ABSTRACT

JIANG, JIAMING. Reflective Attribute-based Access Control. (Under the direction of Jon Doyle.)

Access control (AC) concerns the problem of deciding whether a party is authorized to read, write, or otherwise access some entity in a system. Attribute-based access control (ABAC) is a kind of AC paradigm in which access policies can use entities' attributes in making authorization decisions. However, current ABAC models do not properly restrict and protect administrative operations such as adding a new policy, potentially giving administrators too much power. These systems also give non-administrators too little power, denying them sufficient ability to manage the entities in a system. Moreover, most current ABAC models do not provide a way of specifying the relationships among attributes, thus burdening administrators with additional maintenance work when values of related attributes are updated. Such a system is also prone to disclosing sensitive information. If a piece of public information is related to a sensitive one, releasing the public piece may leak the sensitive one.

We propose a reflective ABAC paradigm called *RefABAC* to address these issues with current ABAC models. We use *reflection* to enable the system to represent and reason about itself and its components, and to monitor and control its own behaviors. We formalize the structure (subjects, objects, actions) and content (attributes, policies, action effects) of an AC system in way by using a variation of dynamic logic. The semantics of the logic provides an implementation-independent expression of the meanings of attributes and policies and other AC system components, so that RefABAC's logical formalization serves as a formal specification of the AC system. We formalize four kinds of RefABAC models. *Core RefABAC* serves as a baseline framework. Through *reification*, all operations of all users can be properly governed by policies, including adding a new policy. We specify relationships among attributes using *constraints*, which are similar to TBox axioms in description logics and assertions in SQL. Any kinds of entities may have attributes, including attributes themselves. *DC (Disclosure Control) RefABAC* improves upon Core RefABAC by allowing a system to make authorization decisions based on what the system has disclosed in the past. *Coco RefABAC* adds a policy-conflict-resolution mechanism to Core RefABAC. A conflict among two policies can be resolved using a *dominance policy*, which asserts the dominance of a policy over the other. *PCR (Policy Conflict Resolution) RefABAC* generalizes the method in Coco by defining multi-level dominance policies. We illustrate the application of these concepts in terms of a healthcare scenario, and provide a sample implementation in SQL.

Reflective Attribute-based Access Control

by
Jiaming Jiang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2020

APPROVED BY:

Jon Doyle Chair of Advisory Committee	Chris Martens
Rada Chirkova	Christopher Healey

DEDICATION

Thank you to my adviser who guided me in this process and the committee who kept me on track.

BIOGRAPHY

The author, Jiaming Jiang, was born in Yantai, Shandong, China. She received her Bachelor's degree in Computer Science in 2014 from Calvin College (renamed to Calvin University in 2019). After her undergraduate studies, she started to work with Dr. Jon Doyle as a Ph.D. student in Computer Science at North Carolina State University. Her research focuses on using mathematical logics to define and mechanize security-related policies and systems.

ACKNOWLEDGEMENTS

I would like to extend my sincere gratitude to the following people who helped me bring this work into reality:

To my thesis advisor, Dr. Jon Doyle, who continuously supported and guided me throughout of my Ph.D. study and research.

To Dr. Rada Chirkova for her immense knowledge and ample time spent on my research projects.

To all my committee members—Dr. Rada Chirkova, Dr. Chris Martens, and Dr. Christopher Healey—for their constructive comments and suggestions.

I would also like to thank my family and my friends for their moral support.

My studies were supported in part by the US Department of Defense through a grant to the Science of Security Lablet of NC State University, and in part by the SAS Institute Chair of Computer Science endowment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 Problem Overview

An access control (AC) system uses authorization policies to make decisions on whether a *subject* (a person, a system or any active entity) is authorized to access some resource in the system. We identify and address three major problems of current AC models. First of all, system administrators often have too much power in performing actions in a system, while non-administrators may not have enough power in doing so. Current AC models assume the existence of several administrative subjects who, by default, are authorized to perform all administrative actions, such as defining and changing the properties of other subjects and defining what policies there are in the systems. On the other hand, non-administrative subjects are not allowed to perform these administrative actions at all. We argue that whether if a subject is an administrator or not, its ability to perform any actions—reading an object or adding a new policy—should be limited and regulated by policies.

Second, current AC models do not have adequate mechanisms to protect the sensitive properties and policies. They assume that every policy and every property of an entity stored in a system are either private and not accessible to subjects, or public and accessible to everyone. Either way, the models do not provide a way to identify the sensitive properties and policies nor provide a mechanism for a subject to ask for the value of some property or for the definition of a policy. With disclosures of such governed by other policies, we argue that similar to accessing objects, accessing properties and policies should be regulated by policies as well.

The third problem is that most current AC models do not take into account the relationships among properties when defining the components in a system or when making an authorization decision. Suppose one property indicates the diseases a patient has and another property indicates

the medicines a patient takes. A possible relationship between the two properties is that if a patient is taking blood thinners then they are likely to have atrial fibrillation. Therefore, one may infer what diseases a person has based on the medicines he or she takes. If a patient's disease is considered private, then the system leaks sensitive information by disclosing a patient's medicines. We argue that an AC model should provide a way of defining such relationships among properties in order to prevent from information leakage through inference.

In this dissertation, we address the above problems of 1) regulating administrative actions, 2) protecting sensitive information, and 3) defining relationships among properties in the context of attribute-based access control (ABAC), with extension to distributed systems. In an ABAC system, policies are expressed using **attributes**, i.e., properties of entities in the system. For convenience, we call a subject, an object, a policy, an attribute, the value of an attribute, or any component in a system an **entity**. Subjects are active entities that are able to perform an **action** in a system. Common AC actions are reading and writing an entity. Administrative actions include 1) creating and deleting an entity; and 2) adding, deleting, and changing the value of some entity's attribute.

In a distributed setting, multiple systems need to communicate with each other and share information. Such practices often happen among hospitals who share their data for research. Suppose a doctor in a hospital h_1 makes a request to another hospital h_2 to access some medical records controlled by h_2. Then h_1 may need to share information—e.g., the values of the doctor's attributes—with h_2 so that h_2 can decide whether the doctor's request should be granted.

In the context of distributed ABAC, we summarize the problems we will address in this dissertation as follows.

1. How to govern and regulate actions of both the administrators and the non-administrative subjects in a uniform way?
2. How to account for the relationships among attributes?
3. How to protect the disclosure of sensitive information stored in attributes, relationships among attributes, and policies?
4. How to extend the solutions of the problems to distributed systems? Specifically,
   (a) What if a subject in one system needs to access an object or a piece of information in another system?
   (b) What if a system needs information from other systems to make a decision, i.e., a policy is expressed using attributes from more than one systems?
   (c) What if several attributes that belong to different systems relate to each other?

The problems are closely related to and dependent on one another. Consider the case where two attributes, a public one $Attr_1$ and a private one $Attr_2$, are directly related to each other in a way such that if a subject knows the value of one, he or she can infer the value of the other. If a subject is authorized to read the values of $Attr_1$, the sensitive information stored in $Attr_2$ is leaked to the

subject, and vice versa. Designers of complicated systems need to be conscious of these underlying privacy issues. We do not attempt to exhaustively identify or address all such complications in this dissertation. One potential future work is to formally define these complications so that one can analyze them.

1.2 Motivation: Scenario and Examples

We summarize the motivations mentioned above and provide examples from the following scenario adapted from an Australian aged care facility [EB04].

Scenario 1.1. An aged care facility offers accommodation for some residents. The care facility has one manager, who is in charge of the administrative duties including hiring and firing workers. The workers include a front desk, a cashier, and several nurses. The front desk is responsible for logging visitors and answering general questions. The cashier manages residents' insurance and payment information. The nurses conduct daily checkups of the residents and help the visiting doctors with monthly checkups and emergency situations. The nurses need to write up the results of the daily checkups. The visiting doctors are doctors from the two nearby hospitals. Each resident has a visiting doctor as his or her responsible doctor. Each resident's responsible doctor conducts an examination of the resident monthly. The visiting doctors need to write up the results of the monthly examinations. They may also take private notes, which are only accessible to themselves. The care facility also has multiple volunteers. Different volunteers have different responsibilities and they are not allowed to do other volunteers' work. Some of them help the nurses with daily checkups and take care of residents when the nurses are not available. Some distribute medicines at set times. Some arrange meetings for residents with their responsible doctors and visitors.

Example 1.1 (Distributed Systems)**.** A visiting doctor must be a doctor from the nearby hospitals.

Example 1.2 (Regulated Administrative Actions)**.** A resident may change his or her own responsible doctor, but such a changes requires the signature of the current responsible doctor, of the doctor the resident requests to change to, and of the manager.

Example 1.3 (Sensitive Attributes)**.** The manager and the volunteers who are responsible for arranging meetings between residents and their responsible doctors are authorized to know the time of the meetings. The attending parties of a meeting are also authorized to know the time of the meeting.

Example 1.4 (Sensitive Policies and Decision Explanation)**.** Resident Carol makes a request to change her responsible doctor from Bob to George. Carol does not have the signature of the manager. According to the policies in Example 1.2, Carol's request is denied. Carol inquires about the reason for the denial and finds out that it is due the lack of the manager's signature.

Example 1.5 (Related Attributes)**.** If a resident has heart diseases, then the resident's responsible doctor is Bob, a doctor from one of the nearby hospitals. Only the manager, the resident and his or her responsible doctor are authorized to read who is the responsible doctor of the resident. If a

non-manager subject reads that a resident has heart diseases, then the subject can infer who is the resident's responsible doctor.

1.3 Proposed Solution: Reflective ABAC

To solve the above problems, we propose a reflective ABAC paradigm called RefABAC.

RefABAC uses a version of dynamic logic to state a logical specification of desired AC behavior. Using a formal logic has several major advantages. First, formal logics provide a clear semantics of defining the meaning of the entities—attributes, policies, and actions, etc.—in a model. Second, the logical formalization separates the implementation-independent specification of the correct AC behavior from interface languages used to implement this behavior, which might use an existing AC language such as XACML [Oas13]. Third, the logical language provides a simple way of defining relationships among attributes. The axioms used to define relationships among attributes together with attribute instances form the knowledge base of a system.

RefABAC extends dynamic logic with a uniform reflection mechanism that *reifies* entities, through which one can assign a name to a formula so as to use the name to refer to the formula. In this way, an attribute can be defined on any entity, including policies and attributes themselves.

The logical language of RefABAC provides uniformity in treating other AC concepts as well. All actions of reading, adding, and deleting an entity can be formalized as actions of reading, adding, and deleting the value of an attribute for some entity. Adding a policy p to a system `sys` means that $\mathit{Policy}(\mathtt{sys}, p)$ becomes true and p is a new entity in the system. Therefore, all administrative and non-administrative actions are treated in the same way and regulated by policies.

RefABAC also accommodates distributed systems. In our definition of RefABAC, systems are also subjects in the sense that systems can perform actions. A system may make a request to another system to access some piece of information. Such communication often happens among hospitals, who need to share research data and findings. Our formalism also provides a way of indicating in which system a piece of information is stored. For instance, suppose a system s makes a request to another system s' to access some information ϕ. After s' discloses ϕ to s, s can learn about that the fact ϕ is stored in s'. Therefore, one may track from which system each piece of information comes and what is true in other systems.

CHAPTER

2

BACKGROUND

We identified some issues with the current access control (AC) models in Chapter 1, regarding the management of administrative actions, relationships among attributes, and protection of sensitive information. To address the issues, we propose a reflective ABAC (RefABAC) paradigm using dynamic logics. Before we discuss the details of paradigm, we first introduce and clarify some common terminology used in the AC literature (Section 2.1), and compare several popular AC paradigms (Section 2.2). We also informally discuss what reflection is and how it has been used (Section 2.3).

2.1 Clarification of Terminology

2.1.1 User, Subject and Object

In some AC literature, the distinctions between users and subjects, and between subjects and objects are not explicitly defined. In *Discretionary Access Control (DAC)*, subjects are system processes, and objects are files, folders, programs, etc.. Thus, objects include subjects in DAC. Users are often user accounts that own the subjects, though they are not immediately relevant in DAC. Some ABAC models [Jin12] explicitly specify that users are user accounts; subjects are login sessions of user accounts; and objects are resources and documents. Thus, objects and subjects are exclusive. However, in some ABAC models for web services [YT05], a service can be both a subject and an object. For clarity and simplicity, we define users, subjects and objects as follows in Definition 2.1.

Definition 2.1. A **user** is a human being. A **subject** is an active entity that is able to or has the potential to perform an action on an entity, e.g., to access a file. An **object** is an inactive entity that is not able to perform an action, such as a text document.

We will use the term *subject* as a generalization of any *active* entity. A subject can be a human user, an application, a service, a login process of a user account, a system process, and so on. We avoid the usage of *user* to keep the definitions for our model simple. When we do use the term user, we refer to a human entity. On the other hand, we use *objects* as a generalization of all the regular inactive and accessible entities. Based on this definition, subjects and objects are exclusive.

In our RefABAC paradigm, an attribute or a policy is also accessible in the sense that it can be disclosed and modified. To be consistent with the common notion of objects, we do not consider the accessible entities such as attributes and policies as objects.

2.1.2 Operation, Right and Permission

Definition 2.2. An **operation** is an action that is performed or requested to be performed by a subject on an entity. A **right**, or **access right**, is a type of actions, such as read and write. A **permission** is a pair (r, e), where r is a right and e is an entity, meaning actions of type r can be performed on e.

The term *permission* has been used inconsistently in the AC literature. In [Jin12; Jin14], a permission is essentially a type of actions. In [Jha15], a permission is a pair (o, e), where o is an action and e is an entity. In [San96], a permission is defined as "an approval of a particular mode of access to one or more objects"; however, it does not provide a definitive interpretation of permissions. Due to such inconsistent usages, we avoid using the term permission. But when we do use it, a permission is a pair (r, e) as defined in Definition 2.2. When we say a subject s has permission (r, e), we mean s is authorized or permitted to perform an action of type r on e.

2.1.3 Paradigm, Framework, Model, Configuration, and System

The terms *paradigm*, *framework*, *model*, and *system* have been used inter-changeably. According to the online Merriam-Webster Dictionary, a framework [Fra] is "a basic conceptual structure," and a model [Mod] is "a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs." In other words, they both provide an abstract rather than concrete structure or description of some concept or problem. In this dissertation, a framework covers a model, and a paradigm covers a framework. They are informally described as follows.

> A **model** is an abstract representation of a system. A **framework** is a basic structure of how multiple systems relate and communicate with each other. A **paradigm** describes a family of frameworks that have common structures.

We say RefABAC is a paradigm, which has various versions called frameworks, such as the baseline framework Core RefABAC we will define in Chapter 5. Based on the specification of a RefABAC framework, one may define a RefABAC model for a set of systems. Each RefABAC model contains at least one system model that defines the domain-dependent entities of the corresponding system.

2.2 AC Paradigms

This section briefly discusses some popular AC paradigms, namely Discretionary (DAC), Mandatory (MAC) and Role-based (RBAC) access control paradigms. We also compare them with Attribute-based Access Control (ABAC). We will refer to DAC, MAC and RBAC as the "traditional" AC paradigms for convenience.

2.2.1 Discretionary Access Control

Discretionary access control (DAC) [Har76; Ini13; Def85] was primarily used in operating systems. Subjects are system processes. Objects are files, programs, etc., and they include subjects.

In a DAC model, the subject ownership of an object and delegations of an ownership decide whether a subject can access the object. The main feature of DAC is that an access right of a subject can be delegated to or shared with another subject. Specifically, a most common policy in DAC states that the owner subject of an object is authorized to share an access right of a particular object with another subject. Similarly, the owner subject can also revoke a previously shared right. Depending on the design, some DAC models also have policies that authorize a non-owner subject to share an access right it has.

Different DAC models use different mechanisms to store and enforce the policies. The DAC model in [Har76] uses an access control matrix to store which subject has what access rights on which objects. An **access control matrix** has subjects as rows, objects as columns, and the entry at row i and column j indicates the access right(s) that the corresponding subject has on the corresponding object. A distinguished right *own* indicates that the subject is the owner of the object. Any right other than *own* is shareable. For instance, suppose the entry for the subject s_1 and the object o is $\{own, read, execute\}$, then s_2 is authorized to share the *read* and *execute* rights on o with any another subject s_2. Upon sharing, s_2 would be authorized to read and execute o as well.

2.2.2 Mandatory Access Control

Mandatory access control (MAC) [BL73; San73] is mainly used in military related applications. Subjects can be human users, user accounts, or login sessions of user accounts. Objects are usually files and documents.

In a MAC model, each subject and object is assigned a **security label** and the security labels are comparable. Different MAC models use different policies on how to compare the security labels. The BLP MAC model defined in [San73] focuses on preventing information from flowing from higher-security sources to lower-security sources. Let λ denote the security label of a subject or object. The reading policy in BLP states that a subject s can read an object o *only if* s's security label is not lower then o's, i.e., $\lambda(s) \geq \lambda(o)$. The writing policy states that s can write o *only if* $\lambda(s) \leq \lambda(o)$. Similarly, for any actions that modify the state of an object, such as creating and deleting an entity, the $\lambda(s) \leq \lambda(o)$ policy applies.

MAC is better to be used together with DAC. If a subject s is the owner of an object o and $\lambda(s) > \lambda(o)$, then even if DAC authorizes s to write o, MAC prohibits it. If s is a login session of a user account, the account needs to log in using a lower security label, i.e., at least as low as $\lambda(o)$, to write the object o.

2.2.3 Role-based Access Control

Role-based access control (RBAC) [FK07; San96] has been mainly used in companies and organizations. The users are user accounts, and the subjects are the login sessions of the user accounts. Objects are often files and documents.

RBAC decides authorizations based on subjects' roles, such as manager, intern, doctor, and nurse. [San96] defines several levels of RBAC models. We will only discuss the base model, RBAC_0. The essential feature of RBAC_0 is two many-to-many relations. The first is a permission-to-role assignment relation, $PA \subseteq P \times R$, between a set of permissions P and a set of roles R; the second is a user-to-role assignment relation $UA \subseteq U \times R$, between a set of users U and a set of roles R. As mentioned earlier, the interpretation of a permission in [San96] is unclear. We adopt the definition discussed before (Definition 2.2), i.e., a permission is a pair (r, o) where r is an access right and o is an object. For example, $((\mathit{read}, o), \mathit{manager}) \in PA$ means managers can read the object o, and $(u, \mathit{manager}) \in UA$ means the user account u takes on the *manager* role. When a user account logs in, the login session can only take on roles that the user account already has. Whether a subject is authorized to access an object is determined by the PA relation. For example, if a subject has role *manager*, then it can read the object o mentioned earlier. Note that the symbols "*read*" and "*manager*" are meaningless until formally interpreted; in other words, "*read*" does not necessarily mean the access right of reading, and "*manager*" does not necessarily mean the manager role in other contexts.

2.2.4 Attribute-based Access Control

Attribute-based access control (ABAC) [Hu13; Jin12; Wan04; Zha05; JFC04] determines authorizations by evaluating properties, called **attributes**, of entities. The policies in an ABAC model are built upon attributes of entities, such as age and IP address. The current ABAC models follow the basic structure in Figure 2.1.

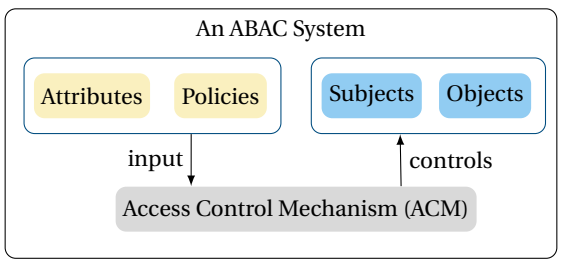

Figure 2.1 Basic Structure of Current ABAC Models

The main components of an ABAC model are attributes, policies, subjects, objects, and an access control mechanism (ACM). Depending on the model specifications, the kind of attributes that can be used in expressing policies varies. In ABAC_α [Jin12], only user, subject and object attributes are used in policies. While in ABAC_β [Jin14], attributes of the environment can also be used. We abstract away the minor difference by just including the attributes component as a whole in Figure 2.1. Similarly, the kind of policies that are defined in an ABAC model also varies. Some policy languages only include authorization policies such that if a request is not explicitly authorized by existing policies, the request is prohibited. Some policy languages include both authorization and prohibition policies, and thus some request may be both authorized and prohibited, or neither authorized nor prohibited. Furthermore, the definitions for subjects and objects in different ABAC models are also different. We follow the definitions provided in Definition 2.1.

The major difference between a traditional AC paradigm and ABAC can be viewed from two perspectives. The first perspective is from the authentication point of view: the traditional AC paradigms are identity-based, while ABAC is identity-less [Wan04]. In the DAC model that uses an AC matrix [Har76], when a subject s requests to read an entity e, once the identity of s is authenticated, the matrix is able to tell whether s is authorized to read e by finding the entry for the corresponding row and column. Similarly, in MAC, the security level of a subject is immediately available once its identity is authenticated, and the same works for roles in RBAC. While in an ABAC model, it is the attributes, instead of the identity, of an entity that need to be authenticated, although in some cases, of course, the identity needs to be authenticated as well. By focusing on authenticating attributes rather than identities [Oas13], ABAC is specially suited for problem domains where identities are unknown beforehand, such as collaborative organizations, and Internet applications and services. For instance, in a collaborative organization, an entity in an organization may not be known in another organization. For a social media, the identities of its users are often not, and cannot be completely authenticated. Whether a subject s_1 is authorized to read another subject s_2's posts would depend on the s_2's privacy settings and related attributes.

The other perspective is from the flexibility and granularity point of view. One may think of a traditional AC model as an ABAC model with restricted attributes. Thus, ABAC offers greater flexibility and granularity by allowing any domain-dependent attributes [Hu13]. For instance, a DAC model is an ABAC model with two attributes: one for ownership and the other for the shared access rights. For each subject, the ownership attribute indicates the entities the subject owns, and the sharing attribute indicates what access rights on what entities have been shared with the subject. When an entity e_1 requests to read an entity e_2, the policies determine whether to authorize the request by evaluating the values of e_1's ownership and sharing attributes. Similarly, a MAC model can be thought of as an ABAC model with a security label attribute defined on all entities. Finally, in RBAC, the many-to-many relation UA between user accounts and roles is essentially an attribute and the elements in the relation PA can be expressed in terms of policies. Specifically, for $(u_1, \mathit{manager}) \in UA$, the role attribute for u_1 has value *manager*. When u_1 logs in as a *manager*, the value for the login subject's role attribute is created correspondingly. For $((\mathit{read}, e), \mathit{manager}) \in PA$,

there is a corresponding policy stating that if the role attribute for a subject has value *manager*, then the subject is authorized to perform an action of type *read* on the entity e.

The major difference between the traditional AC paradigms and ABAC brings ABAC many advantages. ABAC is more suitable for applications where authorizations are characteristics-based instead of identity-based. ABAC is also more flexible and fine-grained than traditional AC paradigms. Furthermore, if an ABAC model is properly defined, it will be able to perform risk analysis by analyzing assurance levels of attributes [Hu13]. However, as we discussed before, these advantages also bring about multiple security and privacy issues, and current ABAC models have not taken the full advantages of ABAC.

2.3 Reflection

Reflection refers to the capacity for a model to reflect upon itself. The conceptual structure of a reflective model is depicted in Figure 2.2 [Cox05]. A reflective model can be viewed to be composed of two levels, a **base level** and a **meta-level**. [1] Monitoring occurs when information flows from the base level to the meta-level, while control occurs when information flows from the meta-level to the base level. In control, reflection uses the meta-level to update and modify the base-level. In monitoring, reflection informs the meta-level about the state of the base level, so that the meta-level can act accordingly.

What and how to control can be stated using **reflection principles**. Given a theory, a reflection principle decides what additional statements are implicitly acceptable in the theory. In [Fef62], it is stated that a reflection principle is "a description of a procedure for adding to any set of axioms A certain new axioms whose validity follow from the validity of the axioms A and which formally express, within the language of A, evident consequences of the assumption that all the theorems of A are valid."

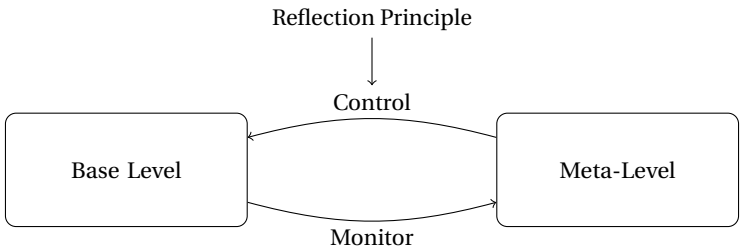

Figure 2.2 Conceptual Structure of A Reflective Model

Reflection has been used for various purposes. Kreisel et al. [KL68] describe a reflection principle for a system S as "the formal statement stating the soundness of S." They then define three reflection

[1] In the literature on reflection, the base level is often called the **object level**. We use "base level" to avoid confusion with the notion of objects in the access control literature.

principles: global, local, and uniform. The global reflection principle states full soundness. Thus, if the global reflection principle is accepted, the soundness of S is also implicitly accepted. Feferman [Fef91] defines a unary predicate T for truth. When extending the language $\mathscr{L}_{\mathsf{PA}}$ for Peano arithmetic with T and corresponding axioms, the resulting axiomatic theory $\mathsf{T}(\mathsf{PA})$ proves the soundness of Peano arithmetic, i.e., it proves the global reflection principle. Reflection has also been used to control the reasoning of default logic [All96]. Finally, [Bar00] uses reflection to formalize a knowledge representation paradigm, in which new knowledge can be derived using reflection principles given the knowledge of some domain. For example, suppose $\omega : \mathit{tell}(\phi^1, A^1)$ means "the agent ω tells agent ϕ that A," and $\phi : \mathit{told}(\omega^1, A^1)$ means "ϕ is told by ω that A." The connection between *tell* and *told* is formalized as a reflection principle which basically says if ω tells ϕ about A under some conditions, then under the same condition, A is told to ω by ϕ.

Reflection can be roughly categorized into two types: amalgamated or non-amalgamated, depending on the language(s) used for the base level and meta-level. The language used for the based level is called the **base language**, and the language for the meta-level is **metalanguage**. When the base language and the metalanguage are the same, we say they are **amalgamated**; otherwise, they are **non-amalgamated**. Among the examples we mentioned earlier, [KL68; Fef91; All96] use a non-amalgamated approach while [Bar00] is amalgamated. According to [Bar00], amalgamated approaches are more suitable for applications in knowledge representation, where expressing knowledge about a domain can be seen as properties of the domain, as well as properties of the knowledge itself.

CHAPTER

3

OVERVIEW OF REFABAC

The Reflective Attribute-based Access Control (RefABAC) paradigm improves upon and generalizes the traditional ABAC paradigm in three major aspects. First, any entity can have attributes. In most ABAC models, only subjects and objects have attributes. [1] By utilizing reflection, any entities, including policies, axioms and attributes, can be *reified* and treated in the same way as a concrete individual. Therefore, any entities can have attributes and they can also be used as values for other attributes. Second, a reified entity can also be modified, which allows RefABAC to define administrative actions as well as regular access control actions. Finally, RefABAC works for distributed settings. Given a connected network consisted of multiple systems, the RefABAC framework can define how the entities in different systems relate to each other and what happens if a subject in some system requests to access an entity in another system.

The basic structure of a RefABAC framework includes one (shared) conceptual model and at least one system model. The conceptual model defines the fundamental entities and properties that are shared among the systems, and each system model defines the domain-dependent entities and properties of the corresponding system. The entities and properties are specified using attributes, constraints, actions, transition rules and policies. Constraints are axioms for attributes. Transition rules describe the pre- and post-conditions of actions. Policies are authorization policies the decide whether a request should be authorized or not. Different conceptual models in different RefABAC frameworks may have different specifications. For instance, the conceptual model in one framework may regard all subjects as objects, while the conceptual model in another framework may require subjects and objects to be mutually exclusive.

This chapter provides an overview of the RefABAC paradigm to give the audience a high-level

[1] With a few exceptions, $ABAC_\beta$ [Jin14] allows meta-attributes, i.e., attributes about attributes.

picture of what it looks like. Then we formally discuss the details in the rest of the dissertation.

3.1 Structure of a RefABAC framework

3.1.1 Conceptual vs. System Models

Suppose there are multiple access control (AC) systems and their AC systems are related in one way or another. The RefABAC framework for the systems consists of one conceptual model and multiple system models, one for each of the AC systems. The **conceptual model** defines the fundamental individuals, axioms, policies, etc. that are shared among all of the AC systems. The **system model** for each AC system includes the domain-dependent entities as well as the ones defined in the shared conceptual model. Figure 3.1 illustrates an example RefABAC framework with m ($m \geq 1$) systems. Each system corresponds to a system model. The *same* conceptual model is shared among all of the m system models. One may think of the conceptual model as a common subset of all the system models.

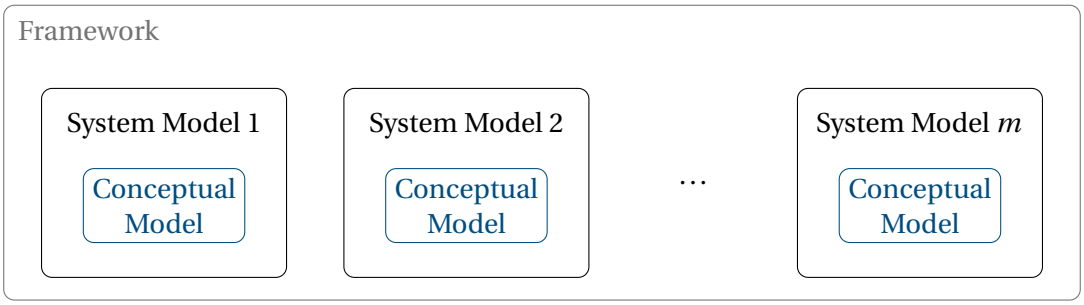

Figure 3.1 Basic RefABAC Framework Structure

Figure 3.2 shows the structure for the RefABAC framework we will be using to formalize Scenario 1.1. There are three organizations in the scenario: the care facility and two nearby hospitals that have doctors visiting the care facility. Each organization has one AC system, and thus one system model. The conceptual model is shared among all of the three system models. Example entities defined in the conceptual model include the types containing all subjects and objects respectively. The system model for the care facility defines a domain-dependent type containing the residents living in the facility. We will discuss more in Section 3.2 on what kinds of entities are defined in a model.

