributes *grant*(S) and *deny*(S) for a subject S using the agreement operator $\oplus$. Note that the result is a conflict ($\top$) if the subject S is both granted and denied access. Rule *(R1)* resolves such conflicts using the conflict override operator ($\overset{\top}{\mapsto}$): A subject S with conflicting grant and deny credentials is granted access if S has the attribute *whitelist*(S), which represents that the subject is in a whitelist.

The input attributes for this policy are: *whitelist*, *researcher*, *give-access*, and *deny-access*. The algorithm for generating these attributes is parameterized by:

- N, the number of subjects,

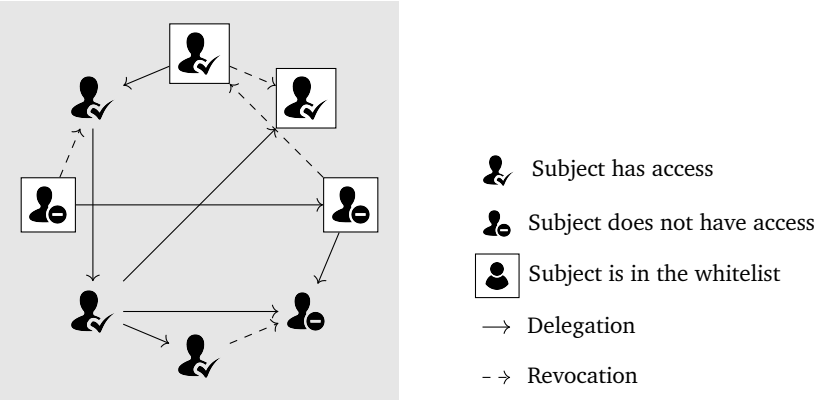


Figure 4.6: The figure shows which subjects have access according to the delegation group with conflict resolution policy. The top-most subject is the only researcher. Solid arrows represent delegations and dashed arrows revocations. Subjects that are in the whitelist are depicted in a white box. The parameters used to generate this example are $N = 8$, $p_g = 0.1$, $p_d = 0.07$, $p_r = 0.125$, and $p_w = 0.5$.

- p_g, the probability that a subject grants another subject,
- p_d, the probability that a subject denies another subject,
- p_r, the probability that a subject is a researcher,
- p_w, the probability that a subject is in the whitelist.

The algorithm for generating attributes constructs a directed random graph where each node is a subject. Grant and deny edges are added to represent delegations (*grant-access* attributes) and revocations (*deny-access* attributes), respectively. Each grant edge is added with probability p_g and each deny edge with probability p_d. For a subject S, an attribute *researcher*(S) is generated with probability p_r and an attribute *whitelist*(S) is generated with probability p_w. In Figure 4.5 we depict an access-control scenario generated with parameters $N = 8$, $p_g = 0.1$, $p_d = 0.07$, $p_r = 0.125$, and $p_w = 0.5$.

We generate access requests by sampling the set of subjects uniformly at random.

Policy 3: Corporate Document Repository The third policy defines how employees access the documents stored in a corporate document repository. The documents are hierarchically organized into directories. The employees

are also hierarchically organized as well: an employee may have a manager, who may in turn have another manager, and so forth. The policy enforced by repository's access-control system is as follows:

$$
\begin{array}{lcll}
pol(S,D) & \leftarrow & access(S,D), \neg revoked(S) & (R1)\\
access(S,D) & \leftarrow & grant(S,D) & (R2)\\
access(S,D) & \leftarrow & grant(S,F), contains(F,D) & (R3)\\
grant(S,D) & \leftarrow & owner(O,D), O:give\text{-}access(S,D) & (R4)\\
grant(M,D) & \leftarrow & manager(M,S), grant(S,D) & (R5)\\
contains(F_1,F_2) & \leftarrow & \mathsf{fs}:subfolder(F_1,F_2) & (R6)\\
contains(F_1,F_3) & \leftarrow & contains(F_1,F_2), contains(F_2,F_3) & (R7)\\
manager(M,S) & \leftarrow & direct\text{-}manager(M,S) & (R8)\\
manager(M,S) & \leftarrow & manager(M,M'), manager(M',S) & (R9)\\
revoked(S) & \leftarrow & manager(M,S), M:revoke(S) & (R10)
\end{array}
$$

Rule *(R1)* states that a subject S can access a document D if S has access to that document and S is not revoked. Rules *(R2-3)* specify that a subject S has access to a document D if S is (directly) granted access to D or S is granted access to a parent folder F that (transitively) contains D. Subjects are granted access to a document D if D's owner O issues the attribute $O : give\text{-}access(S,D)$ (see rule *(R4)*). Rule *(R5)* specifies that managers inherit all grants issued to their subordinates. Rules *(R6-7)* specify that folders contain their subfolders and all folders transitively contained in their subfolders. Rules *(R8-9)* transitively define the manager relation. Finally, rule *(R10)* specifies that a manager M can revoke any of their subordinates S by issuing the credential $M:revoke(S)$.

The input attributes for this policy are: *give-access*, *subfolder*, *revoke*, *direct-manager*, and *owner*. The algorithm used to generate these attributes is parameterized by:

- N, the number of subjects,
- D, the number of folders and documents,
- p_g, the probability that a subject grants access to another subject,
- p_r, the probability that a subject revokes another subject.

The attributes *give-access* and *subfolder* define the subject hierarchy and the directory hierarchy, respectively. To generate these two attributes, the algorithm constructs random trees that define these sets of attributes. For example, to generate the tree that represents the subject hierarchy, the algorithm starts from the root and recursively expand its leafs in a breath-first manner until the tree contains the desired number N of subjects. While expanding the leafs, the algorithm chooses a branching factor from the set $\{4,6,8\}$ uniformly at random. The directory hierarchy is generated

similarly, with the difference that the branching factor is chosen from the set $\{4, 16, 32\}$. All leafs in the generated tree represent the documents stored in the repository. For each document and folder D, the algorithm selects a subject O as its owner uniformly at random and generates the attribute $owner(O, D)$. For any pair of subjects (M, S), the algorithm generates a credential O : *give-access*(S, D) with probability p_g and a credential M : *revoke*(S) with probability p_r.

Access requests are generated by sampling the set of subjects and documents uniformly at random.

4.5.2 Experiments

We now report on our experiments. We first report on the parameters used to generate attributes for the policies and then we turn to our results.

Parameters To investigate the PDP's response time, we varied the parameters in the attribute generation algorithms as follows.

For the policy delegation chains, we set $N = 10^5$. To investigate the impact of the delegation length on the PDP's response time, we selected the values for parameter l from the set $\{1, 3, 7, 15\}$. For each pair $(N, l) \in \{10^5\} \times \{1, 3, 7, 15\}$, we calculated the value for p as follows:

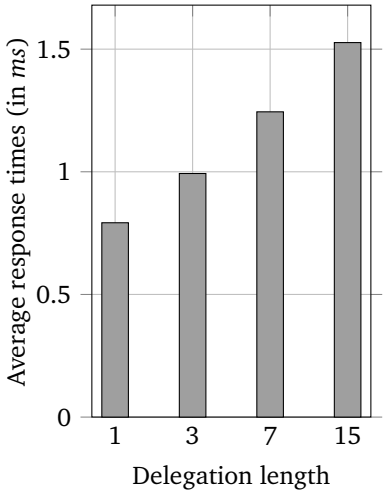
$$p_{N,l} = 10^5 \div \left(l \cdot \left(\frac{N}{l+1} \right)^2 \right)$$

We get $P_{10^5,1} = 4.0 \cdot 10^{-4}$, $P_{10^5,3} \approx 5.3 \cdot 10^{-4}$, $P_{10^5,7} \approx 9.1 \cdot 10^{-4}$, $P_{10^5,15} \approx 17.1 \cdot 10^{-4}$. The value of $p_{N,l}$ is calculated such that the expected number of delegations is 10^5.

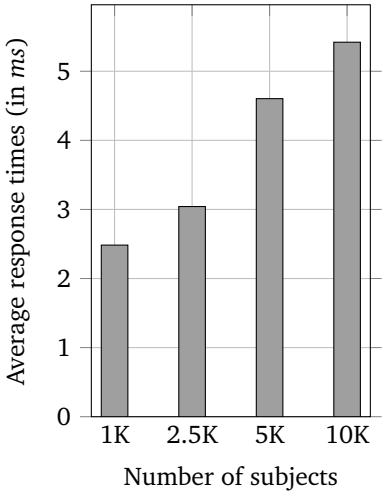
For the policy delegation group with conflict resolution, we set $p_g = 5 \cdot 10^{-3}$, $p_d = \cdot 10^{-3}$, $p_r = 5 \cdot 10^{-3}$, and $p_w = 0.2$. The experimented with values for parameter N from the set $\{1000, 2500, 5000, 10000\}$.

For the policy corporate document repository, we set $p_g = 5 \cdot 10^{-3}$ and $p_r = 5 \cdot 10^{-3}$. We selected the values for parameter N from the set $\{1000, 2500, 5000, 10000\}$.

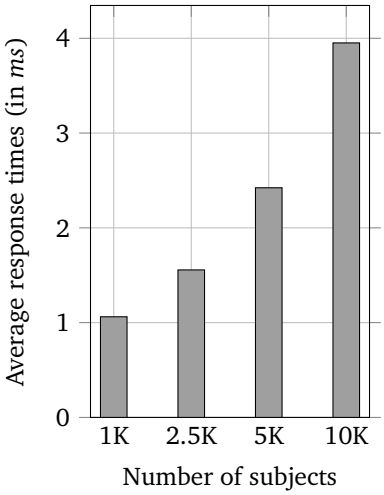
Results For each set of parameters, we generated 10 different sets of attributes and access requests. For each set of attributes, we evaluated all generated access requests and measured the PDP's response time. All times reported in this chapter are in milliseconds. We ran all experiments on a Macbook Pro "Mid 2015" (2.5GHz Intel Core i7/16GB/256GB-Flash).



(a) Average PDP response times for the delegation chains policy.



(b) Average PDP response times for the delegation group with conflict resolution policy.



(c) Average PDP response times for the corporate document repository policy.

Figure 4.7: Average PDP response times for the three policies.

In Figure 4.7 we show three bar charts that depict the PDP's average response times for the three policies, respectively. Each bar chart has multiple bars, where different bars shows the PDP's response time for different set of parameters. The height of each bar shows the PDP's average response time computed over all access requests generated for the given policy and set of parameters.

The first bar chart, given in Figure 4.7a, shows the PDP's average response times for the delegation chains policy. For delegation lengths 1, 3, 7, and 15, the PDP's average response times are 0.792, 0.993, 1.244, and 1.526 milliseconds, with standard deviations 0.176, 0.228, 0.601, and 1.448 milliseconds, respectively. As expected, the PDP's response time increases with the length of the delegation chains. We remark that even when all access requests have delegation chains of length 15, the PDP's response time is reasonable (roughly 1.5ms). Furthermore, we remark that the delegation lengths are short in practice (e.g. OAuth has 1).

In Figure 4.7b we show the PDP's response times for the delegation group with conflict resolution policy. For 1K, 2.5K, 5K, and 10K subjects, the PDP's average response times are 2.483, 3.041, 4.603, and 5.420 milliseconds, with standard deviations 4.936, 4.339, 3.833, and 6.413 milliseconds, respectively. The data shows that the PDP's response time scales to large number of subjects.

Finally, the bar chart of Figure 4.7c presents the PDP's average response times for the corporate document repository policy. For 1K, 2.5K, 5K, and 10K subjects, the PDP's average response times are 1.061, 1.556, 2.423, and 3.951 milliseconds, with standard deviations 0.804, 2.129, 4.409, and 8.588 milliseconds, respectively. Compared to the second policy, the PDP's response time is faster despite that this policy has more BELLOG policy rules.

Overall, the data shows that our BELLOG PDP is efficient: its response time is on average under 10ms per access request for all policies and parameters used in our experiments.

4.6 Related Work

The closest related works to the specification and verification of decentralized composite policies are policy algebras, formal delegation languages, and XACML v3.0, which is an informal policy language.

Policy algebras — such as PBel [28], PTaCL [36], and D-Algebra [71] — are languages for composing a set of policies. A composite policy is a tree, where the internal nodes are composition operators, and the leaf

nodes are core policies. Existing policy algebras cannot express arbitrarily long delegation chains and therefore cannot be used for decentralized composite access control. Moreover, they lack operators for composing *intensionally* defined policy sets, i.e. policy sets that are not fixed at the policy specification time, such as the examples described in Section 5.3.

Delegation languages — such as KeyNote2 [25], DKAL [53], SecPAL [21], RT [65], GP [48], and DCC [4] — allow a policy writer to delegate to other principals authority over attributes and policy decisions. In contrast to BELLOG, these languages support only the permit-override operator for composing policies. Although the permit-override operator is sufficient in their access control setup, this is not the case for decentralized composite policies. Most existing delegation languages are founded on logic programming. We remark that although many-valued extensions for logic programming exist [39, 44, 67], they also cannot express all composition operators found in policy algebras, e.g. the only-one-applicable operator; that is, they are functionally incomplete.

XACML 3 is currently the only access control language supporting decentralized composite access control. Similarly to BELLOG, XACML 3 has four policy decisions and operators for encoding delegation and policy composition. In contrast to BELLOG, XACML is informal and some aspects are underspecified; for example, loop handling in delegation chains is left to implementations. Moreover, XACML 3 has a fixed set of composition operators and new operators cannot be added as syntactic extensions. Kolovski et al. [63] give a formalization of XACML 3 which focuses on delegations and supports only three composition operators. BELLOG, in contrast, supports all finitary composition operators.

Finally, we remark that BELLOG is not meant to be an all-encompassing policy specification language. For example, the constraint-based conditions of [21] are not expressible in BELLOG.

4.7 Technical Details and Proofs

In this section, we formally define the translation of BELLOG's intensional operators defined in Section 4.2.4. We also prove Theorem 14, which establishes the correctenss of BELLOG's translation to stratified Datalog.

4.7.1 Semantics of Intensional Operators

In this section, we define the semantics of the intensional operators $\bigvee$, $\bigwedge$, $\bigoplus$, and $\bigotimes$ as their translation into BELLOG using the function $\mathcal{T}$:

$$\mathcal{T}(p(\vec{X}) \leftarrow \bigvee b(\vec{X} \cup \vec{Y})) = \{p(\vec{X}) \leftarrow b(\vec{X} \cup \vec{Y})\}$$
$$\mathcal{T}(p(\vec{X}) \leftarrow \bigwedge b(\vec{X} \cup \vec{Y})) = \{p(\vec{X}) \leftarrow \neg p_{\mathsf{fresh}}(\vec{X}), p_{\mathsf{fresh}}(X) \leftarrow \neg b(\vec{X} \cup \vec{Y})\}$$
$$\mathcal{T}(p(\vec{X}) \leftarrow \bigoplus b(\vec{X} \cup \vec{Y})) = \{p(\vec{X}) \leftarrow b(\vec{X} \cup \vec{Y}) \wedge \top, p(\vec{X}) \leftarrow \neg p_{\mathsf{fresh}}(\vec{X}),$$
$$p_{\mathsf{fresh}}(X) \leftarrow \neg b(\vec{X} \cup \vec{Y})\}$$
$$\mathcal{T}(p(\vec{X}) \leftarrow \bigotimes b(\vec{X} \cup \vec{Y})) = \{p(\vec{X}) \leftarrow b(\vec{X} \cup \vec{Y}) \wedge \bot, p(\vec{X}) \leftarrow \neg p_{\mathsf{fresh}}(\vec{X}),$$
$$p_{\mathsf{fresh}}(X) \leftarrow \neg b(\vec{X} \cup \vec{Y})\}$$

where $\vec{X} = vars(p)$, $\vec{Y} = vars(b) \setminus \vec{X}$.

As an example we illustrate the operator $\bigoplus$. Consider the simple policy rule $p(X) \leftarrow \bigoplus q(X, Y)$. We have $\vec{X} = \{X\}$, and $\vec{Y} = \{Y\}$. According to the translation function $\mathcal{T}$, this policy rule is translated into the following set of rules:

$$p(X) \leftarrow q(X, Y) \wedge \top \qquad (r_1)$$
$$p(X) \leftarrow \neg p_{\mathsf{fresh}}(X) \qquad (r_2)$$
$$p_{\mathsf{fresh}}(X) \leftarrow \neg q(X, Y) \qquad (r_3)$$

where p_{fresh} is a fresh predicate symbol. For the policy domain $\Sigma = \{a, b\}$, grounding the variable Y in rule r_3 results in two rules, which are (by default) combined with $\vee$

$$p_{\mathsf{fresh}}(X) \leftarrow \neg q(X, a) \vee \neg q(X, b)$$

We rewrite r_2 by replacing $p_{\mathsf{fresh}}(X)$ with $\neg q(X, a) \vee \neg q(X, b)$ and get

$$p(X) \leftarrow \neg(\neg q(X, a) \vee \neg q(X, b)) \qquad (r_4)$$

We simplify r_4 to $p(X) \leftarrow q(X, a) \wedge q(X, b)$. Finally, we ground the variable Y in r_1 and combine the result with the simplified rule r_4:

$$p(X) \leftarrow (q(X, a) \wedge \top) \vee (q(X, b) \wedge \top) \vee (q(X, a) \wedge q(X, b)) \,,$$

which can be simplified, according to the derived operators in Section 3.4, to

$$p(X) \leftarrow q(X, a) \oplus q(X, b) \,.$$

$$\begin{aligned}
\mathcal{T}(p, \forall X.cond) &:= \{p(\vec{Y}) \leftarrow \neg p_{\mathsf{fresh1}}(\vec{Y}), p_{\mathsf{fresh1}}(\vec{Y}) \leftarrow \neg p_{\mathsf{fresh2}}(\{X\} \cup \vec{Y})\} \\
&\quad \cup \mathcal{T}(p_{\mathsf{fresh2}}, cond), \text{ where } \vec{Y} = vars(cond) \setminus \{X\} \\
\mathcal{T}(p, attr \preceq v) &:= \{p(\vec{X}) \leftarrow attr(\vec{X}) \preceq v\}, \text{ where } \vec{X} = vars(attr) \\
\mathcal{T}(p, v \preceq attr) &:= \{p(\vec{X}) \leftarrow v \preceq attr(\vec{X})\}, \text{ where } \vec{X} = vars(attr) \\
\mathcal{T}(p, \neg cond) &:= \{p(\vec{X}) \leftarrow \neg p_{\mathsf{fresh}}(\vec{X})\} \cup \mathcal{T}(p_{\mathsf{fresh}}, cond), \\
&\quad \text{where } \vec{X} = vars(cond) \\
\mathcal{T}(p, cond_1 \wedge cond_2) &:= \{p(\vec{X}) \leftarrow p_{\mathsf{fresh1}}(\vec{X_1}) \wedge p_{\mathsf{fresh2}}(\vec{X_2})\} \cup \mathcal{T}(p_{\mathsf{fresh1}}, cond_1) \\
&\quad \cup \mathcal{T}(p_{\mathsf{fresh2}}, cond_2), \text{ where } \vec{X_1} = vars(cond_1), \\
&\quad \vec{X_2} = vars(cond_2), \vec{X} = \vec{X_1} \cup \vec{X_2} \\
\mathcal{T}(p, \mathsf{t}) &:= \{p \leftarrow \mathsf{t}\}
\end{aligned}$$

Figure 4.8: Translating a policy containment condition *cond* to a set of BELLOG rules.

4.7.2 Reducing Policy Containment to Query Validity

Theorem 13. *Policy containment is polynomially reducible to query validity.*

Proof. Fix a domain Σ and two programs P_1 and P_2 defined over Σ such that $\mathsf{idb}_{P_1} = \mathsf{idb}_{P_2}$. We reduce the problem of deciding $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$ to the problem of query validity $P \models_\Sigma \phi$, where P and ϕ are constructed as follows.

Let $P = \emptyset$. For all rules in P_1 we rename every predicate symbol p in idb_{P_1} to p_1. Similarly, we rename every predicate symbol p from idb_{P_2} in P_2's rules to p_2. The renamed rules are added to P.

In Figure 4.8, we define the function $\mathcal{T}$ that translates a predicate a policy containment condition *cond* to a set of BELLOG rules. The operator $\preceq$ which appears in the generated rule bodies is defined as $p \preceq q = \mathsf{t}$ iff $p \preceq q$, otherwise $p \preceq q = \mathsf{f}$. Note that by Theorem 6 this operator can be expressed in BELLOG. The rules generated by $\mathcal{T}(p_{cond}, cond)$, where *cond* is the containment condition in $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$, are added to P.

Finally, we define the operator $p \rightarrow q$ in the standard way $\neg p \vee q$, and add the following rule to P:

$$\phi \leftarrow (p_{cond}(S, R) \rightarrow (pol_1(S, R) \preceq pol_2(S, R)))$$

By construction we get $P \models_\Sigma \phi$ iff $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$. □

4.7.3 Reducing BELLOG to Stratified Datalog

To show that our translation from BELLOG to stratified Datalog is correct, we proceed as follows. First, we prove that the function ρ correctly encodes the truth values of a BELLOG literal l through the Datalog atoms $\rho(l, \top)$ and $\rho(l, \bot)$. Second, we show that the least fixed point of the BELLOG operator T_{P_i} associated with the BELLOG strata P_i are linked to the least fixed points of their corresponding Datalog operators $T^D_{\mathcal{R}(P_i)}$. Finally, we show that the model of a BELLOG program P is linked to the model of the Datalog translation $\mathcal{R}(P)$ (Theorem 14).

Lemma 9. *For any* BELLOG *interpretation I and any ground literal $l \in \mathcal{L}_{\Sigma(\emptyset)}$, we have*

$$\begin{array}{lcl} I(l) \succeq \bot & \Leftrightarrow & \delta(I) \models_D \rho(l, \bot) \\ I(l) \succeq \top & \Leftrightarrow & \delta(I) \models_D \rho(l, \top) \end{array}$$

Proof. By case distinction on the literal l. The case where the literal l is a positive atom a follows immediately from the definitions of δ and ρ. Below, we consider the cases where l is $\sim a$ and $\neg a$.

- Case $l = \sim a$. We have $\rho(\sim a, \bot) = \rho(a, \top)$ and $\rho(\sim a, \top) = \rho(a, \bot)$.
  - Assume $I(l) \succeq \bot$. Then $I(\sim a) \in \{\bot, \mathrm{t}\}$, and so $I(a) \in \{\top, \mathrm{t}\}$. It follows that $I(a) \succeq \top$, and by definition of δ it must be that $\rho(a, \top) \in \delta(I)$. Therefore, $\delta(I) \models_D \rho(a, \top)$. Since $\rho(l, \bot) = \rho(a, \top)$, we get $\delta(I) \models_D \rho(l, \bot)$.
  - Assume $\delta(I) \models_D \rho(l, \bot)$. Then, $\rho(a, \top) \in \delta(I)$. By definition of δ, we have $I(a) \succeq \top$, and so $I(a) \in \{\top, \mathrm{t}\}$. Since $l = \sim a$, we conclude that $I(l) \in \{\bot, \mathrm{t}\}$, and thus $I(l) \succeq \bot$.
  - Assume $I(l) \succeq \top$. Then $I(\sim a) \in \{\top, \mathrm{t}\}$, and so $I(a) \in \{\bot, \mathrm{t}\}$. It follows that $I(a) \succeq \bot$, and we get $\rho(a, \bot) \in \delta(I)$. Therefore, $\delta(I) \models \rho(a, \bot)$, and since $\rho(l, \top) = \rho(a, \bot)$, we conclude that $\delta(I) \models_D \rho(l, \top)$.
  - Assume $\delta(I) \models_D \rho(l, \top)$. Then $\rho(a, \bot) \in \delta(I)$, and we get $I(a) \succeq \bot$, and so $I(a) \in \{\bot, \mathrm{t}\}$. Since $l = \sim a$, we conclude that $I(l) \in \{\top, \mathrm{t}\}$, and so $I(l) \succeq \top$.
- Case $l = \neg a$. We have $\rho(\neg a, \bot) = \neg\rho(a, \top)$ and $\rho(\neg a, \top) = \neg\rho(a, \bot)$.
  - Assume $I(l) \succeq \bot$. Then $I(\neg a) \in \{\bot, \mathrm{t}\}$, and so $I(a) \in \{\bot, \mathrm{f}\}$. We have $I(a) \not\succeq \top$, and so $\rho(a, \top) \notin \delta(I)$. Therefore, $\delta(I) \not\models_D \rho(a, \top)$. Since $\rho(l, \bot) = \neg\rho(a, \top)$, we get $\delta(I) \models \rho(l, \bot)$.

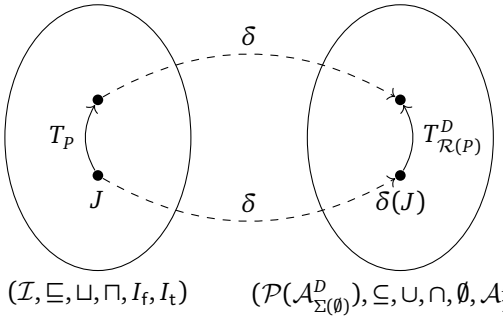


Figure 4.9: The BELLOG domain (left) and the Datalog domain (right). By Lemma 10, the result of applying BELLOG operator T_P on I is identical to the result of applying $T^D_{\mathcal{R}(P)}$ on $\delta(I)$.

- Assume $\delta(I) \models_D \rho(l, \bot)$. Then, $\rho(a, \top) \notin \delta(I)$, and so $I(a) \not\succeq \top$. We get $I(a) \in \{\bot, \mathrm{f}\}$, and since $l = \neg a$ we have $I(l) \in \{\bot, \mathrm{t}\}$. We conclude that $I(l) \succeq \bot$.
- Assume $l(l) \succeq \top$. Then, $I(\neg a) \in \{\top, \mathrm{t}\}$, and so $I(a) \in \{\top, \mathrm{f}\}$. We have $I(a) \not\succeq \bot$, and so $\rho(a, \bot) \notin \delta(I)$. Therefore, $\delta(I) \not\models_D \rho(a, \bot)$. Since $\rho(l, \top) = \neg\rho(a, \bot)$, we conclude that $\delta(I) \models_D \rho(l, \top)$.
- Assume $\delta(I) \models_D \rho(l, \top)$. Then, $\rho(a, \bot) \notin \delta(I)$, and so $I(a) \not\succeq \bot$. We get $I(a) \in \{\top, \mathrm{f}\}$, and since $l = \neg a$ we have $I(l) \in \{\top, \mathrm{t}\}$. We conclude that $I(l) \succeq \top$.

This concludes our proof. □

Lemma 10. *Let P be a* BELLOG *program. For any* BELLOG *interpretation $I \in \mathcal{I}$, we have*

$$\delta(T_P(I)) = T^D_{P_D}(\delta(I))$$

where $P_D = \mathcal{R}(P)$ is the Datalog translation of P.