In some cases, one organization may have more than one AC systems. One can handle such cases by treating the multiple AC systems separately. Each of the AC systems still corresponds to a single system model. The system models can contain extra restrictions and requirements regarding how the entities in different systems relate to one another. For instance, suppose the engineering and human resource departments in a company use two different AC systems. The AC system for

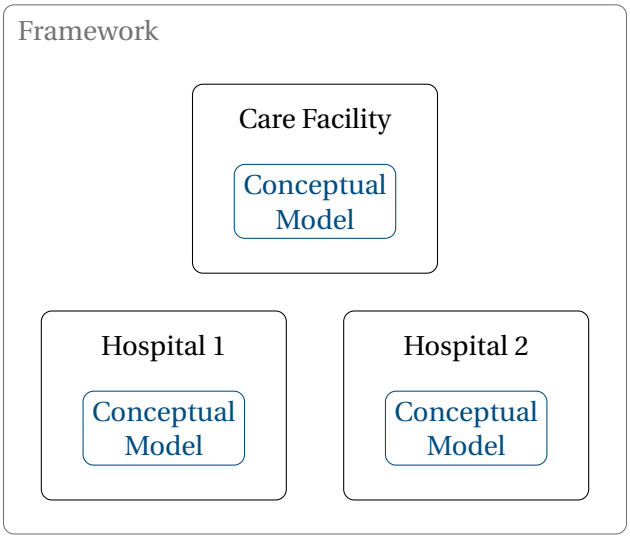

Figure 3.2 RefABAC Framework Structure for Scenario 1.1

the engineering department contains personnel documents for its employees, and it is required that every personnel document in the engineering department's system has a copy in the human resource department's system.

The framework also works when only a single system is involved, as shown in Figure 3.3. The conceptual model is contained in the only system model. In this case, one may choose not to distinguish between the conceptual and system models. However, defining a conceptual model makes it easier to adjust if further systems need to be added. Moreover, the conceptual model represents the minimal we can express in all the system models.

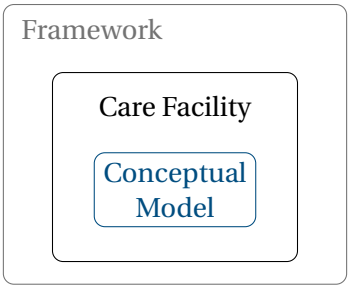

Figure 3.3 RefABAC Framework Structure For A Single System

3.1.2 System State

Given a RefABAC framework, its current state consists of the current **system states** of all system models in the framework. A system state specifies what entities *currently* exist in a system. It contains all the entities and properties defined in the system model, and more. For instance, a system state

for the care facility's system model includes an entity called Bob belonging to the type for residents, which means Bob is currently a resident at the time instance represented by the system state. Another system state that corresponds to another time instance might not contain the entity Bob or the fact that Bob is a resident at the facility.

One might think of the relation between a system model and one of its system states in various different ways. First, the system model can be thought of as a logical signature together with a knowledge base. The system state is a logical interpretation of the logical signature and it satisfies the axioms defined in the knowledge base. One may also think of the system model in description logics as a TBox and the system state as an ABox. Finally, in terms of database theory, the system model is a database schema and the system state is a database state.

3.2 Basic Concepts in RefABAC

We identify several important concepts that are present in many AC frameworks and include them as the basic building blocks in RefABAC. These concepts and their relationships are depicted in Figure 3.4. *Attribute instances*, which instantiate attributes, describe the facts stored in a system. *Constraints* define relationships among attributes. Attribute instances and constraints together constitute the *knowledge base* of a system. *Actions*, which instantiate action types, modify entities and update a system's states. Transition rules define the effects of actions. Finally, policies authorize or prohibit actions.

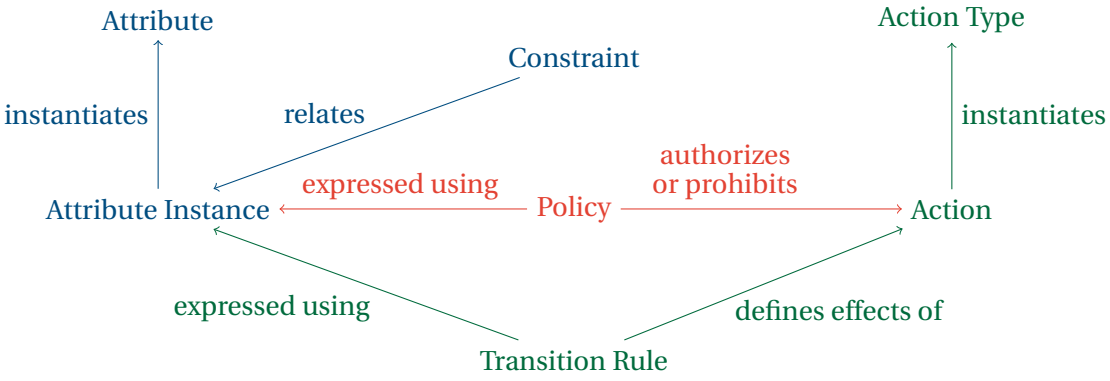

Figure 3.4 Basic Concepts and Their Relationships

3.2.1 Attributes and Constraints

An **attribute** describes a property of entities. In RefABAC, an attribute can be used to define a property of a concrete individual, e.g., the owner of a document, or of an abstract entity, e.g., the creation date and time of a policy. We also use attributes to categorize the entities in a system, such

as subjects, objects, policies and attributes. We call such an attribute a **type attribute** or a **type** for short.

We define several attributes in the conceptual model, called **conceptual attributes**. The conceptual attributes are essential for most AC paradigms and for developing the reflective AC frameworks. The conceptual attributes used as types include the essential ones that are common in most AC paradigms, e.g., subjects, objects, attributes, and policies. We also define several conceptual attributes for convenience and recording keeping. For instance, we define attributes to keep track of the requesting subject, the requested action, and the status—requested, granted, denied, and/or fulfilled—of each request. The domain-dependent attributes defined in a system model are called **system attributes**. For instance, the system model for a hospital defines doctors, nurses, and patients as subtypes of the *subject* conceptual attribute.

Constraints are axioms for attributes. Constraints can be used to define the required relationships among attributes and the properties of attributes. As with attributes, we distinguish between conceptual and system constraints. A **conceptual constraint** defines the relationships among and the properties of the conceptual attributes. Specifically, in the Core RefABAC framework, we have a conceptual constraint that says the types of subjects and objects are mutually exclusive. A **system constraint** defines the relationships and properties of system attributes. For instance, a constraint we use in defining the AC system for the care facility in Scenario 1.1 says that the type of residents is a sub-type of the conceptual type of subjects. Another constraint we have requires that every resident must have one of the visiting doctors as his or her responsible doctor.

The constraints and attribute values in a system state might not always be consistent. Although requiring some work, checking consistency is a standard yet undecidable procedure in mathematical logics. The system administrators are responsible for ensuring the consistency of a system state during the initial setup. Since we allow administrative actions in systems, a subject action might modify the entities in a system, resulting in an inconsistent system state. One way of handling the situation is disallowing all such actions. Another way is alerting the administrators and letting them make the final decision. Complicated systems may also require various methods in belief revision. However, as response time is a priority in access control, any kind of belief revision may take too much time and thus not the best option here.

3.2.2 Actions, Action Types and Transition Rules

An **action** is a specific operation performed by a subject. In RefABAC, we consider systems as active entities so that they are a special kind of subjects. For instance, reading an object is a subject action. Making a request is a subject action. Granting a request is a system action. We also categorize actions into **action types** such that each action instantiates its corresponding action type. In the conceptual models for the RefABAC frameworks we will be formalizing in the rest of dissertation, we define **conceptual action types** that are used to categorize the actions of a subject making, granting, and denying a request and of a policy authorizing and prohibiting a request, respectively. The domain-dependent **system action types** defined in each system model usually define the possible types of

AC and administrative actions in the system. For instance, in formalizing the system model for the care facility in Scenario 1.1, we define the common AC action types for a subject reading and writing an object, as well as the administrative action types for a subject adding a policy or modifying the value of some attribute.

An action type can have more than one transition rule. Suppose the system model for the care facility defines an action type of reading objects. One of the transition rules for the action type specifies that if the object to be read is the medical record of a resident at the facility, then the system must record the purpose of the action. Another transition rule for the action type specifies that if the object to be read is the insurance information of a resident, then the system does not need to record the purpose. The two transition rules both apply to the action type of reading objects. However, one may argue that in such cases, there should be two action types defined, one for reading medical record and the other for reading insurance information. We here do not argue which choice is better. In this dissertation, we assume each action may trigger multiple transition rules. In case where the post-conditions of the transition rules for some action type are in conflict, we would result in an inconsistent system state. As we discussed in the previous section on inconsistent constraints, standard yet undecidable procedures can be used.

3.2.3 Policies

A **policy** describes the conditions of whether a request should be authorized or prohibited. We distinguish the notions of a policy authorizing or prohibiting a request from the notions of a system granting or denying a request. Given a policy and a request such that the policy is **applicable** for the request, the policy either **authorizes** or **prohibits** the request. A request may be authorized and prohibited by different policies at the same time. It is up to the system to make the final decision, depending on the specifications of its system and conceptual models. In the Core RefABAC framework, we formalize a simple and straightforward method for resolving conflicts among policies. A system denies a request if it is prohibited by any policies. A system grants it if it is not prohibited by any policies and authorized by at least one policy.

3.3 General Workflow

How does a system act when a subject makes a request? The answer depends on many factors. From a subject making a subject to the subject performing the requested action (if the request is granted), the process involves many steps and each step involves many options. The basic steps of an AC system's workflow are shown in Figure 3.5. First, we assume some subjects have made requests to a system. We say a request is **active** if it is requested but has not been decided yet. Among the active requests, the system selects one of them to focus on. For the selected request, the system evaluates the policies and finds the applicable ones. Then the system decides whether to grant or deny the request using the applicable policies. Finally, the system enforces the decision. Specifically, if the request is granted, the system may enforce it by letting the requester perform the requested

action. Otherwise, if the request is denied, the requested action should not be performed. Additional logging might take place to record the decision.

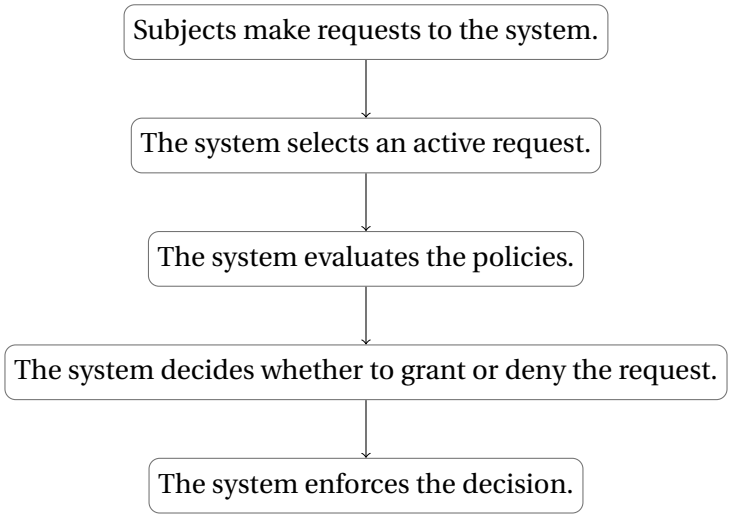

Figure 3.5 Basic Steps of an AC System's Workflow

What each step means and how each step should behave depends on the specific design of a system. One needs to consider various options for each step when designing an AC system. For example, if there are multiple active requests, the system needs a way to select one of them, such as randomly or by using a queue. We list some major issues that need to be considered when designing an AC system. Then we will discuss more on how we address them in the three RefABAC frameworks in the following chapters.

The first major issue is how to handle unknown information. A simple and fast approach is treating all missing information as logical false. Suppose a policy says that if a subject is over six years old, then he or she is authorized to watch the movie. If the system doesn't know the age of a subject, then it assumes the subject is not over six years old and thus does not authorize the subject to watch the movie. This is compliant with the **denial-by-default** principle, which specifies that if a request is not explicitly authorized, then it should be prohibited. Another approach might be to ask the subject or the administrators for the missing information.

Another major issue is how to handle conflicting policies. Given a request, multiple policies might be applicable for the request. If one applicable policy authorizes the request while another prohibits it, then we say the two policies are in conflict. One way to handle conflicting policies is to deny a request if it is prohibited by any policy. Many mechanisms for resolving conflicts among policies have been developed [Koc02; Ben03; SF16; Elk15]. We have proposed a mechanism [Ajm16] using dominance relations to resolve conflicts among *norms*, which are similar as authorization policies but with more general post-conditions. We will adopt the mechanism into RefABAC to resolve the conflicts among AC authorization policies.

When designing an AC system, one also needs to consider which steps need to be conducted at runtime and which can be conducted offline to reduce the running time overhead. We assume all the basic steps in Figure 3.5 are performed at runtime. While other tasks such as checking the consistency of the information in a system and formulating policies are performed offline.

3.4 Conclusion

We informally discussed the basic structure of and the main concepts in RefABAC. Given a set of systems, a RefABAC framework is composed of one conceptual model and multiple system models, one for each of the systems. The conceptual model defines the common and essential entities shared among the systems. Each system model defines the domain-dependent entities of its corresponding system. The major buildings blocks in a RefABAC model consist of attributes, attribute instances, constraints, action types, actions, transition rules, and policies. Informally, attributes instances, or attributions, are instances of attributes, and constraints describe the relationships among attributes. Actions are instances of action types, and transition rules describe the pre- and post-conditions of action types. Policies are authorization or prohibition policies. The general workflow of a RefABAC framework is first, a subject makes a request to a system to perform some action. Next, the applicable policies decide whether the request should be authorized or prohibited. The system to which the request is made makes the final decision on whether the request should be granted or denied. Finally, if the request is granted, the requesting subject may perform the requested action.

CHAPTER

4

LOGICAL LANGUAGE OF REFABAC

The representation of any system should at least be able to represent all the kinds of entities that will be considered by the system. In a RefABAC system, the basic kinds of entities are attributes, attribute instances, constraints, action types, actions, transition rules, and policies (Chapter 3). The logical language of RefABAC must be able to represent these kinds of entities. Moreover, new entities can be formed from existing entities. For instance, a new entity, specifically an attribute instance, can be formed by defining a value for some attribute of a policy, such as the creation date of the policy. We also require that the representation allow attributes to be defined for every entity. Not only subjects and objects but also policies and attributes themselves can have attributes.

We introduce the logical language used to formalize a RefABAC framework in this chapter. The logical language extends Dynamic Logic (DL) by adding referencing and dereferencing operators. We denote the logic by DL^+. We first review the syntax and semantics of DL (Section 4.1) and then define DL^+ (Section 4.2).

4.1 Dynamic Logic (DL)

DL [Pra76] is a first-order modal logic with actions. Each state is identified by the formulas contained in the state. An action allows one state to transition to another.

Conventionally, an action is called a program in dynamic logics, and an atomic action is called an atomic program to be consistent with programming terminology. In this dissertation, we use the terms action and atomic action instead to focus on the fact that the changes in a system are caused by explicit subject actions. Moreover, in common syntax of dynamic logics, an atomic action is usually an assignment statement of the form $x{:=}t$, where x is a variable and t is a term. In our

definition, we use $a(t_1,...,t_m)$ instead for readability and comprehensibility, where a is called an action type, and $t_1,...,t_m$ are terms. We use action types to classify actions.

We introduce DL in a similar manner as in first-order logic by first defining its logical signature.

Definition 4.1. A **logical signature** of DL is a tuple $\langle \mathscr{P}, \mathscr{A}, n \rangle$ where $\mathscr{P}$ is a countable set of **predicate symbols**, $\mathscr{A}$ is a countable set of **action type symbols** disjoint from $\mathscr{P}$, and $n : \mathscr{P} \cup \mathscr{A} \to \mathbb{N}$ is an **arity function** that assigns to each of the symbols a natural number.

4.1.1 Syntax of DL

The syntax for DL is defined in the following.

Definition 4.2. Given a logical signature $\langle \mathscr{P}, \mathscr{A}, n \rangle$ of DL, the syntax of terms, actions and formulas are

$$\begin{aligned} \text{term } t &::= x \mid \mathsf{c} \\ \text{action } \pi &::= a(t,...,t) \mid \pi;\pi \mid \pi \cup \pi \mid \pi^* \mid \phi? \\ \text{formula } \phi &::= \top \mid \bot \mid P(t,...,t) \mid \neg\phi \mid \phi \vee \phi \mid \forall x.\phi \mid [\pi]\phi, \end{aligned}$$

where x is a variable, c is a constant, $P \in \mathscr{P}$, and $a \in \mathscr{A}$. All variables in a formula must be bounded.

Additionally, we also have the abbreviations $\phi \wedge \phi'$ for $\neg(\neg\phi \vee \neg\phi')$, $\phi \to \phi'$ for $\neg\phi \vee \phi'$, $\exists x.\, \phi$ for $\neg\forall x.\, \neg\phi$, and $\langle\pi\rangle\phi$ for $\neg[\pi]\neg\phi$.

We use words beginning with capital-case letters for predicate symbols, e.g., *Subject*; ϕ, ψ for formulas. We use words beginning with lower-case letters for action type symbols and terms. We use π for actions. Among the lower-case letters, we use the *italics* font for action type symbols and variables, and the `typewriter` font for constants. Subscripts and superscripts may be added as well. For example, we define an action type *makesRequest* and a subject `bob` to formalize Scenario 1.1.

An action $\pi_1;\pi_2$ means a serial performance of actions π_1 and π_2, i.e., do π_1 then do π_2. An action $\pi_1 \cup \pi_2$ means nondeterministically performing one of the actions π_1 and π_2. An action π^* means π is performed finitely, but nondeterministically determined, number of times. Finally, an action $\phi?$ means if ϕ is true, then proceed; otherwise, fail. Formulas are Boolean combinations of predicates, similar to predicate logic. Additionally, $[\pi]\phi$ is also a formula, meaning that after π is performed, it is necessary that ϕ is true. However, π might not terminate.

Notice that we require all variables in a formula be bounded. Therefore, given a set of individuals Δ and an action type symbol a of arity n, we are able to get all possible atomic actions of a by using the individuals in Δ as arguments for a. Similarly, given a predicate symbol P with arity n, we can get all the atomic testing actions for P by using the individuals in Δ as arguments for P. Following the tradition of dynamic logics, we denote by Π_0 the set of all atomic actions, and Π the set of all actions, which is constructed by applying the action operators ;, $\cup$, * and ? to the atomic actions in Π_0.

More conventional programming constructs can be defined from the actions in Definition 4.2:

$$\texttt{skip} = \top?$$
$$\texttt{fail} = \bot?$$
$$\texttt{if } \phi \texttt{ then } \pi_1 \texttt{ else } \pi_2 \texttt{ fi} = \phi?;\pi_1 \cup \neg\phi?;\pi_2$$
$$\texttt{while } \phi \texttt{ do } \pi \texttt{ od} = (\phi?;\pi)^*;\neg\phi?.$$

The `if - then - else` construct is called a **condition test**, and the last one is called a **while loop**. Dynamic logics also define several typical classes of actions [Lee90]. The class R of **regular actions** contains all actions formed from atomic actions, $\cup$, ;, *, and ?, such that in a test $\phi?$ the formula ϕ must be quantifier-free. The class *DWP* of **deterministic `while` programs** is the subclass of R in which $\cup$, ?, and * only appear in `skip`, `fail`, conditional tests, and while loops.

4.1.2 Semantics of DL

A look-up table is needed in defining the semantics for DL.

Definition 4.3. Given a universe Δ of individuals, a **look-up table** for Δ is a function $\ell : \mathscr{V} \to \Delta$ from the set of variables $\mathscr{V}$ to Δ. We denote by $\ell[x \mapsto c]$ the look-up table which maps x to c and any other variable y to $\ell(y)$.

We use a **labeling function** that maps each state to a first-order interpretation. In the first-order interpretation of a state, each predicate symbol corresponds to the set of tuples of individuals which make the predicate true in the state, as with the semantics for first-order logic. The following is the formal definition, where Π_0 denotes the set of all atomic actions and Π the set of all actions.

Definition 4.4. Let $\langle \mathscr{P}, \mathscr{A}, n \rangle$ be a logical signature of DL. An **interpretation** of $\langle \mathscr{P}, \mathscr{A}, n \rangle$ is a Labeled Transition System (LTS) $\mathscr{M} = \langle \Sigma, \Delta, \Rightarrow, L \rangle$ where

- Σ is a nonempty set of states;
- Δ is a nonempty set of individuals;
- $\Rightarrow : \Pi \to 2^{\Sigma \times \Sigma}$ is a mapping from the set Π of all actions into a set of partial functions in Σ such that [1]
  - $\sigma_1 \overset{\pi_1;\pi_2}{\Longrightarrow} \sigma_2$ iff there exists $\sigma \in \Sigma$ such that $\sigma_1 \overset{\pi_1}{\Rightarrow} \sigma$ and $\sigma \overset{\pi_2}{\Rightarrow} \sigma_2$
  - $\sigma_1 \overset{\pi_1 \cup \pi_2}{\Longrightarrow} \sigma_2$ iff $\sigma_1 \overset{\pi_1}{\Rightarrow} \sigma_2$ or $\sigma_1 \overset{\pi_2}{\Rightarrow} \sigma_2$
  - $\sigma_1 \overset{\pi^*}{\Rightarrow} \sigma_2$ iff there exists a non-negative integer u and states $x_0, ..., x_u$ such that $x_0 = \sigma_1$, $x_u = \sigma_2$, and for all $k = 1, ..., u$, $x_{k-1} \overset{\pi}{\Rightarrow} x_k$;
  - $\sigma_1 \overset{\phi?}{\Rightarrow} \sigma_2$ iff $\sigma_1 = \sigma_2$ and $\mathscr{M}, \sigma_2 \models^{\ell} \phi$.

[1] Notation: Given an action $\pi \in \Pi$ which maps a state σ to another σ', the standard notation is $(\sigma, \sigma') \in \Rightarrow(\pi)$, where $\Rightarrow(\pi)$ returns a set of pairs of states. We abbreviate it as $\sigma \overset{\pi}{\Rightarrow} \sigma'$.

- L is a labeling function that maps each $\sigma \in \Sigma$ to a first-order interpretation $L(\sigma) = \langle \Delta, \cdot^{L(\sigma)} \rangle$, where $\cdot^{L(\sigma)}$ is a function that assigns each predicate symbol $P \in \mathscr{P}$ with arity $n > 0$ a subset of Δ^n, i.e., $P^{L(\sigma)} \subseteq \Delta^n$.

Following the above definition for a DL interpretation, given a LTS $\mathscr{M} = \langle \Sigma, \stackrel{\cdot}{\Rightarrow} V \rangle$, a state $\sigma \in \Sigma$, a formula ϕ and a look-up table $\ell : \mathscr{V} \to \Delta$, the satisfiability relation $\mathscr{M}, \sigma \models \phi$ is defined as:

- $\mathscr{M}, \sigma \models^{\ell} P(t_1, ..., t_n)$ iff $(t_1'^{L(\sigma)}, ..., t_n'^{L(\sigma)}) \in P^{L(\sigma)}$, where $t_1', ..., t_n'$ are the terms after replacing all variables with their values according to ℓ
- $\mathscr{M}, \sigma \models^{\ell} \neg\phi$ iff $\mathscr{M}, \sigma \not\models^{\ell} \phi$
- $\mathscr{M}, \sigma \models^{\ell} \phi \wedge \phi'$ iff $\mathscr{M}, \sigma \models^{\ell} \phi$ and $\mathscr{M}, \sigma \models^{\ell} \phi'$
- $\mathscr{M}, \sigma \models^{\ell} [\pi]\phi$ iff for all $\sigma' \in \Sigma$, if $\sigma \to_{\pi} \sigma'$, then $\mathscr{M}, \sigma' \models^{\ell} \phi$
- $\mathscr{M}, \sigma \models^{\ell} \forall x.\, \phi$ iff $\mathscr{M}, \sigma \models^{\ell[x \mapsto c]} \phi$ for all $c \in \Delta$
- $\mathscr{M}, \sigma \models^{\ell} \exists x.\, \phi$ iff $\mathscr{M}, \sigma \models^{\ell[x \mapsto c]} \phi$ for some $c \in \Delta$.

Some of the example valid formulas in DL are

- $[\pi_1; \pi_2]\phi \longleftrightarrow [\pi_1][\pi_2]\phi$
- $[\pi_1 \cup \pi_2]\phi \longleftrightarrow [\pi_1]\phi \wedge [\pi_2]\phi$
- $[\pi^*]\phi \longleftrightarrow \phi \wedge [\pi][\pi^*]\phi$
- $[\phi_1?]\phi_2 \longleftrightarrow (\phi_1 \to \phi_2)$

4.2 DL$^+$

DL$^+$ extends DL by adding a referencing operator $\uparrow$ and a dereferencing operator $\downarrow$. Given a formula ϕ or an action $a(t_1, ..., t_n)$, $\uparrow\phi$ or $\uparrow a(t_1, ..., t_n)$ returns an individual that represents the **name** of the formula or action, respectively. Given a name $\mathtt{n}$, $\downarrow\mathtt{n}$ returns the corresponding entity e, i.e., $\downarrow\uparrow e = e$.

4.2.1 Syntax of DL$^+$

A logical signature in DL$^+$ is still identified as a tuple $\langle \mathscr{P}, \mathscr{A}, n \rangle$, where $\mathscr{P}$ is a countable set of predicate symbols, $\mathscr{A}$ is a countable set of action symbols, and $n : \mathscr{P} \cup \mathscr{A} \to \mathbb{N}$ is an arity function. The syntax of DL$^+$ is defined formally as follows.

Definition 4.5. Given a logical signature $\langle \mathscr{P}, \mathscr{A}, n \rangle$ of DL$^+$, the syntax is defined as

$$\begin{aligned}
\text{term } t &::= x \mid \mathtt{c} \mid \mathtt{n} \\
\text{name } \mathtt{n} &::= \mathtt{an} \mid \mathtt{fn} \\
\text{action name } \mathtt{an} &::= \uparrow \pi \\
\text{formula name } \mathtt{fn} &::= \uparrow \phi \\
\text{action } \pi &::= a(t, ..., t) \mid \pi; \pi \mid \pi \cup \pi \mid \pi^* \mid \phi? \mid \downarrow \mathtt{an} \\
\text{formula } \phi &::= P(t, ..., t) \mid \neg\phi \mid \phi \vee \phi \mid \forall x.\ \phi \mid [\pi]\phi \mid \downarrow \mathtt{fn},
\end{aligned}$$

where x is a variable, c is a constant, $P \in \mathscr{P}$, and $a \in \mathscr{A}$. All variables in a formula must be quantified.

Similar to DL, we also have the abbreviations $\phi \wedge \phi'$ for $\neg(\neg\phi \vee \neg\phi')$, $\phi \rightarrow \phi'$ for $\neg\phi \vee \phi'$, $\exists x.\ \phi$ for $\neg\forall x.\ \neg\phi$, and $\langle\pi\rangle\phi$ for $\neg[\pi]\neg\phi$. We use words beginning with capital-case letters for predicate symbols, e.g., *Subject*; ϕ, ψ for formulas. We use words beginning with lower-case letters for action type symbols and terms. We use π for actions. Among the lower-case letters, we use the *italics* font for action type symbols and variables, and the `typewriter` font for constants. Subscripts and superscripts may be added as well. For example, we define an action type *makesRequest* and a subject `bob` to formalize Scenario 1.1.

The meaning of each operator on actions are the same as in DL. Specifically, $\pi_1; \pi_2$ is for serial execution; $\pi_1 \cup \pi_2$ is for nondeterministic selection; π^* is for repetition; and $\phi?$ is for testing. Finally, the set of all atomic actions Π_0 and the set of all actions Π are constructed in the same way, i.e., by using individuals from Δ as arguments for each $a \in \mathscr{A}$ and $P \in \mathscr{P}$.

The **referencing** and **dereferencing** operators $\uparrow$ and $\downarrow$ behave similarly to functions. Given an action π, $\uparrow\pi$ returns an individual called the **name** of the action π, or the **action name** of π for short, such that $\downarrow\uparrow\pi$ returns the original action π, i.e., $\downarrow\uparrow\pi = \pi$. Similarly, given a formula ϕ, $\uparrow\phi$ returns an individual called the **name** of ϕ or the **formula name** of ϕ for short, such that $\downarrow\uparrow\phi = \phi$. For simplicity, we assume that all systems in a RefABAC formalization use the same pair of such naming functions.

4.2.2 Semantics of DL$^+$

An interpretation of DL$^+$ is defined as follows.

Definition 4.6. Let $\langle \mathscr{P}, \mathscr{A}, n \rangle$ be a logical signature of DL$^+$. An **interpretation** of $\langle \mathscr{P}, \mathscr{A}, n \rangle$ is a Labeled Transition System (LTS) $\mathscr{M} = \langle \Sigma, \Delta, \Rightarrow, L \rangle$ where

- Σ is a nonempty set of states;
- Δ is a nonempty set of individuals;
- $\Rightarrow: \Pi \rightarrow 2^{\Sigma \times \Sigma}$ is a mapping from the set Π of all actions into a set of pairs of states in Σ such

that [2]

- $\sigma_1 \xRightarrow{\pi_1;\pi_2} \sigma_2$ iff there exists $\sigma \in \Sigma$ such that $\sigma_1 \overset{\pi_1}{\Rightarrow} \sigma$ and $\sigma \overset{\pi_2}{\Rightarrow} \sigma_2$
- $\sigma_1 \xRightarrow{\pi_1 \cup \pi_2} \sigma_2$ iff $\sigma_1 \overset{\pi_1}{\Rightarrow} \sigma_2$ or $\sigma_1 \overset{\pi_2}{\Rightarrow} \sigma_2$
- $\sigma_1 \overset{\pi^*}{\Rightarrow} \sigma_2$ iff there exists a non-negative integer n and states $x_0, ..., x_n$ such that $x_0 = \sigma_1$, $x_n = \sigma_2$, and for all $k = 1, ..., n$, $x_{k-1} \overset{\pi}{\Rightarrow} x_k$;
- $\sigma_1 \overset{\phi?}{\Rightarrow} \sigma_2$ iff $\sigma_1 = \sigma_2$ and $\mathscr{M}, \sigma_2 \models^\ell \phi$
- $\sigma_1 \xRightarrow{\downarrow \mathtt{an}} \sigma_2$ iff $\sigma_1 \overset{\pi}{\Rightarrow} \sigma_2$ where $\uparrow \pi = \mathtt{an}$.

- L is a labeling function that maps each $\sigma \in \Sigma$ to a first-order interpretation $L(\sigma) = \langle \Delta, \cdot^{L(\sigma)} \rangle$, where $\cdot^{L(\sigma)}$ is a function that assigns each predicate symbol $P \in \mathscr{P}$ with arity $n > 0$ a subset of Δ^n, i.e., $P^{L(\sigma)} \subseteq \Delta^n$.

Finally, given a state $\sigma \in \Sigma$, and a look-up table $\ell : \mathscr{V} \to \Delta$, the satisfaction relation $\models^\ell$ is defined as:

- $\mathscr{M}, \sigma \models^\ell P(t_1, ..., t_n)$ iff $(t_1'^{L(\sigma)}, ..., t_n'^{L(\sigma)}) \in P^{L(\sigma)}$, where $t_1', ..., t_n'$ are the terms after replacing all variables with their values according to ℓ
- $\mathscr{M}, \sigma \models^\ell \neg\phi$ iff $\mathscr{M}, \sigma \not\models^\ell \phi$
- $\mathscr{M}, \sigma \models^\ell \phi \wedge \phi'$ iff $\mathscr{M}, \sigma \models^\ell \phi$ and $\mathscr{M}, \sigma \models^\ell \phi'$
- $\mathscr{M}, \sigma \models^\ell [\pi]\phi$ iff for all $\sigma' \in \Sigma$, if $\sigma \to_\pi \sigma'$, then $\mathscr{M}, \sigma' \models^\ell \phi$
- $\mathscr{M}, \sigma \models^\ell \forall x.\, \phi$ iff $\mathscr{M}, \sigma \models^{\ell[x \mapsto c]} \phi$ for all $c \in \Delta$
- $\mathscr{M}, \sigma \models^\ell \exists x.\, \phi$ iff $\mathscr{M}, \sigma \models^{\ell[x \mapsto c]} \phi$ for some $c \in \Delta$
- $\mathscr{M}, \sigma \models^\ell \downarrow \mathtt{fn}$ iff $\mathscr{M}, \sigma \models^\ell \phi$ where $\uparrow \phi = \mathtt{fn}$.

[2] Notation: Given an action $\pi \in \Pi$ which maps a state σ to another σ', the standard notation is $(\sigma, \sigma') \in \Rightarrow(\pi)$, where $\Rightarrow(\pi)$ returns a set of pairs of states. We abbreviate it as $\sigma \overset{\pi}{\Rightarrow} \sigma'$.

CHAPTER

5

CORE REFABAC FRAMEWORK

In Chapter 3, we discussed the structure of a RefABAC model. Given a set of m ($m \geq 1$) systems, a RefABAC model consists of one conceptual model and m system models, one for each of the n systems. The conceptual model contains the common entities that are shared among the system models. Each system model contains the domain-dependent entities of its system. In Chapter 4, we defined the logical language DL^+ used to formalize a RefABAC model.

In this chapter, we define a baseline RefABAC framework called **Core RefABAC**. Core RefABAC defines the essential components one might often find in many current AC models, such as the types used to categorize entities. We use a simple denial-by-default approach in Core RefABAC. Given a request to access an entity, if the request is prohibited or not authorized by any policies, the request is denied and the requested action cannot be performed.

The rest of the chapter is organized as follows. We first define the conceptual model of Core RefABAC (Section 5.1), including the attributes, constraints, action types, and transition rules contained in the conceptual model. We also discuss in detail the decision-making procedure of a Core RefABAC model. Next, we formalize the basic types of entities in a system model in Core RefABAC, especially the constraints, transition rules and policies (Section 5.2). Finally, we discuss what entities are included in a system state given the system's model specification (Section 5.4). We demonstrate the usage of Core RefABAC by formalizing various examples from Scenario 1.1.

5.1 Formalism of Conceptual Model in Core RefABAC

The logical structure of the conceptual model [1] for Core RefABAC is defined in Definition 5.1. We use $\mathscr{C}$ as a shorthand for the conceptual model while `cm` as a constant in the logical formalism.

Definition 5.1. The **logical structure of the Core RefABAC (shared) conceptual model** $\mathscr{C}$ is a tuple $\langle \mathscr{C}_{Attr}, \mathscr{C}_C, \mathscr{C}_{act}, \mathscr{C}_T \rangle$, where

- $\mathscr{C}_{Attr}$ is a finite set of attributes, called **conceptual attributes**;
- $\mathscr{C}_C$ is a finite set of constraints, called **conceptual constraints**;
- $\mathscr{C}_{act}$ is a finite set of action types, called **conceptual action types**; and
- $\mathscr{C}_T$ is a finite set of transition rules, called **conceptual transition rules**.

The conceptual model defines the basic attributes, constraints, action types, and transition rules shared among Core RefABAC system models. Note that the Core RefABAC conceptual model does not contain any policies. We formally define each of the components in the rest of the section.

5.1.1 Conceptual Attributes

The conceptual attributes 1) define the common types of entities in a RefABAC framework, and 2) describe the essential properties of entities. Given the conceptual model $\mathscr{C} = \langle \mathscr{C}_{Attr}, \mathscr{C}_C, \mathscr{C}_{act}, \mathscr{C}_T \rangle$ in Core RefABAC, the set of conceptual attributes $\mathscr{C}_{Attr}$ is listed in Table 5.1. We categorize the conceptual attributes for readability and convenience.

Table 5.1 Core RefABAC Conceptual Attributes

Category Name	List of Attributes
Common Types (of arity 1)	*System*, *Subject*, *Object*, *Attribute*, *AttributeInst*, *Policy*, *Constraint*, *Request*, *ActionType*, *Action*, *TransitionRule*
Request Parameters (of arity 2)	*Requester*, *RequestedAct*, *AuthorizedBy*, *ProhibitedBy*
Request Status (of arity 1)	*Requested*, *Granted*, *Denied*, *Fulfilled*, *Active*, *Authorized*, *Prohibited*
Action Status (of arity 1)	*Performed*
Other Properties (of arity 2)	*TransitionOf* , *InSys*

[1] Although every Core RefABAC model contains a conceptual model, the definitions for the conceptual models are the same. So we say *the* conceptual model rather than *a* conceptual model in Core RefABAC.

5.1.1.1 Common Types

The first category of conceptual attributes defines the common types of entities. They are represented as unary predicate symbols. They include all of the basic types of entities we have discussed so far. Specifically, *Subject* and *Object* are types for subjects and objects, i.e., $\mathit{Subject}(x)$ or $\mathit{Object}(x)$ indicates x is a subject or object, respectively. *Attribute* is for attributes. Since an attribute is represented as a predicate symbol, we use the name of the attribute as an argument for *Attribute*. We write $\mathit{Attribute}(\uparrow \mathit{Attr})$ to denote *Attr* is an attribute. The rest of the common types are used in the same way. Specifically,

$$\mathit{System}(x), \mathit{Request}(x), \mathit{AttributeInst}(x), \mathit{Policy}(x), \mathit{Constraint}(x),$$
$$\mathit{ActionType}(x), \mathit{Action}(x), \text{ or } \mathit{TransitionRule}(x)$$

means x is a system or a request, or the name of an attribute instance, a policy, a constraint, an action type, an action, or a transition rule, respectively. Moreover, *System* is a subtype of *Subject* such that a system is also able to perform actions, although they may be subject to different transition rules and policies than the "regular" subjects, depending on the system models.

5.1.1.2 Request Parameters

The second category of conceptual attributes records the parameters of a request. $\mathit{Requester}(r, s)$ means the requesting subject of request r is s. $\mathit{RequestedAct}(r, n)$ means the requested action of r is the entity with name n. For convenience, we often say *the requested action* of r is n, instead of *the name of the requested action* of r is n. The meaning should be clear from context. For example, `bob` make a request `req` for himself to read document `carolMedRec`, carol's medical record. The action for bob reading the document is represented as $\mathit{read}(\texttt{bob}, \texttt{carolMedRec})$. Thus, `bob` is the requester of `req`, i.e., $\mathit{Requester}(\texttt{req}, \texttt{bob})$, and $\mathit{read}(\texttt{bob}, \texttt{carolMedRec})$ is the requested action of `req`, i.e., $\mathit{RequestedAct}(\texttt{req}, \uparrow \mathit{read}(\texttt{bob}, \texttt{carolMedRec}))$. Given the name `p` of a policy ϕ, $\mathit{AuthorizedBy}(r, \texttt{p})$ (respectively, $\mathit{ProhibitedBy}(r, \texttt{p})$) means a request r is authorized (respectively, prohibited) by the policy ϕ corresponding to `p`.

5.1.1.3 Request Status

The attributes for request status are unary predicate symbols. Figure 5.1 summarizes the status update process given request r.

Initially, request r has not been requested and it is not active. When a subject makes the request r, it becomes requested and active, i.e., $\mathit{Requested}(r) \wedge \mathit{Active}(r)$. Next, the policies are being evaluated and they decide whether r is authorized or prohibited. If the request does not have any applicable policies, the request is neither authorized nor prohibited. Finally, the system decides whether to grant or deny the request based on the policy decisions, i.e., the request becomes either $\mathit{Granted}(r)$ or $\mathit{Denied}(r)$. Once a decision has been made, the request becomes inactive, i.e., $\neg \mathit{Active}(r)$. In Core

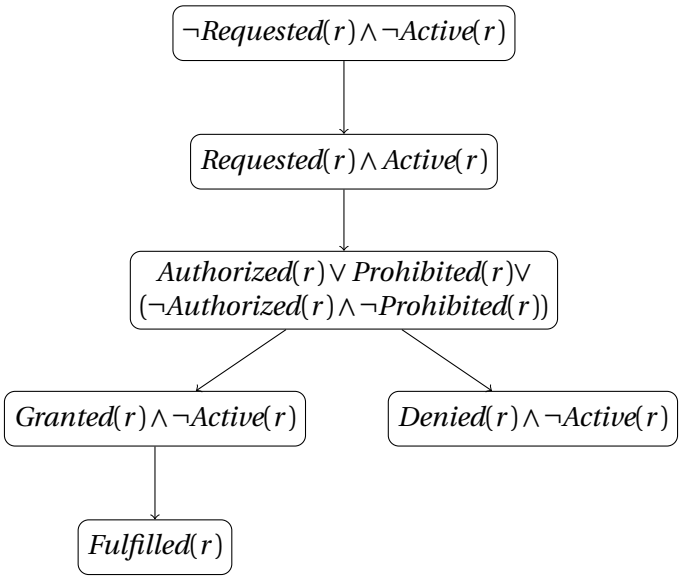

Figure 5.1 Request Status Update in Core RefABAC

RefABAC, a request cannot be both granted and denied, nor can it be neither granted or denied. If r is denied, then the system selects another active request to evaluate. On the other hand, if r is granted, the requested action can be performed, and we say r is fulfilled, i.e., *Fulfilled*(r).

5.1.1.4 Action Status

The conceptual attribute *Performed* for action status is to indicate whether an action has been performed. It can be also considered as a subtype of *Action*. For instance, *Action*$(\uparrow$*readObj*$(s,o))$ means *readObj*(s,o) is an action, or an entity of type *Action* in the formalism. Only after the subject s performs the action by reading the object o, the action is performed and *Performed*$(\uparrow$*readObj*$(s,o))$ becomes true. [2]

5.1.1.5 Other Properties

Finally, we have two attributes to indicate the transition rule for each action type and to which system each entity belongs. *TransitionOf*$(\uparrow\phi,\uparrow at)$ means ϕ is a transition rule of the action type *at*. Again, we use the names of the transition rule and of the action type as arguments.

InSys(e,sys) means the entity e is in the system *sys*, where e is a term. For example, Carol is a subject in the care facility. The system of the care facility stores the fact using attribute instance *Subject*(`carol`). Suppose the nearby hospital $\texttt{hospital}_1$ needs this piece of information to communicate with the care facility. The hospital contains attribute instances *InSys*(`carol`, `careFacility`) to indicate that Carol is an individual in the care facility and *InSys*($\uparrow$*Subject*(`carol`), `careFacility`) to indicate that the fact "Carol is a subject" is stored in the care facility.

[2] We came up with the conceptual attribute when translating the formalism to SQL. To indicate that an action has actually happened in the database, the action needs to be inserted into the table for the *Performed* attribute.

5.1.2 Conceptual Constraints

Conceptual constraints define relationship among the conceptual attributes. We categorize conceptual constraints by their functionality: (1) subtype relations; (2) types of arguments; (3) disjointness relations; (4) number restrictions of the arguments; and (5) other more complicated relations. We follow description logics' naming conventions for the categorization of conceptual constraints. Readers may notice the similarity between RefABAC's formalism and description logics. We use the notions of attributes and constraints while description logics use concepts and terminological axioms.

5.1.2.1 Subtype Relations

Given two type attributes $Type_1$ and $Type_2$, a constraint that expresses a subtype relation between the two attributes is in the form of

$$\forall x.\ Type_1(x) \rightarrow Type_2(x),$$

meaning every entity of type $Type_1$ is also of type $Type_2$.

Table 5.2 Core RefABAC Conceptual Constraints for Subtype relations 2

$\forall s.\ System(s) \rightarrow Subject(s)$
$\forall a.\ Performed(a) \rightarrow Action(a)$
$\forall r.\ Fulfilled(r) \rightarrow Granted(r)$
$\forall r.\ Active(r) \rightarrow Requested(r)$
$\forall r.\ Requested(r) \rightarrow Request(r)$
$\forall r.\ Granted(r) \rightarrow Request(r)$
$\forall r.\ Denied(r) \rightarrow Request(r)$
$\forall r.\ Fulfilled(r) \rightarrow Request(r)$
$\forall r.\ Authorized(r) \rightarrow Request(r)$
$\forall r.\ Prohibited(r) \rightarrow Request(r)$

Table 5.2 lists the conceptual constraints for defining subtype relations among conceptual type attributes. The first equation in the table expresses that every system is a subject. In Core RefABAC, a system can also perform AC actions and make a request. So we regard them as a special subtype of subjects. The second equation means that performed actions are actions. The third equation says every fulfilled request must have been granted. The fourth one says if a request is active, then it must have been requested. The rest of the equations in the table simply make sure that any entity that is in the status of requested, granted, denied, fulfilled, active, authorized, or prohibited must be a request.

5.1.2.2 Argument Types

Given an n-ary attribute *Attr*, a constraint that expresses the types of arguments of *Attr* is generally in the form of

$$\forall x_1, \ldots, x_n.\ Attr(x_1, \ldots, x_n) \rightarrow Type_1(x_1) \wedge \ldots \wedge Type_n(x_n),$$

where each $Type_i$ is a type attribute. The constraint may be more complicated where each $Type_i$ is a Boolean combination of type attributes.

Table 5.3 Core RefABAC Conceptual Constraints for Argument Types

$\forall r, s.\ Requester(r, s) \rightarrow Request(r) \wedge Subject(s)$
$\forall r, a.\ RequestedAct(r, a) \rightarrow Request(r) \wedge Action(a)$
$\forall r, p.\ AuthorizedBy(r, p) \rightarrow Request(r) \wedge Policy(p)$
$\forall r, p.\ ProhibitedBy(r, p) \rightarrow Request(r) \wedge Policy(p)$
$\forall t, at.\ TransitionOf(t, at) \rightarrow TransitionRule(t) \wedge ActionType(at)$
$\forall x, s.\ InSys(x, s) \rightarrow System(s) \vee s = \texttt{cm}$

Table 5.3 lists the conceptual constraints that are used to define the types of arguments of each conceptual attribute. The first equation means that the type of the attribute *Requester*'s first argument is *Request* and the type of its second argument is *Subject*. In other words, the requester of a request must be a subject. Recall that a system is a subject as well. The second equation says the requested action of a requester is an action. The third and fourth ones specify that if r is authorized or prohibited by p, then r is a request and p is a policy, respectively. The next one says that if t is a transition rule of at, i.e., $TransitionOf(t, at)$, then t is (the name of) a transition rule and at is (the name of) an action type. Finally, if x is in s, i.e., $InSys(x, s)$, then s must be either a system or the conceptual model `cm`.

5.1.2.3 Disjointness Relations

The third category of conceptual constraints is to express the disjointness relations among the conceptual attributes. The basic form of such a constraint is

$$\forall \vec{x}.\ Attr_1(\vec{x}) \rightarrow \neg Attr_2(\vec{x}),$$

where $Attr_1$ and $Attr_2$ are attributes. Every pair of the conceptual attributes for types of entities, except for *System* and *Subject*, is disjoint. We list the conceptual constraints that express the disjoint relations between *Subject* and other conceptual attributes in Table 5.4. We here omit the conceptual constraints that express disjointness between other type attributes.

We also define the disjointness relations among the status of requests in Table 5.5. First of all, a

Table 5.4 Core RefABAC Conceptual Constraints for Disjointness Between *Subject* and Other Types

$\forall x.\ Subject(x) \rightarrow \neg Object(x)$
$\forall x.\ Subject(x) \rightarrow \neg Attribute(x)$
$\forall x.\ Subject(x) \rightarrow \neg AttributeInst(x)$
$\forall x.\ Subject(x) \rightarrow \neg Policy(x)$
$\forall x.\ Subject(x) \rightarrow \neg Constraint(x)$
$\forall x.\ Subject(x) \rightarrow \neg Request(x)$
$\forall x.\ Subject(x) \rightarrow \neg ActionType(x)$
$\forall x.\ Subject(x) \rightarrow \neg Action(x)$
$\forall x.\ Subject(x) \rightarrow \neg TransitionRule(x)$

request cannot be both granted and denied. Next, since a request is no longer active after a system decides whether to grant or deny it, a request cannot be both granted and active or be both denied and active. The last equation can be proved from the conceptual transition rules for granting a request. If a request is prohibited by any policies, then the system cannot grant it. Thus, a request cannot be both prohibited (by a policy) and granted (by a system).

Table 5.5 Core RefABAC Conceptual Constraints for Disjointness Among Request Status

$\forall x.\ Granted(x) \rightarrow \neg Denied(x)$
$\forall x.\ Granted(x) \rightarrow \neg Active(x)$
$\forall x.\ Denied(x) \rightarrow \neg Active(x)$
$\forall x.\ Prohibited(x) \rightarrow \neg Granted(x)$

5.1.2.4 Number Restrictions

A conceptual constraint for expressing a number-restriction relation defines how many entities should be allowed for some argument of an attribute. The number restrictions for the conceptual attributes are listed in Table 5.6. We divide them into two categories: there exists *at least one* or *at most one* entity for a particular argument.

In summary, any requested request must have at least and at most one requester and one requested action. In other words, a requested request must have *exactly one* requester and one requested action. The last equation in the at-least-one category says that if an entity is not a system, then it must belong to some system. A system s' might be a sub-system of another system s but s itself does not necessarily belong to any other systems. We add this constraint to avoid an infinite

Table 5.6 Core RefABAC Conceptual Constraints For Number Restrictions

At Least One
$\forall r.\ \mathit{Requested}(r) \rightarrow \exists s.\ \mathit{Requester}(r, s)$
$\forall r.\ \mathit{Requested}(r) \rightarrow \exists a.\ \mathit{RequestedAct}(r, a)$
$\forall at.\ \mathit{ActionType}(at) \rightarrow \exists t.\ \mathit{TransitionOf}(t, at)$
$\forall x.\ \neg \mathit{System}(x) \rightarrow \exists s.\ \mathit{InSys}(x, s)$
At Most One
$\forall r, s, s'.\ \mathit{Requester}(r, s) \wedge \mathit{Requester}(r, s') \rightarrow s = s'$
$\forall r, a, a'.\ \mathit{RequestedAct}(r, a) \wedge \mathit{RequestedAct}(r, a') \rightarrow a = a'$

inclusion of systems.

5.1.2.5 Special Conceptual Constraints

We also have several special conceptual constraints. The following says everything contained in the conceptual model `cm` is also contained in every system, i.e.,

$$\forall x, y.\ \mathit{System}(x) \wedge \mathit{InSys}(y, \mathtt{cm}) \rightarrow \mathit{InSys}(y, x).$$

A In this way, we ensure that the Core RefABAC conceptual model is shared among all system models.

The following two constraints specify that if a request is authorized (respectively, prohibited) by any policy, then the request is authorized (respectively, prohibited):

$$\forall r.\ (\exists p.\ \mathit{AuthorizedBy}(r, p)) \rightarrow \mathit{Authorized}(r),$$

$$\forall r.\ (\exists p.\ \mathit{ProhibitedBy}(r, p)) \rightarrow \mathit{Prohibited}(r).$$

5.1.3 Conceptual Action Types

The conceptual action types in Core RefABAC are

$$\mathit{makesRequest}, \mathit{grantsRequest}, \mathit{deniesRequest}, \tag{5.1}$$

$$\mathit{authorizesRequest}, \mathit{prohibitsRequest}. \tag{5.2}$$

The three action types in Line 5.1 are subject action types. The two action types in Line 5.2 are subject or policy action types. An action $\mathit{makesRequest}(s, r, \mathit{sys}, a)$ means a subject s makes a request r to system sys to perform action (with name) a. An action $\mathit{authorizesRequest}(p, r)$ or $\mathit{prohibitsRequest}(p, r)$ means policy (with name) p authorizes or prohibits request r, respectively. An action $\mathit{grantsRequest}(\mathit{sys}, r)$ or $\mathit{deniesRequest}(\mathit{sys}, r)$ means system sys grants or denies request r, respectively.

The following example demonstrates the usage of the conceptual action types. The system

action type *readObj* is defined in the system model for the care facility such that *readObj*(s, o) means subject s reads object o.

Example 5.1. Suppose Carol makes a request to the care facility to read object `o`. The action of Carol making the request is represented as *makesRequest*(`carol`, `r`, `careFacility`, ↑*readObj*(`carol`, `o`)). Suppose request `r` has two applicable policies with names $\mathtt{p}_1$ and $\mathtt{p}_2$, where $\mathtt{p}_1$ authorizes `r` and $\mathtt{p}_2$ prohibits `r`. The actions are represented as *authorizesRequest*($\mathtt{p}_1$, `r`) and *prohibitsRequest*($\mathtt{p}_1$, `r`), respectively. Since the request is prohibited by some policy, the care facility denies the request, i.e., *deniesRequest*(`careFacility`, `r`).

Our formalism allows a subject to make a request for someone else. For instance, if Carol makes a request to the care facility *for Alice* to read the object o, then we have

$$\mathit{makesRequest}(\mathtt{carol}, \mathtt{r}, \mathtt{careFacility}, \uparrow \mathit{readObj}(\mathtt{alice}, \mathtt{o})).$$

Whether such a request should be granted or denied may depend on the attributes of both the requester `carol` and the performer `alice`. In this way, we have a simple delegation representation built in our formalism. One may incorporate various kinds of delegation or revocation protocols [CK08; PO17]. This would be an interesting future direction to explore.

5.1.4 Conceptual Transition Rules

We define three transition rules for the three conceptual action types *makesRequest*, *grantsRequest* and *deniesRequest*, respectively. The action types *authorizesRequest* and *deniesRequest* do not have transition rules but policies, which are defined in system models.

The transition rule for *makesRequest* is defined as

$$\begin{aligned}
&\forall s, r, sys, a.\ \mathit{Request}(r) \wedge \mathit{InSys}(r, sys) \wedge \neg \mathit{Requested}(r) && (5.3)\\
&\quad \rightarrow [\mathit{makesRequest}(s, r, sys, a)](\mathit{Requested}(r) \wedge \mathit{Active}(r) && (5.4)\\
&\qquad \wedge \mathit{Requester}(r, s) \wedge \mathit{RequestedAct}(r, a) && (5.5)\\
&\qquad \wedge \mathit{Performed}(\uparrow \mathit{makesRequest}(s, r, sys, a))). && (5.6)
\end{aligned}$$

Line 5.3 states the pre-condition of making a request, which is that r is a non-requested request in the system *sys*. Lines 5.4-5.6 state the post-condition. The request r's status is updated to requested and active (Line 5.4). Requester s and requested action a are recorded accordingly (Line 5.5). The action of s making the request is recorded as performed (Line 5.6).

The transition rule for *grantsRequest* is

$$\begin{aligned}
&\forall r, sys.\ \mathit{Authorized}(r) \wedge \neg \mathit{Prohibited}(r) \wedge \mathit{Active}(r) \wedge \mathit{InSys}(r, sys)\\
&\quad \rightarrow [\mathit{grantsRequest}(sys, r)](\mathit{Granted}(r) \wedge \neg \mathit{Active}(r)\\
&\qquad \wedge \mathit{Performed}(\uparrow \mathit{grantsRequest}(sys, r)))
\end{aligned}$$

The pre-condition for a system *sys* granting a request r includes that the request r (1) is authorized by some policy; (2) is not prohibited by any policy; (3) is still active; and (4) exists in system *sys*. After *sys* grants r, the status of r becomes granted and inactive. The action of *sys* granting the request is recorded as performed. Since the pre-condition states that the request r must be active and not prohibited, the system cannot grant any already prohibited or inactive requests. Moreover, if a request does not have any applicable policies, then the request is not authorized nor prohibited, i.e., $\neg \mathit{Authorized}(r) \wedge \neg \mathit{Prohibited}(r)$. The system cannot grant the request in this case as well.

The transition rules for *deniesRequest* is defined as

$$\begin{aligned} \forall r, \mathit{sys}.\ & (\mathit{Prohibited}(r) \vee \neg \mathit{Authorized}(r)) \wedge \mathit{Active}(r) \wedge \mathit{InSys}(r, \mathit{sys}) \\ & \rightarrow [\mathit{deniesRequest}(\mathit{sys}, r)](\mathit{Denied}(r) \wedge \neg \mathit{Active}(r) \\ & \wedge \mathit{Performed}(\uparrow \mathit{deniesRequest}(\mathit{sys}, r))). \end{aligned}$$

The pre-condition is that the request r is (1) prohibited or not authorized; (2) still active; (3) exists in the system *sys*. We include $\neg \mathit{Authorized}$ as part of the pre-condition to handle the situation where a request does not have any applicable policies. When a request does not have applicable policies or when it is prohibited by a policy, the system *sys* denies the request. The status of r is updated as denied and inactive, and the action of denying the request is recorded as performed.

Note that we do not need additional transition rules to make sure a granted, denied, and/or fulfilled request is not requested again. In our framework, once if a request becomes requested, it stays requested, no matter if it is granted, denied or fulfilled.

5.1.5 Decision Making Procedure

The decision making procedure of Core RefABAC is illustrated in Figure 5.2 as an extension of Figure 5.1 for request status update. In Figure 5.1, we only mentioned the status each request would go through. Here in Figure 5.2, we also discuss what actions would cause a change in a request's status and when would the parameters of a request be recorded.

Initially, entity r is an inactive and non-requested request in a system *sys*. At Step 1, a subject makes the request r, after which the status of r is updated to $\mathit{Requested}(r)$ and $\mathit{Active}(r)$. The requester and requested action of r are also recorded.

At Step 2, each applicable policy either authorizes or prohibits r. In the figure, we assume two policies with names p and p' respectively. If p prohibits r while p' authorizes r (Case 2(a)), r is both prohibited and authorized. If p prohibits r and r is not authorized by any policies (Case 2(b)), r is prohibited but not authorized. If none of the policies authorizes or prohibits r (Case 2(c)), r is neither prohibited nor authorized. Finally, if p' authorizes r and r is not prohibited by any policies (Case 2(d)), r is authorized and not prohibited.

At Step 3, if any of the first three cases Case 2(a)-2(c) holds, system *sys* denies request r, and r becomes denied and inactive. If the fourth case Case 2(d) holds, the system grants the request, and r becomes granted and inactive.

At the last step, if r is granted, then the requested action may be performed, and r is fulfilled.

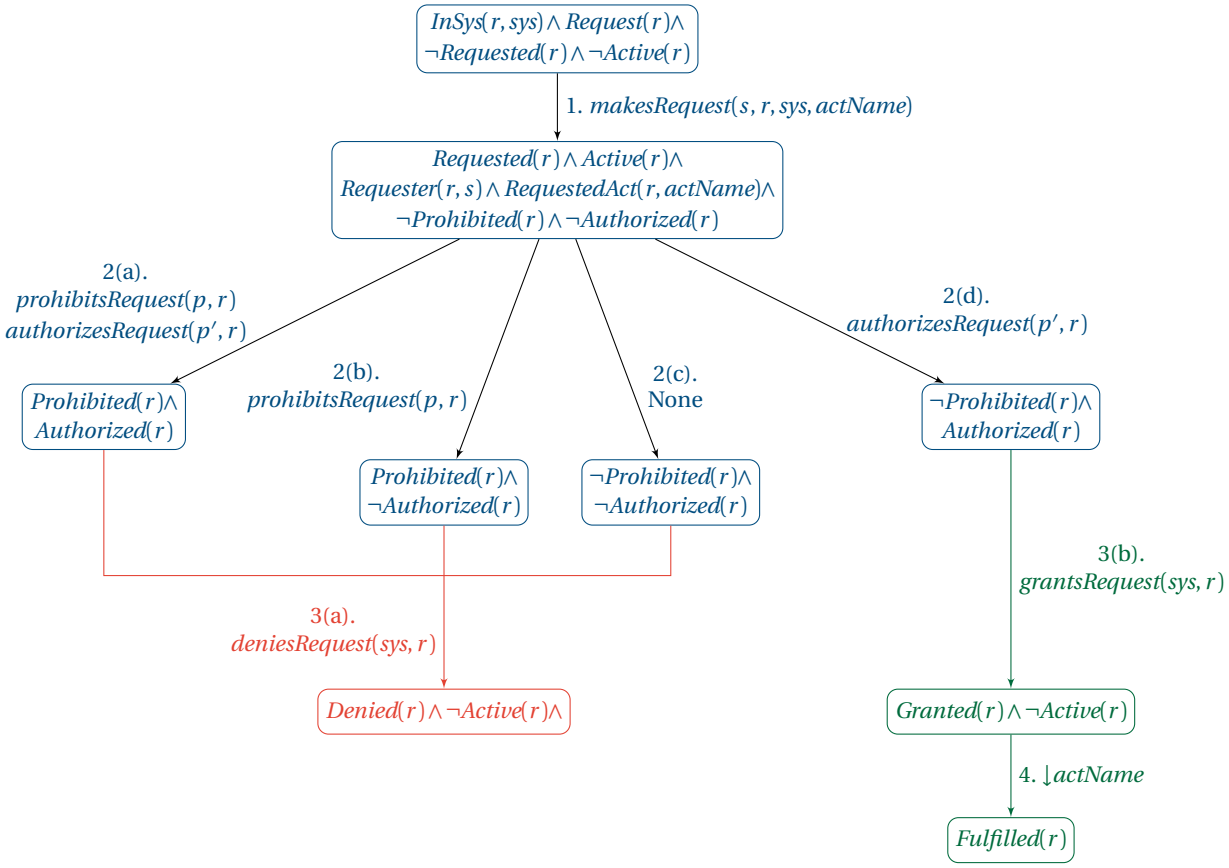

Figure 5.2 Decision Making Procedure in Core RefABAC

5.2 Formalism of System Model in Core RefABAC

Given a RefABAC model for a set of systems, the system model for each of the systems is used to define the domain-dependent attributes, constraints, action types, transition rules, and policies of the system. In this section, we define the formalism of a system model in Core RefABAC.

The logical structure for a system model in Core RefABAC is defined in Definition 5.2. We discuss each of the components in the rest of the section using examples from Scenario 1.1. We use $\mathscr{S}$ as a shorthand for a system model, but `sys` as a constant to represent a system in the formalism.