Proof. We first prove that $\delta(T_P(I)) \subseteq T^D_{P_D}(\delta(I))$ and then that $\delta(T_P(I)) \supseteq T^D_{P_D}(\delta(I))$.

- Assume $\rho(a, \bot) \in \delta(T_P(I))$. Then we have $T_P(I)(a) \succeq \bot$. By definition fo T_P and by the monotinicity of $\wedge$ and $\vee$, there is a rule $a \leftarrow l_1, \ldots, l_n$ in $P^{\downarrow}$ such that $I(l_i) \succeq \bot$ for $1 \leq i \leq n$ (1).

By definition of $\mathcal{R}(P)$, there is a rule $\rho(a,\bot) \leftarrow \rho(l_1,\bot),\ldots,\rho(l_n,\bot)$ in $\mathcal{R}(P)$. From (1) and Lemma 9, we get $\delta(I) \models_D \rho(l_i,\bot)$ for $1 \leq i \leq n$. By definition of $T^D_{P_D}$, we get $\rho(a,\bot) \in T^D_{P_D}(\delta(I))$. The case for atoms of the form $\rho(a,\top)$ is analogous.

- Assume $\rho(a,\bot) \in T^D_{P_D}(\delta(I))$. By definition of $T^D_{P_D}$, there is a rule $\rho(a,\bot) \leftarrow \rho(l_1,\bot),\ldots,\rho(l_n,\bot)$ in $\mathcal{R}(P)$ such that $\delta(I) \models_D l_i$ for $1 \leq i \leq n$. By Lemma 9, we have $I(l_i) \succeq \bot$ for $1 \leq i \leq n$. By definition of $\mathcal{R}(P)$, we must have a rule $a \leftarrow l_1,\ldots,l_n$ in P. We conclude that $T_P(I)(a) \succeq \bot$. Therefore $\rho(a,\bot) \in \delta(T_P(I))$.

This concludes our proof. □

Lemma 11. *Let P be a a* BELLOG *stratum. For any input I for P, we have*

$$\delta([\![P]\!]_I \sqcup I) = T^D_{P_D} \uparrow^\omega (\delta(I))$$

where $P_D = \mathcal{R}(P)$ is the Datalog translation of P.

Proof. For the left-hand-side, recall that $[\![P]\!]_I$ is computed as the least fixed point of the operator $T_{P^{\downarrow\lhd I}}$, which can be iteratively computed as $T_{P^{\downarrow\lhd I}} \uparrow^{i+1} = T_{P^{\downarrow\lhd I}}(T_{P^{\downarrow\lhd I}} \uparrow^i)$ where $T_{P^{\downarrow\lhd I}} \uparrow^0 = I_{\mathsf{f}}$. Furthermore, $\delta([\![P]\!]_I \sqcup I) = \delta([\![P]\!]_I) \cup \delta(I)$.

For the right-hand-side, recall also that $T^D_{P_D} \uparrow^\omega (\delta(I))$ is computed as

$$T^D_{P_D} \uparrow^{i+1} (\delta(I)) = T^D_{P_D}(T^D_{P_D} \uparrow^i) \cup T^D_{P_D} \uparrow^i (I)$$

where $T^D_{P_D} \uparrow^0 = \delta(I)$. Note that I is an input, and therefore we have

$$T^D_{P_D} \uparrow^\omega (\delta(I)) = T^D_{P_D^{\downarrow\lhd\delta(I)}} \uparrow^\omega (\emptyset) \cup \delta(I)$$

because for any edb atom $\rho(a,\bot) \in T^D_{P_D} \uparrow^i (\delta(I))$ iff $\rho(a,\bot) \in \delta(I)$, for $i \geq 0$.

To prove the lemma, it remains to show that $\delta([\![P]\!]_I) = T^D_{P_D^{\downarrow\lhd\delta(I)}} \uparrow^\omega (\emptyset)$. This follows by induction on the iterations, where the inductive step follows from Lemma 10. □

Theorem 14. *Given a stratified* BELLOG *program P and an input I for P, we have $[\![P]\!]_I = \bar{\delta}([\![\mathcal{R}(P) \cup \delta(I)]\!]_D)$.*

Proof. Since $\delta(\overline{\delta}([\![\mathcal{R}(P) \cup \delta(I)]\!]^D)) = [\![\mathcal{R}(P) \cup \delta(I)]\!]^D$, it is sufficient to prove that $\delta([\![P]\!]_{I,\Sigma}) = [\![\mathcal{R}(P) \cup \delta(I)]\!]^D$. The proof is straightforward by induction on the models computed for each of the strata $P_1, \cdots, P_n$. The only non-trivial observation is that the model of a stratum P_i is the join of the model of the previous stratum P_{i-1} and the least fixed point of the current stratum's operator. It is easy to see however that the function δ is join-preserving; that is, for any two BELLOG interpretations I, J, we have $\delta(I \sqcup J) = \delta(I) \cup \delta(J)$. □

Chapter 5
Fail-Security

Modern access-control systems are often distributed, and therefore subject to communication and component failures. If failures affect the availability of information needed for security decisions, then access-control systems must, either implicitly or explicitly, handle these failures. This concern permeates all access-control domains. For example, firewalls must operate even when their log engines crash [33] or rule updates fail [77], web applications must service requests even if authentication services are unresponsive [72], delegation systems must evaluate requests even when they cannot update their revocation lists, and perimeter security systems must control access even when the wireless channels to their central database are jammed. In such settings, the access decisions of a PDP cannot be understood without considering the PDP's failure handlers as well.

The access-control community has not thus far rigorously studied the effects of failure handlers on access decisions. One reason for this is that simply interpreting failures as denies appears sufficient to conservatively approximate the PDP's desired behavior. This would suggest that the policy writer need not overly concern himself with analyzing the PDP's failure handlers. However, failures can affect the PDP's decisions in surprising and unintended ways. Such simplistic approximations are not only inflexible, they also do not necessarily result in secure systems. As an example, we describe later how the conservative approach of replacing failures with denies had been originally adopted in the XACML v3.0 standard, and was later dropped due to its insecurity.

Given that failure handling influences the PDP's access decisions, it follows that formal analysis frameworks for access-control should account for the PDP's failure handlers. Only then can security guarantees be derived for the PDP's access decisions, both in the presence and absence of failures. Analysis techniques for obtaining such security guarantees would be of immediate practical value because existing access-control systems separate failure handling from the "normal" (typically declarative) policy interpreted by the PDP, i.e. the policy that defines the PDP's decisions when no failures occur. The logic that decides access requests is therefore split into two parts. This separation makes the PDP's behavior difficult to understand and analyze.

Existing formal analysis frameworks for access-control policies are inadequate for the task at hand. This is neither an issue with the expressiveness

of their formal languages nor the complexity of their decision problems. Rather, they lack (i) a system and attacker model tailored for failure scenarios, (ii) idioms for specifying failure handlers, and (iii) methods for verifying *fail-security requirements*, i.e. security requirements that describe how distributed access-control systems ought to handle failures. Thus, currently it is all but impossible to derive security guarantees that extend beyond the PDP's normal behaviors. In this paper, we show how to realize these three artifacts using the BELLOG analysis framework [88].

In this chapter, we systematically analyze the role of failure handling in access-control systems. We give examples of systems that exhibit different failure-handling flaws; a common thread in these systems is their seeming conformance to security common sense.

We also demonstrate how the PDP, including its failure handlers, can be modeled and analyzed using the BELLOG access-control framework. In particular: (i) We investigate seven real-world access-control systems and use these to extract a system and an attacker model tailored for analyzing the effect of failures on the PDP's decisions. (ii) We derive common failure-handling idioms from these systems, which can be readily encoded in BELLOG. (iii) Through examples, we show how to express fail-security requirements and we provide a tool to automatically verify them for a given PDP with respect to our attacker model. We argue that our verification method is effective by demonstrating how the three kinds of security flaws mentioned above can be discovered.

Organization In Section 5.1, we give examples of PDP failure handlers and fail-security requirements for access-control systems. In Section 5.2 extend our system model from Chapter 2 with an attacker that can cause communication and component failures. In Section 5.3, we show how BELLOG is used to specify the examples from Section 5.1. In Section 5.4, we analyze these examples with respect to their fail-security requirements. We discuss related work in Section 5.5.

5.1 Motivating Examples

As motivation, we use the XACML v3.0 standard to show that approximating failures with denials, although seemingly conservative, can lead to insecure systems. Through our second and third examples, taken from the web application and grid computing domains, we illustrate the common PDP implementation pattern that treats failure handlers as a separate add-on to the normal policy engine. We show how this separation makes

```
1 evaluate(Request req)
2   Set decisions
3   foreach (pol in policies) // loop through all policies
4     try
5       decision = pol.evaluate(req) // evaluate the current policy
6       if (pol.issuer == admin) or authorize(pol, req)
7         // consider policies that are issued
8         // or authorized by admin
9         decisions.add(decision)
10      else
11        // ignore policies that are neither issued
12        // nor authorized by admin
13        pass
14    catch (EvaluationException e)
15      // ignore policies that cannot be evaluated
16      pass
17  return compositionOperator.apply(decisions)
```

Figure 5.1: PDP module for evaluating XACML v3.0 policy sets. The methods pol.evaluate(req) and authorize(pol, req) throw an exception if the PDP fails to execute them.

understanding and analyzing PDPs particularly difficult, resulting in systems open to attacks.

5.1.1 XACML v3.0

XACML v3.0 is an OASIS standard for specifying access-control policies [94]. XACML v3.0 policies are issued by principals and evaluated by a PDP. A policy issued by the PDP's administrator is called *trusted*; otherwise, it is *non-trusted*. The administrator specifies whether a non-trusted policy is authorized to decide a given request. XACML v3.0 policies are grouped into *policy sets* and their decisions are combined with composition operators, such as permit-overrides, which grants access if at least one policy grants access. To decide a given request, the PDP first computes the decisions of all policies in the set. Afterwards, it checks which non-trusted policies are authorized by the administrator. Finally, the PDP combines the decisions of the trusted policies and the authorized non-trusted policies using the policy set's composition operator.

An XACML v3.0 PDP obtains all information needed for policy evaluations, such as attributes and credentials, from PIPs. The XACML v3.0 standard, up to Revision 16, stated that the PDP should refrain from using policies that could not be evaluated or authorized due to communication and PIP failures. This decision follows the intuitive idea that all *suspicious*

policies should be excluded from the PDP's decision. Figure 5.1 specifies such a PDP, including its failure handler, in pseudo-code. Although this failure handler is inflexible, the committee did not anticipate other consequences on the PDP's decisions apart from always making them more conservative (i.e. less permissive). This however turned out to be wrong.

When the proposed failure-handling behavior was considered together with the deny-overrides composition operator, the following attack was discovered [95]. Consider a request r and a policy set P that contains one trusted policy P_1 that grants r and one authorized non-trusted policy P_2 that denies r. P's decisions are combined with deny-overrides. If the PDP successfully evaluates P_1 and fails to evaluate P_2, then the PDP will grant r, even though it does not have all the necessary information to make this decision. In this case, the attacker can simply launch denial-of-service attacks against PIPs and obtain a grant decision for r. In Section 5.4 we show how this attack can be found through automated analysis using our BELLOG access-control framework.

This example illustrates that a PDP's failure handlers, regardless of their simplicity, can affect access decisions in surprising ways. In this example, the failure-oblivious composition of sub-policies is the root of the security flaw. To remedy this flaw, the XACML v3.0 standard currently uses a designated policy decision (the indeterminate *IN*) for every policy that cannot be evaluated due to failures. Consequently, failure handling is now a concern of the security engineer.

5.1.2 Authorizations in Web Apps

Web applications use access-control frameworks to specify and manage user permissions. Examples include the Java Authentication and Authorization Service JAAS, Apache Shiro, and Spring Security. Basic policies can be specified using declarative policy languages. The PDP loads policies and evaluates them within its *authorization* method. A reoccurring problem is that the PDP fails to load a policy due to syntactic errors or missing files. To deal with this problem, administrators often maintain a *default policy* that serves as a fallback option. Use of the default policy is typically conditioned on whether logging is enabled. This fallback approach imposes the following fail-security requirement:

Fail-security requirement 1 (FR1): *When the PDP cannot compute an access decision due to malformed or missing policy, then it uses the default policy if logging is enabled, and it denies access otherwise.*

```
1 isAuthorized(User usr, Object obj, List aclIDs)
2   try
3     foreach (id in aclIDs) // loop through all ACLs
4       if (readAcl(id).grants(usr,obj))
5         // grant access if an ACL evaluates to true
6         return true
7   catch (ReadAclException e)
8     // evaluate the default ACL if an exception is thrown
9     return def.grants(usr,obj) and logger.isEnabled()
10  return false
```

Figure 5.2: A PDP module for the web app example.

To illustrate this, consider the case where the PDP composes finitely many access-control lists (ACLs) using the permit-overrides operator, which permits access if at least one of the ACLs permits access, and denies access otherwise. To adhere to **FR1**, the PDP must invoke the failure handler if and only if none of the ACLs permits access and at least one of them is malformed. The failure handler in this example would evaluate the default ACL `def` and check whether logging is enabled. Figure 5.2 gives a straightforward authorization method for this scenario in pseudo-code. The method takes as input a user object `usr`, the requested object `obj`, and a list `aclIDs` of ACL identifiers. The method `readAcl(id)` returns the ACL object corresponding to `id`, and throws a `ReadAclException` exception when it cannot find or parse the associated ACL. The default ACL `def` is hard-coded in the method.

The pseudo-code describes a correct permit-overrides operator for ACLs under normal conditions, i.e. when there are no failures. The catch block is also correct as it intuitively follows the structure of **FR1**. However, the failure handling is overly eager in that if a `ReadAclException` is thrown while evaluating an input ACL then the PDP stops evaluating the remaining input ACLs and jumps to the catch block. This method therefore does not satisfy **FR1**: if a list of two ACL identifiers is passed to the method and the first ACL fails to load, then the method immediately consults `def`, which would be wrong if the second ACL would permit access.

This problem is rooted in the overly eager invocation of the failure handler. The problem here is not an instance of syntactic vulnerability patterns, such as *overly-broad throws declaration* and *overly-broad catch block* [56], and it cannot be solved for example by simply moving the try-catch construct inside the `for` loop. One solution would be to delay the invocation of the failure handler until all the ACLs have been evaluated.

To conclude, because existing web access-control frameworks typically separate failure handling from the normal policy of the PDP, it is difficult to gain confidence in their security. To rise to this challenge, policy specification languages and their analysis frameworks should also account for the interactions that result from the separation. In Section 5.3 we give a formal specification of the method of Figure 5.2, and we verify the specification against **FR1** in Section 5.4, which reveals the discussed problem.

5.1.3 Authorizations in Grids

In grid computing platforms, resources (such as storage space) are located in different domains. Each domain has an owner, and only one PDP controls access to the domain's resources. It is however infeasible for each PDP to manage authorizations for all subjects from all domains. Domain owners therefore delegate authorization management to *trusted* subjects, possibly from other domains. These subjects may then issue tokens to authorize other subjects and to further delegate their rights. All tokens are stored as digital credentials. Subjects then submit their credentials, alongside their access requests, to a PDP. In addition, it is sometimes necessary to revoke subject's credentials, for example when dealing with ex-employees. A common solution is to store all revoked credentials on a central revocation server.

A *(delegation) chain* for a subject S is a transitive delegation from the domain owner to S. We say that a delegation chain is non-revoked if none of the delegations in the chain has been revoked. A given domain's PDP grants access if the subject has at least one *non-revoked* delegation chain or the subject is the domain's owner. The revocation server may sometimes be unavailable, for example due to lost network connectivity. Denying all access in the case of failures may be too restrictive as the unavailability of some resources, to selected subjects, would be too costly [91]. One fail-security requirement that reflects this notion is:

Fail-security requirement 2 (FR2): *When the PDP cannot check whether a subject has at least one non-revoked delegation chain due to failures, the PDP grants access if the subject is a direct delegate of the owner; otherwise it denies access.*

The rationale is that the owner rarely revokes his direct delegates. This requirement also states that the owner chooses to ignore all delegations issued by subjects, including his direct delegates, whose delegation chains cannot be checked. Figure 5.3 illustrates one delegation scenario and shows which subjects are granted access according to **FR2**. In the depicted

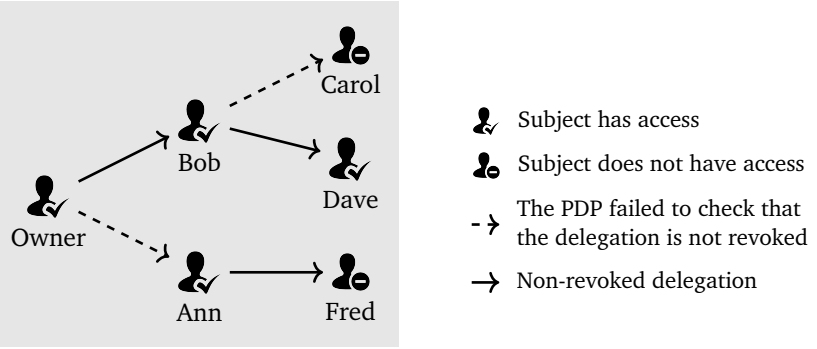


Figure 5.3: The figure shows which subjects in the depicted scenario have access according to **FR2**

scenario, Ann has access because her delegation is issued by the owner. Bob and Dave have access because they have non-revoked chains. Fred and Carol are denied access because they do not have non-revoked chains and they are not the owner's direct delegates.

Existing delegation languages do not specify failure handling within policy specifications, but rely on having failure handlers within the PDP. This approach, which separates the delegation logic from failure handling, is described in [25]. Based on these guidelines, Figure 5.4 depicts a possible PDP design for our grid access-control scenario. In Figure 5.4a, we specify the normal policy of the PDP using two BELLOG policy rules. The policy grants access to a subject X if X is an owner or has a (transitive) delegation chain from an owner. Before evaluating the policy, the PDP checks whether each supplied delegation is still valid by querying the revocation server. If it is revoked then the PDP discards the delegation.

Considered separately, the normal policy and the failure handler of Figure 5.4 intuitively conform to *FR2*. Their interaction however leads to a subtle attack. The attack, described in Section5.4, results from the preemptive masking of failures. We were unable to find the attack before specifying this PDP in BELLOG; we believe this applies to most security engineers.

Finally, we remark that our goal is not to promote particular fail-security requirements; they can be determined for example from a risk analysis of each deployed system. Our goal is instead to raise and address the need for analyzing access-control systems in the presence of (malicious) failures with respect to their security requirements. We stress that even systems

```
pol(X) :- owner(X)
pol(X) :- pol(Y), grant(Y,X)
```

(a) Access-control policy del.policy.

```
isAuthorized(Subject subj, List delegations)
  datalogEngine.load(del.policy) // load the policy
  // loop through all delegations
  foreach ((delegator, delegatee) in delegations)
    try
      if (rev.query(delegator, delegatee) == false)
        // keep only non-revoked delegations
        datalogEngine.assert(grant(delegator, delegatee))
      else
        // discard revoked delegations
        pass
    catch (QueryException e)
      if isOwner(delegator)
        // keep delegations issued by the owner
        datalogEngine.assert(grant(delegator, delegatee))
  return datalogEngine.check(pol(subj))
```

(b) PDP module, where datalogEngine represents a Datalog interpreter. The method rev.query() may throw an exception.

Figure 5.4: A PDP module for the grid example.

that are intended to conform to simple conservative requirements, such as the fail-safe principle (deny all access if there is any failure) [79], are not exempt from failure-handling flaws, and thus should also be analyzed.

5.2 Attacker Model

We now present our attacker model. As described in Chapter 2, PDPs obtains attributes from PIPs through remote queries. In our system model, PDPs and PEPs cannot fail, whereas PIPs can fail. Furthermore, the communication channels between the PDP and the PIPs can fail, while all other channels (e.g. PEP-to-PDP) are reliable.

We consider an attacker who can cause any remote query to fail. Note that our attacker model subsumes all failures due to benign causes. The attacker can in particular cause complete channel failure by causing all remote queries through that channel to fail. In Figure 5.5, we depict in red the components that our attacker can cause to fail.

The attacker cannot, however, forge credentials or forge and replay past remote queries and obsolete responses. To this end, we assume that

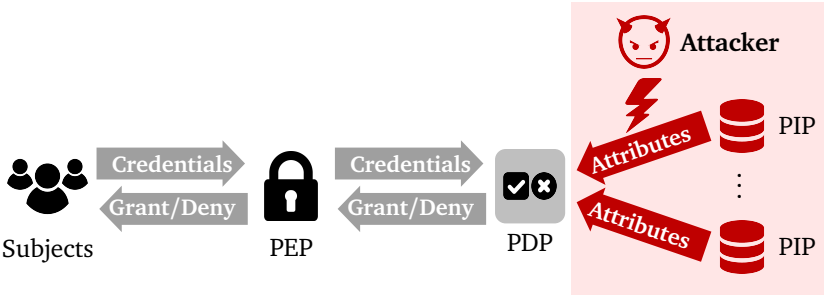


Figure 5.5: Extended system model with an attacker that can cause communication and component failures.

all communication channels are authentic and have freshness guarantees (through timestamps, nonces, etc.).

5.3 Specifying PDPs with Failure Handling

In this section, we first describe three failure-handling idioms, derived by analyzing seven existing access-control systems and their failure handlers. These idioms are abstractions we use for modeling failure-handling mechanisms. We then show how BELLOG can be used to specify the failure-handling idioms and the PDPs of Section 5.1, including their failure handlers.

5.3.1 Failure-handling Idioms

To understand how existing systems handle communication failures, we have inspected the documentation of seven access-control systems; see Table 5.1. Our analysis revealed three failure-handling idioms, which are sufficient to describe how failures are handled in these systems. To describe the idioms, we abstract a PDP as evaluating a request through a finite sequence of computation and communication steps; hereafter referred to as *events*. We assume that computation events always terminate successfully, while communication events either terminate successfully or fail. Note that similar abstractions exist for exception handling in programming languages [55, 64].

Fallback The fallback idiom abstracts the failure handlers that use *fallback* information sources when the communication channels to the primary information sources fail. If a communication event fails then it is re-executed

System	Failure-handling Idioms
Cisco IOS [31]	Catch
KABA KES-2200 [60]	Catch
Kerberos [62]	Fallback
RedHat Firewall [77]	Catch
Spring Framework [83]	Propagate, Catch
WebSphere [93]	Catch
XACML PDPs [96]	Propagate, Catch

Table 5.1: Analyzed access-control systems and their failure-handling idioms.

using the fallback source. The fallback source can be, for example, a backup of a primary information source. This idiom is used in access-control systems whose primary authentication services are unreliable. For example, Kerberos [62] can fall back on local user/password lists when its primary LDAP authentication service is unavailable.

To instantiate this idiom, a fallback source must be configured for each information source that may fail. Although the fallback source may be periodically synchronized with the information source, it may nevertheless provide stale information of inferior quality.

Catch This idiom abstracts the failure handlers that catch failures and then enforce alternative access-control policies. The catch idiom is analogous to exception handling in programming languages where the failure to execute a given procedure is handled by a designated procedure. In terms of the PDP's execution, whenever an event fails, the execution branches to another (alternative) sequence of events.

We can use this idiom to implement a system that meets **FR2**. The system's alternative access-control policy would contain only the grants for the owners' direct delegates. Systems that employ this idiom include: KABA KES-2200 [60], which is a token-based physical access-control system that upon power failures is configured to either grant or deny all requests; IBM WebSphere [93], whose exception handlers evaluate designated error-override policies; and Cisco IOS [31] and RedHat Firewall [77], which in case of failures use alternative rule sets.

Propagate Both the fallback and the catch idioms handle failed events immediately upon failure. In contrast, **FR1** requires failures to be handled

after all the ACLs have been evaluated. The propagate idiom abstracts the mechanisms for meeting requirements with such "delayed" failure handling. Whenever an event fails, the PDP pushes a designated *error value* as the input to all subsequent events.

For example, to meet **FR1**, ideally an error value that indicates a failure to evaluate an ACL is propagated. The default ACL is evaluated iff no ACL grants a given request and the PDP failed to evaluate at least one ACL. Note that the failure handler of Figure 5.2 implementing **FR1** is, however, an instance of the catch idiom. Systems that employ the propagate idiom include XACML PDPs [96], which propagate indeterminate policy decisions, and Spring-based applications [83], which propagate data access exceptions.

5.3.2 Specifying PDPs in BELLOG

We now explain how a PDP, i.e. its normal policy and its failure handlers, can be specified in BELLOG. We illustrate this by specifying the examples of Section 5.1.

A PDP's behavior is determined by three elements:

(i) the PDP inputs, namely credentials forwarded by a PEP, attributes stored locally at the PDP or obtained from PIPs,
(ii) the policy evaluated by the PDP, and
(iii) the failure-handling procedures used when the communication channels between the PDP and PIPs fail.

In the following, we describe how these elements can be specified in BELLOG.

Inputs We represent attributes as described in Section 4.2. For example, ann:*public*(file) is interpreted as "Ann asserts that file is public". We model attributes obtained from PIPs as *remote queries*, which check whether a specified attribute is stored at a designated PIP. We write remote queries as ann : *public*(file)@pip, where ann : *public*(file) is an attribute and pip is a PIP identifier. Formally, remote queries are represented as atoms where the PIP identifier is appended to the predicate symbol; for example, ann:*public*(file)@pip is represented with the atom *public_pip*(ann, file).

The PDP's input consists of credentials forwarded by a PEP, attributes stored at the PDP, and attributes obtained using remote queries to PIPs. We model a PDP's input as BELLOG input. Given a BELLOG input I and a credential *cred*, the truth value I(*cred*) is: t if *cred* is a credential forwarded by the PEP, and f if *cred* is not forwarded by the PEP. Given an attribute

attr, the truth value $I(attr)$ is: t if *attr* is stored at the PDP, and f if *cred* is not stored by the PEP. For a remote query *attr*@pip, $I(attr@\mathsf{pip})$ is: t if *attr* is stored at pip, f if *attr* is not stored at pip, and $\bot$ if a failure prevents the PDP from obtaining *attr* from pip.

Policies For simplicity, in this chapter we confine our attention to PDPs that enforce delegation policies, such as SecPAL [21], RT [65], Binder [38], and DKAL [53]. Any policy written in these languages can be specified in BELLOG; see Section 4.2 for details about specifying policies in BELLOG. We do not consider PDPs that enforce policies with composition operators because we use BELLOG's truth value $\bot$ to represent policy evaluation failures. This technical limitation can be lifted by extending BELLOG with additional truth values.