Definition 5.2. Let `sys` be one of the systems in a RefABAC framework $\mathscr{F}$ and $\mathscr{C} = \langle \mathscr{C}_{Attr}, \mathscr{C}_C, \mathscr{C}_{act}, \mathscr{C}_T \rangle$ the conceptual model of $\mathscr{F}$. Then the **system model** for `sys` is a tuple $\mathscr{S}_{\texttt{sys}} = \langle \mathscr{S}_{Attr}, \mathscr{S}_C, \mathscr{S}_{Act}, \mathscr{S}_T, \mathscr{S}_P \rangle$, where

- $\mathscr{S}_{Attr}$ is a set of attributes called **system attributes** s.t. $\mathscr{C}_{Attr} \subseteq \mathscr{S}_{Attr}$;

- $\mathscr{S}_C$ is a set of constraints called **system constraints** s.t. $\mathscr{C}_C \subseteq \mathscr{S}_C$;
- $\mathscr{S}_{act}$ is a set of action types called **system actions types** s.t. $\mathscr{C}_{act} \subset \mathscr{S}_{act}$;
- $\mathscr{S}_T$ is a set of transition rules called **system transition rules** s.t. $\mathscr{C}_T \subseteq \mathscr{S}_T$; and
- $\mathscr{S}_P$ is a set of policies called **system policies**.

In the formalism for Scenario 1.1, we have three systems, one for the care facility and the other two for the two nearby hospitals. We use the constants `careFacility`, $\texttt{hospital}_1$, and $\texttt{hospital}_2$ to represent them respectively.

5.2.1 System Attributes

System attributes define an entity's domain-dependent properties in a system. As with as conceptual attributes, system attributes can also be used to categorize entities by either defining new types or defining subtypes of the conceptual types. In `careFacility`, we define a new type *Disease* to include all the possible or recognized diseases in the care facility. We also define subtypes of *Subject*, such as *Resident* for residents and *VisitingDoctor* for the doctors that visit the care facility. *Object* has subtypes *DailyLog* for the daily logs the nurses write for the residents during daily checkups, and *PrivateNote* for the private notes the visiting doctors write for their residents.

Besides defining types, a more common usage of system attributes is to define the relationships among the entities in a system. The care facility defines an attribute *ResponsibleDoctorOf* where *ResponsibleDoctorOf*(`d`, `r`) means the visiting doctor `d` is the responsible doctor of the resident `r`. A system attribute may also define a relationship among the entities in different systems. For instance, in an emergency, any doctor from the nearby hospitals may read a resident's medical records and we use *ReviewedBy*(o, d) to indicate a medical record o is reviewed by a doctor d from nearby hospitals. Since not all doctors from nearby hospitals are recorded in `careFacility`, d is an entity—or a subject to be more specific—in either $\texttt{hospital}_1$ or $\texttt{hospital}_2$.

In Core RefABAC, we allow system attributes of arbitrary arities. An attribute of arity greater than 2 can be mapped to multiple binary attributes without loss of generality. For instance, *Meeting*(`r`, `d`, `t`) indicates that the resident `r` has a meeting with the doctor `d` at time `t`. We could transform the ternary attribute *Meeting* to three binary attributes: *MeetingOfResident*, *MeetingWithDoctor* and *MeetingTime*. *MeetingOfResident*(`m`, `r`) means `m` is a meeting of resident `r`, *MeetingWithDoctor*(`m`, `d`) means `m` is a meeting with doctor `d`, and *MeetingTime*(`m`, `t`) means the meeting time of `m` is `t`.

Note that we *choose* the attributes *ResponsibleDoctorOf* and *Meeting* to represent the responsible doctor of a resident and the meeting time between a resident and his or her responsible doctor, respectively, for the purpose of demonstrating the usage of our RefABAC paradigm. Depending on the requirements of an AC system, these facts may be stored as the content in an object instead of as attributes of entities.

5.2.2 System Constraints

A system constraint expresses a domain-dependent relationship among the system attributes. The representation of a system constraint in Core RefABAC is defined as follows.

Definition 5.3. Let sys be a system in a Core RefABAC model. The system model for sys is specified as $\mathscr{S}_{\texttt{sys}} = \langle \mathscr{S}_{Attr}, \mathscr{S}_C, \mathscr{S}_{act}, \mathscr{S}_T, \mathscr{S}_P \rangle$. Then a system constraint in $\mathscr{S}_C$ is a DL$^+$ formula in the following form:

$$\forall \vec{x}.\, \phi \rightarrow \exists \vec{y}.\, \psi,$$

where ϕ and ψ are Boolean combinations of system attributes defined in $\mathscr{S}_{Attr}$.

Notice that existentially quantified variables appear only on the right side of the implication so that we can have a more straightforward mapping to SQL statements. This is also the basic form of formulas in Datalog with disjunction and existential quantifiers [Alv12]. Though a constraint does not have any DL$^+$ action modalities, it may contain names of formulas which mention actions. For instance, in our formalization, policies contain actions $authorizesRequest(p, r)$ or $prohibitsRequest(p, r)$ to indicate whether the policy with name p authorizes or prohibits request r. One may define an attribute $CreatedBy(p, s)$ to indicate that the policy with name p is created by subject s, and a constraint $\forall p, s.\ CreatedBy(p, s) \rightarrow Policy(p) \wedge Manager(s)$ to indicate that all policies are created by managers.

The common functionalities of system constraints are similar to those of conceptual constraints. Specifically, they can be used to define (1) subtype relations among attributes that are used to define types of entities; (2) types of arguments in an attribute (with more than one arity); (3) disjointness relations among attributes; and (4) number restrictions of the arguments in an attribute; and (5) other more complicated relations.

If a constraint in a system sys involves an attribute instance $Attr(\vec{x})$ that belongs to another system sys$'$, then instead of using $Attr(\vec{x})$, one should write $InSys(\uparrow Attr(\vec{x}), \texttt{sys}')$ because $Attr(\vec{x})$ is not contained in the knowledge base of *sys* but of *sys*$'$. We first list some examples where all attributes are from the same system to which the constraint belongs (Example 5.2-Example 5.3), then discuss an example where some attributes from other systems are involved (Example 5.4).

Example 5.2. The care facility defines type attributes *Resident* and *VisitingDoctor*, for the residents in the care facility and the doctors visiting the care facility from the nearby hospitals. The attributes are subtypes of the conceptual attribute *Subject*, specified using the following system constraints:

$$\forall x.\, Resident(x) \rightarrow Subject(x),$$

$$\forall x.\, VisitingDoctor(x) \rightarrow Subject(x).$$

Let c_1 and c_2 denote the names of the two example system constraints above. Since they are

constraints in the care facility, the following statements are also true:

$$\mathit{InSys}(\mathtt{c}_1, \mathtt{careFacility}), \text{ and } \mathit{InSys}(\mathtt{c}_2, \mathtt{careFacility}).$$

Example 5.3. In the care facility, each resident must have one visiting doctor as his or her responsible doctor.

$$\forall r.\ \mathit{Resident}(r) \rightarrow \exists d.\, \mathit{VisitingDoctor}(d) \wedge \mathit{ResponsibleDoctorOf}(d, r)$$

$$\forall r, d, d'.\ \mathit{Resident}(r) \wedge \mathit{ResponsibleDoctorOf}(d, r) \wedge \mathit{ResponsibleDoctorOf}(d', r) \rightarrow d = d'$$

The system constraints in the above example are the at-least-one and at-most-one number restrictions on the attribute *ResponsibleDoctorOf*, respectively. Each resident has exactly one responsible doctor.

The attributes used in a system constraint may be from one or multiple systems. Given a system constraint in a system $\mathtt{sys}_1$, if an attributes *Attr* in the constraint is from another system $\mathtt{sys}_2$, then the constraint should use $\mathit{InSys}(\uparrow \mathit{Attr}(\vec{x}), \mathtt{sys}_2)$ instead of $\mathit{Attr}(\vec{x})$ for clarification. Otherwise, the system $\mathtt{sys}_1$ to which the constraint belongs will analyze whether $\mathit{Attr}(\vec{x})$ is true in its own system state and it will always evaluate to false if $\mathtt{sys}_1$ does not have the attribute *Attr*. The following example from Example 1.1 demonstrates the usage of *InSys* in a constraint.

Example 5.4. A visiting doctor must be a doctor from the nearby hospitals.

$$\forall d.\ \mathit{VisitingDoctor}(d) \rightarrow \mathit{InSys}(\uparrow \mathit{Doctor}(d), \mathtt{hospital}_1) \vee \mathit{InSys}(\uparrow \mathit{Doctor}(d), \mathtt{hospital}_2).$$

In the above example, $\mathit{Doctor}(d)$ means d is a doctor and both of the system models for the two nearby hospitals define it as a system attribute. We assume that for the same doctor, all the three systems in the framework—the care facility and the two nearby hospitals—use the same identifier. When the system of the care facility evaluates whether the constraint holds, it knows it should check with $\mathtt{hospital}_1$ and $\mathtt{hospital}_2$ to see if d is a doctor.

5.2.3 System Action Types

The system action types in a system define the AC action types that a subject may perform in the system. The common AC action types are a subject reading and writing an object. In Core RefABAC, one may also define various administrative action types, such as modifying the value of an attribute, adding a new policy, and deleting a constraint.

In the formalism for the care facility's system model, we define a system action type *addAttr* such that $\mathit{addAttr}(s, \uparrow \mathit{Attr})$ means a subject adds an attribute *Attr* to the system. We also define an action type *modifyAttrInst* such that $\mathit{modifyAttrInst}(s, \uparrow \mathit{Attr}(\vec{\mathtt{a}}), \uparrow \mathit{Attr}(\vec{\mathtt{b}}))$ means a subject s modifies the attribute instance $\mathit{Attr}(\vec{\mathtt{a}})$ to $\mathit{Attr}(\vec{\mathtt{b}})$. How each action should behave is defined by its corresponding transition rules, discussed in the next section.

5.2.4 System Transition Rules

The system transition rules in a system sys define the pre- and post-conditions of the system action types defined in sys. Specifically, given an action type *act*, under what conditions an action of *act* can be performed and what happens after the action is performed. In Core RefABAC, the pre-condition for any system action type includes that the corresponding request for the action is granted but not fulfilled. Definition 5.4 defines the specific syntax of a system transition rule in Core RefABAC.

Definition 5.4. Let sys be a system in a Core RefABAC model. The system model for sys is specified as $\mathscr{S}_{\texttt{sys}} = \langle \mathscr{S}_{Attr}, \mathscr{S}_C, \mathscr{S}_{act}, \mathscr{S}_T, \mathscr{S}_P \rangle$. Then a system transition rule in $\mathscr{S}_T$ is a DL$^+$ formula in the following form:

$$\begin{aligned} \forall r, \vec{x}.\ & \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{sys}) \wedge \mathit{RequestedAct}(r, \uparrow \mathit{act}(\vec{x})) \wedge \phi_1 \\ & \rightarrow [\mathit{act}(\vec{x})](\mathit{Fulfilled}(r) \wedge \mathit{Performed}(\uparrow \mathit{act}) \wedge \phi_2) \end{aligned}$$

where $\mathit{act} \in \mathscr{S}_{act}$, and ϕ_1 and ϕ_2 are conjunctions of system attribute instances and of their negations.

Our formalism of transition rules is expressive enough to define regular AC actions as well as administrative actions. Note that we do not have existential quantifiers in a transition rule and the pre- and post-conditions only involve conjunctions of attribute instances and their negations. In this way, each transition rule is deterministic in the sense that given a system state σ in which the preconditions hold, there is only one possible system state σ' to which σ can transition.

Similar to a system constraint, a system transition rule may also involve attribute instances from other systems. If the pre-condition of a transition rule in a system sys involves an attribute instance $\mathit{Attr}(\vec{x})$ from another system sys$'$, then the transition rule should use $\mathit{InSys}(\uparrow \mathit{Attr}(\vec{x}), \texttt{sys}')$ instead. Whether the requested action can be performed would depend on whether sys contains that piece of information from sys$'$. If not, the pre-condition is not satisfied and the action cannot be performed. On the other hand, we do not allow attribute instances from other systems to be used in the post-conditions of a transition rule. We add the restriction for security and running time concerns. If an entity y from another system sys$'$ is used in the post-condition, then there are three main approaches to handle it: (1) let sys makes a request to sys$'$ to modify entity y; (2) let sys$'$ modify y; or (3) ignore entity y. The first approach is time consuming. The second one is not secure. The last one is similar to our approach of disallowing entities from other systems to be used in the post-conditions.

We first discuss two examples regarding defining system transition rules using only entities from the same system as the transition rule (Example 5.5-Example 5.6), then another example using entities from different systems (Example 5.7).

Example 5.5. The care facility has a regular AC action type *readObj*, representing the actions of a

subject reading an object. The transition rule for *readObj* is defined as:

$$\forall r, s, o.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{RequestedAct}(r, \uparrow \mathit{readObj}(s, o)) \\ \rightarrow [\mathit{readObj}(s, o)](\mathit{Fulfilled}(r) \wedge \mathit{Performed}(\uparrow \mathit{readObj}(s, o))).$$

The above example transition rule means that given a request r in `careFacility` whose requested action is for a subject s to read an object o, then if r is granted but not fulfilled yet, s may perform the requested action and the request's status becomes fulfilled afterwards.

Example 5.6. We define an action type *addAttrInst* in the care facility to represent the actions of a subject a new attribute instance is added, i.e., adding a new value of an attribute for some entity. The transition rule for *addAttrInst* is defined as follows:

$$\forall r, s, o.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{RequestedAct}(r, \uparrow \mathit{addAttrInst}(s, a)) \\ \rightarrow [\mathit{addAttrInst}(s, a)](\mathit{Fulfilled}(r) \wedge \mathit{Performed}(\uparrow \mathit{addAttrInst}(s, a)) \\ \wedge \mathit{AttributeInst}(a) \wedge \mathit{InSys}(a, \texttt{careFacility})).$$

The above example is the transition rule for the system action type *addAttrInst*. Let r be a granted but not yet fulfilled request in `careFacility` with the requested action $\mathit{addAttrInst}(s, a)$, where a is the name of an attribute instance. When adding a new attribute instance, e.g., $\mathit{Attr}(c_1, c_2)$, the arguments in the instance must be constants rather than variables. For example, suppose the resident `bob` requests to add an emergency contact `eve`. The action of `bob` adding an emergency contact `eve` is represented as

$$\mathit{addAttrInst}(\texttt{bob}, \uparrow \mathit{EmergencyContactOf}(\texttt{bob}, \texttt{eve})).$$

After `bob` performs the action, the request becomes fulfilled. The attribute instance $\mathit{EmergencyContactOf}(\texttt{bob}, \texttt{eve})$ that `bob` requests to add is recorded as a new attribute instance in `careFacility`, i.e.,

$$\mathit{AttributeInst}(\uparrow \mathit{EmergencyContactOf}(\texttt{bob}, \texttt{eve})), \text{ and}$$
$$\mathit{InSys}(\uparrow \mathit{EmergencyContactOf}(\texttt{bob}, \texttt{eve}), \texttt{careFacility}).$$

We do not need to include $\mathit{InSys}(\uparrow \mathit{AttributeInst}(a), \texttt{careFacility})$ in the consequence of the transition rule. By the definition of a system state (Definition 5.6), if an attribute instance $\mathit{Attr}(\vec{x})$ is currently in the system, i.e., $\mathit{InSys}(\uparrow \mathit{Attr}(\vec{x}), \texttt{sys})$, then $\mathit{InSys}(\uparrow \mathit{AttributeInst}(\uparrow \mathit{Attr}(\vec{x})), \texttt{sys})$ is also in the system.

Example 5.7. Suppose a doctor d_1 from $\texttt{hospital}_1$ has invited a doctor d_2 from $\texttt{hospital}_2$ to treat a patient p together in $\texttt{hospital}_1$. Then d_2 is authorized to read p's medical records. However,

when d_2 reads p's medical records, $\texttt{hospital}_1$ must record the time of d_2's action.

$$\begin{aligned}\forall r, s, o, t.\ &\mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{hospital}_1) \wedge \mathit{RequestedAct}(r, \uparrow \mathit{readObj}(s, o)) \\ &\wedge \mathit{MedRecord}(o) \wedge \mathit{InSys}(\uparrow \mathit{Doctor}(s), \texttt{hospital}_2) \wedge \mathit{CurrentTime}(t) \\ &\rightarrow [\mathit{readObj}(s, o)](\mathit{Fulfilled}(r) \wedge \mathit{Performed}(\uparrow \mathit{readObj}(s, o)) \\ &\quad \wedge \mathit{PerformedTime}(\uparrow \mathit{readObj}(s, o), t)).\end{aligned}$$

The above transition rule is defined in $\texttt{hospital}_1$. But the subject s is a doctor from $\texttt{hospital}_2$, i.e., $\mathit{InSys}(\uparrow \mathit{Doctor}(s), \texttt{hospital}_2)$. If $\texttt{hospital}_1$ does not contains the piece of information that says s is a doctor in $\texttt{hospital}_2$, the action cannot be performed even though the request r has been granted.

5.2.5 System Policies

A system policy defines under what conditions a request should be authorized or prohibited. The generic form for a policy is a DL^+ formula in one of the following forms:

$$\forall r, \vec{x}.\ \exists \vec{y}.\ \mathit{Active}(r) \wedge \phi \rightarrow [\mathit{authorizesRequest}(\texttt{pol}, r)](\psi)$$

$$\forall r, \vec{x}.\ \exists \vec{y}.\ \mathit{Active}(r) \wedge \phi \rightarrow [\mathit{prohibitsRequest}(\texttt{pol}, r)](\psi),$$

where ϕ and ψ are Boolean combinations of attribute instances.

In Core RefABAC, the representation of a policy is formally defined in the following.

Definition 5.5. Let $\texttt{sys}$ be a system in a Core RefABAC model. The system model for $\texttt{sys}$ is specified as $\mathscr{S}_{\texttt{sys}} = \langle \mathscr{S}_{Attr}, \mathscr{S}_C, \mathscr{S}_{act}, \mathscr{S}_T, \mathscr{S}_P \rangle$. Then a system policy in $\mathscr{S}_P$ with name $\texttt{pol}$ is a DL^+ formula in one of the following forms:

$$\begin{aligned}\forall r, \vec{x}.\ &\mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{sys}) \wedge \phi \\ &\rightarrow [\mathit{authorizesRequest}(\texttt{pol}, r)](\mathit{Authorized}(r) \wedge \mathit{AuthorizedBy}(r, \texttt{pol}))\end{aligned}$$

$$\begin{aligned}\forall r, \vec{x}.\ &\mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{sys}) \wedge \phi \\ &\rightarrow [\mathit{prohibitsRequest}(\texttt{pol}, r)](\mathit{Prohibited}(r) \wedge \mathit{ProhibitedBy}(r, \texttt{pol})).\end{aligned}$$

where ϕ is a conjunction of system attribute instances and of their negations.

We disallow existentially quantified variables and disjunction in Core RefABAC policies. Given a system state satisfying the preconditions of a policy, a state of a Core RefABAC system state has exactly one possible successor state. Moreover, the only post-conditions of a Core RefABAC policy are to change the status change of the request r and to record which policy authorized r.

The formula ϕ in the precondition of a policy usually involves the attributes for the requester and the requested action of r. The following example shows how to define a system policy.

Example 5.8. Policy `logPol`: The front desks are authorized to read the visitor logs.

$$\begin{aligned}\forall r, s, o.\ & \mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \\ & \wedge \mathit{Requester}(r, s) \wedge \mathit{FrontDesk}(s) \\ & \wedge \mathit{RequestedAct}(r, \uparrow \mathit{readObj}(s, o)) \wedge \mathit{VisitorLog}(o) \\ & \rightarrow [\mathit{authorizesRequest}(\texttt{logPol}, r)](\mathit{Authorized}(r) \wedge \mathit{AuthorizedBy}(r, \texttt{logPol}))\end{aligned}$$

The above example is a simple policy in `careFacility`. The preconditions state that (1) the request r in question is active and exists in `careFacility`; (2) the requester s of r is a front desk; (3) the requested action of r is for s reading an object o and the object o is a visitor log. The post-condition states that after the policy with name $\texttt{pol}_1$ authorizes the request r, r's status becomes authorized and it is authorized by $\texttt{pol}_1$.

5.3 Formalism of System State in Core RefABAC

Given a RefABAC model, its current state is composed of the current system state for each of systems in the model. The current system state of a system contains all the information that is currently stored in the system. If a system model is analogous to a database schema, then a system state is analogous to a database state. The elements included in a system state in Core RefABAC are defined formally in the following definition.

Definition 5.6. Let `sys` be a system in a Core RefABAC model. The system model for `sys` is specified as $\mathscr{S}_{\texttt{sys}} = \langle \mathscr{S}_{Attr}, \mathscr{S}_C, \mathscr{S}_{act}, \mathscr{S}_T, \mathscr{S}_P \rangle$. Then a system state σ of `sys` contains the following kinds of elements:

(1) $\mathit{Attribute}(\uparrow \mathit{Attr})$ and $\mathit{InSys}(\uparrow \mathit{Attr}, \texttt{sys})$, for each $\mathit{Attr} \in \mathscr{S}_{Attr}$;

(2) ϕ, $\mathit{Constraint}(\uparrow \phi)$ and $\mathit{InSys}(\uparrow \phi, \texttt{sys})$, for each $\phi \in \mathscr{S}_C$;

(3) $\mathit{ActionType}(\uparrow \mathit{act})$ and $\mathit{InSys}(\uparrow \mathit{act}, \texttt{sys})$, for each $\mathit{act} \in \mathscr{S}_{act}$;

(4) ϕ, $\mathit{TransitionRule}(\uparrow \phi)$ and $\mathit{InSys}(\uparrow \phi, \texttt{sys})$, for each $\phi \in \mathscr{S}_T$;

(5) ϕ, $\mathit{Policy}(\uparrow \phi)$ and $\mathit{InSys}(\uparrow \phi, \texttt{sys})$, for each $\phi \in \mathscr{S}_P$;

(6) individuals in the system and $\mathit{InSys}(c, \texttt{sys})$, for each individual c;

(7) actions of the action types in $\mathscr{S}_{act}$, and $\mathit{Action}(\uparrow a)$ and $\mathit{InSys}(\uparrow a, \texttt{sys})$, for each action a;

(8) attribute instances of the attributes in $\mathscr{S}_{Attr}$, and $\mathit{AttributeInst}(\uparrow a)$ and $\mathit{InSys}(\uparrow a, \texttt{sys})$, for each attribute instance $\mathit{Attr}(\vec{x})$ that is not of the form $\mathit{InSys}(a, b)$.

Recall that conceptual action types are included in system action types. So we do not need to list them separately. The same holds for conceptual constraints, action types and transition rules. Moreover, to avoid recursive inclusion of elements in a system state, we do not have attribute instances in the form of $\mathit{InSys}(\uparrow \mathit{InSys}(x, \texttt{sys}), \texttt{sys})$, or $\mathit{AttributeInst}(\uparrow \mathit{AttributeInst}(x))$, etc.; otherwise, the size of a system state will be infinite.

Example 5.9. Suppose `carol` is a resident at the care facility in the current system state. Then the system state contains the following facts.

Reference	Facts
Definition 5.6 (1)	$\mathit{Attribute}(\uparrow \mathit{Resident})$
	$\mathit{InSys}(\uparrow \mathit{Resident}, \texttt{careFacility})$
Definition 5.6 (2)	$\forall x.\mathit{Resident}(x) \rightarrow \mathit{Subject}(s)$, with name `resSubCons`
	$\mathit{Constraint}(\texttt{resSubCons})$
	$\mathit{InSys}(\texttt{resSubCons}, \texttt{careFacility})$
Definition 5.6 (6)	$\mathit{InSys}(\texttt{carol}, \texttt{careFacility})$
	$\mathit{InSys}(\texttt{cm}, \texttt{careFacility})$
Definition 5.6 (8)	$\mathit{Resident}(\texttt{carol})$
	$\mathit{Subject}(\texttt{carol})$
	$\mathit{AttributeInst}(\uparrow \mathit{Attribute}(\uparrow \mathit{Resident}))$
	$\mathit{AttributeInst}(\uparrow \mathit{Constraint}(\texttt{resSubCons}))$
	$\mathit{AttributeInst}(\uparrow \mathit{Resident}(\texttt{carol}))$
	$\mathit{AttributeInst}(\uparrow \mathit{Subject}(\texttt{carol}))$
	$\mathit{InSys}(\uparrow \mathit{Attribute}(\uparrow \mathit{Resident}), \texttt{careFacility})$
	$\mathit{InSys}(\uparrow \mathit{Constraint}(\texttt{resSubCons}), \texttt{careFacility})$
	$\mathit{InSys}(\uparrow \mathit{Resident}(\texttt{carol}), \texttt{careFacility})$
	$\mathit{InSys}(\uparrow \mathit{Subject}(\texttt{carol}), \texttt{careFacility})$

5.4 Conclusion

In this chapter, we formalized a baseline RefABAC framework called Core RefABAC. Given a set of systems, its Core RefABAC model contains a conceptual model and at least one system model, one for each of the systems. We use the logical language DL^+ to formalize Core RefABAC. The language uses reflection to assign names to entities such that they can be referenced. Therefore, we are able to define attributes on any entities.

The conceptual model defines several basic types to categorize entities in a system, such as the types of subjects, objects, policies, and attributes. It also defines basic action types for a subject

making a request, a policy authorizing or prohibiting a request, and a system granting or denying a request. In a system model, one may define regular AC actions as well as administrative actions. Regular AC actions include reading and writing an object. An administrative action may create, delete, or modify any entity in a system. We have defined example action types with transition rules on reading an object and adding an attribute instance.

Core RefABAC uses the denial-by-default approach in its decision-making procedure. When a subject makes a request, the request becomes requested and active. After the request's applicable policies authorize or prohibit the request, the request becomes authorized or prohibited, or neither authorized nor prohibited. A request may be both authorized and prohibited. If a request is prohibited or not authorized by any policies, then the system denies the request. Otherwise, the system grants the request.

CHAPTER

6

FORMALIZATION OF THE CARE FACILITY SCENARIO IN CORE REFABAC

We defined Core RefABAC as a baseline RefABAC framework in the preceding chapter. Core RefABAC allows attributes to be defined on any entities. One may also define regular AC as well as administrative actions in Core RefABAC. Core RefABAC follows the denial-by-default approach such that a request is denied if it is prohibited or not authorized by any policies.

In this chapter, we demonstrate the usage of Core RefABAC by formalizing Scenario 1.1. Scenario 1.1 contains three systems: one for the care facility and two for the nearby hospitals that have doctors visiting the care facility. We use the constants $\texttt{careFacility}$, $\texttt{hospital}_1$, and $\texttt{hospital}_2$ to represent the three systems respectively in the formalism. We will define the attributes and action types mentioned in the scenario, along with relevant constraints and transition rules. We also formalize a couple of example policies from the motivating examples in Section 1.2.

6.1 System Attributes

Table 6.1 lists the system attributes used in the care facility. We categorize the attributes according to their functionality. We identify four *direct* subtypes of the conceptual type *Subject*: *Staff* for the staff, *VisitingDoctor* for doctors from nearby hospitals, *Resident* for the residents, *EmergencyContact* for the people who are listed as emergency contacts of residents, and *Volunteer* for the volunteers who help out at the care facility. We also define several *indirect* subtypes of *Subject*. For *Staff*, we define the subtypes *Manager*, *FrontDesk*, *Cashier*, and *Nurse*, which include the individuals that are managers, front desks, cashiers, and nurses respectively. We also define a subtype of

Resident, *SpecialCare*, which is for the residents that need special care. We also define two subtypes of *Volunteer*: *VolunteerForDailyCheckup* for the volunteers who are responsible for helping the nurses with daily checkups, and *VolunteerForArrangingMeeting* for the volunteers who are responsible for arranging meetings for residents with their responsible doctors and visitors.

Table 6.1 System Attributes in `careFacility`

Category Name	List of Attributes
Subject Subtypes (of arity 1)	*Staff*, *Manager*, *FrontDesk*, *Cashier*, *Nurse*, *Resident*, *Volunteer*, *EmergencyContact*, *VisitingDoctor*, *SpecialCare*, *VolunteerForDailyCheckup*, *VolunteerForArrangingMeeting*
Object Subtypes (of arity 1)	*DailyLog*, *PrivateNote*, *MedRecord*, *ExamResult*, *InsuranceInfo*, *PaymentInfo*, *VisitorLog*
Other types (of arity 1)	*Disease*, *HeartDisease*, *Time*, *Date*
Properties of Entities (of arity 2)	*OwnerOf*, *ResponsibleDoctorOf*, *Meeting*, *HasDisease*, *EmergencyContactOf*, *LogDate*

The subtypes of *Object* categorize the objects in the care facility as (1) *DailyLog* for the logs that nurses and volunteers take for residents during their daily checkups; (2) *PrivateNote* for visiting doctors' private notes about their residents, (3) *MedRecord* for residents' medical records, (4) *ExamResult* for residents' monthly exam results, (5) *InsuranceInfo* for the documents containing residents' insurance information, (6) *PaymentInfo* for the documents containing residents' payment information, and (7) *VisitorLog* for the visitor logs. These subtypes of *Object* are exclusive of each other.

We also define two additional types of entities. The type *Disease* is for the diseases of the residents at the care facility. *HeartDisease* is for the heart diseases and is a subtype of *Disease*. The types *Time* and *Date* define the timestamps and dates.

We list several example system attributes for describing properties of entities in the care facility. $\mathit{OwnerOf}(s, o)$ means subject s is the owner of object o. $\mathit{ResponsibleDoctorOf}(d, r)$ means visiting doctor d is the responsible doctor of resident r. $\mathit{Meeting}(r, d, t)$ specifies that resident r and visiting doctor d has a meeting at time t. $\mathit{EmergencyContactOf}(s, r)$ means s is listed as an emergency contact of resident r. Finally, $\mathit{LogDate}(o, d)$ means the date when daily log o is recorded is d. Similar as conceptual attributes, the system attributes are subject to constraints that specify the types of arguments, the number restrictions, and so on. We will formally define them in the next section (Section 6.2).

The system attributes for the two nearby hospitals $\texttt{hospital}_1$ and $\texttt{hospital}_2$ are the same, as listed in Table 6.2. The subtypes of *Subject* in the hospitals are *Doctor*, *Nurse*, *FrontDesk*, and

Patient for doctors, nurses, front desks, and patients, respectively . We also define a system attribute *ResponsibleDoctorOf* such that *ResponsibleDoctorOf*(d, p) indicates d is the responsible doctor d of patient p.

Table 6.2 System Attributes in $\texttt{hospital}_1$ and $\texttt{hospital}_2$

Category Name	List of Attributes
Subject Subtypes (of arity 1)	*Doctor*, *Nurse*, *FrontDesk*, *Patient*
Properties of Entities (of arity 2)	*ResponsibleDoctorOf*

In our formalism, we allow different systems to use the same identifier for some entities, such as the attribute *Doctor* in $\texttt{hospital}_1$ and $\texttt{hospital}_2$. The conceptual attribute *InSys* can be used to clarify any confusions. For instance, *InSys*$(\uparrow \textit{Doctor}, \texttt{hospital}_1)$ refers to the *Doctor* attribute in $\texttt{hospital}_1$, while *InSys*$(\uparrow \textit{Doctor}, \texttt{hospital}_2)$ the *Doctor* attribute in $\texttt{hospital}_2$.

6.2 System Constraints

We categorize the system constraints in the formalism of Scenario 1.1 in the same way as we did for the conceptual constraints in Core RefABAC (Subsection 5.1.2). Specifically, the system constraints are used to define (1) subtype relations among system attributes that are used to define types of entities; (2) types of arguments in a system attribute (with more than one arity); (3) number restrictions of the arguments in a system attribute; and (4) disjointness relations among the system attributes; (5) other more complicated relations.

6.2.1 Subtype Relations

Table 6.3 lists the system constraints for defining subtype relations among the attributes in the care facility system. The system attributes *Staff*, *Resident*, *Volunteer*, and *EmergencyContact* are subtypes of the conceptual attribute *Subject*, specified using a system constraint of the form

$$\forall x.\, \textit{Attr}(x) \rightarrow \textit{Subject}(x),$$

where *Attr* is any of these attributes. Furthermore, as shown in the table, we divide *Staff* into more specific types of entities according to their roles at the care facility. *SpecialCare* is a subtype of *Resident*. *VolunteerForDailyCheckup* and *VolunteerForArrangingMeeting* are subtypes of *Volunteer*. Similarly, the system attributes used to define subtypes of *Object* are subject to the system constraints in the form of

$$\forall x.\, \textit{Attr}(x) \rightarrow \textit{Object}(x).$$

Table 6.3 System Constraints for Sub-Type Relations in `careFacility`

Subject Subtypes	$\forall s.\ \mathit{Staff}(s) \rightarrow \mathit{Subject}(s)$
	$\forall s.\ \mathit{Resident}(s) \rightarrow \mathit{Subject}(s)$
	$\forall s.\ \mathit{Volunteer}(s) \rightarrow \mathit{Subject}(s)$
	$\forall s.\ \mathit{EmergencyContact}(s) \rightarrow \mathit{Subject}(s)$
	$\forall s.\ \mathit{VisitingDoctor}(s) \rightarrow \mathit{Staff}(s)$
	$\forall s.\ \mathit{Manager}(s) \rightarrow \mathit{Staff}(s)$
	$\forall s.\ \mathit{FrontDesk}(s) \rightarrow \mathit{Staff}(s)$
	$\forall s.\ \mathit{Cashier}(s) \rightarrow \mathit{Staff}(s)$
	$\forall s.\ \mathit{Nurse}(s) \rightarrow \mathit{Staff}(s)$
	$\forall s.\ \mathit{SpecialCare}(s) \rightarrow \mathit{Resident}(s)$
	$\forall s.\ \mathit{VolunteerForDailyCheckup}(s) \rightarrow \mathit{Volunteer}(s)$
	$\forall s.\ \mathit{VolunteerForArrangingMeeting}(s) \rightarrow \mathit{Volunteer}(s)$
Object Subtypes	$\forall o.\ \mathit{DailyLog}(o) \rightarrow \mathit{Object}(o)$
	$\forall o.\ \mathit{PrivateNote}(o) \rightarrow \mathit{Object}(o)$
	$\forall o.\ \mathit{MedRecord}(o) \rightarrow \mathit{Object}(o)$
	$\forall o.\ \mathit{ExamResult}(o) \rightarrow \mathit{Object}(o)$
	$\forall o.\ \mathit{InsuranceInfo}(o) \rightarrow \mathit{Object}(o)$
	$\forall o.\ \mathit{PaymentInfo}(o) \rightarrow \mathit{Object}(o)$
	$\forall o.\ \mathit{VisitorLog}(o) \rightarrow \mathit{Object}(o)$
Other Subtypes	$\forall x.\ \mathit{HeartDisease}(x) \rightarrow \mathit{Disease}(x)$

Finally, we define *HeartDisease* as a subtype of *Disease*.

The system constraints in $\texttt{hospital}_1$ and $\texttt{hospital}_2$ for defining subtype relations are the same and listed together in Table 6.4. They specify that the attributes *Doctor*, *Nurse*, *FrontDesk*, and *Patient* are subtypes of *Subject*.

Table 6.4 System Constraints for Sub-Type Relations in $\texttt{hospital}_1$ and $\texttt{hospital}_2$

$\forall x.\ \mathit{Doctor}(x) \rightarrow \mathit{Subject}(x)$
$\forall x.\ \mathit{Nurse}(x) \rightarrow \mathit{Subject}(x)$
$\forall x.\ \mathit{FrontDesk}(x) \rightarrow \mathit{Subject}(x)$
$\forall x.\ \mathit{Patient}(x) \rightarrow \mathit{Subject}(x)$

6.2.2 Argument Types and Number Restrictions

We list the system constraints used for defining types of the arguments in a system attribute in Table 6.5, and the number restrictions in Table 6.6.

Table 6.5 System Constraints for Argument Types

careFacility	$\forall r, o.\ OwnerOf(r, o) \rightarrow Resident(r) \wedge Object(o)$
	$\forall d, r.\ ResponsibleDoctorOf(d, r) \rightarrow VisitingDoctor(d) \wedge Resident(r)$
	$\forall r, d.\ HasDisease(r, d) \rightarrow Resident(r) \wedge Disease(d)$
	$\forall s, r.\ EmergencyContactOf(s, r) \rightarrow EmergencyContact(s) \wedge Resident(r)$
	$\forall l, d.\ LogDate(l, d) \rightarrow DailyLog(l) \wedge Date(d)$
	$\forall r, d, t.\ Meeting(r, d, t) \rightarrow Resident(r) \wedge VisitingDoctor(d) \wedge Time(t)$
$\texttt{hospital}_1$, $\texttt{hospital}_2$	$\forall d, p.\ ResponsibleDoctorOf(d, p) \rightarrow Doctor(d) \wedge Patient(p)$

Table 6.6 System Constraints for Number Restrictions

careFacility	$\forall r, r', o.\ OwnerOf(r, o) \wedge OwnerOf(r', o) \rightarrow r = r'$
	$\forall d, d', r.\ ResponsibleDoctorOf(d, r) \wedge ResponsibleDoctorOf(d', r) \rightarrow d = d'$
	$\forall l, d, d'.\ LogDate(l, d) \wedge LogDate(l, d') \rightarrow d = d'$
	$\forall o.\ Object(o) \rightarrow \exists r.\ OwnerOf(r, o)$
	$\forall r.\ Resident(r) \rightarrow \exists d.\ ResponsibleDoctorOf(d, r)$
	$\forall r.\ Resident(r) \rightarrow \exists s.\ EmergencyContactOf(s, r)$
	$\forall l.\ DailyLog(l) \rightarrow \exists d.\ LogDate(l, d)$
	$\forall d, r.\ ResponsibleDoctorOf(d, r) \rightarrow \exists t.Meeting(r, d, t)$
$\texttt{hospital}_1$, $\texttt{hospital}_2$	$\forall d, d', p.\ ResponsibleDoctorOf(d, p) \wedge ResponsibleDoctorOf(d', p) \rightarrow d = d'$
	$\forall p.\ Patient(p) \rightarrow \exists d.\ ResponsibleDoctorOf(d, p)$

In the care facility, $OwnerOf(r, o)$ means a resident r is the owner of an object o. Based on the number restrictions for $OwnerOf$, each object must have exactly one resident as its owner. $ResponsibleDoctorOf(d, r)$ means a visiting doctor d is the responsible doctor of a resident r, and each resident must have exactly one responsible doctor. Third, $HasDisease(r, d)$ indicates the a resident has a disease d. $EmergencyContactOf(s, r)$ means s is an emergency contact of a resident r and each resident must have at least one person listed as his/her emergency contact. $LogDate(l, d)$ means the recorded date of a daily log file l is d and each daily log file has exactly one corresponding

recorded date.

In $\texttt{hospital}_1$ and $\texttt{hospital}_2$, $\mathit{ResponsibleDoctorOf}(d, p)$ means doctor d is the responsible doctor of patient p, and each patient has exactly one responsible doctor.

6.2.3 Disjointness Relations

Table 6.7 System Constraints for Disjointness Among Subject Subtypes in `careFacility`

Parent Type	Constraints for disjoint subtypes
Subject	$\forall s.\ \mathit{Staff}(s) \rightarrow \neg \mathit{Resident}(s)$
	$\forall s.\ \mathit{Staff}(s) \rightarrow \neg \mathit{Volunteer}(s)$
	$\forall s.\ \mathit{Resident}(s) \rightarrow \neg \mathit{Volunteer}(s)$
Staff	$\forall s.\ \mathit{VisitingDoctor}(s) \rightarrow \neg \mathit{Manager}(s)$
	$\forall s.\ \mathit{VisitingDoctor}(s) \rightarrow \neg \mathit{FrontDesk}(s)$
	$\forall s.\ \mathit{VisitingDoctor}(s) \rightarrow \neg \mathit{Cashier}(s)$
	$\forall s.\ \mathit{VisitingDoctor}(s) \rightarrow \neg \mathit{Nurse}(s)$
	$\forall s.\ \mathit{Manager}(s) \rightarrow \neg \mathit{FrontDesk}(s)$
	$\forall s.\ \mathit{Manager}(s) \rightarrow \neg \mathit{Cashier}(s)$
	$\forall s.\ \mathit{Manager}(s) \rightarrow \neg \mathit{Nurse}(s)$
	$\forall s.\ \mathit{Cashier}(s) \rightarrow \neg \mathit{Nurse}(s)$
Volunteer	$\forall s.\ \mathit{VolunteerForDailyCheckup}(s) \rightarrow \neg \mathit{VolunteerForArrangingMeeting}(s)$

We also need to specify which attribute is disjoint from another. In the care facility, all subtypes of *Subject* defined in Table 6.3 are disjoint from each other, as listed in Table 6.7. The first part lists the disjointness constraints for the *direct* subtypes of *Subject*. We do not require the type *EmergencyContact* to be disjoint from other subject subtypes. Thus, any subject can be listed as an emergency contact. The second part is the disjointness constraints for the subtypes of *Staff*, and the third part for the subtypes of *Volunteer*.

We omit the other disjointness constraints here. In summary, any two distinct subtypes of *Object* are disjoint from each other. The other types of entities *Disease*, *Time* and *Date* are disjoint from each other and from all the other type attributes.

In the nearby hospitals, *Doctor*, *Nurse*, and *FrontDesk* are disjoint from each other. Note that we do not require *Patient* to be disjoint from *Doctor*, *Nurse* or *FrontDesk* since any staff at the hospital can be a patient at the hospital as well.

6.2.4 Multi-system Constraints

Finally, we define the system constraints that use system attributes from different systems, as listed in Table 6.8. We define one such system constraint for each of the three systems in Scenario 1.1.

Table 6.8 System Constraints Involving Attributes from Different Systems

System	Constraint
careFacility	$\mathtt{c}_1 : \forall d.\ \mathit{VisitingDoctor}(d)$ $\rightarrow \mathit{InSys}(\uparrow \mathit{Doctor}(d), \mathtt{hospital}_1) \vee \mathit{InSys}(\uparrow \mathit{Doctor}(d), \mathtt{hospital}_2)$
$\mathtt{hospital}_1$	$\mathtt{c}_2 : \forall d.\ \mathit{Doctor}(d) \rightarrow \mathit{InSys}(\uparrow \mathit{Doctor}(d), \mathtt{hospital}_2)$
$\mathtt{hospital}_2$	$\mathtt{c}_3 : \forall d.\ \mathit{InSys}(\uparrow \mathit{Doctor}(d), \mathtt{hospital}_1) \rightarrow \mathit{Doctor}(d)$

The constraint $\mathtt{c}_1$ in the care facility is from Example 5.4, which means that every visiting doctor at the care facility is a doctor from either of the nearby hospitals. The two constraints $\mathtt{c}_2$ and $\mathtt{c}_3$ in $\mathtt{hospital}_1$ and $\mathtt{hospital}_2$, respectively, essentially express the same idea that any doctor in $\mathtt{hospital}_1$ is also a doctor in $\mathtt{hospital}_2$. However, since $\mathtt{c}_2$ is a constraint in $\mathtt{hospital}_1$, the constraint needs to specify that the *Doctor* attribute used on the right of the implication belongs to $\mathtt{hospital}_2$. On the other hand, since $\mathtt{c}_3$ is a constraint in $\mathtt{hospital}_2$, the *Doctor* attribute used on the left of the implication belongs to $\mathtt{hospital}_1$.

6.3 System Action Types

The system action types for the care facility are listed in Table 6.9.

The action types on subjects are creating and deleting a subject, i.e., *createSub* and *deleteSub*, and the action types on objects are creating, deleting, reading and writing an object, i.e., *createObj*, *readObj*, *writeObj*, and *deleteObj*. In Core RefABAC, we regard subjects and objects as mutually exclusive. A subject is an active entity that is able to perform an action in a system, such as a user of the system. While an object is an inactive entity. Thus, we assume that one cannot read or write a subject.

We also define the administrative action types for creating, deleting, and modifying any other entity that is not a subject or object. The possible action types one may perform on an attribute are *addAttr* and *deleteAttr*. Action $\mathit{addAttr}(s, \uparrow \mathit{Attr})$ (respectively, $\mathit{deleteAttr}(s, \uparrow \mathit{Attr})$) means subject s adds (respectively, deletes) attribute *Attr*. Note that when *Attr* is deleted, its attribute instances $\mathit{Attr}(\vec{x})$ need to be deleted as well. We will discuss how to achieve it using transition rules in the next section (Section 6.4).

The action types one may perform on action types are similar to the action types on attribute. Action $\mathit{addActionType}(s, \uparrow \mathit{act})$ (respectively, $\mathit{deleteActionType}(s, \uparrow \mathit{act})$) means subject s adding (re-

Table 6.9 System Action Types in `careFacility`

Regular AC Action Types	
On subjects	*createSub*, *deleteSub*
On objects	*createObj*, *readObj*, *writeObj*, *deleteObj*
Administrative Action Types	
On attribute	*addAttr*, *deleteAttr*
On action types	*addActionType*, *deleteActionType*
On attribute instances	*readAttrInst*, *modifyAttrInst*, *addAttrInst*, *deleteAttrInst*
On constraint	*readConstraint*, *addConstraint*, *deleteConstraint*
On transition rules	*readTransition*, *addTransition*, *deleteTransition*
On policies	*readPolicy*, *addPolicy*, *deletePolicy*

spectively, deleting) an action type *act*. When an action type is deleted, all the its transition rules need to be deleted as well.

The types of actions a subject may perform on an attribute instance include reading, adding, deleting and modifying. Action $readAttrInst(s, \uparrow Attr(\vec{x}))$ means subject s reading attribute instance $Attr(\vec{x})$. For instance, action $readAttrInst(s, \uparrow ResponsibleDoctorOf(\texttt{bob}, \texttt{carol}))$ means s reading that `bob` is the responsible doctor of `carol`. The reading-attribute-instance action type is useful when a subject makes a request that asks the care facility who is the responsible doctor of `carol`. Action $addAttrInst(s, \uparrow Attr(\vec{x}))$ (respectively, $deleteAttrInst(s, \uparrow Attr(\vec{x}))$) represents an action of a subject adding (respectively, deleting) attribute instance $Attr(\vec{x})$, which means making $Attr(\vec{x})$ true (respectively, false) in our logical language. Finally, action $modifyAttrInst(s, \uparrow Attr(x_1, ..., x_n), \uparrow Attr(y_1, ..., y_n)$ means modifying attribute instance $Attr(x_1, ..., x_n)$ to $Attr(y_1, ..., y_n)$. In this dissertation, we do not specify which of the arguments can or cannot be modified. Given attribute instance $Meeting(r, d, t)$ that indicates a meeting time between a resident r and a doctor d is t. A subject s can perform action $modifyAttrInst(s, \uparrow Meeting(r, d, t), \uparrow Meeting(r, d, t'))$ to change resident r and doctor d's meeting time from t to t'. Logically speaking, $Meeting(r, d, t)$ becomes false, while $Meeting(r, d, t')$ becomes true.

We also define administrative action types for adding, deleting, and reading a policy. Action $addPolicy(s, \uparrow\phi)$ (respectively, $deletePolicy(s, \uparrow\phi)$) means subject s adding (respectively, deleting) a policy represented by the formula ϕ. If subject s knows the name `p` of policy ϕ, i.e., $\texttt{p} = \uparrow\phi$, but does not know the actual representation of the policy, then the subject may perform action $readPolicy(s, \texttt{p})$ to read the representation or definition of the policy with name `p`.

The action types on constraints and transition rules are similar to the ones on policies. Specifically, $addConstraint(s, \uparrow\phi)$ (respectively, $deleteConstraint(s, \uparrow\phi)$ and $readConstraint(s, \uparrow\phi)$) means subject s adds (respectively, deletes and reads) constraint ϕ. Similarly, $addTransition(s, \uparrow\phi, \uparrow act)$

(respectively, $deleteTransition(s, \uparrow\phi, \uparrow act)$ and $readTransition(s, \uparrow\phi, \uparrow act)$) means subject s adds (respectively, deletes and reads) transition rule ϕ of action type act.

6.4 System Transition Rules

We divide our discussion of system transition rules based on the categories we used for system action types in the previous section.

6.4.1 For Action Types On Subject and Object

Table 6.10 and Table 6.11 list the transition rules for the action types on subjects and objects respectively. We also define a name n for each transition rule ϕ such that $n = \uparrow\phi$.

Table 6.10 System Transition Rules in `careFacility` for Action Types on Subjects

Name	Transition Rule
`createSubRule`	$\forall r, s, s'.\ Granted(r) \wedge \neg Fulfilled(r) \wedge InSys(r, \texttt{careFacility})$ $\wedge\ RequestedAct(r, \uparrow createSub(s, s'))$ $\rightarrow [createSub(s, s')](Performed(\uparrow createSub(s, s')) \wedge Fulfilled(r)$ $\wedge\ Subject(s') \wedge InSys(s', \texttt{careFacility}))$
`deleteSubRule`	$\forall r, s, s'.\ Granted(r) \wedge \neg Fulfilled(r) \wedge InSys(r, \texttt{careFacility})$ $\wedge\ RequestedAct(r, \uparrow deleteSub(s, s'))$ $\rightarrow [deleteSub(s, s')](Performed(\uparrow deleteSub(s, s') \wedge Fulfilled(r)$ $\wedge\ \neg InSys(s', \texttt{careFacility}))$

For subject s to create a new subject s', we first assume that there exists an individual s' in the system. After s performs the action of creating s' as a new subject, the corresponding request becomes fulfilled, s' is added as a subject in the system for the care facility. On the other hand, after subject s deletes subject s', the corresponding request becomes fulfilled, and `careFacility` no longer has s' in its system, i.e., $\neg InSys(s', \texttt{careFacility})$. Note that we do not have $\neg Subject(s')$ as a consequence of *deleteSub*. The reason is that even though s' does not exist in `careFacility` anymore, s' may still be a subject. This is useful for recording keeping.

The transition rules for creating and deleting an object are similar to those for creating and deleting a subject. Given an individual o in the system, after subject s creates o as an object, the corresponding request becomes fulfilled, o is added as an object in `careFacility`. If s deletes object o, then o no longer exists in `careFacility`.

Similar to deleting a subject, o is still a object in the system, i.e., we do not have $\neg Object(o)$

Table 6.11 System Transition Rules in `careFacility` for Action Types on Objects

Name	Transition Rule
`createObjRule`	$\forall r, s, o.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{createObj}(s, o))$ $\rightarrow [\mathit{createObj}(s, o)](\mathit{Performed}(\uparrow \mathit{createObj}(s, o)) \wedge \mathit{Fulfilled}(r)$ $\wedge\ \mathit{Object}(o) \wedge \mathit{InSys}(o, \texttt{careFacility}))$
`DeleteObjRule`	$\forall r, s, o.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{deleteObj}(s, o))$ $\rightarrow [\mathit{deleteObj}(s, o)](\mathit{Performed}(\uparrow \mathit{deleteObj}(s, o)) \wedge \mathit{Fulfilled}(r)$ $\wedge\ \neg \mathit{InSys}(o, \texttt{careFacility}))$
`ReadObjRule`	$\forall r, s, o.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{readObj}(s, o))$ $\rightarrow [\mathit{readObj}(s, o)](\mathit{Performed}(\uparrow \mathit{readObj}(s, o)) \wedge \mathit{Fulfilled}(r))$
`WriteObjRule`	$\forall r, s, o.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{writeObj}(s, o))$ $\rightarrow [\mathit{writeObj}(s, o)](\mathit{Performed}(\uparrow \mathit{writeObj}(s, o)) \wedge \mathit{Fulfilled}(r))$

as a consequence. Moreover, we also have transition rules for reading and writing an object. In this formalism, the only consequence after reading or writing an object is that the corresponding request is fulfilled. More complicated transition rules may also record the subject who has most recently read or written an object. We omit formal representations of the complicated cases as they are straightforward extension of the transitions rules here.

6.4.2 For Action Types On Attribute and Action Types

The transition rules for creating and deleting an attribute are defined in Table 6.12. After subject s adds an attribute with name a, i.e., $\mathit{addAttr}(s, a)$, $\downarrow a$ is recorded as an attribute in `careFacility`, i.e., $\mathit{Attribute}(a)$ and $\mathit{InSys}(a, \texttt{careFacility})$.

On the other hand, the transition rule for deleting an attribute requires the arity of the deleted attribute because all of the attribute instances of the deleted attribute need to be deleted as well. Let *Attr* be a binary attribute with name a and $\mathit{Attr}(x, y)$ be any attribute instance of *Attr*. After a subject deletes *Attr*, *Attr* and $\mathit{Attr}(x, y)$ no longer exist in `careFacility`. Complications arise when deleting an attribute. For instance, if a policy's pre-condition is expressed using a deleted attribute, the policy will never become applicable for any requests. In this case, one may need to manually revise the policy. More sophisticated transition rules can be written to handle the situation, e.g., by

Table 6.12 System Transition Rules in `careFacility` for Action Types on Attributes

Name	Transition Rule
`addAttrRule`	$\forall r, s, a.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{addAttr}(s, a))$ $\rightarrow [\mathit{addAttr}(s, a)](\mathit{Performed}(\mathit{addAttr}(s, a)) \wedge \mathit{Fulfilled}(r)$ $\wedge\ \mathit{Attribute}(a) \wedge \mathit{InSys}(a, \texttt{careFacility}))$
`deleteAttrRule`	$\forall r, s, a, x, y.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{deleteAttr}(s, a))$ $\wedge (a = \uparrow \mathit{Attr}) \wedge \mathit{Attr}(x, y)$ $\rightarrow [\mathit{deleteAttr}(s, a)](\mathit{Performed}(\mathit{deleteAttr}(s, a)) \wedge \mathit{Fulfilled}(r)$ $\wedge\ \neg \mathit{InSys}(a, \texttt{careFacility})$ $\wedge\ \neg \mathit{InSys}(\uparrow \mathit{Attr}(x, y), \texttt{careFacility}))$

deleting all the policies that use the deleted attribute.

Table 6.13 System Transition Rules in `careFacility` for Action Types on Action Types

Name	Transition Rule
`addActionTypeRule`	$\forall r, s, a.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{addActionType}(s, a))$ $\rightarrow [\mathit{addActionType}(s, a)](\mathit{Performed}(\uparrow \mathit{addActionType}(s, a)$ $\wedge\ \mathit{Fulfilled}(r) \wedge \mathit{ActionType}(a)$ $\wedge\ \mathit{InSys}(a, \texttt{careFacility}))$
`deleteActionTypeRule`	$\forall r, s, a, t.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{deleteActionType}(s, a)) \wedge \mathit{TransitionOf}(t, a)$ $\rightarrow [\mathit{deleteActionType}(s, a)](\mathit{Fulfilled}(r)$ $\wedge\ \mathit{Performed}(\uparrow \mathit{deleteActionType}(s, a))$ $\wedge\ \neg \mathit{InSys}(a, \texttt{careFacility}) \wedge \neg \mathit{InSys}(t, \texttt{careFacility}))$ $\wedge\ \neg \mathit{InSys}(\uparrow \mathit{TransitionOf}(t, a), \texttt{careFacility})$

The transition rules for creating and deleting an action type are defined similarly (Table 6.12). After adding action type a, $\uparrow a$ is recorded as an action type in `careFacility`. As for deleting action type *act* with name a, *act* is no longer in `careFacility`, and each transition rule t of *act* is deleted as well. Specifically, the system does not record t as a transition rule of a, i.e., $\neg \mathit{InSys}(\uparrow \mathit{TransitionOf}(t, a), \texttt{careFacility})$, and t does not exists in the care facility any more,

i.e., $\neg InSys(t, \texttt{careFacility})$.

6.4.3 For Action Types On Attribute Instance

Table 6.14 lists one way of defining transition rules for the action types on attribute instances. The transition rules in the table are for binary attributes only, for the reason we will discuss later in the section. However, one may easily adapt them to accommodate attributes of other arities.

Table 6.14 System Transition Rules in `careFacility` for Action Types on Attribute Instance

Name	Transition Rule
`addAttrInstRule`	$\forall r, s, a, x, y.\ Granted(r) \wedge \neg Fulfilled(r) \wedge InSys(r, \texttt{careFacility})$ $\wedge\ RequestedAct(r, \uparrow addAttrInst(s, \uparrow Attr(x, y)))$ $\wedge (a = \uparrow Attr) \wedge InSys(a, \texttt{careFacility})$ $\rightarrow [addAttrInst(s, \uparrow Attr(x, y))](Fulfilled(r)$ $\wedge\ Performed(\uparrow addAttrInst(s, \uparrow Attr(x, y)))$ $\wedge\ InSys(\uparrow Attr(x, y), \texttt{careFacility}))$
`deleteAttrInstRule`	$\forall r, s, a, x, y.\ Granted(r) \wedge \neg Fulfilled(r) \wedge InSys(r, \texttt{careFacility})$ $\wedge\ RequestedAct(r, \uparrow deleteAttrInst(s, \uparrow Attr(x, y)))$ $\rightarrow [deleteAttrInst(s, \uparrow Attr(x, y))](Fulfilled(r)$ $\wedge\ Performed(\uparrow deleteAttrInst(s, \uparrow Attr(x, y)))$ $\wedge\ \neg InSys(\uparrow Attr(x, y), \texttt{careFacility}))$
`modifyAttrInstRule`	$\forall r, s, a, x, y, y'.\ Granted(r) \wedge \neg Fulfilled(r) \wedge InSys(r, \texttt{careFacility})$ $\wedge\ RequestedAct(r, \uparrow modifyAttrInst(s, \uparrow Attr(x, y), \uparrow Attr(x, y')))$ $\wedge (y \neq y') \wedge (a = \uparrow Attr) \wedge InSys(\uparrow Attr(x, y), \texttt{careFacility})$ $\rightarrow [modifyAttrInst(s, \uparrow Attr(x, y), \uparrow Attr(x, y'))](Fulfilled(r)$ $\wedge\ Performed(\uparrow modifyAttrInst(s, \uparrow Attr(x, y), \uparrow Attr(x, y')))$ $\wedge\ \neg InSys(\uparrow Attr(x, y), \texttt{careFacility})$ $\wedge\ InSys(\uparrow Attr(x, y'), \texttt{careFacility}))$
`readAttrInstRule`	$\forall r, s, a, x.\ Granted(r) \wedge \neg Fulfilled(r) \wedge InSys(r, \texttt{careFacility})$ $\wedge\ RequestedAct(r, \uparrow readAttrInst(s, \uparrow Attr(x))) \wedge (a = \uparrow Attr)$ $\wedge\ InSys(a, \texttt{careFacility})$ $\rightarrow [readAttrInst(s, \uparrow Attr(x))](Fulfilled(r)$ $\wedge\ Performed(\uparrow readAttrInst(s, \uparrow Attr(x))))$

To add an attribute instance $Attr(x, y)$, an important condition is that the attribute $Attr$ to which the instance belongs must exist, i.e., $InSys(a, \texttt{careFacility})$, where $a = \uparrow Attr$. Then after the action of adding the instance is performed, the instance is recorded in the system. Deleting an attribute

instance $Attr(x, y)$ is similar to adding one. After $Attr(x, y)$ is deleted, it no longer exists in the system. Modifying an attribute instance from $Attr(x, y)$ to $Attr(x, y')$ $(y \neq y')$ is a combination of adding $Attr(x, y)$ and deleting $Attr(x, y')$. However, we require $Attr(x, y)$ to be defined in the system when modifying it. Otherwise, the transition rule would simply add $Attr(x, y')$, but adding an attribute instance and modifying one might have different applicable policies.

Reading an attribute instance is tricky. First, we need to be clear on what it means to read an attribute instance. In this dissertation, a subject s reading an attribute instance $Attr(x, y)$ means given an attribute *Attr* and an entity x, s "knows" that the value of *Attr* for x is y. We assume that for each binary attribute *Attr*, there is a corresponding unary attribute with the same symbol *Attr*, such that

$$\forall x, y.\ Attr(x, y) \rightarrow Attr(x).$$

Then in the transition rule for reading an attribute instance, we use the unary *Attr* instead. The unary *Attr* acts in a similar way as a function such that when given an entity x, it is able to return the values of *Attr* for y. The usage of the unary *Attr* is clearer when we add information disclosure to Core RefABAC (Chapter 8). The formalism can be easily adapted for a ternary attribute *Attr*. One can define a corresponding binary *Attr* or a unary *Attr*, depending on which arguments of *Attr* a subject can read.

6.4.4 For Action Types On Constraint, Policy and Transition Rule

Finally, we define the transition rules for action types on constraints, policies, and transition rules in Table 6.15, Table 6.16, and Table 6.17, respectively. The consequences for *reading* a constraint, a policy, and a transition rule are the same. Specifically, after such an action is performed, the corresponding request becomes fulfilled. For simplicity, we do not discuss other possible but more complicated consequences for these action types. A system may require additional information to be logged after a subject reads a policy. A system may also need to keep track of to which subjects it has disclosed the policy.

Table 6.15 System Transition Rules in `careFacility` for Action Types on Constraint

Name	Transition Rule
`readConstraintRule`	$\forall r, s, a.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{readConstraint}(s, a))$ $\rightarrow [\mathit{readConstraint}(s, a)](\mathit{Performed}(\uparrow \mathit{readConstraint}(s, a))$ $\wedge\ \mathit{Fulfilled}(r))$
`addConstraintRule`	$\forall r, s, a.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{addConstraint}(s, a))$ $\rightarrow [\mathit{addConstraint}(s, a)](\mathit{Performed}(\uparrow \mathit{addConstraint}(s, a))$ $\wedge\ \mathit{Fulfilled}(r) \wedge \mathit{Constraint}(a) \wedge \mathit{InSys}(a, \texttt{careFacility}))$
`deleteConstraintRule`	$\forall r, s, a.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{deleteConstraint}(s, a))$ $\rightarrow [\mathit{deleteConstraint}(s, a)](\mathit{Performed}(\uparrow \mathit{deleteConstraint}(s, a))$ $\wedge\ \mathit{Fulfilled}(r) \wedge \neg \mathit{InSys}(a, \texttt{careFacility}))$

After *adding* a constraint, a policy, or a transition rule, the corresponding request becomes fulfilled. In addition, the added constraint (respectively, policy and transition rule) is recorded as a *Constraint* (respectively, *Policy* and *TransitionRule*) in `careFacility`. Note that when adding a transition rule t, one also needs to specify with which action type a transition rule is associated so that after subject s performs the action of adding t, the system also records that t is a transition rule of a in `careFacility`.