Failure Handling We define the *error-override* operator as

$$p \blacktriangleright q := p \overset{\bot}{\mapsto} q \ ,$$

where p and q are rule bodies. The construct $p \blacktriangleright q$ evaluates to q's truth value if p's truth value is $\bot$; otherwise, the result of p is taken. Using this operator, we can model the failure-handling idioms given in Section 5.3.1. Consider the remote query *cred*@pip, which checks whether the credential *cred* is stored at pip. To instantiate the *fallback* idiom, where fallback is the fallback PIP's identifier, we write *cred*@pip $\blacktriangleright$ *cred*@fallback.

To illustrate the *catch* idiom's specification, consider a PDP with the following two policies.

$$pol_1(X) \leftarrow empl(X)@\mathsf{db} \qquad \text{(Policy } P_1\text{)}$$
$$pol_2(X) \leftarrow stud(X) \qquad \text{(Policy } P_2\text{)}$$

Here the atom $pol_i(X)$ denotes policy P_i's decision. The communication between the PDP and the PIP db can fail. Imagine that the PDP instantiates the catch idiom and uses P_2 whenever it cannot evaluate P_1 due to failures. We can specify this failure handler as

$$pol(X) \leftarrow pol_1(X) \blacktriangleright pol_2(X) \ .$$

The *propagate* idiom is the default failure handler used in BELLOG specifications. That is, we need not explicitly encode it using BELLOG rules. This is because we represent failures with $\bot$, and this truth value is always propagated unless it is explicitly handled with an operator such as error-override.

5.3.3 Examples

We now specify the PDPs discussed in Section 5.1.

XACML v3.0 We first observe that the failure handling in Figure 5.1 is independent of the policies in a policy set and of the composition operator used to compose their decisions. Therefore, to illustrate the specification of a complete PDP (i.e. one that contains both a normal policy and failure handling), we choose deny-overrides as the designated composition operator for the policies. In BELLOG, the deny-overrides operator corresponds to the infinitary meet $\bigwedge$ over the truth ordering $\preceq$; see Section 4.2.4.

The following BELLOG program models the XACML v3.0 PDP's failure handling with the deny-overrides operator:

PDP Specification 1 (S1):

$$
\begin{aligned}
pol\_set(Req) &\leftarrow \bigwedge \big(X : pol(Req) \triangleleft auth(X, Req) \triangleright t\big)\\
auth(X, Req) &\leftarrow admin(X)\\
auth(X, Req) &\leftarrow auth(X, Req)@\mathsf{check} \blacktriangleright f\\
X : pol(Req) &\leftarrow pol(X, Req)@\mathsf{eval} \blacktriangleright t
\end{aligned}
$$

We use Req to denote access requests and X to denote principals. For brevity, we assume that each principal X has one policy for all requests, denoted by $X : pol(Req)$. The outcome of evaluating the policy issued by the principal X is represented by $pol(X, Req)@\mathsf{eval}$, where eval represents the PDP's policy evaluation procedure. To represent whether X is authorized for a given Req, we write $auth(X, Req)$. Therefore $auth(X, Req)@\mathsf{check}$ is a query to the procedure check to check whether a non-trusted policy issued by X is authorized to give decisions for the request Req.

To encode that a policy is dropped if a PDP cannot evaluate it, we use the (_ ► t) pattern. This is because t is the identity element for the $\bigwedge$ operator. Thus, if there is an error while evaluating a policy, then t is returned, which does not influence the final outcome of the composition. It formalizes that the policy was ignored. If we were modeling another composition operator, then that operator's identity element would be used.

To specify that a policy is dropped if a PDP cannot check its authorization, we use the (_ ► f) pattern. This means that a policy is treated as unauthorized and thus its decision is ignored (i.e. mapped to t through the if-then operator).

Finally, the for-loop is implicitly modeled using $\bigwedge$ and the if-then operator. The $\bigwedge$ operator returns the decision evaluated over the set of policies

of all principals. Those policies that are not authorized are treated as the identity element and thus do not influence the result.

Authorizations in Web Applications To model the web application scenario given in Section 5.1, we suppose that there are n input ACLs and one default ACL. We specify the authorization method given in Figure 5.2 as follows.

PDP Specification 2 (S2):

$$\begin{aligned} pol(U,O) \leftarrow & (isGranted(U,O)@\mathsf{acl}_1 \overset{f}{\mapsto} \cdots \\ & \cdots \overset{f}{\mapsto} isGranted(U,O)@\mathsf{acl}_n) \\ & \blacktriangleright (isGranted(U,O)@\mathsf{def} \wedge logging) \end{aligned}$$

We model the ACL i's evaluation of the access request (U,O) with the atom $isGranted(U,O)@\mathsf{acl}_i$, where U represents the user and O the requested object. We model the logger's status with the credential *logging*, and instantiate the catch idiom using the error-override operator. To specify the list iterator of Figure 5.2, we unroll the loop's n iterations. We use the f-override operator ($\overset{f}{\mapsto}$) to capture that the PDP evaluates the ACL i if the ACL $i-1$ does not permit the request. This models the exit from the loop when the decision is grant. Similarly, the exit from the loop when there is a failure is captured with the catch idiom using the $\blacktriangleright$ operator. This is because if $isGranted(U,O)@\mathsf{acl}_i$ evaluates to $\bot$ then the entire expression on the left-hand side of $\blacktriangleright$ is evaluated to $\bot$ as well.

We recall that this specification violates *FR1* because the PDP does not evaluate all ACLs if it fails to evaluate, for example, the first ACL. The reason is that the catch block is invoked prematurely. In Section 5.4, we show how our analysis reveals this security flaw, and how the flaw can be fixed.

Authorizations in Grids A BELLOG specification of the PDP for the grid scenario (see Figure 5.4) is as follows.

PDP Specification 3 (S3):

$$\begin{aligned} pol(X) &\leftarrow owner(X) \\ pol(X) &\leftarrow pol(Y) \wedge X : grant(Y) \\ X : grant(Y) &\leftarrow X : delegate(Y) \wedge \\ &\qquad ((\neg X : revoke(Y)@\mathsf{rev}) \blacktriangleright owner(X)) \end{aligned}$$

The PDP stores a credential *owner*(X) for each domain owner X. We represent a delegation from the subject X to the subject Y with the credential X : *delegate*(Y). The credential X : *revoke*(Y) represents that the subject X has revoked Y, and the remote query X : *revoke*(Y)@rev checks whether the revocation server stores such revocations.

The top two BELLOG rules encode the policy of Figure 5.4. The last BELLOG rule encodes the check for revoked credentials. Note that the for-loop is implicitly encoded, since this BELLOG rule is evaluated for all principals and subjects. The rule establishes that X grants Y if X delegates to Y and has not revoked this delegation. The failure handler is invoked for each delegation separately whenever the revocation check cannot be made. This follows the inner-loop logic of Figure 5.4.

To summarize, these examples demonstrate the use of BELLOG and its modeling capabilities. We believe that the failure-handling idioms considered in this paper, as well as other common authorization idioms, map naturally to BELLOG constructs. This makes BELLOG a suitable language for specifying PDPs. Of course, there are limitations to BELLOG's modeling power. Not all procedural constructs map naturally to BELLOG's declarative specifications, for example see the list iterator of the web app example. A further investigation of BELLOG's expressiveness is orthogonal to our results and outside the scope of this paper.

5.4 Verifying PDPs against Fail-Security Requirements

The goal of our analysis is to check a PDP's access decisions in the presence of failures. In the following, we first show how one can *simulate* a PDP using *entailment* questions in BELLOG. As an example, we use simulation to discover the previously described security flaw in XACML v3.0. Second, we show how given a BELLOG PDP specification P, and a fail-security requirement r, one can formulate the problem of checking whether P meets r as a *containment* problem in BELLOG. We use this to determine whether P conforms to r for all possible PDP inputs in our attacker model. As examples, we check whether the PDPs given in Section 5.1 meet their requirements, and we use the analysis framework to reveal flaws that violate the fail-security requirements **FR1** and **FR2**.

5.4.1 Simulating PDPs

Given a PDP input and a request, one can use the PDP's specification to *simulate* the PDP and check whether it grants or denies the request also

in the presence of failures. A PDP can be simulated by posing entailment questions to its BELLOG specification S as follows. First, the PDP input is encoded as a BELLOG input I, defined over some domain Σ, and the request is encoded as a BELLOG atom r, as described in Section 5.3.2. Second, to check whether the PDP grants or denies r, we pose the entailment question $S \models^{I}_{\Sigma} r$.

To illustrate, we simulate the XACML v3.0 PDP and describe how one can find the attack described in Section 5.1. The PDP's specification is XACML v3.0-spec, given in Section 5.3.2, and we consider the following scenario. There are two policies, one issued by Ann and one by Bob. Ann is the PDP's administrator. Let req be a request such that Ann's policy grants req, while Bob's policy denies req. Imagine that Bob's policy is authorized to give decisions for req. The PDP must therefore deny req because Ann and Bob's policies are composed using the deny-overrides operator. The following BELLOG input models this scenario.

$$\begin{array}{rlll} I = \{ & \mathit{admin}(\mathsf{ann}) & \mapsto & \mathsf{t} \\ & \mathit{pol}(\mathsf{ann}, \mathsf{req})@\mathsf{eval} & \mapsto & \mathsf{t} \\ & \mathit{pol}(\mathsf{bob}, \mathsf{req})@\mathsf{eval} & \mapsto & \mathsf{f} \\ & \mathit{auth}(\mathsf{bob}, \mathsf{req})@\mathsf{check} & \mapsto & \mathsf{t} \end{array}$$

Here the input I describes a *no-failure* scenario where the PDP successfully evaluates both policies and successfully checks that Bob's policy is authorized.

To simulate how the PDP behaves in the presence of failures, we may check the PDP's decision for the input

$$\begin{array}{rlll} I_{\mathsf{fail}} = \{ & \mathit{admin}(\mathsf{ann}) & \mapsto & \mathsf{t} \\ & \mathit{pol}(\mathsf{ann}, \mathsf{req})@\mathsf{eval} & \mapsto & \mathsf{t} \\ & \mathit{pol}(\mathsf{bob}, \mathsf{req})@\mathsf{eval} & \mapsto & \mathsf{f} \\ & \mathit{auth}(\mathsf{bob}, \mathsf{req})@\mathsf{check} & \mapsto & \bot \end{array}$$

The only difference here is that the PDP fails to check whether Bob's policy is authorized for req. We observe that for this scenario we have $S1 \vdash_{I_{\mathsf{fail}}} \mathit{pol\_set}(\mathsf{req})$, i.e. the PDP grants req because the PDP's failure handler drops Bob's policy decision. As the XACML committee discovered, this behavior is undesirable because an adversary may gain access by forcing the PDP to drop authorized policy decisions.

Note that our simulation method is similar to fault injection in software testing [86, 92]: The system's behavior is tested in various failure scenarios. The difference is that we do not directly execute the PDP's code and instead work with its specification.

5.4.2 Verifying Fail-security Requirements

To verify that a PDP specification *S meets a requirement r*, we formulate a number of containment problems. Each containment problem is defined using two BELLOG specifications, where one of them is the PDP specification *S* and the other one constrains the PDP's permissiveness, as prescribed by the requirement *r*. In the following, we formulate and verify whether the web app and grid PDPs from Section 5.1 meet their fail-security requirements. We also give an example of a generic fail-security requirement and show that it can be verified similarly.

Authorizations in Web Applications. Consider the PDP specification *S2* and the fail-security requirement **FR1**, which states that *when the PDP cannot compute an access decision due to malformed or missing specifications, then it uses the default specification if logging is enabled; otherwise, it denies access.*

To determine whether *S2* meets **FR1**, we first write a condition that is satisfied by the inputs for which the PDP cannot compute an access decision due to failures. Since the ACLs are composed with the permit-overrides operator, the PDP grants a request if any of the ACLs grant the request, and it denies it if all the ACLs deny it; otherwise, the PDP cannot compute a decision and it must, as prescribed by **FR1**, evaluate the default ACL and check the logging status. We encode the containment condition as

$$c_{\mathsf{error}} = \neg\Big(\big(\mathit{isGranted}(U,O)@\mathsf{acl}_1 = \mathsf{t} \ \vee \cdots \vee \mathit{isGranted}(U,O)@\mathsf{acl}_\mathsf{n} = \mathsf{t}\big) \\ \vee \big(\mathit{isGranted}(U,O)@\mathsf{acl}_1 = \mathsf{f} \wedge \cdots \wedge \mathit{isGranted}(U,O)@\mathsf{acl}_\mathsf{n} = \mathsf{f}\big)\Big) \ .$$

We then construct the BELLOG specification R_{error}:

$$R_{\mathsf{error}} = \{\mathit{pol}(U,O) \ \leftarrow \ (\mathit{isGranted}(U,O)@\mathsf{def} \wedge \mathit{logging}\} \ .$$

The specification R_{error} evaluates to grant if the PDP's default ACL evaluates to grant and logging is enabled; otherwise it evaluates to deny. Finally, to check whether the specification *S2* meets **FR1**, we formulate the containment problem

$$c_{\mathsf{error}} \ \Rightarrow \ S = R_{\mathsf{error}} \ .$$

Our analysis tool shows that the specification *S2* violates the requirement **FR1** for the PDP input

$$I = \{ \quad \begin{array}{lll} \mathit{isGranted}(\mathsf{ann},\mathsf{file})@\mathsf{acl}_1 & \mapsto & \bot, \\ \mathit{isGranted}(\mathsf{ann},\mathsf{file})@\mathsf{acl}_2 & \mapsto & \mathsf{t}, \\ \mathit{isGranted}(\mathsf{ann},\mathsf{file})@\mathsf{def} & \mapsto & \mathsf{f} \quad \} \ . \end{array}$$

```
1 isAuthorized(User u, Object o, List aclIDs)
2   error = false
3   for (id in aclIDs) // loop through all ACLs
4     try
5       if (readAcl(id).grants(u,o))
6         // grant if the current ACL grants
7         return true
8     catch (NotFoundException e)
9       // remember if the PDP failed to evaluate an ACL
10      error = true
11  if error
12    // check the default ACL and the logger's status
13    return def.grants(u,o) and logger.isEnabled()
14  return false
```

Figure 5.6: A PDP module that meets **FR1**.

S2 violates **FR1** because it denies the request $pol(\mathsf{ann}, \mathsf{file})$ even though ACL 2 grants this request.

To meet **FR1**, the PDP must correctly implement the propagate failure-handling idiom and apply the failure handler only if it fails to evaluate an ACL and all remaining ACLs deny access. We correct the PDP's specification as follows.

PDP Specification 4 (S4):

$$pol(U,O) \leftarrow \big(isGranted(U,O)@\mathsf{acl}_1 \vee \cdots \vee isGranted(U,O)@\mathsf{acl}_n\big) \\ \blacktriangleright \big(isGranted(U,O)@\mathsf{def} \wedge logging\big)$$

To automatically check whether *S4* meets **FR1**, we translate domain policy containment problems into propositional validity problems, which can be answered using off-the-shelf SAT solvers. Note that we cannot check whether *S4* meets **FR1** for any domain since containment of BELLOG programs is, in general, undecidable.

For a PDP with 10 ACLs, for all PDP inputs in a fixed domain of 10 constants. The verification takes 0.03 seconds. Naturally, the verification time increases with the number of ACLs and the domain size. For example, the verification time for a PDP with 100 ACLs and inputs ranging over domains of size 10, 100, and 1000 is 0.13, 2.09, and 34.42 seconds, respectively.

We give the pseudo-code for the authorization method that implements *S4* in Figure 5.6. This method delays handling failures until all ACLs have been evaluated. The PDP correctly implements the propagate idiom,

i.e. it consults the ACL `def` only if no input ACL grants the request and the PDP has failed to evaluate at least one of them (recorded in the `error` variable).

Authorizations in Grids Consider the PDP specification *S3* and the fail-security requirement **FR2**, which states that *when the PDP cannot check whether a subject has at least one non-revoked delegation chain due to failures, the PDP grants access if the subject is a direct delegate of the owner; otherwise it denies access.*

To verify that the specification meets the requirement, we formulate two containment problems. The first problem checks whether the PDP correctly evaluates the requests made by direct delegates and the second one checks whether the PDP correctly evaluates the requests made by non-direct delegates. We formulate these containment problems as the BELLOG program

$$R_{\mathsf{chain}} = \{ \quad chain(X) \leftarrow owner(X) \\ chain(X) \leftarrow chain(Y) \wedge Y : delegate(X) \wedge \neg Y : revoke(X)@\mathsf{rev}\} \ .$$

Given a subject X, $chain(X)$ is: (1) t if the PDP checks that X has at least one non-revoked chain, (2) $\bot$ if the PDP fails to check whether X has at least one non-revoked chain, and (3) f if X has no chains or the PDP checks that X has only revoked chains. We use the containment condition

$$c_{\mathsf{direct}} = (\exists Y.\ owner(Y) = \mathsf{t} \wedge Y : delegate(X) = \mathsf{t} \ \wedge Y : revoke(X)@\mathsf{rev} \neq \mathsf{t}) \ ,$$

which is satisfied by a PDP input iff the subject X who makes the request is a direct delegate and the owner has either not revoked the delegation or the PDP cannot check if the delegation is revoked.

We formulate the first containment problem as

$$c_{\mathsf{direct}} \ \Rightarrow \ S = R_{\mathsf{direct}} \ , \text{where} \\ R_{\mathsf{direct}} = R_{\mathsf{chain}} \cup \{pol(X) \leftarrow chain(X) \blacktriangleright \mathsf{t}\} \ .$$

The condition c_{direct} restricts PDP inputs to direct delegates and R_{direct} specifies which direct delegates S must grant and deny access to. Since the PDP must grant access to a direct delegate X iff the PDP either checks, or fails to check, that X has at least one non-revoked chain, R_{direct} conflates $\bot$ and t into the grant decision using the $(\_ \blacktriangleright \mathsf{t})$ pattern.

We formulate the second problem as

$$(\neg c_{\mathsf{direct}}) \ \Rightarrow \ S = R_{\mathsf{non-direct}} \ , \text{where} \\ R_{\mathsf{non-direct}} = R_{\mathsf{chain}} \cup \{pol(X) \leftarrow chain(X) \blacktriangleright \mathsf{f}\} \ .$$

The condition $\neg c_{\mathsf{direct}}$ restricts PDP inputs to non-direct delegates and revoked direct delegates, and $R_{\mathsf{non-direct}}$ specifies which ones S must grant and deny access to. Since the PDP must deny access to a non-direct delegate X iff the PDP fails to check that X has at least one non-revoked chain or X has only revoked chains, $R_{\mathsf{non-direct}}$ conflates $\bot$ and f into the deny decision using the (_ ► f) pattern.

Our analysis tool shows that the PDP specification *S3* does not meet **FR2** because the problem

$$(\neg c_{\mathsf{direct}}) \Rightarrow S = R_{\mathsf{non-direct}}$$

is answered negatively. The tool outputs the following PDP input:

$$\begin{array}{lllll} I = \{ & \mathsf{admin}:\mathit{owner}(\mathsf{piet}) & \mapsto & \mathsf{t}, & \\ & \mathsf{piet}:\mathit{delegate}(\mathsf{ann}) & \mapsto & \mathsf{t}, & \\ & \mathsf{piet}:\mathit{revoke}(\mathsf{ann})@\mathsf{rev} & \mapsto & \bot, & \\ & \mathsf{ann}:\mathit{delegate}(\mathsf{fred}) & \mapsto & \mathsf{t}, & \\ & \mathsf{ann}:\mathit{revoke}(\mathsf{fred})@\mathsf{rev} & \mapsto & \mathsf{f} & \} . \end{array}$$

In this scenario, Piet is the owner, and he delegates access to Ann, who further delegates access to Fred. Furthermore, the PDP fails to check whether Piet's delegation to Ann is revoked, and it succeeds in checking that Ann has not revoked Fred; see Figure 5.3. The PDP must deny access to Fred because he does not have a non-revoked delegation chain and he is not a direct delegate. The PDP, however, grants access to Fred, thus violating **FR2**. This flaw stems from the preemptive masking of failures. The adversary Fred can exploit this flaw and force an unintended grant decision by preventing the PDP from checking whether the owner's delegation to Ann is revoked. To confirm the attack, we simulated the attack scenario using our BELLOG interpreter.

To meet **FR2**, we modify the specification as follows.

PDP Specification 5 (S5):

$$\begin{aligned} &\mathit{pol}(X) \leftarrow \mathit{grant}(X) \blacktriangleright \big(\mathit{owner}(Y) \wedge Y:\mathit{delegate}(X) \wedge (\neg Y:\mathit{revoke}(X)@\mathsf{rev})\big) \\ &\mathit{grant}(X) \leftarrow \mathit{owner}(X) \\ &\mathit{grant}(X) \leftarrow \mathit{grant}(Y) \wedge Y:\mathit{delegate}(X) \wedge (\neg Y:\mathit{revoke}(X)@\mathsf{rev}) \end{aligned}$$

In the original specification *S3*, errors are not propagated through delegation chains. In contrast, the specification *S5* propagates errors through

delegation chains and thus denies access to subjects who are not direct delegates and do not have a non-revoked chain. The pseudo-code that reflects *S5* would have to, in effect, distinguish between permissions solely due to direct delegation versus permissions due to non-revoked chains.

Our analysis tool shows that the specification *S5* meets **FR2** for all PDP inputs in a fixed policy domain with eight constants; the verification takes 149.38 seconds. Our tool did not terminate in a reasonable time for larger domains.

We remark that domain containment gives weaker security guarantees than (general) policy containment because the guarantees are only for the given policy domain. Hence, domain policy containment does not account for possible attacks in other domains. For example, domain policy containment misses the attack described in our grid example if the policy domain has only two constants (e.g., two subjects). This is because the adversary must assume the role of a subject who is delegated access by a direct delegate, and such a subject does not exist in a domain with fewer than three constants.

Other Requirements In addition to the aforementioned requirements, one can verify whether a PDP meets certain generic security requirements. For example, one may want to ensure that a PDP handles all failures, i.e. it always evaluates requests to either grant or deny decisions. We refer to this requirement as *error-freeness*, and show how it can be checked by formulating suitable containment problems.

Let S be the PDP specification and $pol(X)$ be the atom used to denote the PDP's access decisions. We construct a specification R as follows. Let $R = \emptyset$. We rename the predicate symbol *pol* to *tmp* in S's rules and add the changed rules to R. Finally, we add the rule

$$pol(X) \leftarrow tmp(X) \blacktriangleright \mathsf{f}\,,$$

to R. We formulate the containment problem as $S = R$. By construction, R denies all requests that S evaluates to $\bot$. Therefore, if S evaluates a request to $\bot$, then R is not equal to S; otherwise, S is error-free. Note that one can similarly verify that any atom other than $pol(\cdot)$ in the PDP's specification is error-free.

To conclude, these examples show that our simulation and verification methods can reveal security flaws in PDPs that handle failures incorrectly. Our preliminary experiments show that our simulation tool scales well to realistic problems. The runtimes for our analysis tool, however, are mixed.

In our grid example the analysis tool does not terminate in a reasonable amount of time for a domain with nine constants, whereas in the web app example the tool terminates in less than a minute for domains with thousands of constants.

5.5 Related Work

Although fail-security requirements have been discussed in the security literature [24, 91], there has been no rigorous, systematic treatment of fail-secure access control. The existing access control specification languages, such as [21, 26, 38, 46, 53, 65], do not explicitly deal with failure handlers in their analysis. Although failures are considered in [35], failure-handling mechanisms are not dealt with.

Static and dynamic policy analysis frameworks such as [20, 28, 40, 43, 49, 63] can potentially be tailored to reason about PDPs with failure handling, similarly to BELLOG. In particular, PBel's analysis framework [28] also supports policies with many-valued policy decisions and can, if delegations are excluded, express our failure-handling idioms. We remark that dynamic analysis frameworks, such as [20, 40, 49], consider history-based access decisions, which fall outside the scope of this thesis.

Chapter 6
Access Control Synthesis for Physical Spaces

In this chapter, we present our access-control synthesis algorithm for synthesizing local policies from global requirements. The focus in this chapter is on physical access-control systems, which are used to restrict access to physical spaces. For example, they control who can access which parts of an office building or how personnel can move within critical spaces such as airports or military facilities. As physical spaces are usually comprised of subspaces, such as rooms connected by doors, policies are enforced by multiple policy enforcement points (PEPs). Each PEP is associated with a control point, like a door, and enforces a *local* policy.

Consider, for example, an office building. An electronic door lock might control access to an office by enforcing a policy that states that only an *employee* may enter the office. This policy is local in the sense that its scope is limited to an individual enforcement point, here the office's door. The policy therefore does not guarantee that non-employees cannot enter the office, since the office may have other doors. Neither does it guarantee that employees can actually access the office. If employees cannot enter the corridor leading to the office's door, then the local policy is useless.

In contrast to the local policies for enforcement points, access-control requirements for physical spaces are typically *global*. They express constraints on the access paths through the entire space. In the example above, a requirement might be that employees should be able to access the office from the lobby. This requirement is global in that no single PEP alone can guarantee its satisfaction. A standard electronic lock, which enforces only local policies such as *grant access to employees*, is oblivious to the physical constraints of the office building and what policies the other PEPs enforce. It therefore cannot address this requirement.

Problem Statement The discrepancy between global requirements and local policies creates an abstraction gap that must be bridged when configuring access-control mechanisms. We consider the problem of automatically synthesizing a set of PEP policies that together enforce global access-control requirements in a given physical space.

This problem is nontrivial. A given physical space usually constrains the ways subjects may access its subspaces. These constraints must be

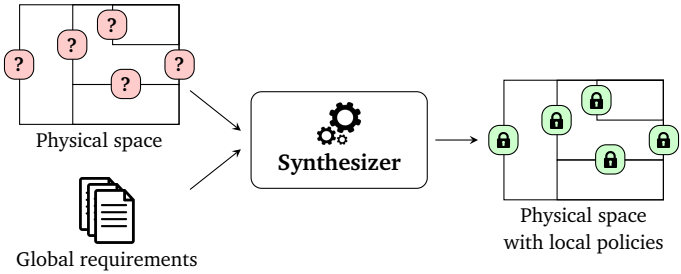


Figure 6.1: Access control synthesis for physical spaces

accounted for when configuring the individual PEPs. Moreover, global access-control requirements may have interdependencies and hence their individual solutions may not contribute to an overall solution. To illustrate this lack of compositionality, suppose in addition to the requirement that employees can access an office room from the lobby, we require that they must not enter the area where auditing documents are stored. Giving employees access to their office through *any* path satisfies the first requirement, but it would violate the second one if the path goes through the audit area.

In practice, constructing local policies for a physical space is a manual task where a security engineer writes individual policies, one per PEP, that collectively enforce the space's global requirements. This manual process results in errors, such as granting access to unauthorized subjects or denying access to authorized ones; the literature contains numerous examples of such problems [19, 45, 47]. Moreover, engineers must manually revise their policies whenever requirements are changed, or when the physical space changes, e.g. due to construction work. In short, writing local policies manually is error-prone and scales poorly. Our thesis is that it is also unnecessary: the automatic synthesis of local policies with system-wide security guarantees is a viable alternative.