Deleting a constraint, a transition rule or a policy is similar as well. After a subject adds a constraint with name a, the corresponding request becomes fulfilled. In addition, the deleted entity is no longer in the system.

Table 6.16 System Transition Rules in `careFacility` for Action Types on Policies

Name	Transition Rule
`readPolicyRule`	$\forall r, s, p.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{readPolicy}(s, p))$ $\rightarrow [\mathit{readPolicy}(s, p)](\mathit{Performed}(\uparrow \mathit{readPolicy}(s, p)) \wedge \mathit{Fulfilled}(r))$
`addPolicyRule`	$\forall r, s, p.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{addPolicy}(s, p))$ $\rightarrow [\mathit{addPolicy}(s, p)](\mathit{Performed}(\uparrow \mathit{addPolicy}(s, p)) \wedge \mathit{Fulfilled}(r)$ $\wedge\ \mathit{Policy}(p) \wedge \mathit{InSys}(p, \texttt{careFacility}))$
`deletePolicyRule`	$\forall r, s, p.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$ $\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{deletePolicy}(s, p))$ $\rightarrow [\mathit{deletePolicy}(s, p)](\mathit{Performed}(\uparrow \mathit{deletePolicy}(s, p)) \wedge \mathit{Fulfilled}(r)$ $\wedge\ \neg \mathit{InSys}(p, \texttt{careFacility}))$

Table 6.17 System Transition Rules in `careFacility` for Action Types on Transition Rules

Name	Transition Rule
`readTransRule`	$\forall r, s, t, a.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$
	$\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{readTransition}(s, t, a))$
	$\rightarrow [\mathit{readTransition}(s, t)](\mathit{Performed}(\uparrow \mathit{readTransition}(s, t))$
	$\wedge\ \mathit{Fulfilled}(r))$
`addTransRule`	$\forall r, s, t, a.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$
	$\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{addTransition}(s, t, a))$
	$\rightarrow [\mathit{addTransition}(s, t, a)](\mathit{Performed}(\uparrow \mathit{addTransition}(s, t, a))$
	$\wedge\ \mathit{Fulfilled}(r) \wedge \mathit{TransitionRule}(t) \wedge \mathit{InSys}(t, \texttt{careFacility})$
	$\wedge\ \mathit{TransitionOf}(t, a)$
	$\wedge\ \mathit{InSys}(\uparrow \mathit{TransitionOf}(t, a), \texttt{careFacility}))$
`deleteTransRule`	$\forall r, s, t, a.\ \mathit{Granted}(r) \wedge \neg \mathit{Fulfilled}(r) \wedge \mathit{InSys}(r, \texttt{careFacility})$
	$\wedge\ \mathit{RequestedAct}(r, \uparrow \mathit{deleteTransition}(s, t, a))$
	$\rightarrow [\mathit{deleteTransition}(s, t, a)](\mathit{Performed}(\uparrow \mathit{deleteTransition}(s, t, a))$
	$\wedge\ \mathit{Fulfilled}(r) \wedge \neg \mathit{InSys}(t, \texttt{careFacility})$
	$\wedge\ \neg \mathit{InSys}(\uparrow \mathit{TransitionOf}(t, a), \texttt{careFacility})$

6.5 System Policies and Motivating Examples

In this section, we formalize the policies used in the motivating examples Example 1.3 and Example 1.2.

6.5.1 Motivating Example: Sensitive Attributes

The motivating example for sensitive attributes (Example 1.3) is described as follows.

> The manager and the volunteers who are responsible for arranging meetings between residents and their responsible doctors are authorized to read the time of the meetings. The attending parties of a meeting are also authorized to read the time the meeting.

The example is about who are authorized to read the meeting times between a resident and his or her responsible doctor. We use an attribute $\mathit{Meeting}(r, d, t)$ to indicate that the meeting time

between a resident r and a doctor d is t. We also define a binary attribute *Meeting* such that

$$\forall x, y, t.\ \mathit{Meeting}(x, y, t) \rightarrow \mathit{Meeting}(x, y).$$

For details on the binary *Meeting*, see Subsection 6.4.3. Thus, reading the meeting times between a resident r and a doctor d means reading all the attribute instances $\mathit{Meeting}(r, d, t)$ that make $\mathit{Meeting}(r, d)$ true. The example can be formalized using the following two policies.

1. $\texttt{meetingPol}_1$: A manager or a volunteer who is responsible for arranging meetings is authorized to read the meeting times between any residents and doctors.

$$\begin{aligned}\forall r, s, x, y.\ &\mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{Requester}(r, s)\\ &\wedge \mathit{Manager}(s) \wedge \mathit{RequestedAct}(r, \uparrow \mathit{readAttrInst}(s, \uparrow \mathit{Meeting}(x, y)))\\ &\rightarrow [\mathit{authorizesRequest}(\texttt{meetingPol}_1, r)](\mathit{AuthorizedBy}(r, \texttt{meetingPol}_1)\\ &\quad \wedge \mathit{Authorized}(r))\end{aligned}$$

2. $\texttt{meetingPol}_2$: A resident (resp. a doctor) is authorized to read the meeting time between himself or herself and his or her doctor (resp. resident).

$$\begin{aligned}\forall r, s, x, y.\ &\mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{Requester}(r, s) \wedge ((s = x) \vee (s = y))\\ &\wedge \mathit{RequestedAct}(r, \uparrow \mathit{readAttrInst}(s, \uparrow \mathit{Meeting}(x, y)))\\ &\rightarrow [\mathit{authorizesRequest}(\texttt{meetingPol}_2, r)](\mathit{AuthorizedBy}(r, \texttt{meetingPol}_2)\\ &\quad \wedge \mathit{Authorized}(r))\end{aligned}$$

6.5.2 Motivating Example: Regulated Administrative Operations

The motivating example for regulated administrative operations (Example 1.2) is described as follows.

> A resident may change her or his own responsible doctor, but such a change requires the signatures of the manager.

We define an attribute *SignedBy* such that $\mathit{SignedBy}(r, s)$ means a request r is signed by a subject s. One may think of a request as a piece of paper that can be submitted. A manager or other subjects may sign it, depending on to whom the paper is submitted. We do not discuss the logistics here and simply assume a request can be signed. We use the following two policies `changeDoctorPolAut`

and changeDoctorPolPro, respectively, to formalize the example.

$$\begin{aligned}\forall r, s, d_1, d_2, x.\ &\mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{Requester}(r, s) \\ &\wedge \mathit{RequestedAct}(r, \uparrow \mathit{modifyAttrInst}(s, \uparrow \mathit{ResponsibleDoctorOf}(s, d_1), \\ &\qquad \uparrow \mathit{ResponsibleDoctorOf}(s, d_2))) \\ &\wedge (d_1 \neq d_2) \wedge \mathit{SignedBy}(r, x) \wedge \mathit{Manager}(x) \\ &\rightarrow [\mathit{authorizesRequest}(\texttt{changeDoctorPolAut}, r)](\mathit{Authorized}(r) \\ &\quad \wedge \mathit{AuthorizedBy}(r, \texttt{changeDoctorPolAut}))\end{aligned}$$

$$\begin{aligned}\forall r, s, d_1, d_2, x.\ &\mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{Requester}(r, s) \\ &\wedge \mathit{RequestedAct}(r, \uparrow \mathit{modifyAttrInst}(s, \uparrow \mathit{ResponsibleDoctorOf}(s, d_1), \\ &\qquad \uparrow \mathit{ResponsibleDoctorOf}(s, d_2))) \\ &\wedge (d_1 \neq d_2) \wedge \neg \mathit{SignedBy}(r, x) \wedge \mathit{Manager}(x) \\ &\rightarrow [\mathit{prohibitsRequest}(\texttt{changeDoctorPolPro}, r)](\mathit{Prohibited}(r) \\ &\quad \wedge \mathit{ProhibitedBy}(r, \texttt{changeDoctorPolPro}))\end{aligned}$$

6.6 Conclusion

We formalized multiple example attributes, action types and policies along with relevant constraints and transition rules from Scenario 1.1. We defined subtypes of *Subject* and *Object*. We also defined attributes to indicate the owner of an object, the responsible doctor of a resident, and the emergency contact of a resident, etc.. The system constraints for the attributes specify the types of the arguments, the number restrictions, the subtype and disjointness relations of the example attributes. We also showed some example constraints that use system attributes from different systems. We covered regular AC action types as well as administrative action types. We defined the transition rules for action types on the basic types of entities, including subjects, objects, attributes, attribute instances, action types, constraints, policies, and transition rules. Finally, we formalized example policies from Example 1.3 and Example 1.2.

CHAPTER

7

SAMPLE IMPLEMENTATION OF CORE REFABAC IN SQL

A Core RefABAC model for m systems consists of one conceptual model and m system models, one for each of the systems. Each system contains the following basic types of entities: attributes, attribute instances, constraints, action types, actions, transition rules, and policies. An *attribute* describes a kind of property of entities and is represented as a predicate in the logical language DL^+. When the arguments of an attribute are filled, we say it is an *attribute instance*. For instance, *Age* is an attribute describing the age property of subjects, while *Age*(`bob`, 30) is an attribute instance indicating that the age of `bob` is 30. A *constraint* expresses the relationships among the attributes and it is represented as a formula without actions. An action type specifies the kinds of actions that can be performed by a subject in a system. When an *action type* is instantiated, we have an *action* and what the action does depends on the transition rules defined for its corresponding action type. A *transition rule* is represented as a formula in DL^+ with actions. A *policy* is a DL^+ formula where given an antecedent, it evaluates whether a request should be authorized or prohibited.

In this chapter, we provide a sample implementation of Core RefABAC in SQL. The basic idea of the implementation is that each system model maps to a database in the corresponding system. The *database* contains all the information that the system model has. A *state of the database* corresponds to a system state. For simplicity, we assume each system's database has only one schema. Each attribute or action type maps to a table. An attribute instance or an action maps to a row in the corresponding table. A constraint maps to an assertion. A transition rule or policy maps to a trigger.

We discuss the details of the implementation with examples in the rest of the chapter. The

examples in the chapter are implemented in MySQL 5.7 on MacOS. [1]

7.1 Attributes and Attribute Instances in SQL

An **attribute** *Attr* of arity n is implemented as a table Attr with $n+1$ columns, with the extra column serving as the primary key. [2] An **attribute instance** $Attr(\vec{c})$ of *Attr* maps to a row in the table Attr. In terms of our formalism for Core RefABAC, the value in the first column is the name or reference of the attribute instance represented by the corresponding row.

Example 7.1. The binary attribute *ResponsibleDoctorOf* indicates the responsible doctor of a resident at the care facility. The attribute is implemented as a table ResponsibleDoctorOf with three columns. We use the first column as the primary key, and call it inst_name. The second column corresponds to the first argument of *ResponsibleDoctorOf*, and the third column the second argument. An attribute instance *ResponsibleDoctorOf*(bob, carol) maps to the row (id, bob, carol) in the table.

In the above example, the row (id, bob, carol) in the ResponsibleDoctorOf table corresponds to the attribute instance *ResponsibleDoctorOf*(bob, carol). In terms of our DL$^+$ formalism, id serves as the name of the attribute instance. Let id be the corresponding symbol for id in DL$^+$. Then we have id $=\uparrow$*ResponsibleDoctorOf*(bob, carol).

The extra column in the table for an attribute allows us to represent names of attribute instances and to refer to them elsewhere. The unary attribute *AttributeInst* maps to table AttributeInst, with two columns. This table contains a row with values (id', id), where id is the unique identifier for row (id, bob, carol) in Example 7.1. The unique identifier id' serves as the name for the attributes instance *AttributeInst*($\uparrow$*ResponsibleDoctorOf*(bob, carol)).

Table 7.1 lists the schema for the tables of the conceptual attributes. Subject(inst_name, subject) refers to the table that corresponds to the *Subject* attribute. The first column refers to the attribute instances of *Subject*, while the second column corresponds to the only argument of *Subject*. The table can be created using the following SQL statement. [3]

```
CREATE TABLE Subject (
  inst_name varchar(255) PRIMARY KEY,
  subject varchar(255) NOT NULL);
```

[1] For more information on MySQL 5.7, please refer to https://dev.mysql.com/doc/refman/5.7/en/.

[2] We use the same symbol but different fonts in the DL$^+$ formalism and SQL implementation for the same entity.

[3] In real-world database implementations of a Core RefABAC model, the first column is usually of the integer type. Thus, when adding the AUTO_INCREMENT constraint on the column, the primary key can be automatically generated. For the purpose of readability, we use a list of characters instead.

Table 7.1 Database Schema for Core RefABAC Conceptual Attributes

Category Name	Schema
Common Types	System(inst_name, system) Subject(inst_name, subject) Object(inst_name, object) Attribute(inst_name, attribute_name) AttributeInst(inst_name, attributeInst_name) Policy(inst_name, policy_name) Constraint(inst_name, constraint_name) Request(inst_name, request) ActionType(inst_name, actionType_name) Action(inst_name, action_name) TransitionRule(inst_name, transitionRule_name)
Request Parameters	Requester(inst_name, request, subject) RequestedAct(inst_name, request, action_name) AuthorizedByPolicy(inst_name, request, policy_name) ProhibitedByPolicy(inst_name, request, policy_name)
Request Status	Requested(inst_name, request) Granted(inst_name, request) Denied(inst_name, request) Fulfilled(inst_name, request) Active(inst_name, request) Authorized(inst_name, request) Prohibited(inst_name, request)
Action Status	Performed(inst_name, action_name)
Other Properties	TransitionOf(inst_name, transitionOf_name, actionType_name) In(inst_name, entity_name, system)

7.2 Constraints in SQL

We implement constraints using the CREATE ASSERTION statement in SQL. [4] To implement a constraint, we first need to *normalize* it. The procedure is described in the following.

Definition 7.1. Let $\forall \vec{x}.\ \phi \rightarrow \exists \vec{y}.\ \psi$ be a constraint in a Core RefABAC model, where ϕ and ψ are Boolean combinations of attribute instances. The normalization steps are as follows:

1. convert ϕ to its disjunctive normal form (DNF) [Wik20b], i.e., a formula of the form $(P_{11} \wedge ... \wedge P_{1b}) \vee ... \vee (P_{a1} \wedge ... \wedge P_{ab})$, where each P_{ij} is one of the attributes used in ϕ;
2. convert ψ to its conjunctive normal form (CNF) [Wik20a], i.e., a formula of the form $(Q_{11} \vee ... \vee Q_{1b'}) \wedge ... \wedge (Q_{a'1} \vee ... \vee Q_{a'b'})$, where each Q_{ij} is one of the attributes used in ψ;

[4] SQL also has "constraints", and assertions are special kinds of SQL constraints. When the context is unclear, we write *(SQL) constraint* to refer to a SQL constraint, and *(RefABAC) constraint* to refer to a constraint in Core RefABAC.

3. for each conjunct $P_{i1} \wedge ... \wedge P_{ij}$ in the converted ϕ and for each clause $Q_{i'1} \vee ... \vee Q_{i'j'}$ in the converted ψ, construct the formula $\forall \vec{x}.\, P_{i1} \wedge \ldots \wedge P_{ij} \rightarrow \exists \vec{y}.\, Q_{i'1} \vee ... \vee Q_{i'j'}$.

The normalization process can be done during pre-processing instead of during runtime. We understand the normalization process is likely to cause blow-up in the size of the database.

How a formula

$$\forall \vec{x}.\, P_1 \wedge \ldots \wedge P_a \rightarrow \exists \vec{y}.\, Q_1 \vee \ldots \vee Q_{a'}$$

is implemented as an assertion depends on whether it contains existentially quantified variables. Suppose first that the formula does not have any existential quantifiers, i.e., the formula is of the form

$$\forall \vec{x}.\, P_1 \wedge \ldots \wedge P_a \rightarrow Q_1 \vee \ldots \vee Q_{a'}.$$

The basic idea of implementing the assertion is (1) to select all the common rows from the tables that correspond to the attributes P_1, ..., P_a; and (2) to check if *all* of them (or more precisely, the corresponding arguments in each of them), appear in the union of the tables for the attributes Q_1, ..., $Q_{a'}$. We use

$$\forall x, y.\, P_1(x,y) \wedge P_2(x,y) \rightarrow Q_1(x,y) \vee Q_2(x,y)$$

as an example. Recall that each attribute of arity n is implemented as a table with $n+1$ columns. Therefore, we have four tables: P_1, P_2, Q_1, and Q_2, where each table has three columns. We call the first column id; it is the primary key. The second column corresponds to the first argument of an attribute; we denote it fst_arg. The third column corresponds to the second argument; we denote it snd_arg. Then the assertion statement is written as

```
CREATE ASSERTION example
CHECK NOT EXISTS
  (SELECT P1.fst_arg, P1.snd_arg, P2.fst_arg, P2.snd_arg FROM P1, P2
   WHERE P1.fst_arg = P2.fst_arg
     AND P1.snd_arg = P2.snd_arg
     AND (P1.fst_arg, P1.snd_arg) NOT IN
       (SELECT fst_arg, snd_arg FROM Q1
        UNION
        SELECT fst_arg, snd_arg FROM Q2));
```

The assertion can be informally described as there not existing a row that is in both P_1 and P_2, but not in either Q_1 or Q_2.

In the case where the consequent contains existential quantifiers, the SQL assertion statement is similar to the previous construction but has extra conditions. For example, the (normalized)

constraint

$$\forall x, y.\, P_1(x,y) \wedge P_2(x,y) \rightarrow \exists z.\, Q_1(x,z) \vee Q_2(y,z)$$

is implemented as

```
CREATE ASSERTION example
CHECK NOT EXISTS
  (SELECT P1.fst_arg, P1.snd_arg, P2.fst_arg, P2.snd_arg FROM P1, P2
   WHERE P1.fst_arg = P2.fst_arg
   AND P1.snd_arg = P2.snd_arg
   AND (P1.fst_arg, P1.snd_arg) NOT IN
     (SELECT fst_arg, snd_arg FROM Q1 WHERE Q1.fst_arg = P1.fst_arg
      UNION
      SELECT fst_arg, snd_arg FROM Q2 WHERE Q2.fst_arg = P1.snd_arg));
```

The CREATE ASSERTION statement says that among all of the common rows in the tables for P_1 and P_2, there cannot exists at least one matching row either in Q_1, with the first argument equal to the first one in P_1, or in Q_2, with the first argument equal to the second one in P_1.

7.2.1 Constraints for Subtype Relations

In Table 5.2, we listed the constraints expressing the subtype relations among the conceptual attributes. Each of the constraints is a DL$^+$ formula of the form

$$\forall x.\, \mathit{Type}_1(x) \rightarrow \mathit{Type}_2(x).$$

Type_1 and Type_2, respectively, map to tables Type1 and Type2, each with two columns. We translate the constraint to an assertion using

```
CREATE ASSERTION type
CHECK NOT EXISTS
  (SELECT fst_arg FROM Type1
   WHERE Type1.fst_arg NOT IN
     (SELECT fst_arg FROM Type2));
```

Assertions of this simple form can also be implemented as *foreign key constraints* in SQL using

```
ALTER TABLE Type1
ADD FOREIGN KEY (fst_arg) REFERENCES Type2(fst_arg);
```

However, when defining a (SQL) constraint, one needs to specify on which table it is defined. Some (RefABAC) constraints do not provide a clear picture of to the table(s) to which they should belong. So we use assertions instead as a general implementation of (RefABAC) constraints. The following assertion shows how to implement in SQL the conceptual constraint

$$\forall x.\, \mathit{System}(x) \rightarrow \mathit{Subject}(x).$$

```
CREATE ASSERTION systemSubjectSubtype
CHECK NOT EXISTS
  (SELECT system FROM System
   WHERE system NOT IN
     (SELECT subject FROM Subject));
```

7.2.2 Constraints for Argument Types

The conceptual constraint that are used to define the types of arguments of a conceptual attribute with arity n ($n > 1$) (Table 5.3) is a formula of the form

$$\forall x, y.\ Attr(x, y) \rightarrow Type_1(x) \wedge Type_2(y),$$

which is normalized to two formulas:

$$\forall x, y.\ Attr(x, y) \rightarrow Type_1(x)$$
$$\forall x, y.\ Attr(x, y) \rightarrow Type_2(y).$$

The normalized formulas can be implemented in SQL as follows:

```
CREATE ASSERTION argumentType
CHECK NOT EXISTS
  (SELECT fst_arg, snd_arg FROM Attr
   WHERE Attr.fst_arg NOT IN
    (SELECT fst_arg FROM Type1));
CREATE ASSERTION argumentType
CHECK NOT EXISTS
  (SELECT fst_arg, snd_arg FROM Attr
   WHERE Attr.snd_arg NOT IN
    (SELECT snd_arg FROM Type2));
```

For instance, the conceptual constraint

$$\forall r, s.\ Requester(r, s) \rightarrow Request(r) \wedge Subject(s)$$

is implemented as the following two assertions

```
CREATE ASSERTION requesterFirstArgumentType
CHECK NOT EXISTS
  (SELECT request FROM Requester
   WHERE Requester.request NOT IN
    (SELECT request FROM Request));
CREATE ASSERTION requesterSecondArgumentType
CHECK NOT EXISTS
  (SELECT subject FROM Requester
   WHERE Requester.subject NOT IN
    (SELECT subject FROM Subject));
```

7.2.3 Constraints for Disjoint Relations

Table 5.4 lists the conceptual constraints that specify which pairs of conceptual attributes are disjoint. Such a constraint is written as a formula of the form

$$\forall x.\ Type_1(x) \rightarrow \neg Type_2(x).$$

The corresponding SQL statement is as follows:

```
CREATE ASSERTION disjoint
CHECK NOT EXISTS
  (SELECT Type1.fst_arg, Type2.fst_arg FROM Type1, Type2
   WHERE Type1.fst_arg = Type2.fst_arg);
```

meaning that Type1 and Type2 do not have common rows. For instance, since subjects and objects are disjoint in Core RefABAC, we have

```
CREATE ASSERTION disjoint
CHECK NOT EXISTS
  (SELECT Subject.subject, Object.object FROM Subject, Object
   WHERE Subject.subject = Object.object);
```

7.2.4 Constraints for Number Restrictions

The "at-least-one number" restriction is in one of the following two forms:

$$\forall x.\ P(x) \rightarrow \exists y.\ Attr(x, y),$$
$$\forall x.\ P(x) \rightarrow \exists y.\ Attr(y, x).$$

The first one is implemented as

```
CREATE ASSERTION atLeastOneR
CHECK NOT EXISTS
  (SELECT fst_arg FROM P
   WHERE P.fst_arg NOT IN
         (SELECT fst_arg FROM Attr));
```

meaning that there does not exist a value that is in the table for P but does not show up in any rows of $Attr$. Similarly, the second kind of the "at-least-one" constraint is implemented as

```
CREATE ASSERTION atLeastOneL
CHECK NOT EXISTS
  (SELECT fst_arg FROM P
   WHERE P.fst_arg NOT IN
         (SELECT snd_arg FROM Attr));
```

For example, the conceptual constraint that says that each requested request must have at least one requester, i.e.,

$$\forall r.\ Requested(r) \rightarrow \exists s.\ Requester(r, s),$$

translates to

```
CREATE ASSERTION atLeastOneRequester
CHECK NOT EXISTS
  (SELECT request FROM Requested
   WHERE Requested.request NOT IN
         (SELECT request FROM Requester));
```

The "at-most-one" number restriction is in one of the two forms

$$\forall x, y, y'.\ Attr(x, y) \wedge Attr(x, y') \rightarrow y = y',$$
$$\forall x, x', y.\ Attr(x, y) \wedge Attr(x', y) \rightarrow x = x'.$$

We implement it using the respective statements

```
CREATE ASSERTION atMostOne
CHECK NOT EXISTS
  (SELECT * FROM Attr
   GROUP BY fst_arg
   HAVING COUNT(*) > 1);
CREATE ASSERTION atMostOne
CHECK NOT EXISTS
  (SELECT * FROM Attr
   GROUP BY snd_arg
   HAVING COUNT(*) > 1);
```

respectively. The first statement means that the number of rows that have the same value for the first argument is 1. Similarly, the second statement means that the number of rows that have the same value for the second argument is 1. For instance, in Core RefABAC, we specify that each request must have at most one requester, i.e.,

$$\forall r, s, s'.\ Requester(r, s) \wedge Requester(r, s') \rightarrow s = s'.$$

The constraint can be implemented as

```
CREATE ASSERTION atMostOneRequester
CHECK NOT EXISTS
  (SELECT * FROM Requester
   GROUP BY request
   HAVING COUNT(*) > 1);
```

7.3 Action Types and Actions in SQL

We store action types and actions in a similar way to what we did for attributes and attribute instances, respectively (Section 7.1). Specifically, an action type *act* of arity n corresponds to a table act with $n+1$ columns. Each row in the table corresponds to an action of the action type. The first

column represents the reference to the action, and the remaining n columns correspond to the n arguments of the action.

Example 7.2. The *readObj* system action type in `careFacility` has arity 2, such that *readObj*(s, o) means that subject s reads object o. It is implemented as table readObj with the schema

readObj(action_name, subject, object).

The SQL statement for creating the table is as follows: [5]

```
CREATE TABLE readObj (
  action_name varchar(255) NOT NULL,
  subject varchar(255) NOT NULL,
  object varchar(255) NOT NULL
);
```

Suppose the care facility has an action of `bob` reading `carol`'s medical records `carolMedRec`, i.e., *readObj*(`bob`, `carolMedRec`). The action is represented as a row in the readObj table with values (bobReadsObj, bob, carolMedRec). The value bobReadsObj serves as the unique identifier for the row. In our formalism for RefABAC, it corresponds to the name for the action that the row represents. In other words, `bobReadsObj` $= \uparrow$*readObj*(`bob`, `carolMedRec`), where `bobReadsObj` is the logical name in DL$^+$ that corresponds to the entity bobReadsObj in the SQL implementation. In this way, we can use the identifier to refer to the action.

Example 7.3. The *makesRequest* conceptual action type has arity 4, where *makesRequest*(s, r, sys, a) means that subject s makes request r to system *sys* to perform action with name a. The action type is implemented as a table makesRequest with the following schema:

makesRequest(action_name, subject, request, system, requested_action_name).

In the above example for the makesRequest table, the first column act_name stores the unique identifiers for the request-making actions, while the last column requested_action_name stores the identifiers for requested actions. For instance, suppose `bob` makes a request `r` to `careFacility` to perform the action *readObj*(`bob`, `carolMedRec`) with the name `bobReadsObj` mentioned above. Then the request-making action is represented as *makesRequest*(`bob`, `r`, `careFacility`, `bobReadsObj`) in our formalism, and we translate it to the row (bobMakesReq, bob, r, careFacility , bobReadsObj) in the makesRequest table, where bobMakesReq represents the id for the request-making action. In terms of DL$^+$, we have `bobMakesReq` $= \uparrow$*makesRequest*(`bob`, `r`, `careFacility`, `bobReadsObj`).

Similarly, the conceptual action types *grantsRequest* and *deniesRequest* translate to tables with

[5] Similar to implementing an attribute, the first column for the unique identifiers is often implemented as integers and assigned as the primary key for the table. For the purpose of readability, we use a list of characters instead in this dissertation.

the following respectively schemas.

grantsRequest(act, system, request)

deniesRequest(act, system, request)

We have discussed how each action is *recorded* in a database, but not how an action is actually performed and under what conditions an action can be performed. This is specified using transition rules in our formalism, which will be discussed in the next section.

7.4 Transition Rules in SQL

A transition rule for an action $act(\vec{y})$ is in the form of

$$\forall \vec{x}.\ \phi \rightarrow [act(\vec{y})]\psi,$$

where ϕ and ψ are conjunctions of attribute instances, and the variables in $\vec{y}$ are included in $\vec{x}$. We translate each transition rule into two SQL statements: an INSERT statement and a CREATE TRIGGER statement. The INSERT statement mimics the actual performance of the action in the system. The CREATE TRIGGER statement corresponds to the post-conditions *Post*. It states what changes should be made after the action is performed.

We illustrate how to translate a transition rule into SQL by using *makesRequest* as an example. The transition rule for *makesRequest* is restated as follows:

$$\begin{aligned}\forall s, r, sys, a.\ & Request(r) \wedge InSys(r, sys) \wedge \neg Requested(r) \\ & \rightarrow [makesRequest(s, r, sys, a)](Performed(\uparrow makesRequest(s, r, sys, a)) \\ & \quad \wedge Requested(r) \wedge Active(r) \wedge Requester(r, s) \wedge RequestedAct(r, a))\end{aligned}$$

To implement the transition rule in SQL, we first define an attribute *PerformedMakesRequets*, which is a subtype of *Performed* such that *PerformedMakesRequest*(a) means request-making action with name a is performed. We assume that the moment when a request-making action happens is the moment when the action information is inserted into the PerformedMakesRequest table. Thus, the INSERT statement is written as inserting the parameters of the action into PerformedMakesRequest. Suppose the care facility contains an action of `bob` making request `exReq` to the care facility to perform action with name `exAct`, i.e., *makesRequest*(`bob`, `exReq`, `careFacility`, `exAct`). Thus, makesRequest table contains a row with values

(bobMakesRequest, bob, exReq, careFacility , exAct),

where bobMakesRequest denotes the name for the request-making action. Then `bob` performing the action of making the request is analogous to inserting the action name bobMakesRequest into the

PerformedMakesRequest table using the following statement:

```
INSERT INTO PerformedMakesRequest(action_name)
VALUE ("bobMakesRequest");
```

After the corresponding value is successfully inserted into PerformedMakesRequest, we use a trigger to take care of the rest of the post-conditions in ψ. The trigger for *makesRequest* is created as follows.

```
delimiter |
CREATE TRIGGER makesRequest_trigger
AFTER INSERT ON PerformedMakesRequest FOR EACH ROW
BEGIN
  INSERT INTO Requested(request)
  SELECT request FROM makesRequest
  WHERE NEW.action_name = makesRequest.action_name;

  INSERT INTO Active(request)
  SELECT request FROM makesRequest
  WHERE NEW.action_name = makesRequest.action_name;

  INSERT INTO Requester(request, subject)
  SELECT request, subject FROM makesRequest
  WHERE NEW.action_name = makesRequest.action_name;

  INSERT INTO RequestedAct(request, action)
  SELECT request, requested_action_name FROM makesRequest
  WHERE NEW.action_name = makesRequest.action_name;
END;
|
delimiter ;
```

In a MySQL trigger, NEW refers to each row that is inserted into PerformedMakesRequest. Therefore, when the action is performed, i.e., when bobMakesRequest is inserted into PerformedMakesRequest, NEW .action_name is bobMakesRequest, and the values

```
(exReq),
(exReq),
(exReq, bob),
(exReq, exAct)
```

are inserted into the Requested, Active, Requester, and RequestedAct tables respectively.

Readers may notice that we did not check whether the pre-conditions for an action are satisfied. Specifically, we did not make sure the action for making a request has not already been performed when the name of the action is inserted into the PerformedMakesRequest table. The pre-condition can be enforced by simply adding a SQL constraint that says all values in PerformedMakesRequest are unique.

7.5 Policies in SQL

Recall that a policy pol is a DL^+ formula of one of the following two forms:

$$\forall r, \vec{x}.\, \mathit{Active}(r) \wedge \mathit{InSys}(r, \mathtt{sys}) \wedge \phi \\ \rightarrow [\mathit{authorizesRequest}(\mathtt{pol}, r)](\mathit{Authorized}(r) \wedge \mathit{AuthorizedBy}(r, \mathtt{pol}))$$

$$\forall r, \vec{x}.\, \mathit{Active}(r) \wedge \mathit{InSys}(r, \mathtt{sys}) \wedge \phi \\ \rightarrow [\mathit{prohibitsRequest}(\mathtt{pol}, r)](\mathit{Prohibited}(r) \wedge \mathit{ProhibitedBy}(r, \mathtt{pol})).$$

Based on the format, a policy is a restricted form of transition rule. Whenever a subject makes a request, a policy decides if the request should be authorized or prohibited and the result is *inserted* into the corresponding tables: AuthorizedByPolicy and Authorized if the request is authorized, and ProhibitedByPolicy and Prohibited if prohibited. Therefore, we implement policies in a similar way to how we implement transition rules. Specifically, we use an INSERT statement and a trigger. The INSERT statement inserts the request r and the corresponding policy name pol into AuthorizedByPolicy or ProhibitedByPolicy when the preconditions are satisfied. The trigger is triggered on the insertion, and inserts the request into the authorized or prohibited table, respectively.

For example, we have the following policy $\mathtt{meetingPol}_1$ from Subsection 6.5.1:

$$\forall r, s, x, y.\mathit{Active}(r) \wedge \mathit{InSys}(r, \mathtt{careFacility}) \wedge \mathit{Requester}(r, s) \\ \wedge \mathit{Manager}(s) \wedge \mathit{RequestedAct}(r, \uparrow \mathit{readAttrInst}(s, \uparrow \mathit{Meeting}(x, y))) \\ \rightarrow [\mathit{authorizesRequest}(\mathtt{meetingPol}_1, r)](\mathit{AuthorizedBy}(r, \mathtt{meetingPol}_1) \\ \wedge \mathit{Authorized}(r))$$

which states that a manager is authorized to read the meeting times between any residents and doctors. First, the Policy table has a row (attrInstName, meetingPol1) to record that the system has a policy named meetingPol1. Then the INSERT statement is defined as

```
INSERT INTO AuthorizedByPolicy(request, policy_name)
SELECT request, policy_name
FROM Requested, Policy
WHERE policy_name = "meetingPol1"
AND EXISTS
  (SELECT * FROM InSys
   WHERE InSys.entity_name = Requested.request
   AND InSys.system = "careFacility")
AND EXISTS
  (SELECT * FROM Requester, Manager
   WHERE Requester.request = Requested.request
   AND Requester.subject = Manager.manager)
AND EXISTS
  (SELECT * FROM Requester, RequestedAct, readAttrInst, MeetingTwo
   WHERE RequestedAct.request = Requested.request
   AND RequestedAct.action_name = readAttrInst.action_name
   AND readAttrInst.subject = Requester.subject
   AND readAttrInst.inst_name = MeetingTwo.inst_name);
```

The first AND EXISTS condition states that the request request is indeed a requested request and it exists in the care facility. The second AND EXISTS condition checks if the requester of request is a manager. The third one checks if the requested action is an action of reading an instance of the MeetingTwo attribute. Note that in the DL$^+$ formalism of the policy, we use the *derived binary* attribute *Meeting* where $Meeting(r,d)$ is true if there exists any t such that $Meeting(r,d,t)$ is true. In other words, the binary and ternary attributes for meetings have the same name. In the SQL implementation of the policy, we use MeetingTwo for the binary version to avoid confusion. Finally, when the insertion happens, the trigger for inserting the request req into Authorized is triggered. The trigger is defined as follows:

```
delimiter |
CREATE TRIGGER meetingPol1
ON AuthorizedByPolicy
AFTER INSERT
AS
BEGIN
  INSERT INTO Authorized(request)
  VALUES NEW.request;
END;
|
delimiter;
```

7.6 Conclusion

We discussed how to implement a Core RefABAC model in SQL. We also provided examples for each of the basic types of entities throughout the chapter.

In summary, we implement an n-ary attribute *Attr* as a table Attr with $n+1$ columns. The first column is the primary key of the table, and the remaining n columns map to the n arguments of *Attr*, respectively. The value in the first column for each row serves as the name or reference for the attribute instance that the row represents. For instance, the attribute instance *ResponsibleDoctorOf*(`bob`, `carol`) with name `bobDoctor` maps to row (bobDoctor, bob, carol) in the ResponsibleDoctorOf table.

We implement constraints using assertions. A constraint is first normalized to a set of DL^+ formulas. The antecedent of each formula is a conjunction of attribute instances, and the consequent is a disjunction of attribute instances. Then each formula resulting from the normalization is implemented as an assertion that checks if the arguments of the attributes involved in the formula match.

Action types are implemented in the same way as attributes. Each n-ary action type maps to a table with $n+1$ columns, and each action is a row in the corresponding table. The first column is the primary key of the table, which provides the names for the actions stored in the table.

A transition rule is implemented as two parts: one INSERT statement and one TRIGGER statement. A subject performing an action is analogous to inserting some value into a table. Then the TRIGGER statement is triggered on the INSERT statement, which handles the post-conditions of the transition rule. A policy is essentially a transition rule, and thus is implemented as an INSERT statement and a TRIGGER statement as well.

Direct implementation of everything in SQL might not be very efficient, even if a DB is needed to store all the standard attributes. For efficiency, one might need to do some pre-processing or compilation of policies so that one does not have to do chains of inference in processing individual queries (although whether that is necessary is itself unknown at this point). Adding new constraints would also likely require recompiling policies.

CHAPTER

8

DC REFABAC: CORE REFABAC WITH DISCLOSURE CONTROL

A major contribution of RefABAC is to allow AC actions on *any* entities in a system. Traditional ABAC models usually only allow AC actions of reading and writing an object, such as reading a file. Few of the current ABAC models allow addition AC actions of reading, adding, deleting, and modifying the value of an attribute for some entity [Kol07; Sma14], called an attribute value in our formalism.

In Chapter 5, we defined Core RefABAC as a baseline framework for the RefABAC family. In Core RefABAC, a subject can read, add, delete and modify any entities in a system. The decision making procedure of Core RefABAC is first, a subject makes a request to perform an action. Each applicable policy either authorizes or prohibits the request. If the request is prohibited by any policies or if it does not have any applicable policies, then the system denies the request. Otherwise, the system grants the request. Finally, if the request is granted, the requesting subject can perform the requested action.

However, performing actions on entities induces a privacy issue, as discussed in Chapter 1, and Core RefABAC does not address the issue. For instance, suppose the care facility in Scenario 1.1 contains a constraint that says a resident with serious heart diseases have Bob as his or her responsible doctor. If Frank knows about the constraint and knows that Carol has a serious heart disease, then Frank knows who is the responsible doctor of Carol.

In this chapter, we address the privacy issue by proposing an advanced version of Core RefABAC, **DC RefABAC** (Core RefABAC with Disclosure Control). DC RefABAC improves upon Core RefABAC by keeping track of what information has been disclosed to a subject. For instance, if a subject reads

an attribute value in a system, then we say the system discloses (the information stored in) the attribute value to the subject.

8.1 Formalism of DC RefABAC

The formalism of DC RefABAC is very similar to that of Core RefABAC. Thus, we only discuss the changes we make from Core RefABAC to get DC RefABAC.

The conceptual model of DC RefABAC adds a new conceptual attribute *DiscloseTo* to keep track of the disclosed information. $\mathit{DiscloseTo}(s, x, s')$ means a subject s has disclosed information x to another subject s', where x can be any entity, such as an object, an attribute instance, or a policy. For example, the attribute instance

$$\mathit{DiscloseTo}(\texttt{careFacility}, \uparrow \mathit{ResponsibleDoctorOf}(\texttt{bob}, \texttt{carol}), \texttt{alice})$$

represents the fact that the care facility has disclosed to Alice that the responsible doctor of Carol is Bob. We also add the following constraint for *DiscloseTo* to specify its argument types:

$$\forall x, y, z.\ \mathit{DiscloseTo}(x, y, z) \rightarrow \mathit{System}(x) \wedge \mathit{Subject}(z).$$

The decision procedure of DC RefABAC is the same as Core RefABAC. We repeat the diagram Figure 5.2 from Chapter 5 in Figure 8.1 for reference. When a subject makes a request, the request becomes requested and active. Each applicable policy either authorizes or prohibits the request. If the request is prohibited or is not authorized, the system denies the request. Otherwise, the system grants the request, after which the requested action may be performed, and the request is fulfilled.

The level of disclosure control would depend on the specific implementation of a system. In the examples shown in the rest of the chapter, we focus on the level of a system's knowledge base. In other words, we control the disclosure of the facts stored in a RefABAC system, rather than of the information contained in a file which is stored in the system.

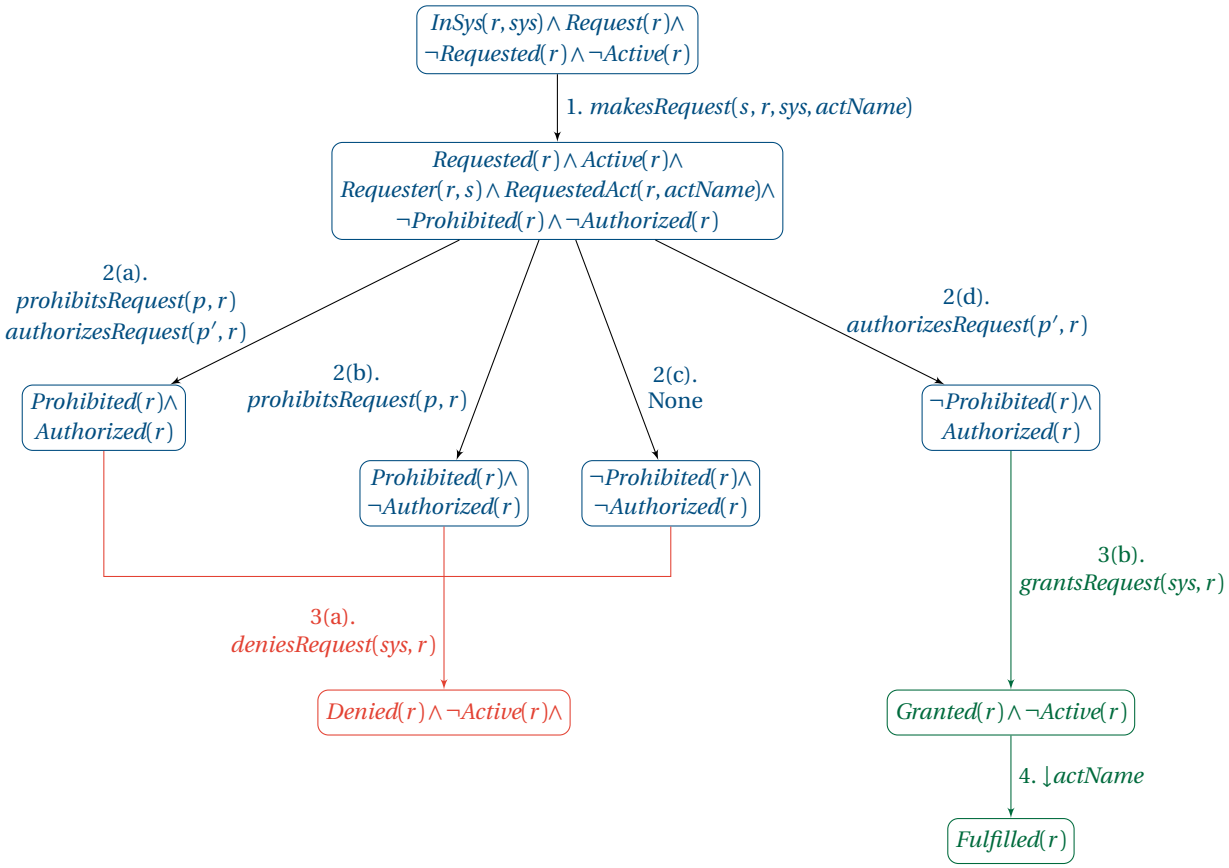

Figure 8.1 Decision Making Procedure in DC RefABAC

8.1.1 Usage of DC RefABAC in Transition Rules

The *DiscloseTo* attribute can be used in transition rules as an additional post-condition of actions. We take the transition rule for reading an attribute instance in the care facility as an example. The other transition rules can be modified in a similar way. Recall that the transition rule for reading the instances of a binary attribute *Attr* is defined as (Table 6.14)

$$\begin{aligned}\forall r, s, a, x.\, &Granted(r) \wedge \neg Fulfilled(r) \wedge InSys(r, \texttt{careFacility}) \\ &\wedge RequestedAct(r, \uparrow readAttrInst(s, \uparrow Attr(x))) \wedge (a = \uparrow Attr) \\ &\wedge InSys(a, \texttt{careFacility}) \\ &\rightarrow [readAttrInst(s, \uparrow Attr(x))](Fulfilled(r) \\ &\quad \wedge Performed(\uparrow readAttrInst(s, \uparrow Attr(x)))).\end{aligned}$$

Given an entity `ent`, when a subject `s` performs the action $readAttrInst(\texttt{s}, \uparrow Attr(\texttt{ent}))$, the system discloses to `s` all attribute instances $Attr(\texttt{ent}, y)$ of *Attr* for `ent`. Therefore, the updated transition

rule is defined as follows:

$$\forall r, s, a, x, y.\, Granted(r) \wedge \neg Fulfilled(r) \wedge InSys(r, \texttt{careFacility}) \tag{8.1}$$
$$\wedge\, RequestedAct(r, \uparrow readAttrInst(s, \uparrow Attr(x))) \wedge (a = \uparrow Attr) \tag{8.2}$$
$$\wedge\, InSys(\uparrow Attr(x, y), \texttt{careFacility}) \tag{8.3}$$
$$\rightarrow [readAttrInst(s, \uparrow Attr(x))](Fulfilled(r) \tag{8.4}$$
$$\wedge\, Performed(\uparrow readAttrInst(s, \uparrow Attr(x))) \tag{8.5}$$
$$\wedge DiscloseTo(\texttt{careFacility}, \uparrow Attr(x, y), s)). \tag{8.6}$$

The changes we made are colored red. In Equation 8.3, instead of requiring the attribute *Attr* to be in the system, we need the attribute instances $Attr(x, y)$ to be in the system. In Equation 8.3, we add the *DiscloseTo* part to indicate that `careFacility` discloses the attribute instances $\uparrow Attr(x, y)$ to the subject s.

8.1.2 Usage of DC RefABAC in Policies

We can also use *DiscloseTo* in the pre-conditions of a policy to control information disclosure. We discuss Example 1.5 in more detail and formalize it to illustrate the usage of DC RefABAC. We repeat Example 1.5 as follows for convenience.

Example 8.1. If a resident has heart diseases, then the resident's responsible doctor should be Bob, a doctor from one of the nearby hospitals. Only the manager, the resident and his or her responsible doctor are authorized to read who is the responsible doctor of the resident. If a non-manager subject s reads that a resident r has heart diseases, then s can infer who is r's responsible doctor.

The care facility in Scenario 1.1 contains a constraint that says if a resident has heart diseases, then the resident's responsible doctor is Bob, a doctor at $\texttt{hospital}_1$. We denote the name of the constraint as `heartDiseaseDoctor` , and the constraint is formalized as:

$$\forall x, d.\, Resident(x) \wedge HasDisease(x, d) \wedge HeartDisease(d) \rightarrow ResponsibleDoctorOf(\texttt{bob}, x).$$

The care facility also has a policy that says if a subject is not a manager, then he or she should not read who is the responsible doctor of a resident. We denote `responsibleDoctorPol` the name of the policy, and the policy is formalized as follows:

$$\forall r, s, x.\, Active(r) \wedge InSys(r, \texttt{careFacility}) \wedge Requester(r, s)$$
$$\wedge \neg Manager(s) \wedge RequestedAct(r, \uparrow readAttrInst(s, \uparrow ResponsibleDoctorOf(x)$$
$$\rightarrow [prohibitsRequest(\texttt{responsibleDoctorPol}, r)](Prohibited(r)$$
$$\wedge\, ProhibitedBy(r, \texttt{responsibleDoctorPol})).$$

Suppose a subject s reads constraint `heartDiseaseDoctor` and that s' has a heart disease.

Thus s can infer that the responsible of s' is `bob`. The care facility can define additional policies to prevent this from happening. First, we define a policy `dcPol1` in `careFacility` that says if the system has disclosed to a non-manager subject s who is the responsible doctor of another subject s', then the system cannot disclose constraint `heartDiseaseDoctor` to s. The policy is formalized as:

$$\begin{aligned}\forall r, s, s', d.\ & \mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{Requester}(r, s) \\ & \wedge \neg \mathit{Manager}(s) \wedge \mathit{RequestedAct}(r, \uparrow \mathit{readConstraint}(s, \texttt{heartDiseaseDoctor})) \\ & \wedge \mathit{DiscloseTo}(\texttt{careFacility}, \uparrow \mathit{ResponsibleDoctorOf}(s', d), s) \\ & \rightarrow [\mathit{prohibitsRequest}(\texttt{dcPol1}, r)](\mathit{Prohibited}(r) \wedge \mathit{ProhibitedBy}(r, \texttt{dcPol1})).\end{aligned}$$

Similarly, if the care facility has disclosed `heartDiseaseDoctor` to a non-manager subject s, then it cannot disclose to s who is the responsible doctor of the subject. We formalize the policy `dcPol2` as follows:

$$\begin{aligned}\forall r, s, s'.\ & \mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{Requester}(r, s) \\ & \wedge \neg \mathit{Manager}(s) \wedge \mathit{RequestedAct}(r, \uparrow \mathit{readAttrInst}(s, \uparrow \mathit{ResponsibleDoctorOf}(s'))) \\ & \wedge \mathit{DiscloseTo}(\texttt{careFacility}, \texttt{heartDiseaseDoctor}, s) \\ & \rightarrow [\mathit{prohibitsRequest}(\texttt{dcPol2}, r)](\mathit{Prohibited}(r) \wedge \mathit{ProhibitedBy}(r, \texttt{dcPol2})).\end{aligned}$$

8.1.3 Usage of DC RefABAC in Policy Explanation

Since a RefABAC model allows actions on reading any entities in a system, we can define an action type

$$\mathit{readDecisionExp}$$

such that $\mathit{readDecisionExp}(s, \mathit{sys}, r)$ means that subject s reads the explanation for system sys's decision on request r. What counts as a decision explanation depends on the system. In this dissertation, we say that the decision explanation for a request includes the policies *cause* the system to grant or deny the request. In the context of DC RefABAC, if request r is denied, the explanation includes all the policies used to *prohibit* r; if r is granted, it includes all the policies used to *authorize* r. The

transition rules for *readDecisionExp* are defined as follows: [1]

$$\begin{aligned}
\forall r_1, r_2, s, p.\ & \mathit{Granted}(r_1) \wedge \neg \mathit{Fulfilled}(r_1) \wedge \mathit{InSys}(r_1, \texttt{careFacility}) \\
& \wedge \mathit{RequestedAct}(r_1, \uparrow \mathit{readDecisionExp}(s, \texttt{careFacility}, r_2)) \\
& \wedge \mathit{Granted}(r_2) \wedge \mathit{AuthorizedBy}(r_2, p) \\
& \rightarrow [\mathit{readDecisionExp}(s, \texttt{careFacility}, r_2)](\mathit{Fulfilled}(r_1) \\
& \quad \wedge \mathit{Performed}(\uparrow \mathit{readDecisionExp}(s, \texttt{careFacility}, r_2)) \\
& \quad \wedge \mathit{DiscloseTo}(\texttt{careFacility}, p, s)
\end{aligned}$$

$$\begin{aligned}
\forall r_1, s, t, p.\ & \mathit{Granted}(r_1) \wedge \neg \mathit{Fulfilled}(r_1) \wedge \mathit{InSys}(r_1, \texttt{careFacility}) \\
& \wedge \mathit{RequestedAct}(r_1, \uparrow \mathit{readDecisionExp}(s, \texttt{careFacility}, r_2)) \\
& \wedge \mathit{Denied}(r_2) \wedge \mathit{ProhibitedBy}(r_2, p) \\
& \rightarrow [\mathit{readDecisionExp}(s, \texttt{careFacility}, r_2)](\mathit{Fulfilled}(r_1) \\
& \quad \wedge \mathit{Performed}(\uparrow \mathit{readDecisionExp}(s, \texttt{careFacility}, r_2)) \\
& \quad \wedge \mathit{DiscloseTo}(\texttt{careFacility}, p, s).
\end{aligned}$$

Let r_1 be a granted and unfulfilled request that is requested for subject s to read the decision explanation for another request r_2. If r_2 is granted and authorized by a policy with name p, then `careFacility` discloses p to s; if r_2 is denied and prohibited by a policy with p, then `careFacility` discloses p to s.

Moreover, we define a policy `decisionExpPol` that says any subject is authorized to read the explanations for the requests himself or herself has made. The policy is formalized as follows:

$$\begin{aligned}
\forall r_1, r_2, s.\ & \mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{Requester}(r_1, s) \\
& \wedge \mathit{RequestedAct}(r_1, \uparrow \mathit{readDecisionExp}(s, \texttt{careFacility}, r_2) \wedge \mathit{Requester}(r_1, s) \\
& \rightarrow [\mathit{authorizesRequest}(\texttt{decisionExpPol}, r_1)](\mathit{AuthorizedBy}(r_1, \texttt{decisionExpPol}) \\
& \quad \wedge \mathit{Authorized}(r_1)).
\end{aligned}$$

We use *readDecisionExp* to formalize Example 1.4, described as follows:

> Resident Carol makes a request to change her responsible doctor from Bob to George. Carol does not have the signature of the manager. According to the policies in Example 1.2, Carol's request is denied. Carol inquires about the reason for the denial and finds out that it is due the lack of the manager's signature.

The policies for changing the responsible doctor of a resident is defined in Subsection 6.5.2,

[1] The *readDecisionExp* action type can be also defined in Core RefABAC by deleting the parts of the transition rules involving *DiscloseTo*, which would demonstrate the usefulness of the *readDecisionExp* action type as well. Thus, we delay the discussion of *readDecisionExp* till this chapter.

repeated as follows for convenience:

$$\begin{aligned}
\forall r, s, d_1, d_2, x.\ & \mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{Requester}(r, s) \\
& \wedge \mathit{RequestedAct}(r, \uparrow \mathit{modifyAttrInst}(s, \uparrow \mathit{ResponsibleDoctorOf}(s, d_1), \\
& \qquad \uparrow \mathit{ResponsibleDoctorOf}(s, d_2))) \\
& \wedge (d_1 \neq d_2) \wedge \mathit{SignedBy}(r, x) \wedge \mathit{Manager}(x) \\
& \rightarrow [\mathit{authorizesRequest}(\texttt{changeDoctorPolAut}, r)](\mathit{Authorized}(r) \\
& \quad \wedge \mathit{AuthorizedBy}(r, \texttt{changeDoctorPolAut}))
\end{aligned}$$

$$\begin{aligned}
\forall r, s, d_1, d_2, x.\ & \mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{careFacility}) \wedge \mathit{Requester}(r, s) \\
& \wedge \mathit{RequestedAct}(r, \uparrow \mathit{modifyAttrInst}(s, \uparrow \mathit{ResponsibleDoctorOf}(s, d_1), \\
& \qquad \uparrow \mathit{ResponsibleDoctorOf}(s, d_2))) \\
& \wedge (d_1 \neq d_2) \wedge \neg \mathit{SignedBy}(r, x) \wedge \mathit{Manager}(x) \\
& \rightarrow [\mathit{prohibitsRequest}(\texttt{changeDoctorPolPro}, r)](\mathit{Prohibited}(r) \\
& \quad \wedge \mathit{ProhibitedBy}(r, \texttt{changeDoctorPolPro}))
\end{aligned}$$

where $\mathit{SignedBy}(r, s)$ means request r is signed by subject s. One may think of a request is a piece of paper that can be submitted. A manager or other subjects may sign it, depending on to whom the paper is submitted. We do not discuss the logistics here and simply assume that a request can be signed.

Let $\texttt{r}_1$ be Carol's request to changer her responsible doctor from Bob to George, i.e.,

$$\begin{aligned}
& \mathit{Requester}(\texttt{r}_1, \texttt{carol}), \\
& \mathit{RequestedAct}(\texttt{r}_1, \uparrow \mathit{modifyAttrInst}(\texttt{carol}, \uparrow \mathit{ResponsibleDoctorOf}(\texttt{carol}, \texttt{bob}), \\
& \qquad \uparrow \mathit{ResponsibleDoctorOf}(\texttt{carol}, \texttt{george}))).
\end{aligned}$$

Carol does not have the manager's signature, i.e., $\neg \mathit{SignedBy}(\texttt{r}_1, \texttt{alice})$, where `alice` is the manager. Thus, `changeDoctorPolPro` prohibits Carol's request $\texttt{r}_1$. Carol makes another request $\texttt{r}_2$ ask about the reason for $\texttt{r}_1$'s denial, i.e.,

$$\begin{aligned}
& \mathit{Requester}(\texttt{r}_2, \texttt{carol}), \\
& \mathit{RequestedAct}(\texttt{r}_2, \uparrow \mathit{readDecisionExp}(\texttt{carol}, \texttt{careFacility}, \texttt{r}_1)).
\end{aligned}$$

Since Carol is the requester of both $\texttt{r}_1$ and $\texttt{r}_2$, $\texttt{r}_2$ is authorized. When no other policies prohibit $\texttt{r}_2$, `careFacility` will grant $\texttt{r}_2$ and disclose `changeDoctorPolPro` to Carol.

8.2 SQL Implementation of DC RefABAC

The implementation of a DC RefABAC model in SQL can be easily modified from the implementation of Core RefABAC. The *DiscloseTo* attribute maps to a table DiscloseTo with the schema

$$\mathsf{DiscloseTo}(\mathsf{inst\_name}, \mathsf{system}, \mathsf{entity\_name}, \mathsf{subject}),$$

where inst_name serves as the names for the attribute instances, and the rest of the three columns map to the three parameters, respectively. The constraint

$$\mathit{DiscloseTo}(\texttt{careFacility}, \uparrow \mathit{ResponsibleDoctorOf}(\texttt{bob}, \texttt{carol}), \texttt{alice})$$

for *DiscloseTo* is implemented as two SQL assertions using

```
CREATE ASSERTION discloseToFirstArgumentType
CHECK NOT EXISTS
  (SELECT system FROM DiscloseTo
   WHERE DiscloseTo.system NOT IN
   (SELECT system FROM System));
CREATE ASSERTION discloseToThirdArgumentType
CHECK NOT EXISTS
  (SELECT subject FROM DiscloseTo
   WHERE DiscloseTo.subject NOT IN
   (SELECT subject FROM Subject));
```

The implementation for transition rules is modified as well. We still use *readAttrInst* as an example. The action type *readAttrInst* maps to a table readAttrInst with the schema

$$\mathsf{readAttrInst}\,(\mathsf{action\_name}, \mathsf{subject}, \mathsf{inst\_name}).$$

Suppose *Attr* is binary. Performing an action $\mathit{readAttrInst}(s, \uparrow \mathit{Attr}(x))$ is analogous to inserting a row with value (exRead, s, exInst) into the PerformedReadAttrInst table, where $\texttt{exInst} = \uparrow \mathit{Attr}(x)$, and $\texttt{exRead} = \uparrow \mathit{readAttrInst}(s, \uparrow \mathit{Attr}(x))$. The TRIGGER statement handles the post-conditions and is written as follows:

```
delimiter |
CREATE TRIGGER readAttrInst_trigger
AFTER INSERT ON PerformedReadAttrInst FOR EACH ROW
BEGIN
  INSERT INTO Fulfilled(request)
  SELECT request FROM Request
  WHERE NEW.action_name = Request.action_name;

  INSERT INTO DiscloseTo(system, entity_name, subject)
  SELECT Attr.fst_arg, Attr.snd_arg, system
  FROM Attr, AttrOne, InSys, readAttrInst
  WHERE readAttrInst.action_name = NEW.action_name
  AND readAttrInst.inst_name = AttrOne.inst_name
  AND system = "careFacility";
END;
|
delimiter ;
```

8.3 Challenges of DC RefABAC

8.3.1 What Information to Track?

A major challenge in defining a DC RefABAC system model is to what extent should a system keep track of the information it has disclosed. In the preceding example transition rule for reading an attribute instance, the only thing that `careFacility` keeps track of is the read attribute instance $Attr(x,y)$, i.e., $DiscloseTo(\texttt{careFacility}, \uparrow Attr(x,y), s)$. However, one may complicate the transition rule by also specifying that $Fulfilled(r)$ and $Performed(\uparrow readAttrInst(s, \uparrow Attr(x)))$ are also disclosed, i.e.,

$$\begin{aligned}
\forall r,s,a,x,y.\, & Granted(r) \wedge \neg Fulfilled(r) \wedge InSys(r, \texttt{careFacility}) \\
& \wedge RequestedAct(r, \uparrow readAttrInst(s, \uparrow Attr(x))) \wedge (a = \uparrow Attr) \\
& \wedge InSys(\uparrow Attr(x,y), \texttt{careFacility}) \\
& \rightarrow [readAttrInst(s, \uparrow Attr(x))](Fulfilled(r) \\
& \quad \wedge Performed(\uparrow readAttrInst(s, \uparrow Attr(x))) \\
& \quad \wedge DiscloseTo(\texttt{careFacility}, \uparrow Attr(x,y), s) \\
& \quad \wedge DiscloseTo(\texttt{careFacility}, \uparrow Fulfilled(r), s) \\
& \quad \wedge DiscloseTo(\texttt{careFacility}, \uparrow Performed(\uparrow readAttrInst(s, \uparrow Attr(x))), s)).
\end{aligned}$$

What kinds of information should be tracked during access control would depend on the problem domain. This issue is similar to that faced in logging, where one should not have too much or too little. We do not discuss in this dissertation where the balance lies but only provide a simple mechanism of representing information disclosure.