Approach and Contributions We propose an algorithm for automatically synthesizing local policies that run on distributed PEPs from a set of global access-control requirements for a given physical space. The main ingredients of our synthesis approach are depicted in Figure 6.1. The key component is a *synthesizer*, which is an algorithm that takes as input a model of the physical space and a set of global requirements. The synthesizer's output is the set of local policies that the PEPs enforce. If the global requirements are satisfiable, then the synthesizer is guaranteed to output

a correct set of local policies; otherwise, it returns unsat to indicate that the requirements cannot be satisfied. Hence, using our synthesis algorithm, engineers can generate local policies from global requirements simply by formalizing the global requirements and modeling the physical space.

Below, we briefly describe the components of our synthesis approach, depicted in Figure 6.1. We use directed graphs to model physical spaces: a node represents an enclosed space, such as an office or a corridor, and an edge represents a PEP, for example installed on a door or turnstile. The nodes are labeled to denote their attributes. These attributes may include the assets the node contains (audit documents), its physical attributes (international terminal), and its clearance level (high security zone). These attributes may be used when specifying policies. Formally, our model of a physical space is a Kripke structure.

We give a declarative language, called SPCTL, for specifying global requirements. Our language is built on the computation tree logic (CTL) [42] and supports subject attributes (e.g., an organizational role), time constraints (e.g., business-hour requirements), as well as quantification over paths and branches in physical spaces. To demonstrate its expressiveness, we show how common physical access-control requirements can be directly written in SPCTL. Moreover, to simplify the task of formalizing such requirements, we develop requirement patterns and illustrate their use through examples.

Our synthesis algorithm outputs attribute-based policies, expressed as constraints over subject attributes and contextual conditions, such as organizational roles and the current time. We restrict our attention to policies without authority delegation and policy composition because synthesizing arbitrary policies specified in BELLOG, which is our policy specification language used in Chapters 4 and 5, is prohibitively expensive. Nonetheless, the attribute-based policies supported by our algorithm cover a wide range of practical setups and scenarios, including attribute-based and role-based access control. We strike a balance between the requirement language's expressiveness and the complexity of synthesizing local policies. The synthesis problem we consider is NP-hard. However, we show that for practically-relevant requirements, it can be efficiently solved using existing SMT solvers. This is intuitively because physical spaces, in practice, induce directed graphs that have short simple-paths. We illustrate our algorithm's effectiveness using three case studies where we synthesize access-control policies for a university building, a corporate building, and an airport terminal. Synthesizing local policies in each case takes less than 30 seconds. The

last two case studies are based on real-world examples developed together with KABA, a leading physical access control company.

To the best of our knowledge, ours is the first framework for synthesizing policies from system-wide access-control requirements. We thereby solve a fundamental problem in access control for physical spaces. An immediate practical consequence is that security engineers can focus on system-wide requirements, and delegate to our synthesizer the task of constructing the local policies with correctness guarantees. We remark that although this work is focused on access control for physical spaces, the ideas presented are general and can be extended to other domains, such as computer networks partitioned into subnetworks by distributed firewalls.

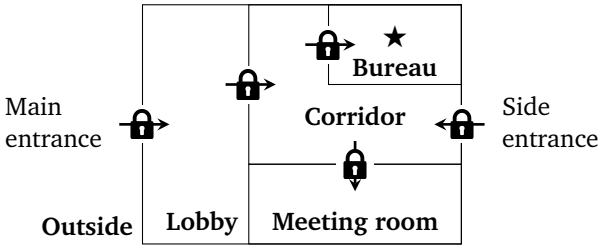
Organization We give an overview of our access-control synthesis approach in Section 6.1. In Section 6.2, we describe and formalize our system model. In Section 6.3, we define our SPCTL language for specifying global requirements, and present requirement patterns. In Section 6.4, we define the policy synthesis problem and prove its decidability. In Section 6.5, we define an efficient policy synthesis algorithm. In Section 6.6, we describe our implementation and report on our experiments. We review related work in Section 6.7. All proofs are given in Section 6.8.

6.1 Overview

We start with a simple example that illustrates the challenges of constructing local policies that cumulatively enforce global access-control requirements. We also explain how our synthesis algorithm is used, that is, we describe its inputs and outputs.

6.1.1 Running Example

Consider a small office space consisting of a lobby, a bureau, a meeting room, and a corridor. The office layout is given in Figure 6.2a. Access within this physical space is secured using electronic locks. Each door has a lock and a card reader. The lock stores a policy that defines who can open the door from the card reader's side. The door can be opened by anyone from the opposite side. We annotate locks with arrows in Figure 6.2a to indicate the direction that the locks restrict access. For example, the lock at the main entrance restricts who can access the lobby, and it allows anyone to exit the office space from the lobby. To open a door from the card reader's side, a subject presents a smartcard that stores the holder's credentials. The



(a) Floor plan (★ marks security zones)

R1: Visitors can access the meeting room between 8AM and 8PM.

R2: Visitors cannot access the meeting room if they have not passed through the lobby.

R3: Employees can access the bureau between 8AM and 8PM.

R4: Employees can access the bureau at any time if they enter their correct PIN.

R5: Non-employees cannot access security zones.

(b) Global requirements

Figure 6.2: Floor plan and global requirements of our running example.

lock can access additional information, such as the current time, needed to evaluate the policy. The lock opens whenever the policy evaluates to grant.

The global requirements for this physical space are given in Figure 6.2b. The requirements **R1**, **R3**, and **R4** define permissions, while **R2** and **R5** define prohibitions. To meet these requirements, the electronic locks must be configured with appropriate local policies. As previously observed, this is challenging because one must account for both spatial constraints and all global access-control requirements. We illustrate these points below.

Spatial Constraints The layout of the physical space prevents subjects from freely requesting access to any resource. For example, the requirement **R1** is not met just because the meeting room's lock grants access to visitors; the visitor must also be able to enter the corridor from the outside. Such constraints must be accounted for when defining the local policies. To satisfy **R1**, we may for instance choose a path from the main entrance to the meeting room and configure all the locks along that path to grant access to visitors.

Global Requirements Each global requirement typically has multiple sets of local policies that satisfy it. The local policies must however be constructed to ensure that *all* requirements are satisfied *simultaneously*. For example, the requirement **R1** is satisfied if the side-entrance lock and the meeting room lock both grant access to visitors between 8AM and 8PM. It can also be satisfied by ensuring that the main entrance, the lobby, and the meeting room locks all grant access to visitors between 8AM and 8PM. Granting visitors access through the side entrance however violates the requirement **R2**, which requires that visitors pass through the lobby. Hence, to meet both requirements, the locks along the path through the lobby must grant access to visitors between 8AM and 8PM, while the side-entrance lock must always deny access to visitors.

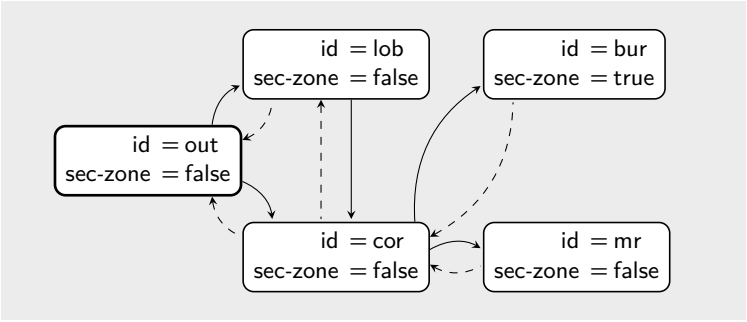
6.1.2 Synthesis Algorithm

Figure 6.3 depicts our synthesis algorithm's input and output for our running example. The input is a model of the physical space and a specification of its global requirements. The output produced by our synthesizer is a set of local policies.

A physical space is modeled as a rooted directed graph called a *resource structure*; see Figure 6.3a. We have depicted the root node in gray. In our example, this corresponds to the public space that surrounds the office space, e.g. public streets. The remaining (non-root) nodes are the spaces inside the building. The locks control access along the edges. A subject can traverse a solid edge of the resource structure only if the lock's policy evaluates to grant, whereas any subject can follow the dashed edges. Hence, the locks effectively enforce the grant-all policy along the dashed edges. We use two attributes to label the physical spaces: the attribute id represents room identifiers, and sec-zone formalizes that a space is inside the security zone.

Global requirements are specified using a declarative language, called SPCTL. In Figure 6.3b we show the formalization of our running example's requirements in SPCTL. For instance, **R1**, which states that visitors can access the meeting room between 8AM and 8PM, is formalized as $\big((\mathsf{role} = \mathsf{visitor}) \wedge (8 \leq \mathsf{time} \leq 20)\big) \Rightarrow \textsc{Grant}(\mathsf{id} = \mathsf{mr})$. This formalization instantiates SPCTL's permission pattern GRANT to state that there is a path from outside to the meeting room such that every lock on the path grants access to any visitor between 8AM and 8PM. We define SPCTL's syntax and semantics and present several patterns in Section 6.3.

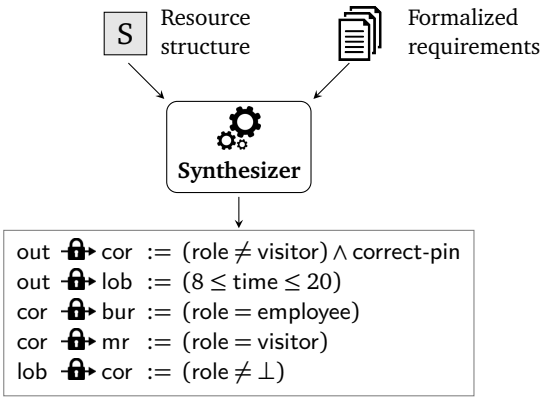
Given these inputs, the synthesizer automatically constructs a local policy for each lock. The synthesized policies are attribute-based policies



(a) Resource structure S

$$
\begin{array}{ll}
((\text{role} = \text{visitor}) \wedge (8 \leq \text{time} \leq 20)) & \Rightarrow \textsc{Grant}(\text{id} = \text{mr}) \\
(\text{role} = \text{visitor}) & \Rightarrow \textsc{Waypoint}(\text{id} = \text{lob}, \text{id} = \text{mr}) \\
((\text{role} = \text{employee}) \wedge (8 \leq \text{time} \leq 20)) & \Rightarrow \textsc{Grant}(\text{id} = \text{bur}) \\
((\text{role} = \text{employee}) \wedge \text{correct-pin}) & \Rightarrow \textsc{Grant}(\text{id} = \text{bur}) \\
(\text{role} \neq \text{employee}) & \Rightarrow \textsc{Deny}(\text{sec-zone} = \text{true})
\end{array}
$$

(b) Formalized requirements



(c) The synthesizer takes as input the resource structure and the formalized requirements and outputs local policies.

Figure 6.3: Synthesizing the local policies for our running example

that collectively enforce the global requirements. The synthesized policies for our running example are given in Figure 6.3c. We write, for example, cor 🔒 bur for the synthesized policy deployed at the bureau's lock. This policy (role = employee) grants access to subjects with the role employee. We define the synthesis problem, and the syntax and semantics of attribute-based local policies in Section 6.4.

6.2 Physical Access Control

We first describe our model for physical spaces, and then formalize it.

6.2.1 Basic notions

Each physical space is partitioned into finitely many enclosed spaces and one open (public) space. We call the enclosed spaces *resources*. Two spaces may be directly connected with a *gate*, controlled by a PEP. Examples of gates include doors, turnstiles, and security checkpoints. Each PEP has its own PDP, which stores a *local policy* mapping access requests to access decisions.

To enter a space, a subject provides his credentials to the PEP that controls the gate. The PEP forwards the subject's credentials to its PDP. The PDP, in turn, queries the PIP if needed, evaluates the policy, and then forwards the access decision — either grant or deny — to the PEP. The PEP then enforces the PDP's decision. See Figure 2.1.

We assume that access requests contain all relevant information for making access decisions. PDPs can thus make their decisions independent of past access requests. Hence it has no bearing on our model whether the PDPs are actually distributed or are realized through a centralized system. This is desirable from a practical standpoint since PDPs and PIPs need not be equipped with logging mechanisms. Moreover, different PDPs and PIPs need not synchronize their local views on the request history. In this sense they are autonomous entities.

As described in Chapter 2, the access requests and local policies we consider are attribute based and may reference three kinds of attributes: subject, contextual, and resource attributes. For example, the resource attributes *floor* and *department* may represent the floor of an office space and the department it belongs to. We use such resource attributes to specify global requirements. They are however not needed for expressing access requests or local policies in our model. This is because the PDPs associated to any resource can be hardwired with all the attributes of that resource. In this sense, each PDP "knows" the space under its control.

Our system model targets electronic PEP/PDPs that can enforce attribute-based policies and read digital credentials, e.g. stored on smart cards and mobile phones. Manufacturers often refer to these as smart locks [12, 50, 66]. In large physical access-control systems, smart locks are rapidly replacing mechanical locks and keys, which can only enforce simple, crude policies.

6.2.2 Formalization

We now formalize the above notions.

Attributes Fix a finite set $\mathcal{A}$ of *attributes* and a set $\mathcal{V}$ of attribute *values*. The *domain* function $\mathsf{dom}\colon \mathcal{A} \to \mathcal{P}(\mathcal{V})$ associates each attribute with the set of values it admits. For instance, the current time attribute is associated with the set of natural numbers, and the clearance level attribute is associated with a fixed finite set of levels. We assume that any attribute can take the designated value $\perp$, representing the situation where the attribute's value is unknown. We partition the set of attributes into subject attributes $\mathcal{A}_S$, contextual attributes $\mathcal{A}_C$, and resource attributes $\mathcal{A}_R$.

Access Requests We represent an access request as a total function that maps subject and contextual attributes to values from their respective domains. This function is computed by PDPs after receiving a subject's credentials and querying PIPs. For instance, the PDP maps the attribute role to visitor when the subject's credentials indicate this. It maps the attribute correct-pin to true when the PIN entered through the keypad attached to the PDP is correct. Finally, it maps the contextual attribute time to 8 after querying a time server at 8AM. We denote the set of all access requests by $\mathcal{Q}$.

A remark on set-valued attributes is due here. In some settings, attributes take a finite set of values, as opposed to a single value. For example, in role-based access control, a subject may activate multiple roles. The attribute role must then be assigned with the set of all the activated roles. We account for such set-valued attributes simply by defining a Boolean attribute for each value; for example, we define role_employee and role_manager. An access request q assigns true to both Boolean attributes whenever a subject has activated both the employee and the manager roles.

Local Policies Local policies map access requests to grant or deny. We extensionally define local policies as subsets of $\mathcal{Q}$: a local policy is defined

as the set of requests that it grants. The structure $(\mathcal{P}(\mathcal{Q}), \subseteq, \cap, \cup, \emptyset, \mathcal{Q})$ is a complete lattice that orders local policies by their permissiveness. The least permissive policy, namely $\emptyset$, denies all access requests, and the most permissive one, i.e. $\mathcal{Q}$, grants them all. In section 6.4.1, we will intensionally define local policies as constraints over subject and contextual attributes. The local policies shown in Figure 6.3, for example, are defined by such constraints.

Resource Structures We now give a formal model of physical spaces. A *resource structure* is a tuple $S = (\mathcal{R}, E, r_e, L)$, where $\mathcal{R}$ is a set of *resources*, $E \subseteq \mathcal{R} \times \mathcal{R}$ is an irreflexive edge relation, $r_e \in \mathcal{R}$ is the *entry resource*, and $L : \mathcal{R} \to (\mathcal{A}_R \to \mathcal{V})$ is a total function mapping resources to resource attribute valuations. We assume that every resource $r \in \mathcal{R}$ is reachable from the entry resource r_e, that is, $(r_e, r) \in E^*$, where E^* is the reflexive-transitive closure of E.

The edges in a resource structure model PEPs. The irreflexivity of E captures the condition that once a subject enters a physical space, he cannot re-enter the space before first leaving it. We assume that resource structures do not contain *deadlocks*. A resource r_0 in S is a deadlock if there does not exists an r_1 such that $(r_0, r_1) \in E$. This assumption is valid in physical-space access control: a deadlock resource corresponds to a "black hole" that no one can leave. Note that dead-end corridors are not deadlocks, provided one can backtrack.

The entry resource r_e represents the public space and the remaining resources denote enclosed spaces. A resource structure describes how subjects can access resources. A subject accesses a resource along a path, which is a sequence of resources connected by edges, starting from the entry resource. For example, before entering a room in a hotel, a subject enters the hotel's lobby from the street, and then goes through the corridor. Figure 6.3(c) gives an example of a resource structure.

Configurations Each edge of a resource structure represents a gate controlled by a local policy installed on the gate's PDP. We therefore define a *configuration* for a resource structure S as a function that assigns to each edge of S a local policy. We write C_S for the set of all configurations for S. The set C_S is partially ordered under the relation $\sqsubseteq_S$, defined as: $c \sqsubseteq_S c'$ if for any edge e of S we have $c(e) \subseteq c'(e)$. Namely, a configuration is less permissive than another configuration if, for any edge, the former assigns a less permissive local policy than the latter.

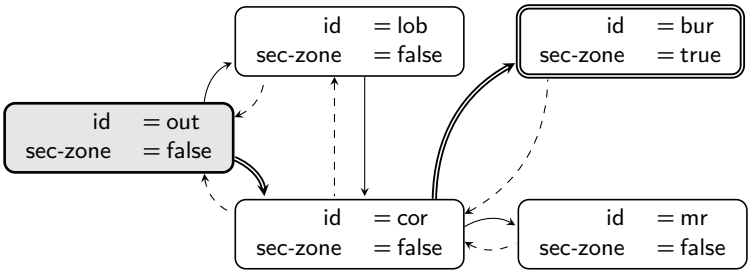


Figure 6.4: The double-lined edges denote the PEPs that deny the access request $q = \{\mathsf{role} \mapsto \mathsf{visitor}, \mathsf{time} \mapsto 10, \mathsf{correct\text{-}pin} \mapsto \bot\}$, given the configuration c from Figure 6.3. The resource structure $S_{c,q}$ is obtained by removing the double-lined edges and nodes.

We can now define which resources are accessible given an access request and a configuration. For a resource structure S, a configuration c for S, and an access request q, we define $S_{c,q}$ as the resource structure obtained by removing all the edges from S whose policies deny q, and then removing all nodes that are not reachable from the entry resource. The structure $S_{c,q}$'s entry resource is the same as S's. To illustrate, consider the resource structure S and the configuration c given in Figure 6.3, and the access request $q = \{\mathsf{role} \mapsto \mathsf{visitor}, \mathsf{time} \mapsto 10, \mathsf{correct\text{-}pin} \mapsto \bot\}$. The side-entrance PEP and the bureau PEP deny q and therefore these two edges are removed from S. The node that represents the bureau is not reachable from the entry resource and it is thus also removed. In Figure 6.4 we depict the removed edges and nodes.

We remark that the structure $S_{c,q}$ is defined for a fixed access request q. Access requests, which assign values to subject and contextual attributes, can however change, for instance when a subject's role is revoked or as time progresses. We abstract away such changes in $S_{c,q}$'s definition. In our running example, this amounts to assuming that a subject's role does not change during this time, and the time needed to move through the office building is negligible compared to the time needed for a subject's access rights to change; for example, the requirements **R1-5** stipulate that subject's access rights may change only twice per day — at 8AM and at 8PM. This abstraction corresponds to taking a snapshot of all the attributes, and then computing $S_{c,q}$ based on the snapshot. We refer to these snapshots as *sessions*. Henceforth we interpret global requirements and local policies in the context of such sessions.

Interpreting requirements and polices in the context of a session is justified for many practical scenarios. This is because, in most practical settings, changes in subject and contextual attributes are addressed through out-of-band mechanisms. To illustrate, consider a subject who has the role visitor and enters the meeting room of our running example at 3PM as permitted by the system's requirements. Now, suppose that the subject's visitor role is revoked at 4PM, or that the subject remains in the meeting room until 10PM. No access-control system can force the subject to leave. In practice, out-of-band mechanisms, such as security guards, address such concerns.

In the following sections, we confine our attention to configurations that do not introduce deadlocks. That is, we consider those configurations c where for any $q \in \mathcal{Q}$, the structure $S_{c,q}$ is deadlock-free. In Section 6.3.3, we describe how this provision can be encoded as a global requirement.

6.3 Specifying Requirements

In this section we define SPCTL, a simple declarative language for specifying requirements. We give the language's syntax and semantics in Section 6.3.1. To simplify the specification of global requirements, in Section 6.3.2 we present four requirement patterns that capture common access-control idioms for physical spaces. Finally, in Section 6.3.3, we illustrate the specification of two generic access-control requirements: deny-by-default and deadlock-freeness.

6.3.1 Requirement Specification Language

The design of SPCTL has been guided by real-world physical access-control requirements. Virtually all such requirements can be formalized as properties that specify which physical spaces subjects can and cannot access, directly and over paths, based on the security context and on the physical spaces they have accessed. In our physical access-control model, subjects choose which physical spaces to access, which induces a branching structure over the spaces they access. We therefore build our requirement specification language SPCTL upon the computation tree logic (CTL) [41], whose branching semantics is a natural fit for physical spaces.

$$
\begin{array}{rcl}
a = c & := & a \in \{c\} \\
a \neq c & := & \neg(a = c) \\
a_{\mathsf{bool}} & := & a_{\mathsf{bool}} = \mathsf{true} \\
a_{\mathsf{num}} \leq n & := & a_{\mathsf{num}} \in \{0, \ldots, n\} \\
a_{\mathsf{num}} \geq n & := & \neg(a_{\mathsf{num}} \leq n-1) \\
n \leq a_{\mathsf{num}} \leq n' & := & (a_{\mathsf{num}} \geq n) \wedge (a_{\mathsf{num}} \leq n')
\end{array}
$$

Figure 6.5: Syntactic shorthands: $a \in \mathcal{A}$ is an attribute, $a_{\mathsf{num}} \in \mathcal{A}_{\mathsf{num}}$ is a numeric attribute, $a_{\mathsf{bool}} \in \mathcal{A}_{\mathsf{bool}}$ is a boolean attribute, $n, n' \in \mathbb{N}$ are natural numbers.

Syntax A requirement specified in SPCTL is a formula of the form $T \Rightarrow \varphi$ given by the following BNF:

$$
\begin{array}{rcl}
T & ::= & \mathsf{true} \mid a_s \in D \mid a_c \in D \mid \neg T \mid T \wedge T \\
\varphi & ::= & \mathsf{true} \mid a_r \in D \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{EX}\varphi \mid \mathsf{AX}\varphi \\
& \mid & \mathsf{E}[\varphi \mathsf{U} \varphi] \mid \mathsf{A}[\varphi \mathsf{U} \varphi] \,.
\end{array}
$$

Here $a_s \in \mathcal{A}_S$ is a subject attribute, $a_c \in \mathcal{A}_C$ is a contextual attribute, $a_r \in \mathcal{A}_R$ is a resource attribute, and $D \subseteq \mathcal{V}$ is a finite subset of values. The formula T is a constraint over subject and contextual attributes that defines the access requests to which the requirement applies. We call T the *target*. The formula φ is a CTL formula over resource attributes. It defines a path property that must hold for all access requests to which the requirement is applicable. We call φ an *access constraint*.

Note that additional Boolean and CTL operators can be defined in the standard way. For example, we write false for $\neg\mathsf{true}$, and define the Boolean connectives $\vee$ and $\Rightarrow$ in the standard manner using $\neg$ and $\wedge$. We will later make use of the CTL operators $\mathsf{EF}\varphi$, $\mathsf{AG}\varphi$, and $\mathsf{A}[\varphi \mathsf{R} \psi]$, which are defined as $\mathsf{E}[\mathsf{true}\ \mathsf{U}\ \varphi]$, $\neg(\mathsf{EF}\neg\varphi)$, and $\neg(\mathsf{E}[\neg\varphi \mathsf{U} \neg\psi])$, respectively. Below we give intuitive explanations of EX, AX, EU, and AU, which are standard CTL connectives.

The connectives *exists-next* EX and *always-next* AX constrain the physical spaces that a subject can access next. In our running example, suppose that a subject has entered the lobby. The subject can next enter the corridor or go to the public space: these are immediately accessible from the lobby. In the lobby, $\mathsf{EX}\varphi$ states that the formula φ is true in at least one of these "next" spaces. In contrast, $\mathsf{AX}\varphi$ states that φ is true both in the corridor and in the public space.

$$
\begin{array}{llll}
S, r_0 & \models \mathsf{true} & & \\
S, r_0 & \models a \in D & \text{if} & L(r_0)(a) \in D \\
S, r_0 & \models \neg\varphi & \text{if} & S, r_0 \not\models \varphi \\
S, r_0 & \models \varphi_1 \wedge \varphi_2 & \text{if} & S, r_0 \models \varphi_1 \text{ and } S, r_0 \models \varphi_2 \\
S, r_0 & \models \mathsf{EX}\varphi & \text{if} & \exists (r_0, r_1, \cdots) \in S(r_0).\ S, r_1 \models \varphi \\
S, r_0 & \models \mathsf{AX}\varphi & \text{if} & \forall (r_0, r_1, \cdots) \in S(r_0).\ S, r_1 \models \varphi \\
S, r_0 & \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2] & \text{if} & \exists (r_0, r_1, \cdots) \in S(r_0).\ \exists i \geq 0. \\
 & & & \qquad S, r_i \models \varphi_2 \wedge \forall j \in [0, i).\ S, r_j \models \varphi_1 \\
S, r_0 & \models \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2] & \text{if} & \forall (r_0, r_1, \cdots) \in S(r_0).\ \exists i \geq 0. \\
 & & & \qquad S, r_i \models \varphi_2 \wedge \forall j \in [0, i).\ S, r_j \models \varphi_1
\end{array}
$$

Figure 6.6: The relation $\models$ between a resource structure $S = (\mathcal{R}, E, r_e, L)$, a resource $r_0 \in \mathcal{R}$, and an access constraints φ.

The operators *exists-until* EU and *always-until* AU relate two access constraints φ_1 and φ_2 over paths. The formula $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ states that there exists a path that reaches a resource r that satisfies φ_2, and any resource prior to r on the path satisfies φ_1. We use this connective to formalize, for example, waypointing requirements such as: visitors cannot access the meeting room until they have accessed the lobby. The formula $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ states that every path reaches some resource r that satisfies φ_2, and that any resource prior to r on the path satisfies φ_1.

To simplify writing attribute constraints in SpCTL, we introduce in Figure 6.5 abbreviations for numeric and boolean attributes. Based on the attributes' domains, we partition the set of attributes $\mathcal{A}$ into *numeric attributes* A_{num}, *boolean attributes* A_{bool}, and *enumerated attributes* A_{enum}: An attribute a is *numeric* if $\mathsf{dom}(a) = \mathbb{N} \cup \{\bot\}$; it is *boolean* if $\mathsf{dom}(a) = \{\mathsf{false}, \mathsf{true}, \bot\}$; otherwise, it is enumerated and $\mathsf{dom}(a)$ is finite. We may write a_{num} or a_{bool} to emphasize that an attribute a is numeric or boolean, respectively.

Semantics We first inductively define the satisfaction relation $\vdash$ between an access request $q \in \mathcal{Q}$ and a target:

$$
\begin{array}{llll}
q & \vdash & \mathsf{true} & \\
q & \vdash & a \in D & \text{if} \quad q(a) \in D \\
q & \vdash & \neg T & \text{if} \quad q \not\vdash T \\
q & \vdash & T_1 \wedge T_2 & \text{if} \quad q \vdash T_1 \text{ and } q \vdash T_2\ .
\end{array}
$$

A requirement $T \Rightarrow \varphi$ is *applicable* to an access request q iff q satisfies the target T, i.e. $q \vdash T$. For example, the requirement $(\text{role} = \text{visitor}) \Rightarrow \varphi$ is applicable to all access requests that assign the value visitor to the subject attribute role.

Let $S = (\mathcal{R}, E, r_e, L)$ be a resource structure. A *path* of S is an infinite sequence of resources $(r_0, r_1, \cdots)$ such that $\forall i \geq 0.\ (r_i, r_{i+1}) \in E$, and we denote the set of all paths rooted at a resource r_0 by $S(r_0)$. In Figure 6.6, we inductively define the satisfaction relation $\models$ between a resource structure, a resource, and an access constraint. A resource structure S with an entry resource r_e *satisfies* an access constraint φ, denoted by $S \models \varphi$, iff $S, r_e \models \varphi$.

Definition 5. *Let S be a resource structure, c a configuration for S, and $T \Rightarrow \varphi$ a requirement. S configured with c* satisfies *$T \Rightarrow \varphi$, denoted by $S, c \Vdash (T \Rightarrow \varphi)$, iff $q \vdash T$ implies $S_{c,q} \models \varphi$, for any access request $q \in \mathcal{Q}$.*

We extend $\Vdash$ to sets of requirements as expected. Given a set of requirements $R = \{T_1 \Rightarrow \varphi_1, \ldots, T_n \Rightarrow \varphi_n\}$, a resource structure S configured with c satisfies R, denoted by $S, c \Vdash R$, iff $S, c \Vdash (T_i \Rightarrow \varphi_i)$ for all i, $1 \leq i \leq n$.

We remark that resource structures can easily be represented using standard Kripke structures [41] by mapping each resource to a Kripke state and each resource attribute valuation to sets of atomic propositions. The access constraints can be similarly mapped to standard CTL formulas by translating attribute constraints into propositional logic. Note however that while Kripke structures are often used to represent changes of, say, a concurrent system's state over time, resource structures model static physical spaces.

6.3.2 Requirement Patterns

SPCTL can be directly used to specify global requirements. However, to illustrate its use and expressiveness, we present the formalization of common physical access-control idioms.

We have studied the requirements of an airport, a corporate building, and a university campus to elicit the common structure of physical access-control requirements. To distill the basic requirement patterns, we split complex requirements into their atomic parts. Our analysis revealed four common patterns, which we formalize below. The first pattern abstracts *positive* requirements, which stipulate that the access-control system must grant certain access requests. The remaining three patterns capture *negative* requirements, which stipulate that the access-control system must deny certain access requests.

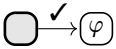
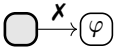
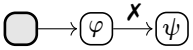
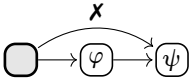
Pattern	Shorthand	Specification	Intuitive Semantics
Permission	$T \Rightarrow \textsc{Grant}(\varphi)$	$T \Rightarrow \mathsf{EF}\ \varphi$	
Prohibition	$T \Rightarrow \textsc{Deny}(\varphi)$	$T \Rightarrow \mathsf{AG}(\neg\varphi)$	
Blocking	$T \Rightarrow \textsc{Block}(\varphi, \psi)$	$T \Rightarrow \mathsf{AG}(\varphi \Rightarrow \mathsf{AG}(\neg\psi))$	
Waypointing	$T \Rightarrow \textsc{Waypoint}(\varphi, \psi)$	$T \Rightarrow \mathsf{A}[\varphi \mathsf{R} \psi]$	

Figure 6.7: BELLOG Patterns: The entry resource in the intuitive semantics is depicted using a gray rectangle. The arrows $\varphi \xrightarrow{\checkmark} \psi$ $\left(\varphi \xrightarrow{\times} \psi\right)$ indicate that there must (must not) exist a path from a φ-space to a ψ-space along which T-requests are granted.

We use the following terminology when describing requirements. Given a target T, we call an access request q a *T-request* if $q \vdash T$, i.e. q satisfies the target T. Given a resource structure S and an access constraint φ, we say that a subject can access a *φ-space* of S if the subject can access a physical space r_0 of S such that $S, r_0 \models \varphi$, i.e. the space r_0 satisfies the access constraint φ. Our patterns are summarized in Figure 6.7.

Permission The *permission* pattern abstracts requirements stating that T-requests can access φ-spaces from the entry resource. Permission requirements have the form $T \Rightarrow (\mathsf{EF}\ \varphi)$. The exists-future operator EF formalizes that a φ-space is reachable from the entry resource. For example, the requirement **R3** stipulating that employees can access the bureau between 8AM and 8PM is formalized as

$$\big((\mathsf{role} = \mathsf{employee}) \wedge (8 \leq \mathsf{time} \leq 20)\big) \Rightarrow \mathsf{EF}\ (\mathsf{id} = \mathsf{bur}).$$

The target $(\mathsf{role} = \mathsf{employee}) \wedge (8 \leq \mathsf{time} \leq 20)$ formalizes that this requirement is applicable only to access requests made by visitors at times between 8AM and 8PM. The access constraint $\mathsf{EF}(\mathsf{id} = \mathsf{bur})$ is satisfied iff the resource structure has a path from the entry resource to the bureau. The requirements **R1** and **R4** of our running example are also instances of the permission pattern.

Prohibition Dual to the permission pattern, the *prohibition* pattern captures requirements stating that T-access requests cannot access a φ-space.

Prohibition requirements have the form $T \Rightarrow \mathsf{AG}(\neg\varphi)$. The operator AG quantifies over all paths reachable from the entry resource. An example taken from our airport requirements is: Passengers cannot access the departure gate zones without a boarding pass. Another example is requirement **R5**, formalized as

$$(\mathsf{role} \neq \mathsf{employee}) \Rightarrow \mathsf{AG}(\neg\, \mathsf{sec\text{-}zone}) \,.$$

The target $\mathsf{role} \neq \mathsf{employee}$ is satisfied by access requests that assign a value other than employee to the attribute role. The access constraint $\mathsf{AG}(\neg\, \mathsf{sec\text{-}zone})$ is satisfied if no path leads to a security zone.

Blocking The *blocking* pattern captures requirements stating that subjects cannot access a ψ-space after they have accessed a φ-space. Intuitively, accessing a φ-space *blocks* the subject from accessing ψ-spaces. At international airports, for example, passengers may not access departure gate zones after they have accessed the baggage claim. Blocking requirements have the form $T \Rightarrow \mathsf{AG}(\varphi \Rightarrow \mathsf{AG}(\neg\psi))$. The airport example is formalized as:

$$\begin{aligned}(\mathsf{role} = \mathsf{passenger}) \Rightarrow \mathsf{AG}\big((\mathsf{zone} = \mathsf{baggage\text{-}claim}) \\ \Rightarrow \mathsf{AG}\ \neg(\mathsf{zone} = \mathsf{departure})\big) \,.\end{aligned}$$

This requirement instantiates the blocking pattern: the target T is $(\mathsf{role} = \mathsf{passenger})$, and the two access constraints ψ and φ are $(\mathsf{zone} = \mathsf{departure})$ and $(\mathsf{zone} = \mathsf{baggage\text{-}claim})$.

Waypointing The *waypointing* pattern captures requirements stipulating that subjects must first access a φ-space before accessing a ψ-space. For example, passengers cannot access an airport's terminal before they have passed through a security check. This is a negative requirement that restricts how passengers can access the terminal. Waypointing requirements have the form $T \Rightarrow (\mathsf{A}[\varphi \mathsf{R} \psi])$. The globally-release operator AR quantifies over all paths from the entry resource and formalizes that if ψ holds at some point, then φ was valid at least once beforehand. The requirement **R2** of our running example is an instance of the waypointing pattern and is formalized as

$$(\mathsf{role} = \mathsf{visitor}) \Rightarrow \mathsf{A}[(\mathsf{id} = \mathsf{lob})\mathsf{R}(\mathsf{id} = \mathsf{mr})] \,.$$

The target specifies that this requirement applies to all access requests made by visitors. The access constraint is satisfied if all paths to the meeting room go through the lobby.

The four idioms just described cover *all* the requirements that arose in the case studies that we report on in Section 6.6.2. We remark though that there are global requirements that are not instances of these four patterns. For example, in corporate buildings, a subject must be able to access the parking lot if he or she has access to an office. Although this requirement cannot be expressed using the above patterns, it can be directly formalized in SPCTL as follows:

$$\mathsf{true} \Rightarrow \big((\mathsf{EF}(\mathsf{zone} = \mathsf{office})) \Rightarrow (\mathsf{EF}(\mathsf{id} = \mathsf{parking\text{-}lot}))\big) .$$

In general, as SPCTL supports all CTL operators, it can specify any branching property expressible in CTL.

6.3.3 Generic Requirements

We now describe two commonly-used generic requirements.

Deny-by-default The deny-by-default principle stipulates that if an access request can be denied without violating the requirements, then it should be denied; cf. [79]. Security engineers often follow this principle to avoid overly permissive local policies. To illustrate, consider our running example and imagine that the role intern is contained in the domain of the attribute role. The requirements given in Figure 6.3(b) do not prohibit an intern from accessing, say, the meeting room. However, denying interns access to the meeting room also complies with these requirements.

The following requirement, called *deny-by-default*, instantiates the above principle: If no positive requirement is applicable to an access request, then only the entry space is accessible to the subject who makes such a request. To formalize this requirement, we first define positive and negative requirements. Let c and c' be two configurations for a given resource structure S. A requirement $T \Rightarrow \varphi$ is *positive* if $S, c \Vdash (T \Rightarrow \varphi)$ and $c \sqsubseteq_S c'$ imply $S, c' \Vdash (T \Rightarrow \varphi)$. A requirement $T \Rightarrow \varphi$ is *negative* if $S, c \Vdash (T \Rightarrow \varphi)$ and $c' \sqsubseteq_S c$ imply $S, c' \Vdash (T \Rightarrow \varphi)$. Intuitively, if a configuration satisfies a positive (negative) requirement, then any more (less) permissive configuration also satisfies the requirement. We remark that although not all requirements are positive or negative, most real-world requirements are, including all requirements specified in this paper.

Let R be a set of requirements that contains only positive and negative requirements, and let $\{T_1 \Rightarrow \varphi, \cdots, T_n \Rightarrow \varphi_n\}$ be the set of all positive requirements contained in R. The deny-by-default requirement for R is

$$(\neg T_1) \wedge \cdots \wedge (\neg T_n) \Rightarrow \mathsf{AX}\ (\mathsf{id} = \mathsf{entry})\ .$$

Here we assume that $L(r_e)(\mathsf{id}) = \mathsf{entry}$, i.e. the entry resource r_e is labeled with entry. Adding this requirement to our running example's requirements would ensure that an intern cannot access, for example, the meeting room. Note that if R contains no positive requirements, then the default-by-requirement is $\mathsf{true} \Rightarrow \mathsf{AX}\ (\mathsf{id} = \mathsf{entry})$, which formalizes that all subjects can only access the entry space.

Deadlock-freeness A *deadlock-freeness* requirement stipulates that there are no deadlocks in a system, i.e. resources that a subject can access and then never leave. For example, the meeting room of our running example would be a deadlock if visitors could enter it, but never leave. As discussed in our system model, local policies that introduce deadlocks are undesirable.

Formally, the deadlock-freeness requirement is defined as:

$$\mathsf{true} \Rightarrow \mathsf{AG}\ \mathsf{EX}\ \mathsf{true}\ .$$

This requirement applies to all access requests. The access constraint $\mathsf{AG}\ \mathsf{EX}\ \mathsf{true}$ states that for any resource a subject can access, there is a resource that the subject can access next. A resource structure S and a configuration c satisfy this requirement iff for any access requests $q \in \mathcal{Q}$, $S_{c,q}$ has no deadlocks.

6.4 Policy Synthesis Problem

We now define the policy synthesis problem. We show that this problem is decidable but NP-hard.

6.4.1 Problem

Definition 6. *The* policy synthesis problem *is as follows:*

Input.	*A resource structure S and a set of requirements R.*
Output.	*A configuration c such that $S, c \Vdash R$, if such a configuration exists, and* unsat *otherwise.*

The synthesized configuration defines the local policies to be deployed at the PEPs. Recall that a policy is extensionally defined as the set of access

requests for which the PEP grants access. As such a set may, in general, be infinite, one cannot simply output an extensional definition of the synthesized configuration. We therefore define local policies intensionally by constraints over subject and contextual attributes, expressed in the same language that we specify requirement targets in Section 6.3. We remark that policies and targets are often formalized with the same language; cf. [36, 87]. The semantics of an intensional local policy P is then simply the function that maps an access request q to grant if $q \vdash P$ and to deny otherwise. Figure 6.3 illustrates the input and output to the policy synthesis problem for our running example.

An example of a local policy defined over the attributes role and time is $(\mathsf{role} = \mathsf{visitor}) \wedge (8 \leq \mathsf{time} \leq 20)$. This local policy grants all access requests that assign the value visitor to the attribute role and a number between 8 and 20 to the attribute time. Note that this local policy is also the target of requirement **R1**.

6.4.2 Decidability

To show that the policy synthesis problem is decidable, we give a synthesis algorithm, called $\mathcal{S}_{\mathsf{cs}}$, that uses controller synthesis as a subroutine. In the following, we first define the controller synthesis problem. We then show how the algorithm $\mathcal{S}_{\mathsf{cs}}$ constructs the PEPs' local policies by solving multiple controller synthesis instances.

Controller Synthesis Problem Controller synthesis algorithms take as input a description of an uncontrolled system, called a plant, along with a specification, and output a controller that restricts the plant so that it satisfies the given specification. In our setting, the plant is the resource structure and the specification is an access constraint, i.e. a CTL formula over resource attributes. The synthesized controller then defines which PEPs must grant or deny the access request so that the access constraint is satisfied. For simplicity, we do not define the controller synthesis problem in its most general form. For our needs the following simpler definition suffices.

Definition 7. *The* controller synthesis problem *is as follows:*

Input.	*A resource structure* $S = (\mathcal{R}, E, r_e, L)$ *and an access constraint* φ.
Output.	*A set* $E' \subseteq E$ *of edges such that* $(\mathcal{R}, E', r_e, L) \models \varphi$*, if such an* E' *exists, and* unsat *otherwise.*

The controller synthesis problem can be reduced to synthesizing a memoryless controller for a Kripke structure given a CTL specification. Deciding whether a controller synthesis instance has a solution is NP-complete [8]. Systems such as MBP [23] can be used to synthesize controllers. For a comprehensive overview of controller synthesis see [76].

Algorithm The algorithm $\mathcal{S}_{\mathsf{cs}}$ is based on two insights. First, for a given access request q, we can use controller synthesis to identify which PEPs must grant or deny q. In more detail, we can compute $(\mathcal{R}, E', r_e, L) \models \varphi_q$, where φ_q conjoins all access constraints of the requirements that are applicable to q. The edges in E' represent the PEPs that must grant q and those in $E \setminus E'$ the PEPs that must deny q. A configuration can thus be synthesized by solving one controller synthesis instance for each access request. However, there are infinitely many access requests. Our second insight is that we can construct a configuration by solving finitely many controller synthesis instances. We partition the set $\mathcal{Q}$ of access requests into $2^{|R|}$ equivalence classes, where two access requests are equivalent if the same set of requirements are applicable to them. Solving one controller synthesis instance for one representative access request per equivalence class is sufficient for our purpose.

The main steps of the algorithm $\mathcal{S}_{\mathsf{cs}}$ are given in Algorithm 1. The algorithm iteratively constructs a configuration c as follows. Initially, it sets all local policies to true (lines 2-3). The algorithm iterates over all subsets $R' = \{T_1 \Rightarrow \varphi_1, \ldots, T_i \Rightarrow \varphi_i\}$ of the requirements R (line 4). The conjunction $T = T_1 \wedge \cdots \wedge T_i \wedge \neg T_{i+1} \wedge \cdots \neg T_n$ constructed at line 5 is satisfied by all access requests to which only the requirements contained in R' are applicable. The set $\{q \in \mathcal{Q} \mid q \vdash T\}$ is an equivalence class of access requests. If this equivalence class is nonempty, i.e. $\exists q \in \mathcal{Q}.\ q \vdash T$, then c must grant and deny all access requests contained in it in conformance with the access constraints defined by R'. Lines 9-14 define how the algorithm $\mathcal{S}_{\mathsf{cs}}$ updates c. First, it constructs the conjunction φ of the access constraints defined by the requirement in R'. It then executes the controller synthesis algorithm, denoted by cs, with the inputs S and φ. If the algorithm cs returns unsat, then the requirements are not satisfiable for the given resource structure, and the algorithm $\mathcal{S}_{\mathsf{cs}}$ thus returns unsat. Otherwise, the algorithm cs returns a set $E' \subseteq E$ of edges. The algorithm updates the configuration c as follows: for any edge in $E \setminus E'$, the configuration is modified to deny access to all requests in the equivalence class defined by R'. The algorithm terminates when all subsets of the global requirements have been considered.

Algorithm 1: The algorithm $\mathcal{S}_{\mathsf{cs}}$ for synthesizing policies using controller synthesis. The controller synthesis algorithm, denoted $\mathsf{cs}(S, \varphi)$, outputs either a subset of E or unsat.

Input: Resource stricture $S = (\mathcal{R}, E, r_e, L)$, a set of requirements R
Output: A configuration c or unsat

```
1  begin
2      for e ∈ E do
3          c(e) ← true
4      for R' ⊆ R do
5          T ← T_1 ∧ ··· ∧ T_i ∧ ¬T_{i+1} ∧ ··· ∧ ¬T_n, where
6              {T_1 ⇒ φ_1, ..., T_i ⇒ φ_i} = R' and
7              {T_{i+1} ⇒ φ_{i+1}, ..., T_n ⇒ φ_n} = R \ R'
8          if ∃q ∈ Q. q ⊢ T then
9              φ ← φ_1 ∧ ··· ∧ φ_i
10             if cs(S, φ) = unsat then
11                 return unsat
12             else
13                 for e ∈ E \ cs(S, φ) do
14                     c(e) ← c(e) ∧ (¬T)
15     return c
```

Theorem 15. *Let S be a resource structure and R a set of requirements. If $\mathcal{S}_{\mathsf{cs}}(S, R) = c$ then $S, c \Vdash R$. If $\mathcal{S}_{\mathsf{cs}}(S, R) = \mathsf{unsat}$ then there is no configuration c such that $S, c \Vdash R$.*

We prove this theorem and give the complexity of $\mathcal{S}_{\mathsf{cs}}$ in Section 6.8.

Example To illustrate $\mathcal{S}_{\mathsf{cs}}$, consider our running example and the requirements **R2** and **R5** formalized as follows:

$$\textbf{R2} := (\mathsf{role} = \mathsf{visitor}) \Rightarrow (\mathsf{A}[(\mathsf{id} = \mathsf{lob})\ \mathsf{R}\ (\mathsf{id} = \mathsf{mr})])$$
$$\textbf{R5} := (\mathsf{role} \neq \mathsf{employee}) \Rightarrow (\mathsf{AG}\ \neg \mathsf{sec\text{-}zone})\ .$$

Recall that $\mathsf{dom}(\mathsf{role}) = \{\bot, \mathsf{visitor}, \mathsf{employee}\}$, and hence the targets $\mathsf{role} = \mathsf{visitor}$ and $\mathsf{role} \neq \mathsf{employee}$ are not equivalent.

To synthesize a configuration, the algorithm $\mathcal{S}_{\mathsf{cs}}$ executes the second for-loop four times. Let the selected subset of requirements in the first

iteration be $\{\mathbf{R2}, \mathbf{R5}\}$. The conjunction T of the targets is $(\mathsf{role} = \mathsf{visitor}) \wedge (\mathsf{role} \neq \mathsf{employee})$, which is equivalent to $(\mathsf{role} = \mathsf{visitor})$. Hence, T is satisfiable. The access constraint φ (see Algorithm 1, line 9) is then $(\mathsf{A}[(\mathsf{id} = \mathsf{lob})\ \mathsf{R}\ (\mathsf{id} = \mathsf{mr})]) \wedge (\mathsf{AG}\ \neg \mathsf{sec\text{-}zone})$. A possible output by the controller synthesis algorithm $\mathsf{cs}(S, \varphi)$ is $E \setminus \{(\mathsf{cor}, \mathsf{bur}), (\mathsf{out}, \mathsf{cor})\}$. The updated configuration c after the first iteration is therefore

$$c(e) = \begin{cases} \mathsf{true} \wedge \mathsf{role} \neq \mathsf{visitor} & \text{if } e = (\mathsf{cor}, \mathsf{bur}) \\ \mathsf{true} \ \wedge \mathsf{role} \neq \mathsf{visitor} & \text{if } e = (\mathsf{out}, \mathsf{cor}) \\ \mathsf{true} & \text{otherwise}\,. \end{cases}$$

Suppose the outputs to the remaining three controller synthesis instances are $\mathsf{cs}(S, \varphi_{\{\mathbf{R2}\}}) = E \setminus \{(\mathsf{out}, \mathsf{cor})\}$, $\mathsf{cs}(S, \varphi_{\{\mathbf{R5}\}}) = E \setminus \{(\mathsf{cor}, \mathsf{bur})\}$, and $\mathsf{cs}(S, \varphi_{\emptyset}) = E$, where φ_X denotes the conjunction of the access constraints of the requirements in X. The simplified configuration c returned by $\mathcal{S}_{\mathsf{cs}}$ is

$$c(e) = \begin{cases} \mathsf{role} = \mathsf{employee} & \text{if } e = (\mathsf{cor}, \mathsf{bur}) \\ \mathsf{role} \neq \mathsf{visitor} & \text{if } e = (\mathsf{out}, \mathsf{cor}) \\ \mathsf{true} & \text{otherwise}\,. \end{cases}$$

Limitations The main limitation of the algorithm $\mathcal{S}_{\mathsf{cs}}$ is that the running time is exponential in the number of requirements, rendering it impractical for nontrivial instances of policy synthesis. For example, while the algorithm $\mathcal{S}_{\mathsf{cs}}$ takes 2 seconds to synthesize a configuration for our running example, it does not terminate within an hour for our case studies, reported in Section 6.6.2. We give a practical policy synthesis algorithm based on SMT solving in Section 6.5.

6.4.3 NP-hardness

To show NP-hardness, we reduce propositional satisfiability to the policy synthesis problem. It is easy to see that a propositional formula φ can be encoded, in logarithmic space, as a target T_φ over Boolean attributes. Consider the policy synthesis problem for the inputs S and $\{(T_\varphi \Rightarrow \mathsf{false})\}$, where S is an arbitrary resource structure. If the output to this policy synthesis instance is unsat then for some access request q, we have $q \vdash T_\varphi$. Hence φ is satisfiable. Alternatively, the output to the policy synthesis problem is a configuration c. Since for any access request q where $q \vdash T_\varphi$ we have $S_{c,q} \models \mathsf{false}$, it is immediate that there is no access request q such that $q \vdash T_\varphi$. Therefore, φ is unsatisfiable.

6.5 Policy Synthesis Algorithm

In this section, we define our policy synthesis algorithm based on SMT solving, called $\mathcal{S}_{\mathsf{smt}}$. The algorithm takes as input a resource structure S, a set R of requirements, and a set C of configurations. The set C is encoded symbolically, as we describe shortly. The algorithm outputs a configuration c such that $S, c \Vdash R$, if there is such a configuration in C; otherwise, it returns unsat. To synthesize a configuration c, the algorithm encodes the question $\exists c \in C.\ S, c \Vdash R$ in a decidable logic supported by standard SMT solvers. Due to its technical nature, we relegate a detailed description of the encoding to the end of this section.

Our algorithm takes as input a set of configurations, and we refer to the symbolic encoding of this set as a *configuration template*. The configuration template enables us to restrict the search space: the algorithm confines its search to the configurations described by the template. Our algorithm $\mathcal{S}_{\mathsf{smt}}$ is sound, independent of the provided configuration template. Its completeness, however, depends on the template. We show that one can construct a template for which $\mathcal{S}_{\mathsf{smt}}$ is complete, but the resulting template would, in practice, encode so many configurations that the resulting SMT problem would be infeasible to solve. We therefore strike a balance between the algorithm's completeness and its efficiency: since real-world local policies often have small syntactic representations, as demonstrated by our experiments in Section 6.6, our policy synthesis tool starts with a template that defines configurations with succinct local policies, and iteratively executes $\mathcal{S}_{\mathsf{smt}}$, increasing the template's size in each iteration. It turns out that in our case studies a small number of iterations is sufficient to synthesize all local policies. Below, we describe the algorithm $\mathcal{S}_{\mathsf{smt}}$'s components.

6.5.1 Configuration Templates

A *configuration template* assigns to each edge of the resource structure a symbolic encoding of a set of local policies. To illustrate this encoding, consider the set of local policies $\{\mathsf{true}, \mathsf{role} = \mathsf{employee}, \mathsf{role} \neq \mathsf{visitor}\}$. We symbolically encode this set for an edge, say $(\mathsf{cor}, \mathsf{bur})$, as a constraint over subject and contextual attributes, as well as a *control* variable $z_{(\mathsf{cor},\mathsf{bur})}$:

$$\begin{aligned} C((\mathsf{cor},\mathsf{bur})) = {} & (z_{(\mathsf{cor},\mathsf{bur})} = 1 \Rightarrow \mathsf{true}) \wedge \\ & (z_{(\mathsf{cor},\mathsf{bur})} = 2 \Rightarrow \mathsf{role} = \mathsf{employee}) \wedge \\ & (z_{(\mathsf{cor},\mathsf{bur})} = 3 \Rightarrow \mathsf{role} \neq \mathsf{visitor})\ . \end{aligned} \tag{T1}$$

The control variable $z_{(\mathsf{cor},\mathsf{bur})}$ encodes the choice of one of three local policies for the edge $(\mathsf{cor}, \mathsf{bur})$. Hence, for this example, the set of configurations

defined by the configuration template contains $3^{|E|}$ elements, where E is the set of edges in the resource structure. Note that for a set of local policies of size n (here $n = 3$), $\lceil \log n \rceil$ propositional variables are sufficient for representing each edge's control variables. To avoid clutter, we will write C_{r_0,r_1} for $C((r_0, r_1))$.

We remark that configuration templates can be used to restrict the search space of configurations to those that satisfy *attribute availability* constraints, which restrict the set of attributes that PEPs can retrieve. Suppose that only the side-entrance door of our running example is equipped with a keypad. To account for this constraint, we will restrict the configurations in the template to those that use the correct-pin attribute only in the local policy of side entrance's lock.

6.5.2 Algorithm

The main steps of $\mathcal{S}_{\mathsf{smt}}$ are given in Algorithm 2. We describe the algorithm with an example: the input to the algorithm consists of the resource structure and the requirements **R2** and **R5** of our running example, along with the above configuration template C, which maps edges to the set of local policies $\{\mathsf{true}, \mathsf{role} = \mathsf{employee}, \mathsf{role} \neq \mathsf{visitor}\}$. The algorithm starts by creating for each requirement a constraint that asserts the satisfaction of the requirement in the resource structure, given the template. This constraint is called ψ in the algorithm, and is expressed in the logic of an SMT solver. This step is implemented by the subroutine ENCODE, defined in Figure 6.8. To encode the satisfaction of access constraints, we follow the standard model-checking algorithm for CTL based on labeling [57]; we explain this encoding at the end of this section.

As an example, the result of ENCODE$(S, \mathbf{R2}, C)$, after straightforward simplifications, is the following constraint:

$$\psi_{R2} := \mathsf{role} = \mathsf{visitor} \Rightarrow (\neg C_{\mathsf{out,cor}} \vee \neg C_{\mathsf{cor,mr}}) .$$

Here role is an *attribute variable*, originating from **R2**'s target, and $C_{\mathsf{out,cor}}$ and $C_{\mathsf{cor,mr}}$ are the symbolic encodings of the local policies for the edges $(\mathsf{out}, \mathsf{cor})$ and $(\mathsf{cor}, \mathsf{mr})$, respectively. This constraint states that if the requirement's target $\mathsf{role} = \mathsf{visitor}$ is satisfied, then one of the PEPs along the path that starts at the entry resource and reaches the meeting room directly through the corridor must deny access. Similarly, ENCODE$(S, \mathbf{R5}, C)$ returns

Algorithm 2: The algorithm $\mathcal{S}_{\mathsf{smt}}$ for synthesizing policies using SMT solving.

Input: A resource structure $S = (\mathcal{R}, E, r, L)$, a set $\{R_1, \cdots, R_n\}$ of requirements, a configuration template C
Output: A configuration c or unsat

```
1  begin
2      φ ← true
3      for R ∈ {R_1, ··· , R_n} do
4          ψ ← ENCODE(S, R, C)
5          φ ← φ ∧ ψ
6      if (∃z⃗.∀a⃗. φ) is sat then
7          M ← MODEL(∃z⃗.∀a⃗. φ)
8          for e ∈ E do
9              c(e) ← DERIVE(C(e), M)
10         return c
11     else
12         return unsat
```

the constraint:

$$\begin{aligned}\psi_{R5} := \mathsf{role} \neq \mathsf{employee} \Rightarrow \\ ((\neg C_{\mathsf{out,cor}} \vee \neg C_{\mathsf{cor,bur}}) \\ \wedge (\neg C_{\mathsf{out,lob}} \vee \neg C_{\mathsf{lob,cor}} \vee \neg C_{\mathsf{cor,bur}}))\,.\end{aligned}$$

This states that any access request that maps the attribute role to a value other than employee must be denied by at least one PEP along the path to the bureau that goes directly through the corridor, and moreover it must be denied by at least one PEP along the path that passes through the lobby.

The conjunction of the constraints created for all the requirements is called ϕ in Algorithm 2. To check whether there is a configuration in C that satisfies the requirements, the algorithm calls an SMT solver to find a model for the formula $\exists\vec{z}.\forall\vec{a}.\ \phi$. Here this is

$$\exists\vec{z}.\forall\vec{a}.\ (\psi_{R2} \wedge \psi_{R5})\,,$$

where $\vec{z}$ and $\vec{a}$ consist, respectively, of all the control and attribute variables. If ϕ is unsatisfiable, then no configuration in C satisfies the requirements. In this case, the algorithm returns unsat. If however the formula is satisfiable,

ENCODE$(S, T \Rightarrow \varphi, C)$ **returns** $T \Rightarrow \tau(\varphi, r_e)$

Rewrite rules $\tau(\varphi, r_0)$:

$$\tau(\mathsf{true}, r_0) \hookrightarrow \mathsf{true}$$

$$\tau(a \in D, r_0) \hookrightarrow \begin{cases} \mathsf{true} & \text{if } L(r_0)(a) \in D \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\tau(\neg\varphi, r_0) \hookrightarrow \neg\tau(\varphi, r_0)$$

$$\tau(\varphi_1 \wedge \varphi_2, r_0) \hookrightarrow \tau(\varphi_1, r_0) \wedge \tau(\varphi_2, r_0)$$

$$\tau(\mathsf{EX}\varphi, r_0) \hookrightarrow \exists r_1 \in E(r_0).\ \left(C_{r_0,r_1} \wedge \tau(\varphi, r_1)\right)$$

$$\tau(\mathsf{AX}\varphi, r_0) \hookrightarrow \forall r_1 \in E(r_0).\ \left(C_{r_0,r_1} \Rightarrow \tau(\varphi, r_1)\right)$$

$$\tau(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0) \hookrightarrow \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \emptyset)$$

$$\tau(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0) \hookrightarrow \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, \emptyset)$$

Rewrite rules $\tau_{\mathsf{U}}(\varphi, r_0, X)$, with $X \subseteq \mathcal{R}$:

$$\tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, X) \hookrightarrow \tau(\varphi_2, r_0) \vee \Big(\tau(\varphi_1, r_0) \wedge \big(\exists r_1 \in E(r_0) \setminus X.\ C_{r_0,r_1} \wedge \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_1, X \cup \{r_0\})\big)\Big)$$

$$\tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, X) \hookrightarrow \tau(\varphi_2, r_0) \vee \Big(\tau(\varphi_1, r_0) \wedge \big(\forall r_1 \in E(r_0) \setminus X.\ C_{r_0,r_1} \Rightarrow \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_1, X \cup \{r_0\})\big) \wedge \big(\forall r_1 \in E(r_0) \cap X.\ \neg C_{r_0,r_1}\big)\Big)$$

Figure 6.8: Encoding the satisfaction of a requirement $T \Rightarrow \varphi$ in a resource structure $S = (\mathcal{R}, E, r_e, L)$, given a template C, into an SMT constraint. The rewrite rules τ reduce an access constraint φ and a resource r_0 to an SMT constraint. For a resource $r_0 \in \mathcal{R}$, we write $E(r_0)$ for $\{r_1 \in \mathcal{R} \mid (r_0, r_1) \in E\}$. The $\exists$ and $\forall$ quantifiers range over a finite domain. Therefore, the former can be expanded as a finite number of disjunctions, and the latter as a finite number of conjunctions.

then the SMT solver returns a model of the formula, which instantiates all the control variables (but not the attribute variables since they are universally quantified). We refer to the SMT solver's procedure that returns such a model as MODEL in Algorithm 2. The model $\mathcal{M}$ generated by the SMT solver in effect identifies the local policy for each edge e: by instantiating the control variables in $C(e)$, we obtain e's local policy; see template T1. This procedure is called DERIVE$(C(e), \mathcal{M})$ in the algorithm. For our example, a model $\mathcal{M}$ that satisfies $\exists\vec{z}.\forall\vec{a}.\ (\psi_{R2} \wedge \psi_{R5})$ maps $z_{(\mathsf{cor},\mathsf{bur})}$ to 2, $z_{(\mathsf{out},\mathsf{cor})}$ to 3, and all other control variables to 1. It is then evident from template T1 that, e.g., the local policy for the edge $(\mathsf{cor}, \mathsf{bur})$ is $(\mathsf{role} = \mathsf{employee})$.

Complexity Let S be a resource structure, R be a set of requirements, and C be configuration template. The running time of the $\mathcal{S}_{\mathsf{smt}}$ algorithm is determined by the size of the generated formula ϕ and the complexity of finding a model of ϕ. The size of the formula ϕ is in $\mathcal{O}(d \cdot |R| \cdot |\mathcal{R}|)$, where d is the size of the largest access constraint that appears in the requirements, R is the set of requirements, and $\mathcal{R}$ is the set of resources in S. The formula ϕ is defined over Boolean control variables $\vec{z}$ and attribute variables $\vec{a}$. The number of control and attribute variables is $\lceil log(|C|)\rceil$ and $|\mathcal{A}|$, respectively. In the worst case, one must check all possible models of the formula ϕ, so finding a model of ϕ is in $\mathcal{O}(2^{\lceil log(|C|)\rceil + k\cdot|\mathcal{A}|})$, where k is the largest domain that appears in the constraints. Note that such domains are always finite. For example, $\mathsf{time} \geq 10$ is a shorthand for $\neg(\mathsf{time} \in \{0,\ldots,9\})$. We conclude that the overall running time of the algorithm $\mathcal{S}_{\mathsf{smt}}$ is in $\mathcal{O}(2^{\lceil log(|C|)\rceil + k\cdot|\mathcal{A}|} + d \cdot |R| \cdot |\mathcal{R}|)$.

6.5.3 Soundness and Completeness

The algorithm $\mathcal{S}_{\mathsf{smt}}$ is sound.

Theorem 16. *Let S be resource structure, R a set of requirements, and C a configuration template. If $\mathcal{S}_{\mathsf{smt}}(S,R,C) = c$ then $S, c \Vdash R$. If $\mathcal{S}_{\mathsf{smt}}(S,R,C) =$ unsat, then there is no configuration c in C such that $S, c \Vdash R$.*

$\mathcal{S}_{\mathsf{smt}}$'s completeness depends on the template C provided as input to the algorithm. We show that one can construct a template for which the algorithm is complete. A template C is *complete* for a given resource structure S and set of requirements R if $\mathcal{S}_{\mathsf{smt}}(S,R,C)$ returns a configuration whenever there is a configuration that satisfies the requirements. For the algorithm's completeness, it is in fact sufficient to start the algorithm with a template $C_{S,R}$ that contains all the configurations that the algorithm

based on controller synthesis, described in Section 6.8.1, may output. The following theorem formalizes this observation.

Theorem 17. *Given a resource structure S and a set R of requirements, the configuration template $C_{S,R}$ is complete for S and R.*

The number of configurations in $C_{S,R}$ is exponential in $|E|$ and $|R|$ (which we prove in [1]). Hence this template, although complete, is not useful in practice as it would overwhelm SMT solvers, rendering $\mathcal{S}_{\mathsf{smt}}$ ineffective. In Section 6.6.1, where we explain our implementation in detail, we describe a configuration template that works well for synthesizing configurations for practically-relevant examples.

We conclude this discussion by pointing out that our synthesis algorithm can be readily used to verify whether a candidate configuration c satisfies a set R of global access-control requirements in a resource structure S. Namely, if the configuration template input to $\mathcal{S}_{\mathsf{smt}}$ consists only of the configuration c, then $\mathcal{S}_{\mathsf{smt}}$ returns c if $S, c \models R$; otherwise, the algorithm returns unsat, which means that the configuration c does not satisfy R.

6.5.4 Encoding into SMT

We now explain Algorithm 2's procedure ENCODE, which translates a resource structure S, a requirement $R = (T \Rightarrow \varphi)$, and a configuration template C, into an SMT constraint $T \Rightarrow \tau(\varphi, r_e)$. The generated constraint encodes that whenever the requirement $T \Rightarrow \varphi$ is applicable to an access request q, i.e. $q \vdash T$, then φ must be satisfied for the entry resource r_e in the structure $S_{c,q}$. Here, c is the configuration selected from the template C. The constraint $\tau(\varphi, r_e)$ is generated using the rewrite rules τ as defined in Figure 6.8.

Given an access constraint φ and a resource r_0, the rewrite rules τ produce an SMT constraint $\tau(\varphi, r_0)$ that encodes $S, r_0 \models \varphi$; see Figure 6.6. The rewrite rules for access constraints of the form true, $a \in D$, $\neg\varphi$, and $\varphi_1 \wedge \varphi_2$ are as expected. The rewrite rule for access constraints of the form $\mathsf{EX}\varphi$ encodes that the access constraint φ is satisfied at r_0 if there is an edge from r_0 to some node r_1 such that C_{r_0,r_1} holds and $S, r_1 \models \varphi$. In this rule, the constraint C_{r_0,r_1} returns the symbolic encoding of the local policies for the edge (r_0, r_1), and $\tau(\varphi, r_1)$ returns the encoding of $S, r_1 \models \varphi$ as an SMT constraint. In contrast to EX, the rewrite rule for $\mathsf{AX}\varphi$ access constraints states that for any resource r_1, such that $(r_0, r_1) \in E$, if C_{r_0,r_1} is true then the constraint $\tau(\varphi, r_1)$ is satisfied.

To encode the semantics of the connectives EU (AU), we use the *until* rewrite rules τ_{U}, which reduce an until construct $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ ($\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$),

a resource $r_0 \in \mathcal{R}$, and a set of resources $X \subseteq \mathcal{R}$ to an SMT constraint. We use the set of resources X to record for which resources the satisfaction of the until access constraint has already been encoded. This is necessary to guarantee the reduction system's termination. The rule for access constraints of the form $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ encodes that either $S, r_0 \models \varphi_2$, or $S, r_0 \models \varphi_1$ and there is an edge from r_0 to some node r_1 such that C_{r_0,r_1} holds and $S, r_1 \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$. Here $\tau_{\mathsf{U}}(E[\varphi_1 \mathsf{U} \varphi_2], r_1, X \cup \{r_0\})$ returns the encoding of $S, r_1 \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$. Note that we add r_0 to X to ensure that no resource is revisited during EU-rewriting. Similarly, the rule for access constraints $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ encodes that either $S, r_0 \models \varphi_2$, or $S, r_0 \models \varphi_1$ holds, for any outgoing edge to a node r_1 we have $S, r_1 \models \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and it has no outgoing edges to nodes in X.

We illustrate our encoding with examples in Section 6.8. There, we also prove that this rewrite system always terminates, and that the generated SMT encoding of access constraints is correct.

6.6 Implementation and Evaluation

We report on an implementation of our policy synthesis algorithm, the case studies we conducted to evaluate its efficiency and scalability, and our empirical results.

6.6.1 Implementation

We have implemented a synthesizer that encodes policy synthesis instances into the QF_LIA and QF_UA logics of SMT-LIB v2 [14] and uses the Z3 SMT solver [37]. Our synthesizer is configured with configuration templates of different sizes. The local policies defined by these configuration templates are in disjunctive normal form. Namely, the local policies are defined as a disjunction of clauses, each clause consisting of a conjunction of terms, where each term is either an equality constraint for non-numerical attributes (e.g. $\mathsf{role} = \mathsf{employee}$) or an interval constraint for numeric attributes (e.g. $t_1 \leq \mathsf{time} \leq t_2$). We denote by C_k the configuration template that defines local policies with k clauses, each consisting of k terms. Note that the local policies defined in the template C_k may refer to at most k^2 attributes.

Our synthesizer implements the following procedure: it iteratively executes $\mathcal{S}_{\mathsf{smt}}(S, R, C_1)$, $\mathcal{S}_{\mathsf{smt}}(S, R, C_2)$, $\mathcal{S}_{\mathsf{smt}}(S, R, C_3)$, ..., stopping with the first call to $\mathcal{S}_{\mathsf{smt}}$ that returns a satisfying configuration, and returning this configuration. By iterating over templates increasing in size, our synthesizer generates small local policies, which is desirable for avoiding redundant attribute checks. For the running example, for instance, our synthesizer's

output includes the constraint correct-pin only for the entrance gates' local policies, and does not include this check, e.g., for the office room's policy. A satisfying solution for each case study can be found in the configuration template C_3. This indicates that real-world local policies have concise representations.

Note that our synthesizer may not terminate in a reasonable amount of time if no configuration satisfies the global requirements for the given resource structure. In our case studies, we used a simple iterative method to pinpoint such unsatisfiable requirements: we start with a singleton set of requirements, consisting of one satisfiable requirement, and iteratively extend this set by one requirement. This helped us identify a minimal set of conflicting requirements and revise problematic ones.

6.6.2 Case Studies

To investigate $\mathcal{S}_{\mathsf{smt}}$'s efficiency and scalability, we have conducted case studies in collaboration with KABA. We used real-world requirements and resource structures, and used our tool to synthesize policy configurations for a university building, a corporate building, and an airport terminal. Our synthesizer and all data are publicly available[1]. Below, we briefly explain the three case studies; relevant complexity metrics are summarized in Table 6.1.

University Building We modeled the main floor of ETH Zurich's computer science building. This floor consists of 66 subspaces including labs, offices, meeting rooms, and shared areas. The subspaces are labeled with four attributes that indicate: the research group to which a physical space is assigned, the physical space type (e.g., office, teaching room, or server room), the room number, and whether the physical spaces belongs to a secretary or a faculty member. Example requirements stipulate that a research group's PhD students can access all offices assigned to the group except those assigned to the faculty members and secretaries. The policies are defined over eight attributes.

Corporate Building We modeled an office space that consists of 20 subspaces, including a lobby, meeting rooms, offices, and restricted areas such as a server room, a mail room, and an HR office. The rooms are connected by three corridors, and they are labeled with attributes to mark public areas and employee-only zones. Access to these spaces is controlled by locks

[1] https://github.com/ptsankov/spctl

		University building	Corporate building	Airport terminal
Complexity metrics	Requirements	14	10	15
	PEPs	127	41	32
	Subspaces	66	20	13
Performance	Synthesis time	10.32	25.30	1.92
	Std. dev.	0.04	0.15	0.01

Table 6.1: Complexity metrics and policy synthesis times (in seconds) for the three cases studies

that are equipped with smartcard readers and PIN keypads. These locks are connected to a time server. Example requirements are that only the postman and HR employees can access the mail room, and that between noon and 1PM employees can access their offices without entering their PIN. The policies are defined over four attributes.

Airport Terminal We modeled the main terminal of a major international airport. The part of the terminal that we modeled includes subspaces such as the boarding pass control, security, and shopping areas. We have used the actual plan of the terminal, and considered 15 requirements, all currently enforced by the airport's access-control system. The area is divided into 13 subspaces, each labeled with zone identifiers (such as check-in and passport control). Example requirements stipulate that no passenger can access departure areas before passing through security, passengers with economy boarding passes cannot pass through the business/first-class ticket-control gates, and that only airport staff can access certain elevators.

6.6.3 Empirical Results

We ran all experiments on a Linux machine with a quad-code i7-4770 CPU, 32GB of RAM, running Z3 SMT v4.4.0. We present two sets of results: (1) the synthesizer's performance when used to synthesize the local policies for the three case studies, and (2) the synthesizer's scalability.

Performance We used our tool to synthesize the local policies for the three case studies, measuring the time taken for policy synthesis. We report the average synthesis time, measured over 10 runs of the synthesizer, in the bottom two rows of Table 6.1. The reported synthesis time is the

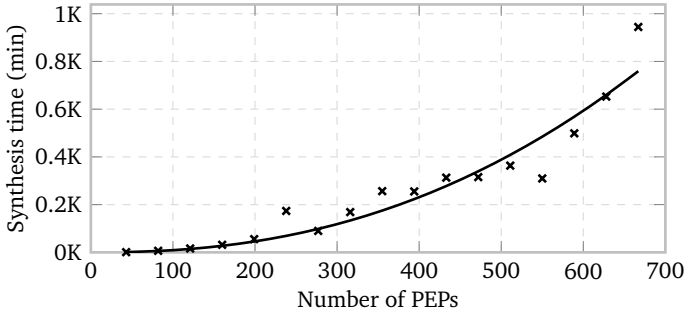


Figure 6.9: Scaling the number of PEPs

sum of the time taken for encoding the policy synthesis instance into SMT constraints, the time for solving the generated SMT constraints, and the time for iterating over the smaller templates for which the synthesizer returns unsat. In all three case studies, our tool synthesizes the local policies in less than 30 seconds. The standard deviation is under 0.2 seconds. This indicates that synthesizing local policies is practical, and can be used for real-world systems.

Scalability Experiments To investigate the scalability of our synthesis tool, we synthetically generated larger problem instances based on the corporate building case study. Although the case study originally consisted of a single floor, we increased the number of the floors in the building. We kept the same labeling for the newly added subspaces, so the original requirements also pertain to the newly added floors. Based on this method, we scaled the number of PEPs up to 670.

The time needed to synthesize local policies for different numbers of PEPs is given in Figure 6.9. The results show that our tool can synthesize a large number of local policies in a reasonable amount of time. For example, synthesizing up to 600 local policies takes less than ten hours. The tool's performance can be further improved using domain-specific heuristics for solving the resulting SMT constraints. Nevertheless, the tool already scales to most real-world scenarios: protected physical spaces usually have less than 500 PEPs.

6.7 Related Work

Physical Access Control The Grey project was an experiment in deploying a physical access-control system at the campus of Carnegie Mellon

University [16, 17]. As part of this project, researchers developed formal languages for specifying policies and credentials, and also developed techniques for detecting policy misconfigurations [18, 19]. The work on credential management, such as delegation, is orthogonal to the specification of the locks' local policies. In contrast to their work on detecting policy misconfigurations, we have developed a framework to synthesize policies that are guaranteed to enforces the global requirements, avoiding misconfigurations.

Several researchers have investigated SAT-based and model-checking techniques for reasoning about physical access control [45, 47]. Similarly to our work, these approaches model spatial constraints, and formalize global requirements that physical access-control systems must enforce. The authors of [45], for instance, model physical spaces using directed graphs and formalize global requirements in first-order logic. Their goal is to identify undesired denials due to blocked paths and unintended grants to restricted zones using SAT solvers. In contrast to these verification approaches, we develop a synthesis framework for generating correct local policies.

Network Policy Synthesis The problems of configuring networks with access-control and routing policies are related to the problem of constructing local policies from global requirements. In the network problem domain, one has an explicit resource structure defined by the network topology and must enforce global requirements using local rules deployed at the switches. Several synthesis algorithms for networks have been studied; e.g. see [15, 54, 68, 70, 73, 75, 97]. The authors of [54] and [15], for example, propose techniques for synthesizing local firewall rules that collectively enforce global network requirements in a given network topology. These approaches are sufficiently expressive for formalizing simple connectivity constraints, such as which hosts can access which services in a network. Similarly to our approach, recent techniques for synthesizing network configurations, such as [68, 70, 75, 97], also leverage SAT and SMT solvers. In addition to access-control constraints, these techniques also consider business constraints, such as deployment cost and usability. However, none of the above approaches for network synthesis supports branching properties, which are necessary for specifying requirements such as those stipulating that a fire-exit is reachable from any office room, as well as those that instantiate our waypointing and blocking requirement patterns; see Section 6.3.2 for examples. These requirements, which can be expressed in our framework, are central to physical access control. Existing network

policy synthesis algorithms, therefore, are not sufficiently expressive for handling access-control requirements for physical spaces.

Policy verification has also been studied in the context of computer networks; see e.g. [6]. However, this line of research is not concerned with synthesis, which is our work's main focus. We remark though that our synthesis algorithm can be readily used for verifying the conformance of a set of local policies to global access-control requirements; see Section 6.5.

Program Synthesis Program synthesis techniques, such as template-based synthesis [7, 82, 84, 85], reactive program synthesis from temporal specifications [32, 58, 69], and program repair techniques [29, 59], are related to policy synthesis for physical spaces. Similarly to our SMT-based algorithm, most of these synthesis frameworks also supplement the logical specification with a template, and exploit SMT solvers to efficiently explore the search space defined by the template. They cannot however express the relevant access-control requirements we have considered, such as those pertaining to branching properties. Our synthesis framework builds upon these techniques, and extends them with support for specifications that are needed for physical spaces.

Methods for synthesizing models of logical formulas, such as those in linear-temporal logic or CTL, have been extensively studied in the literature [9, 32, 51, 52, 74, 76]. In Section 6.4, we have described a policy synthesis algorithm based on CTL controller synthesis. This algorithm however comes at the expense of an exponential blow-up. Therefore, existing CTL synthesis tools and algorithms cannot be readily applied to synthesize attribute-based local policies in practice. Our efficient SMT-based algorithm addresses this practical challenge.

6.8 Proofs

6.8.1 Correctness and Complexity of the Algorithm $\mathcal{S}_{\mathsf{cs}}$

Correctness We now prove the correctness of $\mathcal{S}_{\mathsf{cs}}$.

Theorem 18. *Let S be a resource structure and R a set of requirements. If $\mathcal{S}_{\mathsf{cs}}(S,R) = c$ then $S, c \Vdash R$. If $\mathcal{S}_{\mathsf{cs}}(S,R) = \mathsf{unsat}$ then there is no configuration c such that $S, c \Vdash R$.*

Proof. We prove the two implications by contradiction.

Assume that $\mathcal{S}_{\mathsf{cs}}(S,R)$ returns a configuration c. Suppose for the sake of contradiction that $S, c \not\Vdash R$. Then, by definition of $\Vdash$, there is an access

request q and a requirement $T \Rightarrow \varphi$ such that $q \vdash T$ and $S_{c,q} \not\models \varphi$. Given a subset $R' \subseteq R$ of the requirements, let $T_{R'} = T_1 \wedge ... \wedge T_i \wedge \neg T_{i+1} \wedge \cdots \wedge \neg T_n$, where $\{T_1 \Rightarrow \varphi_1, \ldots, T_i \Rightarrow \varphi_i\} = R'$ and $\{T_{i+1} \Rightarrow \varphi_{i+1}, \ldots, T_n \Rightarrow \varphi_n\} = R \setminus R'$. The constraint $T_{R'}$ corresponds to the target computed at line 7 of Algorithm 1. Let $R_q = \{(T \Rightarrow \varphi) \in R \mid q \vdash T\}$ be the set of all requirements in R that are applicable to q. We have $q \vdash T_{R_q}$ (1). Furthermore, for any $R' \subseteq R$ where $R' \neq R_q$, we have $q \not\vdash T_{R'}$ (2). By definition of $S_{c,q}$, $S_{c,q}$ contains an edge e if e is an edge of S and $q \vdash c(e)$. Algorithm 1 constructs the configuration c by conjoining targets $T_{R'}$, where $R' \subseteq \mathcal{R}$, to the local policies $c(e)$; see line 14. From (1) and (2) we conclude the following: First, adding $\neg T_{R_q}$ to a local policy $c(e)$ removes the edge e in $S_{c,q}$ because $q \not\vdash c(e) \wedge (\neg T_{R_q})$. Second, adding $\neg T_{R'}$ to a local policy $c(e)$, where $R' \neq R_q$, does not remove the edge e in $S_{c,q}$ because $q \vdash c(e) \wedge (\neg T_{R'})$ iff $q \vdash c(e)$. It is immediate that $S_{c,q}$ contains those edges of S for which the target $\neg T_{R_q}$ is not conjoined to the local policy $c(e)$. We conclude that $S_{c,q}$ contains the edges $E' = \mathsf{cs}(S, \varphi_{R_q})$ (see line 13 of Algorithm 1), where φ_{R_q} conjoins the access constraints of all requirements in R_q. By definition of controller synthesis, we have $(S, E', r, L) \models \varphi_{R_q}$. Since $S_{c,q} = (\mathcal{R}, E', r, L)$, $S_{c,q} \models \varphi_{R_q}$. We can now deduce that $S_{c,q} \models \varphi$ because $(T \Rightarrow \varphi) \in R_q$. But previously we deduced that $S_{c,q} \not\models \varphi$. Thus we have a contradiction, and there is no access request q and requirement $T \Rightarrow \varphi$ such that $q \vdash T$ and $S_{c,q} \not\models \varphi$. Therefore, $S, c \Vdash R$.

Assume that $\mathcal{S}_{\mathsf{cs}}(S, R) = \mathsf{unsat}$. Suppose for the sake of contradiction that there is a configuration c such that $S, c \Vdash R$. From $\mathcal{S}_{\mathsf{cs}}(S, R) = \mathsf{unsat}$, by definition of Algorithm 1, it follows that there is a subset $R' = \{T_1 \Rightarrow \varphi_1, \ldots, T_k \Rightarrow \varphi_k\} \subseteq R$ of the requirements and an access request q, such that $q \vdash T_1 \wedge \cdots \wedge T_k$ (1) and $\mathsf{cs}(S, \varphi_1 \wedge \cdots \wedge \varphi_k) = \mathsf{unsat}$ (2). From (1), we know that all requirements in R' are applicable to q . Furthermore, since $S, c \Vdash R$, it must be that $S_{c,q} \models \varphi_i$, for $1 \leq i \leq k$. We get $S_{c,q} \models \varphi_1 \wedge \cdots \wedge \varphi_k$. From (2), by definition of controller synthesis, there is no resource structure $S' = (\mathcal{R}, E', r, L)$, with $E' \subseteq E$, such that $S' \models \varphi_1 \wedge \cdots \wedge \varphi_k$. Thus we have a contradiction, and we conclude that there is no configuration c such that $S, c \Vdash R$.

This concludes our proof. □

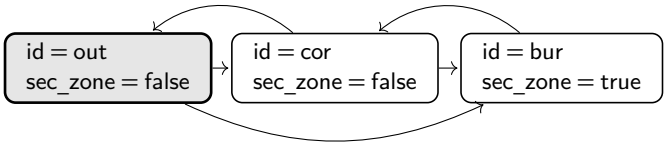
Complexity The running time of algorithm $\mathcal{S}_{\mathsf{cs}}$ is determined by the number of iterations of the loops, the complexity of checking the satisfiability of the conjunction of targets (line 8), and the complexity of solving each controller synthesis instance (line 10). The first loop is executed $|E|$ times,

where $|E|$ is the number of edges, and the second loop is executed $2^{|R|}$ times. The second loop checks one satisfiability instance and one controller synthesis instance. The complexity of checking satisfiability is in $\mathcal{O}(2^{k\cdot|A|})$ where $k = |D_{\mathsf{max}}|$ for the largest set D_{max} of values that appears in the constraint T, and $|A|$ is the number of attributes. Solving a controller synthesis instance requires checking $(\mathcal{R}, E', r_e, L) \models \varphi$ at most $2^{|E|}$ times, where $E' \subseteq E$ and φ is a conjunction of access constraints. The problem $(\mathcal{R}, E', r_e, L) \models \varphi$ can be decided using the model checking algorithm for CTL based on labeling, which is in $\mathcal{O}(|\varphi| \cdot (|\mathcal{R}| + |E'|))$, where $|\varphi|$ is the size of the access constraint φ [28]. The size of the largest access constraint given as input to cs is in $\mathcal{O}(|d| \cdot |R|)$, where d is the largest access constraint that appears in the requirements R. The running time of $\mathcal{S}_{\mathsf{cs}}$ is therefore $\mathcal{O}(2^{|R|} \cdot (2^{k\cdot|A|} + 2^{|E|} \cdot |R| \cdot d \cdot (|\mathcal{R}| + |E|)))$.

6.8.2 SMT Encoding

Example We illustrate the SMT encoding of an exists-until and an always-until access constraint in Figure 6.10. The SMT encoding of the access constraint $\mathsf{E}[(\neg\mathsf{sec\_zone})\mathsf{U}(\mathsf{id} = \mathsf{bur})]$ for the resource out formalizes that the two PEPs $(\mathsf{out}, \mathsf{cor})$ and $(\mathsf{cor}, \mathsf{bur})$ grant access or the PEP $(\mathsf{out}, \mathsf{bur})$ grants access. This guarantees the existence of a path that satisfies the access constraint. The SMT encoding of $\mathsf{A}[(\neg\mathsf{sec\_zone})\mathsf{U}(\mathsf{id} = \mathsf{bur})]$ for the resource out formalizes that the always-until constraint is satisfied along any path that starts from the resource out. Since any path that start with $(\mathsf{out}, \mathsf{bur}, \ldots)$ satisfies the access constraint, the SMT constraint imposes no constrains on the PEP $(\mathsf{out}, \mathsf{bur})$. However, not all paths that start with $(\mathsf{out}, \mathsf{cor}, \ldots)$ satisfy the access constraint. Concretely, the infinite path $(\mathsf{out}, \mathsf{cor}, \mathsf{out}, \mathsf{cor}, \ldots)$ violates the access constraint. The SMT constraint therefore formalizes that if there are paths starting with $(\mathsf{out}, \mathsf{cor}, \ldots)$, i.e. the PEP $(\mathsf{out}, \mathsf{cor})$ grants access, then the PEP $(\mathsf{cor}, \mathsf{out})$ denies access. This guarantees that the path violating the access constraint is not present in the resulting resource structure. Note that, since we consider only deadlock-free resource structures, the absence of the edge $(\mathsf{cor}, \mathsf{out})$ guarantees that the resulting resource structure has the edge $(\mathsf{cor}, \mathsf{bur})$, and therefore all paths starting with $(\mathsf{out}, \mathsf{cor}, \ldots)$ continue along resource bur.

Termination We first prove that the rewrite rules given in Figure 6.8 terminate.



$$\tau_U(\mathsf{E}[(\neg \mathsf{sec\_zone})\mathsf{U}(\mathsf{id} = \mathsf{bur})], \mathsf{out}, \emptyset) \hookrightarrow (C_{\mathsf{out,cor}} \wedge C_{\mathsf{cor,bur}}) \vee C_{\mathsf{out,bur}}$$

$$\tau_U(\mathsf{A}[(\neg \mathsf{sec\_zone})\mathsf{U}(\mathsf{id} = \mathsf{bur})], \mathsf{out}, \emptyset) \hookrightarrow (C_{\mathsf{out,cor}} \Rightarrow (\neg C_{\mathsf{cor,out}}))$$

Figure 6.10: Encoding exists-until and always-until access constraints using SMT constraints.

Theorem 19. *Let $S = (\mathcal{R}, E, r, L)$ be a resource structure. For any resource $r_0 \in \mathcal{R}$ and access constraint φ, the rewrite function $\tau(\varphi, r_0)$ terminates.*

Proof. The proof proceeds by induction on the length of the access constrain φ. Formally, we define the *length* of an access constraint φ, denoted by $l(\varphi)$, as

$$\begin{aligned} l(\mathsf{true}) &= 1 \\ l(a \in D) &= 1 \\ l(\neg\varphi) &= 1 + l(\varphi) \\ l(\mathsf{EX}\varphi) &= 1 + l(\varphi) \\ l(\mathsf{AX}\varphi) &= 1 + l(\varphi) \\ l(\varphi_1 \wedge \varphi_2) &= 1 + \mathsf{max}(l(\varphi_1), l(\varphi_2)) \\ l(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]) &= 1 + \mathsf{max}(l(\varphi_1), l(\varphi_2)) \\ l(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]) &= 1 + \mathsf{max}(l(\varphi_1), l(\varphi_2)) \end{aligned}$$

where $\mathsf{max}(n_1, n_2)$ returns n_1 if $n_1 \geq n_2$, otherwise it returns n_2. Note that $l(\varphi) \geq 1$ for any access constraint φ.

Base Case For the base case, $l(\varphi) = 1$, the access constraint is of the form true or $a \in D$. The rewrite function τ terminates in one step.

Inductive Step Assume that $\tau(\varphi, r_0)$ terminates for any access constraint φ of length $l(\varphi) \leq k$ (H1). We prove that $\tau(\varphi, r_0)$ terminates for any access constraint of length $l(\varphi) = k + 1$.

- For the cases where the access constraint φ is of the form $\neg\varphi_1$, $\mathsf{EX}\varphi_1$, $\mathsf{AX}\varphi_1$, the rewrite function $\tau(\varphi, r_0)$ calls $\tau(\varphi_1, r_0)$. By induction, $\tau(\varphi_1, r_0)$ terminates because $l(\varphi_1) = k$.
- The case where $\varphi = \varphi_1 \wedge \varphi_2$ also terminates because $l(\varphi_1) \leq k$ and $l(\varphi_2) \leq k$.
- For the cases where φ is of the form $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ or $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$, we need to show that $\tau_{\mathsf{U}}(\varphi, r_0, \emptyset)$ terminates. We prove that $\tau_{\mathsf{U}}(\varphi, r_0, X)$ terminates for any set $X \subseteq \mathcal{R}$ by descending induction on the size of the set X. For the base case, we have $|X| = |\mathcal{R}|$. Then, $\tau_{\mathsf{U}}(\varphi, r_0, \mathcal{R})$ calls $\tau(\varphi_1, r_0)$ and $\tau(\varphi_2, r_0)$. By our inductive hypothesis (H1), both $\tau(\varphi_1, r_0)$ and $\tau(\varphi_2, r_0)$ terminate since $l(\varphi_1) \leq k$ and $l(\varphi_2) \leq k$. For the inductive step, assume that $\tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]), r_0, X)$ terminates for any $X \subseteq \mathcal{R}$ of size $k \leq |X| \leq |\mathcal{R}|$ (H2). Consider a set $X' \subseteq \mathcal{R}$ of size $|X'| = k - 1$. Then, $\tau_{\mathsf{U}}(\varphi, r_0, X)$ calls the rewrite functions $\tau(\varphi_1, r_0)$, $\tau(\varphi_2, r_0)$, and $\tau_{\mathsf{U}}(\varphi, r_1, X' \cup \{r_0\})$, for $r_1 \in E(r_0) \setminus X$. The rewrite function $\tau(\varphi_1, r_0)$, $\tau(\varphi_2, r_0)$ terminate by the inductive hypothesis (H1). By the inductive hypothesis (H2), the rewrite function $\tau_{\mathsf{U}}(\varphi, r_1, X' \cup \{r_0\})$ terminates because $|X \cup \{r_0\}| = k$.

This completes our proof. □

Correctness We now prove that the correctness of our SMT-based policy synthesis algorithm. We start with several definitions. Our definitions are similar to those used to describe the decision procedure for CTL satisfiability given in [41]. Let φ_1 and φ_2 be two access constraints and $S = (\mathcal{R}, E, r, L)$ be a resource structure. We assume that S does not contain deadlock resources, i.e. for any resource $r_0 \in \mathcal{R}$, the set $E(r_0) = \{r_1 \in \mathcal{R} \mid (r_0, r_1) \in E\}$ is nonempty. We call access constraints of the form $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ eventuality constraints. We first define the derivation of a rooted directed graph from S for a given access constraint φ_2 and root node $r_0 \in \mathcal{R}$. We call this graph an *eventuality graph*. We then give two conditions over such eventuality graphs. The first condition is satisfied iff $S, r_0 \models \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$, while the second one is satisfied iff $S, r_0 \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$.

We define the eventuality graph $G(S, r_0, \varphi_2)$ as the rooted directed graph obtained by taking the node r_0 and all nodes and edges along all paths emanating from r_0 up to and including the first node r_1 such that $S, r_1 \models \varphi_2$; if there is no such node r_1 along a path, then all nodes and edges along the path are included in $G(S, r_0, \varphi_2)$. We call a node of $G(S, r_0, \varphi_2)$ an *interior node* if it has successors; otherwise, we call it a *frontier node*.

We now define the two conditions. We say that an eventuality graph $G(S, r_0, \varphi_2)$ *fulfills* $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ if

1. the graph is acyclic,
2. for any of its interior nodes r_1 we have $S, r_1 \models \varphi_1$, and
3. for any of its frontier nodes r_2 we have $S, r_2 \models \varphi_2$.

Note that for resource structures without deadlock resources, (1) implies (3). We say that an eventuality graph $G(S, r_0, \varphi_2)$ *fulfills* $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ if

1. the graph contains a frontier node r_2 such that $S, r_2 \models \varphi_2$, and
2. there is a path from r_0 to this frontier node r_2 such that for any interior node r_1 along the path we have $S, r_1 \models \varphi_1$.

From the CTL satisfiability decision procedure of [41], it follows that $S, r_0 \models \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ iff $G(S, r_0, \varphi_2)$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$, and $S, r_0 \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ iff $G(S, r_0, \varphi_2)$ fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$.

To prove the correctness of our SMT-based synthesis algorithm, we first prove that τ correctly encodes access constraint into SMT constraints. Towards this end, Theorem 20 establishes that the SMT encoding is correct for any access constraint and any singleton configuration template $C = \{c\}$, i.e. a template consisting of one configuration. To prove this theorem, we give two lemmas (Lemma 12 and Lemma 13), which show that the rewrite function τ_{U} correctly encodes eventuality access constraints. Afterwards, with Lemma 14 we lift the correctness of the access constraints' encoding to requirements. Finally, we restate and prove Theorem 16.

Theorem 20. *Let $S = (\mathcal{R}, E, r, S)$ be a resource structure. For any configuration c for S, resource $r_0 \in \mathcal{R}$, access request $q \in \mathcal{Q}$, and access constraint φ, we have*

$$S_{c,q}, r_0 \models \varphi \text{ iff } q \vdash \tau(\varphi, r_0).$$

Proof. The proof proceeds by induction on the derivation of $\tau(\varphi, r_0)$.

- For the case $\varphi = \mathsf{true}$, we have $\tau(\varphi, r_0) = \mathsf{true}$. We get $S_{c,q}, r_0 \models \varphi$ and $q \vdash \tau(\varphi, r_0)$.
- For the case $\varphi = (a \in D)$, we have $\tau(\varphi, r_0) = \mathsf{true}$ if $L(r_0)(a) \in D$, and $\tau(\varphi, r_0) = \mathsf{false}$ if $L(r_0)(a) \notin D$. Recall that $S_{c,q}, r_0 \models (a \in D)$ iff $L(r_0)(a) \in D$. It is immediate that $S_{c,q}, r_0 \models \varphi$ iff $q \vdash \tau(\varphi, r_0)$.
- For the case $\varphi = \neg\varphi'$, we have $\tau(\varphi, r_0) = \neg\tau(\varphi', r_0)$.

  $\Rightarrow$: Assume $S_{c,q}, r_0 \models \varphi$. We get $S_{c,q}, r_0 \not\models \varphi'$. By induction, $q \nvdash \tau(\varphi', r_0)$. Therefore $q \vdash \tau(\varphi, r_0)$.

  $\Leftarrow$: Assume $q \vdash \tau(\varphi, r_0)$. We get $q \nvdash \tau(\varphi', r_0)$. By induction, $S_{c,q} \not\models \varphi'$. Therefore $S_{c,q} \models \varphi$.

- For the case $\varphi = \varphi_1 \wedge \varphi_2$, we have $\tau(\varphi, r_0) = \tau(\varphi_1, r_0) \wedge \tau(\varphi_2, r_0)$.

  $\Rightarrow$: Assume $S_{c,q}, r_0 \models \varphi$. Therefore $S_{c,q}, r_0 \models \varphi_1$ and $S_{c,q}, r_0 \models \varphi_2$. By induction, $q \vdash \tau(\varphi_1, r_0)$ and $q \vdash \tau(\varphi_2, r_0)$, and therefore $q \vdash \tau(\varphi, r_0)$.

  $\Leftarrow$: Assume $q \vdash \tau(\varphi, r_0)$. Then $q \vdash \tau(\varphi_1, r_0)$ and $q \vdash \tau(\varphi_2, r_0)$. By induction, $S_{c,q}, r_0 \models \varphi_1$ and $S_{c,q}, r_0 \models \varphi_2$, and therefore $S_{c,q}, r_0 \models \varphi$.

- For the case $\varphi = \mathsf{EX}\varphi'$, we have $\tau(\varphi, r_0) = \exists r_1 \in E(r_0).\ (C_{r_0,r_1} \wedge \tau(\varphi', r_1))$.

  $\Rightarrow$: Assume $S_{c,q}, r_0 \models \varphi$. By definition of $S_{c,q}$, there is an edge (r_0, r_1) in E such that $q \vdash c((r_0, r_1))$ (1) and $S_{c,q}, r_1 \models \varphi'$ (2). Since $C = \{c\}$, we have $C_{r_0,r_1} = c((r_0, r_1))$. From (1), we thus get $q \vdash C_{r_0,r_1}$. From (2), by induction, we get $q \vdash \tau(\varphi', r_1)$. It follows that $q \vdash \tau(\varphi, r_0)$.

  $\Leftarrow$: Assume $q \vdash \tau(\varphi, r_0)$. There is an edge $r_1 \in E(r_0)$ such that $q \vdash C_{r_0,r_1}$ (1) and $q \vdash \tau(\varphi', r_1)$ (2). From (1), we get $q \vdash c((r_0, r_1))$, and thus there is an edge (r_0, r_1) also in $S_{c,q}$. From (2), by induction, we get $S, r_1 \models \varphi'$. Therefore, $S_{c,q}, r_0 \models \varphi$.

- For the case $\varphi = \mathsf{AX}\varphi'$, we have $\tau(\varphi, r_0) = \forall r_1 \in E(r_0).\big(C_{r_0,r_1} \Rightarrow \tau(\varphi', r_1)\big)$.

  $\Rightarrow$: Assume $S_{c,q}, r_0 \models \varphi$. Then, for any edge (r_0, r_1) of $S_{c,q}$ we have $S_{c,q}, r_1 \models \varphi'$. Consider an edge $(r_0, r_1) \in E$ such that $q \vdash C_{r_0,r_1}$. From $q \vdash C_{r_0,r_1}$, we know that (r_0, r_1) is also an edge in $S_{c,q}$. Therefore, $S_{c,q}, r_1 \models \varphi'$. By induction, $q \vdash \tau(\varphi', r_1)$. We get $q \vdash \tau(\varphi, r_0)$.

  $\Leftarrow$: Assume $q \vdash \tau(\varphi, r_0)$. Then for any edge $(r_0, r_1) \in E$, $q \vdash C_{r_0,r_1}$ implies $q \vdash \tau(\varphi', r_1)$. Consider an edge (r_0, r_1) of $S_{c,q}$. We know that $q \vdash C_{r_0,r_1}$, and thus $q \vdash \tau(\varphi', r_1)$. By induction, $S_{c,q}, r_1 \models \varphi'$. Therefore $S_{c,q}, r_0 \models \varphi$.

- For the case $\varphi = \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$, we have $\tau(\varphi, r_0) = \tau_{\mathsf{U}}(\varphi, r_0, \emptyset)$. By induction, for any resource $r_1 \in \mathcal{R}$ and any access request $q \in \mathcal{Q}$ we have

$$\bigwedge_{i \in \{1,2\}} S_{c,q}, r_1 \models \varphi_i \text{ iff } q \vdash \tau(\varphi_i, r_1).$$

  By Lemma 13, we get $S_{c,q}, r_0 \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ iff $q \vdash \tau_{\mathsf{U}}(\varphi, r_0, \emptyset)$.

- For case $\varphi = \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$, have $\tau(\varphi, r_0) = \tau_{\mathsf{U}}(\varphi, r_0, \emptyset)$. By induction, for any resource $r_1 \in \mathcal{R}$ and any access request $q \in \mathcal{Q}$ we have

$$\bigwedge_{i \in \{1,2\}} S_{c,q}, r_1 \models \varphi_i \text{ iff } q \vdash \tau(\varphi_i, r_1).$$

  By Lemma 12, we get $S_{c,q}, r_0 \models \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ iff $q \vdash \tau_{\mathsf{U}}(\varphi, r_0, \emptyset)$.

This concludes our proof. □

Lemma 12. *Let $S = (\mathcal{R}, E, r, S)$ be a resource structure, φ_1 and φ_2 be two access constraints, and $C = \{c\}$ be a configuration template. If for any resource $r_1 \in \mathcal{R}$ and any access request $q \in \mathcal{Q}$ we have*

$$\bigwedge_{i \in \{1,2\}} S_{c,q}, r_1 \models \varphi_i \textit{ iff } q \vdash \tau(\varphi_i, r_1), \tag{A1}$$

then for any resource $r_0 \in \mathcal{R}$ we have

$$S_{c,q}, r_0 \models \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2] \textit{ iff } q \vdash \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, \emptyset).$$

Proof. Assume (A1). Given a set $X \subseteq \mathcal{R}$ of resources, we say that $G(S_{c,q}, r_0, \varphi_2)$ is *X-disjoint* if no node of $G(S_{c,q}, r_0, \varphi_2)$ is contained in X. To avoid clutter, we will write $G[r_0]$ for $G(S_{c,q}, r_0, \varphi_2)$. We prove that for any set $X \subseteq \mathcal{R} \setminus \{r_0\}$ of resources,

$$\begin{array}{l} G[r_0] \text{ fulfills } \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2] \quad \textit{iff} \quad q \vdash \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, X). \\ \text{and } G[r_0] \text{ is } X\text{-disjoint} \end{array}$$

The proof proceeds by descending induction on the size of the set X. Note that for the case $X = \emptyset$ we have $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ iff $q \vdash \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, \emptyset)$. This case proves the lemma because $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ iff $S_{c,q}, r_0 \models \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$.

Before we start, we expand $\tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, X)$ to

$$\begin{aligned} &\tau(\varphi_2, r_0) \vee && (6.1)\\ &\quad \big(\tau(\varphi_1, r_0) && (6.2)\\ &\quad \wedge \big(\forall r_1 \in E(r_0) \cap X.\ \neg C_{r_0, r_1} \big) && (6.3)\\ &\quad \wedge \big(\forall r_1 \in E(r_0) \setminus X. (C_{r_0, r_1} \Rightarrow \\ &\qquad\qquad \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_1, X \cup \{r_0\})) \big) \big), && (6.4) \end{aligned}$$

as defined in Figure 6.8. To avoid clutter, we write, e.g., *(6.3) is true* for $q \vdash \forall r_1 \in E(r_0) \cap X.\ \neg C_{r_0, r_1}$.

Base Case For the base case we have $X = \mathcal{R} \setminus \{r_0\}$.

$\Rightarrow$: Assume $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and $G[r_0]$ is $\mathcal{R} \setminus \{r_0\}$-disjoint. From $\mathcal{R} \setminus \{r_0\}$, $G[r_0]$ consists of a single node, r_0. Furthermore, r_0 is a frontier node, and since $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$, we have $S_{c,q}, r_0 \models \varphi_2$. From (A1), we get $q \vdash \tau(\varphi_2, r_0)$. Since (6.1) is true, we get $q \vdash \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, \mathcal{R} \setminus \{r_0\})$.

$\Leftarrow$: Assume $q \vdash \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, \mathcal{R} \setminus \{r_0\})$. Since the resource structure $S_{c,q}$ is deadlock-free, there is a resource r_1 in $E(r_0) \cap (\mathcal{R} \setminus \{r_0\})$ such that $q \vdash C_{r_0,r_1}$. It follows that (6.3) is false. Therefore, it must be that $q \vdash \tau(\varphi_2, r_0)$. By (A1), $S_{c,q}, r_0 \models \varphi_2$. By definition of the eventuality graph $G[r_0]$, we conclude that it consists of a single node, r_0. It is immediate that $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and that it is $\mathcal{R} \setminus \{r_0\}$-disjoint.

Inductive Step Assume that for any set $X \subseteq \mathcal{R} \setminus \{r_0\}$ of size $k \leq |X| < |\mathcal{R}|$, $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and it is X-disjoint iff $\tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, X)$. We show that this holds for any set $X \subset \mathcal{R} \setminus \{r_0\}$ with $|X| = k - 1$.

$\Rightarrow$: Assume $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and it is X-disjoint.

Case 1: If $S_{c,q}, r_0 \models \varphi_2$, then from (A1) we get $q \vdash \tau(\varphi_2, r_0)$. Since (6.1) is true, it is immediate that $q \vdash \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, X)$.

Case 2: If $S, r_0 \not\models \varphi_2$, then by (A1) we have $q \not\vdash \tau(\varphi_2, r_0)$. Therefore, (6.1) is false, so we need to show that (6.2), (6.3), and (6.4) are all true:

* Since $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$, we have $S, r_0 \models \varphi_1$ because r_0 is an interior node. By (A1), we get $q \vdash \tau(\varphi_1, r_0)$, and thus (6.2) is true.
* If $G[r_0]$ has an edge (r_0, r_1), then it must be that the resource structure S has an edge (r_0, r_1) and $q \vdash c((r_0, r_1))$; otherwise, the edge (r_0, r_1) is removed from $S_{c,q}$. Furthermore, since $C = \{c\}$, C does not contain any control variables, and so $C_{r_0,r_1} = c((r_0, r_1))$. Now, since $G[r_0]$ is X-disjoint, we know that r_0 does not have any successors contained in X. Therefore, for any successor r_1 of r_0, we have $q \not\vdash C_{r_0,r_1}$. We conclude that (6.3) is true.
* Finally, consider an edge $r_1 \in E(r_0) \setminus X$ such that $q \vdash C_{r_0,r_1}$. Since $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and r_1 is a successor of r_0, it follows that $G(S_{c,q}, r_1, \varphi_2)$ also fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$. Furthermore, since $G[r_0]$ is X-disjoint, $G(S_{c,q}, r_1, \varphi_2)$ must be also

X-disjoint. Furthermore, $G(S_{c,q}, r_1, \varphi_2)$ does not contain the node r_0 because $G[r_0]$ is acyclic. We conclude that $G(S_{c,q}, r_1, \varphi_2)$ is $X \cup \{r_0\}$-disjoint and it fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$. By induction, we get $q \vdash \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_1, X \cup \{r_0\})$. Therefore, (6.4) is true.

$\Leftarrow$: Assume $q \vdash \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, X)$.

Case 1: If $q \vdash \tau(\varphi_2, r_0)$, then (6.1) is true. By (A1), $S_{c,q}, r_0 \models \varphi_2$. It is immediate that $G[r_0]$ consists of a single node, namely r_0. Therefore, $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and it is X-disjoint because $X \subset \mathcal{R} \setminus \{r_0\}$.

Case 2: If $q \nvdash \tau(\varphi_2, r_0)$, then (6.1) is false. Therefore, (6.2), (6.3), and (6.4) must be true. From (6.2) and (A1), we have $S_{c,q}, r_0 \models \varphi_1$. Consider any node $r_1 \in E(r_0) \setminus X$ such that $q \vdash C_{r_0,r_1}$. Then, r_1 is a successor of r_0 in the graph $G[r_0]$. From (6.4), we get $q \vdash \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_1, X \cup \{r_0\})$. By induction, $G(S_{c,q}, r_1, \varphi_2)$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and it is $X \cup \{r_0\}$-disjoint. Since r_0 is an internal node, $S, r_0 \models \varphi_1$, and all subgraphs rooted at r_0's successors fulfill $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$, it follows that $G[r_0]$ fulfills $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$. Furthermore, from (6.3) we know that r_0 has no successors in X. Since all subgraphs rooted at r_0's successors are $X \cup \{r_0\}$-disjoint, it follows that $G[r_0]$ is X-disjoint.

This concludes our proof. □

Lemma 13. *Let $S = (\mathcal{R}, E, r, S)$ be a resource structure, φ_1 and φ_2 be two access constraints, and $C = \{c\}$ be a configuration template. If for any resource $r_1 \in \mathcal{R}$ and any access request $q \in \mathcal{Q}$ we have*

$$\bigwedge_{i \in \{1,2\}} S_{c,q}, r_1 \models \varphi_i \text{ iff } q \vdash \tau(\varphi_i, r_1), \tag{A2}$$

then for any resource $r_0 \in \mathcal{R}$ we have

$$S_{c,q}, r_0 \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2] \text{ iff } q \vdash \tau(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \emptyset).$$

Proof. Given a directed graph $G = (\mathcal{R}, E)$ and a subset $X \subset \mathcal{R}$ of resources, we define *the projection of G on X* as $G|_X = (X, \{(r_0, r_1) \in E \mid \{r_0, r_1\} \subseteq X\})$. We will write $G[r_0]$ for $G(S_{c,q}, r_0, \varphi_2)$. We prove by induction on the size of the set X that for any $\{r_0\} \subseteq X \subseteq \mathcal{R}$, we have

$$G[r_0]|_X \text{ fulfills } \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2] \text{ iff } q \vdash \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \mathcal{R} \setminus X).$$

Note that since $G|_{\mathcal{R}} = G$ and $\tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \mathcal{R} \backslash \mathcal{R}) = \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \emptyset)$, the case for $X = \mathcal{R}$ proves the lemma.

We first expand $\tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \mathcal{R} \setminus X)$ to

$$\tau(\varphi_2, r_0) \vee \tag{6.5}$$

$$\Big(\tau(\varphi_1, r_0) \tag{6.6}$$

$$\wedge \exists r_1 \in E(r_0) \backslash (\mathcal{R} \backslash X).\ \big(C_{r_0, r_1}$$

$$\wedge\ \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_1, (\mathcal{R} \setminus X) \cup \{r_0\})\big)\Big) \tag{6.7}$$

Base Case For the base case, we have $X = \{r_0\}$.

$\Rightarrow$: Assume that $G(S_{c,q}, r_0, \varphi_2)|_{\{r_0\}}$ fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$. The graph $G[r_0]|_{\{r_0\}}$ consists of the single node r_0. The node r_0 is a frontier node, and therefore it must be that $S_{c,q}, r_0 \models \varphi_2$. By (A2), we have $q \vdash \tau(\varphi_2, r_0)$. Then (6.5) is true and therefore $q \vdash \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \mathcal{R} \setminus \{r_0\})$.

$\Leftarrow$: Assume that $q \vdash \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \mathcal{R} \setminus \{r_0\})$. Since here $X = \{r_0\}$ and S's edge relation is irreflexive, we have $E(r_0) \setminus (\mathcal{R} \setminus \{r_0\}) = \emptyset$. Therefore, (6.7) is false and it must be that $q \vdash \tau(\varphi_2, r_0)$. By (A2), we have $S_{c,q}, r_0 \models \varphi_2$. It is immediate that the graph $G[r_0]|_{\{r_0\}}$ consists of the single node r_0, and that it fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$.

Inductive Step Assume that $G[r_0]|_X$ fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ iff

$$q \vdash \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \mathcal{R} \setminus X)$$

holds for any set $\{r_0\} \subseteq X \subset \mathcal{R}$ of size $1 \leq |X| \leq k$, for some k, $1 \leq k < |R|$. We show that this also holds for any set $\{r_0\} \subset X \subseteq \mathcal{R}$ of size $|X| = k + 1$.

$\Rightarrow$: Assume $G[r_0]|_X$ fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$.

Case 1: If $S_{c,q}, r_0 \models \varphi_2$, then from (A2) we get $q \vdash \tau(\varphi_2, r_0)$. It is immediate that $q \vdash \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \mathcal{R} \setminus X)$.

Case 2: If $S_{c,q}, r_0 \not\models \varphi_2$, then from (A2) we get $q \nvdash \tau(\varphi_2, r_0)$. Since $G[r_0]|_X$ fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$, r_0 is an internal node and $S_{c,q}, r_0 \models \varphi_1$. By (A2), $q \vdash \tau(\varphi_1, r_0)$, and so (6.6) is true. Furthermore, r_0 has a successor r_1 with $q \vdash C_{r_0, r_1}$ such that r_1 has a path to a node r_n with $S_{c,q}, r_n \models \varphi_2$. We conclude that $G(S_{c,q}, r_1, \varphi_2)|_{(X \setminus \{r_0\})}$ fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$. By induction, since $|X \setminus \{r_0\}| = k - 1$, we

have $q \vdash \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_1, \mathcal{R} \setminus (X \setminus \{r_0\}))$. Since $r_0 \in X$, from $\mathcal{R} \setminus (X \setminus \{r_0\}) = (\mathcal{R} \setminus X) \cup \{r_0\}$ we conclude that (6.7) is also true. We conclude that $q \vdash \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, X)$.

$\Leftarrow$: Assume $q \vdash \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, X)$.

Case 1: If $q \vdash \tau(\varphi_2, r_0)$, then from (A2) we get $S_{c,q}, r_0 \models \varphi_2$. Therefore the graph $G[r_0]|_X$ consists of the single node r_0 with $S_{c,q}, r_0 \models \varphi_2$. It is immediate that $G[r_0]|_X$ fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$.

Case 2: If $q \not\vdash \tau(\varphi_2, r_0)$, then it must be that (6.6) and (6.7) are true. From (6.6) and (A2), we get $S_{c,q}, r_0 \models \varphi_1$. From (6.7), it follows that r_0 has a successor r_1 with $q \vdash C_{r_0,r_1}$ such that $q \vdash \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_1, (\mathcal{R} \setminus X) \cup \{r_0\})$. Since $r_0 \in X$, we have $(\mathcal{R} \setminus X) \cup \{r_0\} = \mathcal{R} \setminus (X \setminus \{r_0\})$. By induction, $G(S_{c,q}, r_1, \varphi_2)|_{X \setminus \{r_0\}}$ fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$, so there is a path from $r_1, \ldots, r_n$ in $G(S_{c,q}, r_1, \varphi_2)|_{X \cup \{r_0\}}$ along nodes in $X \setminus \{r_0\}$ where $S_{c,q}, r_n \models \varphi_2$ and $S_{c,q}, r_i \models \varphi_1$ for $1 \leq i < n$. It is immediate that there is a path $r_0, r_1, \ldots, r_n$ in $G[r_0]|_X$ such that $S_{c,q}, r_n \models \varphi_2$ and $S_{c,q}, r_i \models \varphi_1$ for $1 \leq i < n$. Therefore $G[r_0]|_X$ fulfills $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$.

This concludes our proof. □

Lemma 14. *Given a resource structure $S = (\mathcal{R}, E, r, L)$, a set $R = \{T_1 \Rightarrow \varphi_1, \ldots, T_n \Rightarrow \varphi_n\}$ of requirements, and a configuration template $C = \{c\}$, let $\phi = \textsc{Encode}(S, T_1 \Rightarrow \varphi_1, C) \wedge \cdots \wedge \textsc{Encode}(S, T_n \Rightarrow \varphi_n, C)$. The constraint $\forall a.\ \phi$ is satisfiable iff $S, c \Vdash R$.*

Proof. Note that since $C = \{c\}$, the formula ϕ contains no control variables, i.e. it contains only attribute variables. Since there is a one-to-one mapping from a valuation of the attribute variables $\vec{a}$ to an access request q, we have $\forall \vec{a}.\ \phi$ iff $\forall q \in \mathcal{Q}.\ q \vdash \phi$. We expand the constraint ϕ to $(T_1 \Rightarrow \tau(\varphi_1, r)) \wedge \cdots \wedge (T_n \Rightarrow \tau(\varphi_n, r))$. We get that $\forall \vec{a}.\ \phi$ iff for any access request $q \in \mathcal{Q}$, and for any requirement $T \Rightarrow \varphi$, $q \vdash T$ implies $q \vdash \tau(\varphi, r)$. By Lemma 20, $q \vdash \tau(\varphi, r)$ iff $S_{c,q}, r \models \varphi$. We get $\forall \vec{a}.\ \phi$ iff for any access request $q \in \mathcal{Q}$, and for any requirement $T \Rightarrow \varphi$, $q \vdash T$ implies $S_{c,q}, r \models \varphi$. By definition of $\Vdash$, we get $\forall \vec{a}.\ \phi$ iff $S, c \Vdash R$. □

We now restate and prove Theorem 16.

Theorem 16. *Let S be resource structure, R a set of requirements, and C a configuration template. If $\mathcal{S}_{\mathsf{smt}}(S, R, C) = c$ then $S, c \Vdash R$. If $\mathcal{S}_{\mathsf{smt}}(S, R, C) =$ unsat, then there is no configuration c in C such that $S, c \Vdash R$.*

Proof. Let $C = \{c_1, \ldots, c_n\}$. The formula $\exists \vec{z}.\forall \vec{a}.\ \phi$ generated by Algorithm 2 is equivalent to the formula $(\forall \vec{a}.\ \phi_{c_1}) \vee \cdots \vee (\forall \vec{a}.\ \phi_{c_n})$ where ϕ_{c_i} is the formula obtained by grounding the control variables $\vec{z}$ in ϕ with those values that encode the configuration c_i. Note that each formula ϕ_{c_i} is equivalent to the one obtained when using a configuration template $C_i = \{c_i\}$.

Assume that $\exists c \in C.\ S, c \Vdash R$. By Lemma 14, $\forall \vec{a}.\ \phi_{c_i}$ is satisfiable for some c_i in C. Therefore $\mathcal{S}_{\mathsf{smt}}(S, R, C)$ returns some configuration c_i. Assuming the DERIVE procedure correctly derives a configuration c_i from a model of $\exists \vec{z} \forall \vec{a}.\ \phi$, then $\mathcal{S}_{\mathsf{smt}}(S, R, C) = c_i$ for some c_i such that $\forall \vec{a}.\ \phi_{c_i}$. By Lemma 14, $S, c_i \Vdash R$.

Assume that $\neg \exists c \in C.\ S, c \Vdash R$. By Lemma 14, $\forall \vec{a}.\ \phi_{c_i}$ is not satisfiable for any c_i in C. Therefore $\mathcal{S}_{\mathsf{smt}}(S, R, C)$ returns unsat.

□

Chapter 7
Conclusion

Access-control systems, like all other complex software artifacts, are challenging to get right: Their behavior is influenced by a number of different components, making it difficult for security engineers to ensure that they grant and deny access in compliance with all access-control requirements. The correctness of access-control systems is nevertheless of utmost importance, especially in the presence of active adversaries whose goal is to deliberately exploit unintended access decisions and abuse the protected resources.

Automated techniques for verifying access-control systems offer a promising direction towards ensuring the correctness of access-control systems. The key idea is to first formalize the behavior of both (i) all access-control components, along with their interactions, that define how access decisions are made and (ii) the adversary. Then, we must check whether the formal specification satisfies all access-control requirements in the presence of the adversary. Going further, program synthesis techniques enable the automatic construction of low-level components directly from high-level access-control specifications. This frees security engineers from the tedious task of manually constructing low-level components, such as writing the locks' policies deployed at a physical access-control system.

In this thesis, we have investigated these promising directions and we have developed techniques that address key problems and limitations of existing formal access-control frameworks. Concretely, we presented an access-control framework that can be used to (i) specify access-control policies with authority delegation and policy composition, which are the core policy idioms of modern access-control systems, (ii) verify the behavior of access-control systems in the presence of an attacker who can cause communication and component failures, and (iii) synthesize low-level policies for access-control systems with multiple, distributed PEPs.

In addition to solving practical problems, the techniques presented in this thesis reveal numerous interesting directions for future research. We discuss these briefly below.

Verification Beyond the Policy Our fail-security analysis on real-world access-control systems demonstrates that verifying the policy alone is insufficient. This is because, in practice, there are access-control components other than the policy that often affect access decisions in surprising and

unintended ways. We therefore see our fail-security analysis as a first step towards investigating how components beyond the policy (in our case, failure handlers) affect the PDP's access decisions.

To extend security guarantees beyond the policy, further investigation is needed to understand which components influence access decisions and can be controlled by an adversary. Then, researchers must develop both more expressive system models that take into account these components and new attacker models. One interesting direction for future research is, for example, to extend our system model to multiple communicating PDPs, where PDPs themselves can fail. Furthermore, it would be interesting to consider an attacker who can submit malformed credentials and attributes to the PEP/PDP, and to use this attacker model to investigate whether the PDP correctly implement the semantics of the policy language in the presence of such unexpected inputs.

Synthesis for Access Control Our access-control synthesis algorithm is a first step towards bringing automated program synthesis techniques to the field of access control. This is an important connection to make, as it enables leveraging years of research in program synthesis to solve practical problems in access control.

Towards extending the expressiveness of our synthesis system, it would be interesting to include support for soft constraints. Currently, our synthesis framework returns any configuration that satisfies the high-level access-control requirements. Soft constraints would enable security engineers to specify their preference over local policies and let our synthesizer return more preferred policies. Examples of soft constraints include shortest-path constraints, which can be used to avoid long navigation paths, and optimality constraints, which can be used to synthesize local policies that avoid re-checking attributes that have been checked by other enforcement points. Handling such constraints is important for large-scale access-control systems in practice.

Towards making our synthesis system easier to use, it would be interesting to investigate the synthesis of access-control policies from example scenarios. Such scenarios are often easier to provide compared to writing a complete formal specification of all access-control requirements. The challenge here is to come up with an algorithm that synthesizes policies that generalize well, i.e. that do not overfit to the provided examples. A interactive-based approach would be helpful in dealing with the inherent ambiguity of such an example-based access-control synthesis approach.

To sum up, we believe that both verification and synthesis techniques will be an integral part of the construction of next-generation access-control systems. In this thesis, we developed novel access-controls models that support current standards, such as XACML v3.0, we developed novel concepts in access control, such as fail-security, and we investigated novel approaches to constructing access-control systems in an automated manner. We hope that these approaches will be extended further to help security engineers in constructing access-control systems that are correct by design while reducing the required engineering effort.

Bibliography

[1] Access control synthesis for physical spaces. Technical report.

[2] OASIS: Advancing open standards for the information society. `https://www.oasis-open.org/`.

[3] XSB. `http://xsb.sourceforge.net/`.

[4] MartÃ◌n Abadi. Access Control in a Core Calculus of Dependency. *Electronic Notes in Theoretical Computer Science*, 172(0):5 – 31, 2007.

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[6] E.S. Al-Shaer and H.H. Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *INFOCOM*, 2004.

[7] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. In *FMCAD*, 2013.

[8] M. Antoniotti and B. Mishra. The Supervisor Synthesis Problem for Unrestricted CTL is NP-complete. Technical report, New York University, 1995.

[9] Marco Antoniotti. *Synthesis and Verification of Discrete Controllers for Robotics and Manufacturing Devices with Temporal Logic and the Control-D System*. PhD thesis, NYU, 1995.

[10] K. R. Apt, H. A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pages 89–148. Morgan Kaufmann Publishers Inc., 1988.

[11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *CACM*, 2010.

[12] Augustus: Smart home access products. `http://august.com`.

[13] Axiomatics. Policy Decision Points, 2016. https://axiomatics.com/policy-decision-points.html.

[14] Clark Barrett, Aaron Stump, Cesare Tinelli, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava KrstiÄĞ, Michal Moskal, Leonardo De Moura, Roberto Sebastiani, To David Cok, and Jochen Hoenicke. The SMT-LIB Standard: Version 2.0, 2010.

[15] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: a novel firewall management toolkit. In *S&P*, pages 17–31. IEEE, 1999.

[16] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed Proving in Access-control Systems. In *S&P*, 2005.

[17] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Efficient Proving for Practical Distributed Access-control Systems. In *ESORICS*, 2007.

[18] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Detecting and Resolving Policy Misconfigurations in Access-control Systems. *TISSEC*, 2011.

[19] Lujo Bauer, Yuan Liang, Michael K. Reiter, and Chad Spensky. Discovering Access-control Misconfigurations: New Approaches and Evaluation Methodologies. In *CODASPY*, 2012.

[20] Moritz Y. Becker. Specification and analysis of dynamic authorisation policies. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, pages 203–217, 2009.

[21] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorization Language. *Journal of Computer Security*, 2010.

[22] N. D. Belnap. A Useful Four-Valued Logic. In *Modern Uses of Multiple-Valued Logic*. D. Reidel, 1977.

[23] Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. MBP: A Model Based Planner. In *IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[24] Bob Blakley and Craig Heath. Security Design Patterns. Technical report, The Open Group, 2004.

[25] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System Version 2. RFC 2704 (Informational), September 1999.

[26] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 17th Symposium on Security and Privacy*, pages 164–173, 1996.

[27] Lorenz Breidenbach. Efficient declarative physical access control. Bachelor thesis, ETH Zurich, March 2015.

[28] Glenn Bruns and Michael Huth. Access Control via Belnap Logic: Intuitive, Expressive, and Analyzable Policy Composition. *Transactions on Information and System Security*, 2011.

[29] Francesco Buccafurri, Thomas Eiter, Georg Gottlob, and Nicola Leone. Enhancing Model Checking in Verification by AI Techniques. *Journal of Artificial Intelligence*, 1999.

[30] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.*, pages 146–166, 1989.

[31] Network Admission Control Configuration Guide Cisco IOS Release 15MT. `http://www.cisco.com/en/US/docs/ios-xml/ios/sec_usr_nac/configuration/15-mt/sec-usr-nac-15-mt-book.pdf`.

[32] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, 1982.

[33] Microsoft Corporation. Disabling Firewall Service Lockdown due to Logging Failures. http://technet.microsoft.com/en-us/library/cc302466.aspx, May 2013.

[34] Carlos Cotrini, Thilo Weghorn, David Basin, and Manuel Clavel. Analyzing first-order role based access control. In *CSF*, 2015.

[35] Jason Crampton and Michael Huth. An Authorization Framework Resilient to Policy Evaluation Failures. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 472–487. Springer-Verlag, 2010.

[36] Jason Crampton and Charles Morisset. PTaCL: A language for attribute-based access control in open systems. In *POST*, 2012. doi: 10.1007/978-3-642-28641-4_21.

[37] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[38] John DeTreville. Binder, a Logic-Based Security Language. In *S&P*, 2002.

[39] Changyu Dong and Naranker Dulay. Shinren: Non-monotonic Trust Management for Distributed Systems. In *Trust Management IV*, volume 321 of *IFIP Advances in Information and Communication Technology*, pages 125–140. Springer, 2010.

[40] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[41] E. Allen Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*. MIT Press, 1990.

[42] E.Allen Emerson and Edmund M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 1982.

[43] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and Change-impact Analysis of Access-control Policies. In *ICSE*, 2005.

[44] Melvin Fitting. Bilattices in Logic Programming. In *Multiple-Valued Logic, 1990., Proceedings of the Twentieth International Symposium on*, pages 238–246, 1990.

[45] William M. Fitzgerald, Fatih Turkmen, and Simon N. Foley. Anomaly Analysis for Physical Access Control Security Configuration. In *CRiSIS*, 2012.

[46] Simone Frau and Mohammad Torabi Dashti. Integrated Specification and Verification of Security Protocols and Policies. In *Proceedings of the Computer Security Foundations Symposium*, pages 18 –32, 2011.

[47] Robert Frohardt, Bor-YuhEvan Chang, and Sriram Sankaranarayanan. Access Nets: Modeling Access to Physical Spaces. In *VMCAI*, 2011.

[48] Deepak Garg and Frank Pfenn. Non-Interference in Constructive Authorization Logic. In *Proceedings of the 19th IEEE workshop on Computer Security Foundations*, CSFW '06, pages 283–296, Washington, DC, USA, 2006. IEEE Computer Society.

[49] Deepak Garg and Frank Pfenning. Stateful authorization logic - proof theory and a case study. *Journal of Computer Security*, 20(4):353–391, 2012.

[50] Goji lock. `http://gojiaccess.com/`.

[51] A. Gromyko, M. Pistore, and P. Traverso. A Tool for Controller Synthesis via Symbolic Model Checking. In *WODES*, 2006.

[52] Andrey Gromyko, Marco Pistore, and Paolo Traverso. Supervisory Control via Symbolic Model Checking. Technical report, University of Trento, 2006.

[53] Yuri Gurevich and Itay Neeman. DKAL: Distributed-Knowledge Authorization Language. In *CSF*, 2008.

[54] Joshua D. Guttman. Filtering postures: Local enforcement for global policies. In *S&P*, 1997.

[55] Arno Haase. Java Idioms: Exception Handling. In *Proceedings of the 7th European Conference on Pattern Languages of Programs*, pages 41–70, 2002.

[56] Michael Howard, David LeBlanc, and John Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw Hill, 2009.

[57] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[58] B. Jobstmann and R. Bloem. Optimizations for LTL Synthesis. In *FMCAD*, 2006.

[59] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program Repair As a Game. In *CAV*, 2005.

[60] KABA KES-2200. http://www.kaba.co.nz/Products-Solutions/Access-Control/Electric-Locking/34392-32446/electric-strikes.html.

[61] KABA. KABA Exos 9300, 2013.

[62] Kerberos 5, Release 1.2.8. http://web.mit.edu/kerberos/www/krb5-1.2/krb5-1.2.8/.

[63] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing Web Access Control Policies. In *Proceedings of the 16th international conference on WWW*, pages 677–686. ACM, 2007.

[64] Barbara S. Lerner, Stefan Christov, Leon J. Osterweil, Reda Bendraou, Udo Kannengiesser, , and Alexander Wise. Exception handling patterns for process modeling. *IEEE Transactions on Software Engineering*, pages 162–183, 2010.

[65] Ninghui Li, J.C. Mitchell, and W.H. Winsborough. Design of a Role-based Trust-management Framework. In *S&P*, pages 114–130, 2002.

[66] Bolt: Unlock your door without keys. http://lockitron.com.

[67] Srdjan Marinovic, Robert Craven, Jiefei Ma, and Naranker Dulay. Rumpole: A Flexible Break-glass Access Control Model. In *Symposium on Access Control Models and Technologies*, SACMAT '11, pages 73–82. ACM, 2011.

[68] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient Synthesis of Network Updates. In *PLDI*, 2015.

[69] Andreas Morgenstern and Klaus Schneider. Program Sketching via CTL* Model Checking. In *SPIN Conference on Model Checking Software*, 2011.

[70] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008.

[71] Qun Ni, Elisa Bertino, and Jorge Lobo. D-Algebra for Composing Access Control Policy Decisions. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 298–309, 2009.

[72] OpenSSO Enterprise 8.0, Authentication Service Failover. http://docs.oracle.com/cd/E19681-01/820-3885/gbarl/index.html.

[73] Oded Padon, Neil Immerman, Aleksandr Karbyshev, Ori Lahav, Mooly Sagiv, and Sharon Shoham. Decentralizing SDN Policies. In *POPL'15*. ACM, 2015.

[74] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.

[75] Mohammad Ashiqur Rahman and Ehab Al-Shaer. A formal framework for network security design synthesis. In *ICDCS*, pages 560–570. IEEE, 2013.

[76] P. J. Ramadge and W. M. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization*, 1987.

[77] Red Hat 6.5, 2.8.2.1 Firewall Configuration Tool. http://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/.

[78] Erik Rissanen. XACML 3.0 Additional Combining Algorithms Profile Version 1.0. Technical report, Axiomatics, 2013.

[79] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, pages 1278–1308, 1975.

[80] O. Shmueli. Decidability and Expressiveness Aspects of Logic Queries. In *Proceedings of the ACM Symposium on Principles of database systems*. ACM, 1987.

[81] SNIC. SweGrid: e-Infrastructure for Computing and Storage. http://www.snic.vr.se/projects/swegrid/.

[82] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[83] Spring Security. http://projects.spring.io/spring-security/.

[84] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based Inductive Synthesis for Program Inversion. In *PLDI*, 2011.

[85] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From Program Verification to Program Synthesis. In *POPL*, 2010.

[86] Herbert H. Thompson, James A. Whittaker, and Florence E. Mottay. Software Security Vulnerability Testing in Hostile Environments. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 260–264. ACM, 2002.

[87] Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David Basin. Decentralized Composite Access Control. In *POST*, 2014. doi: 10.1007/978-3-642-54792-8_14.

[88] Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David Basin. Decentralized Composite Access Control. In *Principles of Security and Trust*, pages 245–264, 2014.

[89] Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David Basin. Fail-Secure Access Control. In *CCS*, 2014.

[90] Moshe Y. Vardi. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 137–146, New York, NY, USA, 1982. ACM.

[91] John Viega and Gart McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2002.

[92] Jeffrey M. Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[93] IBM WebSphere. http://www-01.ibm.com/software/websphere/.

[94] eXtensible Access Control Markup Language (XACML) Version 3.0. http://docs.oasis-open.org/xacml/3.0/.

[95] XACML Failure Handling Flaw. https://lists.oasis-open.org/archives/xacml/200703/doc00000.doc.

[96] eXtensible Access Control Markup Language (XACML) Version 2.0. www.oasis-open.org/committees/tc_home.php.

[97] Bin Zhang and Ehab Al-Shaer. On synthesizing distributed firewall configurations considering risk, usability and cost constraints. In *CNSM*, 2011.

Resume

PETAR TSANKOV

Born on May 5, 1984 in Stara Zagora, Bulgaria.
Citizen of Bulgaria and the USA.

Education

2010 – 2011 **Master of Science in Computer Science**
ETH Zurich, Switzerland

2007 – 2009 **Bachelor of Science in Computer Science**, Highest Honor
Georgia Institute of Technology, USA

Work Experience

2012 – 2016 **Research Assistant**
Department of Computer Science, ETH Zurich