8.3.2 What If Something Changes?

Another challenge of DC RefABAC is related to version control and update of information. For example, suppose subject s_2 reads an attribute instance $Attr(x, y)$ from subject s_1, then we have $DiscloseTo(s_1, \uparrow Attr(x, y), s_2)$. However, if another subject s modifies $Attr(x, y)$ to $Attr(x, y')$, s_2 needs not be aware of the modification and might still think the original attribute instance $Attr(x, y)$ is true.

One way of handling such inconsistencies of knowledge is adding an extra parameter to each piece of information to indicate its version. By recording the current version of each information, the system only discloses the information corresponding to its current version. Consider again the preceding example. The attribute instance $Attr(x, y)$ is now represented as $Attr(x, y, v)$, meaning x has value y for $Attr$ at version v. Suppose the current version of $Attr$ is v. When s reads the attribute instances, the system discloses $Attr(x, y, v)$ instead of $Attr(x, y)$ to s. Later on, when s' modifies $Attr(x, y)$, the system also modifies the attribute instance's version. Thus, instead of $Attr(x, y')$, $Attr(x, y', v')$ is true and the current version of $Attr$ is updated to v' accordingly. In our formalism of DC RefABAC, we do not consider version control for simplicity. However, it could be achieved by simply adding the version parameter to every attribute and adding a conceptual attribute *CurrentVersion* where $CurrentVersion(x, v)$ means the current version of x is v.

8.3.3 What Can A System Infer?

A third major challenge is what inference power should a system have so that it can automatically detect privacy leaks. In Subsection 8.1.2, we discussed an example where if a system has disclosed to a subject two related pieces of information x_1 and x_2, the subject is able to infer another sensitive piece of information y that the system is prohibited to disclose to the subject. To address the issue, we defined a policy that if the system has disclosed either of x_1 or x_2, the system is prohibited to disclose y to the same subject. If the system can automatically infer that disclosing x_1 and x_2 will reveal y, we need not define such policies but let the system handle the situation.

However, allowing a system to make full inferences is time consuming. Readers might notice the close relationship between information disclosure and possession of knowledge, or in other words, the relationship between our formalism and epistemic logic. It would be interesting to explore what kinds of inference a system is allowed to make.

8.4 Conclusion

We formalized an advanced version of Core RefABAC, DC (Disclosure Control) RefABAC, in this chapter. DC RefABAC improves upon Core RefABAC by adding a conceptual attribute *DiscloseTo*. $DiscloseTo(s, x, s')$ means subject s has disclosed the entity x or the information contained in x to subject s'. The attribute can be used in the post-conditions of transition rules so that a system can track what information has been disclosed to which subjects. It can also be used in the pre-conditions of a policy to control the information flow. If a system has disclosed to a subject a piece

of information, then one may define policies to limit the disclosure of other sensitive information that might be inferred from the disclosed information. We also briefly discussed some challenges when defining a DC RefABAC model, regarding the sufficiency of tracking disclosed information, information update, and the inference power of a system. The challenges are worth exploring in the future work.

CHAPTER

9

PCR REFABAC: CORE REFABAC WITH POLICY CONFLICT RESOLUTION

In Chapter 5, we defined a baseline RefABAC framework, Core RefABAC. Core RefABAC uses a simple denial-by-default approach where a request is denied if it is prohibited or not authorized by any policies.

In this chapter, we define an advanced RefABAC framework, called **PCR RefABAC** (Core RefABAC with Policy Conflict Resolution) that improves upon Core RefABAC by including a policy conflict resolution mechanism. The mechanism is based on Coco [Ajm16], a conflict resolution mechanism we proposed in previous work. The relevant contribution of Coco is defining a special kind of policies called *dominance policies*. A dominance policy resolves the conflicts between two (regular) policies. However, a drawback of Coco is that two dominance policies may also conflict with each other. PCR RefABAC generalizes Coco by allowing multi-level dominance policies so that a dominance policy at a higher level can resolve the conflicts of two (dominance or regular) policies at a lower level.

This chapter is organized as follows. We first summarize Coco in Section 9.1. Then we define an intermediate RefABAC framework between Core RefABAC and PCR RefABAC, called **Coco RefABAC**. Coco RefABAC incorporates the idea of (single-level) dominance policies from Coco. Finally, we define PCR RefABAC in Section 9.3, which generalizes Coco RefABAC by addling multi-level dominance policies.

9.1 Summary of Coco

Coco [Ajm16] is an approach for reasoning about which specific commitments apply to specific parties in light of general types of commitments, specific circumstances, and dominance relations among specific commitments. A **commitment** expresses the expectations that subjects [1] have of one another. Coco adapts Answer-Set Programming (ASP) to identify a maximal set of non-dominated commitments. The working example used in [Ajm16] is described as follows.

Example 9.1. Alice, a child, takes ill. Society expects Alice's guardian, Bob, to take her immediately to her pediatrician, Carol, provided Alice requires immediate medical attention. Meanwhile, Bob's employer, Steve, expects him to work during business hours. Bob disregards his employer's expectation in light of the medical emergency involving his ward.

Bob is both committed to Steve to work during business hours and also committed to society to take Alice to Carol if Alice is sick. A conflict between the two commitments arises when Alice falls sick during work hours. Notice that unlike in AC, we have two kinds of subjects in a commitment: one acts as the **debtor**, the party that is responsible for performing an action; the other acts as the **creditor**, the party to whom the debtor is responsible.

9.1.1 Commitment Representation

We first define how to represent a commitment in ASP. The notion of commitment actually has two meanings and we refer to them as a commitment schema and a commitment instance. A **commitment schema**, or a **schema** for short, is a *general* statement describing what any subject should do under certain circumstances, called the antecedent or pre-condition of the schema. A **commitment instance**, or an **instance** for short, is a *specific* statement generated from its corresponding schema when the schema is instantiated. The example commitment of a subject taking another to the pediatrician is represented as multiple formulas in ASP as follows: [2]

$$\mathit{Schema}(\texttt{gCom}) \tag{9.1}$$

$$\mathit{GuardianR}(c, g, t) \rightarrow \mathit{Dbt}(\texttt{gCom}, g, t) \tag{9.2}$$

$$\mathit{GuardianR}(c, g, t) \rightarrow \mathit{Crd}(\texttt{gCom}, c, t) \tag{9.3}$$

$$[\mathit{Sick}(c, t) \wedge \mathit{Crd}(\texttt{gCom}, c, t)] \rightarrow \mathit{Ant}(\texttt{gCom}, t) \tag{9.4}$$

$$[\mathit{Bring}(g, c, p, t) \wedge \mathit{PedR}(c, p, t) \wedge \mathit{Dbt}(\texttt{gCom}, g, t) \wedge \mathit{Crd}(\texttt{gCom}, c, t)] \rightarrow \mathit{Con}(\texttt{gCom}, t) \tag{9.5}$$

$$\mathit{DDuration}(\texttt{gCom}, 3). \tag{9.6}$$

[1] We follow the terminology used in access control to refer to an autonomous party as a subject. However, in the original paper of Coco [Ajm16] and in the literature of norms [Sin15], an autonomous party is called a **principal**.

[2] To be consistent with the rest of the dissertation, we follow the notation for DL^+ in this section instead of the notation for ASP. We use words beginning with capital-case letters for predicate symbols, and ϕ, ψ for formulas. We use words beginning with lower-case letters for action type symbols and terms. Among the lower-case letters, we use the *italics* font for variables, and the `typewriter` font for constants. Subscripts and superscripts may be added as well.

The constant gCom denotes the commitment schema. Equation 9.2 and Equation 9.3 mean that if g fills the guardian role for charge c at t, then g is the debtor of gCom and c is the creditor. Equation 9.4 and Equation 9.5 state the antecedent and consequent of gCom, respectively. The antecedent is that c falls sick, and the consequent is that g brings c to c's pediatrician p. Equation 9.6 states that the deadline duration is 3 time units, meaning that once c falls sick, g has 3 time units available to take c to the pediatrician p.

Similarly, the schema wCom defined in the following is for the commitment of being at work during work hours. Equation 9.8 and Equation 9.9 state that if employee e's employer is r at time t, then e is the debtor and R is the creditor. The antecedent of wCom is that time t is a work hour for employee e at employer r (Equation 9.10), and the consequent is e is at work for r at time t (Equation 9.11). The deadline duration is 0 time units, meaning if T is a work hour for e, then e should be at work at t.

$$\mathit{Schema}(\texttt{wCom}) \tag{9.7}$$

$$\mathit{EmployerR}(e, r, t) \rightarrow \mathit{Dbt}(\texttt{wCom}, e, t) \tag{9.8}$$

$$\mathit{GuardianR}(c, g, t) \rightarrow \mathit{Crd}(\texttt{gCom}, c, t) \tag{9.9}$$

$$[\mathit{Dbt}(\texttt{wCom}, e, t) \wedge \mathit{Crd}(\texttt{wCom}, r, t) \wedge \mathit{Workhour}(e, r, t) \rightarrow \mathit{Ant}(\texttt{wCom}, T) \tag{9.10}$$

$$[\mathit{Dbt}(\texttt{wCom}, e, t) \wedge \mathit{Crd}(\texttt{wCom}, r, t) \wedge \mathit{AtWork}(e, r, t) \rightarrow \mathit{Con}(\texttt{wCom}, t) \tag{9.11}$$

$$\mathit{DDuration}(\texttt{wCom}, 0). \tag{9.12}$$

We treat compliance and conflict with respect to instances. If Alice falls sick at time t, the pre-conditions of gCom become true, and we say an **instance** of gCom is instantiated and becomes **detached** at t. We denote the instance as $\mathit{Instance}(\texttt{gCom}, \texttt{bob}, \texttt{alice}, 10)$, which we abbreviate to gComInst in the following. We define predicates *CrdI*, *DbtI*, *AntI*, *ConI*, and *DDurationI* to represent the different parts of an instance, namely the creditor, debtor, antecedent, consequent, and duration of an instance. Thus, the information contained in gComInst can be extracted using the following rules:

$$[\mathit{SInst}(\texttt{gComInst}) \wedge \mathit{IsInstOf}(\texttt{gComInst}, \texttt{gCom})] \rightarrow \mathit{CrdI}(\texttt{gComInst}, \texttt{alice})$$

$$[\mathit{SInst}(\texttt{gComInst}) \wedge \mathit{IsInstOf}(\texttt{gComInst}, \texttt{gCom})] \rightarrow \mathit{DbtI}(\texttt{gComInst}, \texttt{bob})$$

$$\mathit{DDurationI}(\texttt{gComInst}, 3).$$

The antecedent and consequent of an instance, omitted in the listing, are formed by substituting the names of specific debtor and creditor for the corresponding variables in the schema.

9.1.2 Satisfaction and Violation

If Bob takes Alice to her pediatrician within three hours, then the instance is **satisfied**. Otherwise, the instance is **violated**. The satisfaction of an instance is formalized as $\mathit{BecomesSat}(i, t)$, which

means that t is the first time instance at which the consequent of i holds, provided this happens before the deadline; *Satisfied* is true of the instance from that time on. The violation of an instance is realized in a similar way. Specifically, the statement $BecomesViolated(i,t)$ holds at t when i is detached, but not dominated, and its consequent does not hold within the deadline. The definitions for *BecomesSat* and *BecomesViolated* are provided as follows:

$$Detached(i,t') \wedge ConI(s,t) \wedge Difference(t,t',1) \rightarrow BecomesSat(i,t),$$
$$BecomesDetached(i,t_1) \wedge DDurationI(i,t_2) \wedge Addition(t_1,t_2,T) \wedge ConINotHold(i,t_1,t) \wedge \neg Dominated(i,t) \rightarrow BecomesViolated(i,t).$$

The predicate $ConINotHold(i,t_1,t)$ says that the consequent of instance i is not true between times t_1 and t. The two utility predicates $Difference(t_1,t_2,t)$ and $Addition(t_1,t_2,t')$ state that the difference and sum of t_1 and t_2 are t and t', respectively.

9.1.3 Conflict Detection

We define a set of instances as conflicting if the set of their consequents cannot hold simultaneously. Whether a consequent can be true when another consequent is true depends on the domain knowledge of specific scenarios. In the running example, `gComInst` and `wComInst` conflict. The following rule defines the relation $Conflicting(i_1,i_2,t)$. If instances i_1 and i_2 are both detached at time t and their consequents cannot both hold at t, then they conflict at t.

$$Detached(i_1,t) \wedge Detached(i_2,t) \wedge \neg(ConI(s_1,t) \longleftrightarrow ConI(s_2,t)) \rightarrow Conflicting(s_1,s_2,t)$$

In Example 9.1, Bob is required to be at his workplace during work hours and at Carol's office with Alice when Alice is sick. If Alice gets sick during Bob's work hours, both `gComInst` and `wComInst` detach and require Bob to be in two places at the same time. The consequent conditions conflict even when Alice is not sick, but only matter for detached instances; only then may such conflicts cause a commitment violation. The following rules formalize the conflict detection of `gComInst` and `wComInst` under certain circumstances.

$$AtLocation(s,l_1,t) \wedge (l_1 \neq l_2) \rightarrow \neg AtLocation(s,l_2,t) \tag{9.13}$$

$$IsInstOf(i,\texttt{gCom}) \wedge ConI(i,t) \wedge DbtI(i,g) \rightarrow AtLocation(g,\texttt{hospitalLoc},t) \tag{9.14}$$

$$EmployerR(e,r,t) \wedge \neg AtLocation(e,\texttt{companyLoc},t) \rightarrow \neg AtWork(e,r,t) \tag{9.15}$$

$$IsInstOf(i,\texttt{wCom}) \wedge DbtI(i,e) \wedge CrdI(i,r) \wedge \neg AtWork(e,r,t) \rightarrow \neg ConI(i,t) \tag{9.16}$$

$$IsInstOf(i,\texttt{wCom}) \wedge ConI(i,t) \wedge DbtI(i,e) \wedge CrdI(i,r) \rightarrow AtLocation(e,\texttt{companyLoc},t) \tag{9.17}$$

$$GuardianR(c,g,t) \wedge PedR(c,p,t) \wedge \neg AtLocation(g,\texttt{hospitalLoc},t) \rightarrow \neg Bring(g,c,p,t) \tag{9.18}$$

$$IsInstOf(i,\texttt{gCom}) \wedge DbtI(i,g) \wedge CrdI(i,c) \wedge PedR(c,p,t) \wedge \neg Bring(g,c,p,t) \rightarrow \neg ConI(i,t) \tag{9.19}$$

Equation 9.13 says that the same subject cannot be at two different locations at the same time. Equation 9.14-Equation 9.16 imply that `wComInst`'s consequent cannot be true at t when `gComInst`'s consequent holds at t. Equation 9.17-Equation 9.19 handle the converse. Thus, `gComInst` and `wComInst` conflict.

9.1.4 Conflict Resolution

We resolve such conflicts by employing dominance relations, so that satisfaction of a more dominant instance vitiates violation of a less dominant instance. Thus, a subject can comply with one instance while violating another, and still remain in compliance with the entire set of instances. We require the dominance relation be a strict partial order, that is, transitive and irreflexive. A partial order is appropriate because changing commitments and unforeseen conflicts may make a total ordering infeasible; a partial order can suffice to resolve conflicts that do arise. We select from the set of all detached instances a maximal subset of non-dominated instances, that is, a subset of detached instances none of which is less dominant than some conflicting instance in the same set.

The following rule formalizes simple dominance of `gCom` over `wCom`. The formula says that a detached instance of `gCom` dominates a detached instance of `wCom` if they have the same debtor, which is indicated by the utility predicate *SameDbtI*.

$$\begin{aligned} &\mathit{IsInstOf}(i_1, \texttt{gCom}) \wedge \mathit{IsInstOf}(i_2, \texttt{wCom}) \wedge \mathit{Detached}(i_1, t) \wedge \mathit{Detached}(i_2, t) \\ &\quad \wedge \mathit{SameDbtI}(i_1, i_2) \rightarrow \mathit{Dominates}(i_1, i_2, t) \end{aligned}$$

The following rule defines that $\mathit{Dominated}(i_1, t)$ holds at time t in case a detached instance i_1 is in conflict with another detached, non-dominated instance i_2 and i_2 dominates i_1.

$$\begin{aligned} &\mathit{Dominates}(i_2, i_1, t) \wedge \mathit{Conflicting}(i_1, i_2, t) \wedge \neg \mathit{Dominated}(i_2, t) \\ &\quad \wedge \mathit{Detached}(i_1, t) \wedge \mathit{Detached}(i_2, t) \rightarrow \mathit{Dominated}(i_1, t) \end{aligned}$$

9.2 Coco RefABAC: Incorporating Coco Into RefABAC

Before defining PCR RefABAC, we first define an intermediate RefABAC framework between Core RefABAC and PCR RefABAC by incorporating Coco into RefABAC, called **Coco RefABAC**. Notice that Coco talks about the truth of information at explicit time instances. We will eliminate the explicit notion of time and adapt it to DL^+.

9.2.1 Coco RefABAC: Conceptual Model

The logical structure of the conceptual model $\mathscr{C}$ in Coco RefABAC is the same as in Core RefABAC. Specifically, $\mathscr{C}$ is a tuple $\langle \mathscr{C}_{Attr}, \mathscr{C}_C, \mathscr{C}_{act}, \mathscr{C}_T \rangle$, where $\mathscr{C}_{Attr}$ is a set of conceptual attributes, $\mathscr{C}_C$ a set of conceptual constraints, $\mathscr{C}_{act}$ a set of conceptual action types, and $\mathscr{C}_T$ a set of conceptual transition rules.

To incorporate Coco, we first define additional conceptual attributes that are used for policy instances and the dominance relation. We also define the corresponding conceptual constraints for the additional conceptual attributes. Moreover, in Core RefABAC, we say a policy authorizes or prohibits a request. While in Coco RefABAC, we say a *policy instance* authorizes or prohibits a request. We modify the relevant conceptual constraints and transition rules accordingly. We formalize the modifications in the rest of this section.

9.2.1.1 Conceptual Attributes

The additional conceptual attributes we add to the conceptual model for Coco RefABAC include

$$\mathit{PolicyInst}, \mathit{IsInstOf}, \mathit{DominatedBy}, \mathit{Dominated}.$$

PolicyInst is a new conceptual type attribute. $\mathit{PolicyInst}(x)$ means x is (the name of) a policy instance, which is represented as a DL^+ formula without variables. The *PolicyInst* attribute corresponds to the predicate *SInst* in Coco. Given a policy p, an instance of p is generated by substituting all variables with constants. $\mathit{IsInstOf}(x, p)$ indicates that policy instance with name x is an instance of policy with name p, and it corresponds to the predicate *IsInstOf* in Coco. $\mathit{DominatedBy}(x, y)$ means the policy instance with name x is dominated by the policy instance with name y, and $\mathit{Dominated}(x)$ means the policy instance with name x is dominated. [3]

For simplicity, we do not define a conceptual attribute *Conflicting* that corresponds to the *Conflicting* predicate in Coco. In Coco RefABAC, the only condition that would cause two policy instances to be in conflict is that one of them authorizes a request while the other prohibits the same request. We write the condition directly when defining the dominance relation.

In Core RefABAC, $\mathit{AuthorizedBy}(r, p)$ (respectively, $\mathit{Prohibited}(r, p)$) means a request is authorized (respectively, prohibited) by a policy with name p. In Coco RefABAC, we say a request is authorized or prohibited by a *policy instance* instead.

9.2.1.2 Conceptual Constraints

The conceptual constraints for the additional conceptual attributes are listed in Table 9.1. The first group of constraints in the table specify the types of arguments for the additional conceptual attributes. They also change the types of the second arguments in *AuthorizedBy* and *ProhibitedBy* to *PolicyInst* from *Policy*. The second group lists the additional conceptual constraints for specifying number restrictions. Each policy instance must correspond to exactly one policy. Finally, we define the conceptual constraints for the dominance relation. If a policy instance x is dominated by any other policy instance, then x is dominated.

[3] The reason we use $\mathit{DominatedBy}(x, y)$ instead of $\mathit{Dominates}(y, x)$ as in Coco is because *Dominates* sounds like an action according to our naming tradition.

Table 9.1 Coco RefABAC Additional Conceptual Constraints

Category Name	Conceptual Constraints
Argument Types	$\forall p, i.\ IsInstOf(i, p) \rightarrow PolicyInst(i) \wedge Policy(p)$
	$\forall x, y.\ DominatedBy(x, y) \rightarrow PolicyInst(x) \wedge PolicyInst(y)$
	$\forall x, y.\ Dominated(x) \rightarrow PolicyInst(x)$
	$\forall r, i.\ AuthorizedBy(r, i) \rightarrow Request(r) \wedge PolicyInst(i)$
	$\forall r, i.\ ProhibitedBy(r, i) \rightarrow Request(r) \wedge PolicyInst(i)$
Number Restriction	$\forall i.\ PolicyInst(i) \rightarrow \exists p.IsInstOf(i, p)$
	$\forall i, p, p'.\ IsInstOf(i, p) \wedge IsInstOf(i, p') \rightarrow p = p'$
Others	$\forall x.\ \exists y.\ DominatedBy(x, y) \rightarrow Dominated(x)$

9.2.1.3 Conceptual Action Type and Transition Rules

The conceptual action types in Coco RefABAC remain the same as in Core RefABAC. The transition rule for *grantsRequest* and *deniesRequest* are modified as follows respectively to account for the added dominance relation:

$$\forall r, sys.\ Active(r) \wedge InSys(r, sys) \wedge \exists i.\ AuthorizedBy(r, i) \wedge \neg Dominated(i)$$
$$\rightarrow [grantsRequest(sys, r)](Performed(\uparrow grantsRequest(sys, r)) \wedge Granted(r) \wedge \neg Active(r)),$$

$$\forall r, sys.\ Active(r) \wedge InSys(r, sys) \wedge \exists i.\ ProhibitedBy(r, i) \wedge \neg Dominated(i)$$
$$\rightarrow [deniesRequest(sys, r)](Performed(\uparrow deniesRequest(sys, r)) \wedge Denied(r) \wedge \neg Active(r)).$$

Given an active request r in a system *sys*, if r is authorized or prohibited by a non-dominated policy instance i, then *sys* grants or denies the request r, respectively. The action is recorded as performed, and the request becomes inactive and granted (respectively, denied).

9.2.2 Coco RefABAC: System Model

The logical structure for a system model $\mathscr{S}_{sys}$ for a system *sys* in a Coco RefABAC framework includes an extra component containing the dominance policies. Specifically, $\mathscr{S}_{sys}$ is identified as a tuple $\langle \mathscr{S}_{Attr}, \mathscr{S}_C, \mathscr{S}_{Act}, \mathscr{S}_T, \mathscr{S}_P, \mathscr{S}_D \rangle$, where the first five components are the same as a system model in Coco RefABAC, and the last component $\mathscr{S}_D$ is a set of formulas called **system dominance policies**.

The formalism for system attributes, constraints, action types and transition rules are the same as in Core RefABAC. The formalism for system policies is modified such that a system policy is a

DL^+ formula of one of the following forms:

$$\forall r, \vec{x}.\ \exists pi.\ Active(r) \wedge InSys(r, \mathtt{sys}) \wedge PolicyInst(pi) \wedge IsInstOf(pi, \mathtt{pol}) \wedge \phi \\ \rightarrow [authorizesRequest(pi, r)](Authorized(r) \wedge AuthorizedBy(r, pi))$$

$$\forall r, \vec{x}.\ \exists pi.\ Active(r) \wedge InSys(r, \mathtt{sys}) \wedge PolicyInst(pi) \wedge IsInstOf(pi, \mathtt{pol}) \wedge \phi \\ \rightarrow [prohibitsRequest(pi, r)](Prohibited(r) \wedge ProhibitedBy(r, pi)),$$

where ϕ is a conjunction of attribute instances and of their negations. Assuming an entity *pi* is a policy instance of `pol`. When the antecedent of the policy is satisfied, *pi* authorizes or prohibits, respectively, the request.

A dominance policy is defined in the following.

Definition 9.1 (Dominance Policy)**.** Let `p` and $\mathtt{p}'$ be the names of two policies. The policy with name `p` decides whether a request should be prohibited. The policy with name $\mathtt{p}'$ decides whether a request should be authorized. A **dominance policy** between the policy instances of the two policies is a DL^+ formula of either of the following forms:

$$\forall pi, pi', r, \vec{x}.\ IsInstOf(pi, \mathtt{p}) \wedge IsInstOf(pi', \mathtt{p}') \\ \wedge ProhibitedBy(r, pi) \wedge AuthorizedBy(r, pi') \wedge \phi \\ \rightarrow DominatedBy(pi, pi') \wedge Dominated(pi)$$

$$\forall pi, pi', r, \vec{x}.\ IsInstOf(pi, \mathtt{p}) \wedge IsInstOf(pi', \mathtt{p}') \\ \wedge ProhibitedBy(r, pi) \wedge AuthorizedBy(r, pi') \wedge \phi \\ \rightarrow DominatedBy(pi', pi) \wedge Dominated(pi'),$$

where ϕ is a Boolean combination of attribute instances.

9.2.3 Example

The following two examples show two policies regarding a subject reading daily logs. The first policy says that volunteers are prohibited from reading daily logs, while the second one says that volunteers who are responsible for helping the nurses with daily checkups are authorized to read daily logs.

Example 9.2. Volunteers are prohibited from reading the daily logs. Denote the policy by `exPolPro`.

$$\forall r, \vec{x}.\ \exists pi.\ Active(r) \wedge InSys(r, \mathtt{sys}) \wedge PolicyInst(pi) \wedge IsInstOf(pi, \mathtt{exPolPro}) \\ \wedge Requester(r, s) \wedge Volunteer(s) \\ \wedge RequestedAct(r, \uparrow readObj(s, o)) \wedge DailyLog(o) \\ \rightarrow [prohibitsRequest(pi, r)](Prohibited(r) \wedge ProhibitedBy(r, pi)).$$

Example 9.3. Volunteers who are responsible for helping the nurses with daily checkups are authorized to read the daily logs. Denote the policy by `exPolAut`.

$$\begin{aligned}\forall r, \vec{x}.\, \exists pi.\, &Active(r) \wedge InSys(r, \texttt{sys}) \wedge PolicyInst(pi) \wedge IsInstOf(pi, \texttt{exPolAut}) \\ &\wedge Requester(r, s) \wedge VolunteerForDailyCheckup(s) \\ &\wedge RequestedAct(r, \uparrow readObj(s, o)) \wedge DailyLog(o) \\ &\rightarrow [authorizesRequest(pi, r)](Authorized(r) \wedge AuthorizedBy(r, pi)).\end{aligned}$$

If `exPolPro` and `exPolAut` are both applicable for a request, and the requester is a volunteer that helps the nurses with daily checkups, then the request is both prohibited by `exPolPro` and authorized by `exPolAut`. The following example describes a dominance policy resolving the conflicts between the policies instances of `exPolPro` and `exPolAut`.

Example 9.4. A policy instance generated by `exPolAut` dominates one generated by `exPolPro`.

$$\begin{aligned}\forall pi, pi', r, s.&IsInstOf(pi, \texttt{exPolPro}) \wedge IsInstOf(pi', \texttt{exPolAut}) \\ &\wedge ProhibitedBy(r, pi) \wedge AuthorizedBy(r, pi') \\ &\wedge Requester(r, s) \wedge VolunteerForDailyCheckup(s) \\ &\rightarrow DominatedBy(pi, pi') \wedge Dominated(pi)\end{aligned}$$

Since `exPolAut` becomes applicable when the requester is of type *VolunteerForDailyCheckup*, the dominance policy can be simplified by getting rid of the extra conditions, shown as follows:

$$\begin{aligned}\forall pi, pi', r.&IsInstOf(pi, \texttt{exPolPro}) \wedge IsInstOf(pi', \texttt{exPolAut}) \\ &\wedge Prohibited(r, pi) \wedge Authorized(r, pi') \\ &\rightarrow DominatedBy(pi, pi') \wedge Dominated(pi).\end{aligned}$$

In other words, the policy instances of `exPolAut` dominate the policy instances of `exPolPro` when the request is a volunteer who is responsible for daily checkups.

9.3 PCR RefABAC: Generalizing Coco RefABAC

The policy conflict resolution in [Ajm16] and Coco RefABAC does not work properly when every policy instance for a request is dominated by some other policy instance. In this case, there will not be any non-dominated policy instances left for the system to decide whether to grant or deny the request. This happens when the policy instances for the request in question form a *cyclic* dominance relation. Consider a simple case where a request r has two applicable policies p_1 and p_2 such that r is prohibited by an instance of p_1 but authorized by an instance of p_2. Suppose there are two dominance policies. One of them says for the request r, the policy instance of p_1 dominates the one of p_2; while the other says the policy instance of p_2 dominates the one of p_1. Therefore, the only

two policy instances for r are both dominated. The system neither denies nor grants the request, according to the transition rules for *deniesRequest* and *grantsRequest*.

We define another RefABAC framework called **PCR (Policy Conflict Resolution)** RefABAC that improves upon Coco RefABAC by defining multi-level dominance policies. A dominance policy at a higher level resolves the conflicts among policies at a lower level.

9.3.1 PCR RefABAC: Conceptual Model

The logical structure of the conceptual model $\mathscr{C}$ in PCR RefABAC is the same as Core RefABAC and Coco RefABAC. Specifically, $\mathscr{C}$ is a tuple $\langle \mathscr{C}_{Attr}, \mathscr{C}_C, \mathscr{C}_{act}, \mathscr{C}_T \rangle$, where $\mathscr{C}_{Attr}$ is a set of conceptual attributes, $\mathscr{C}_C$ a set of conceptual constraints, $\mathscr{C}_{act}$ a set of conceptual action types and $\mathscr{C}_T$ a set of conceptual transition rules.

The modifications we make in PCR RefABAC from Coco RefABAC include two new conceptual attributes with corresponding conceptual constraints, and a new conceptual action type. We will discuss them in detail in the rest of the section.

9.3.1.1 Conceptual Attributes

The conceptual model in a PCR RefABAC framework defines the following additional conceptual attributes:

$$\textit{Consequent}, \text{ and } \textit{GovernedBy}.$$

$\textit{Consequent}(i, x)$ means the consequent of a dominance policy instance with name i is a formula with name x. For any rule, we usually refer to everything to the right of the implication as the consequent. Recall that a dominance policy in Coco RefABAC is of the form

$$\begin{aligned}\forall pi, pi', r, s.\, &\textit{IsInstOf}(pi, \texttt{exPolPro}) \wedge \textit{IsInstOf}(pi', \texttt{exPolAut}) \\ &\wedge \textit{ProhibitedBy}(r, pi) \wedge \textit{AuthorizedBy}(r, pi') \wedge \phi \\ &\rightarrow \textit{DominatedBy}(pi, pi') \wedge \textit{Dominated}(pi).\end{aligned}$$

Thus, the *Consequent* of a dominance policy instance are $\textit{DominatedBy}(i_1, i_2)$ and $\textit{Dominated}(i_1)$, where i_1 and i_2 are names for policy instances. In other words, for any instance i of the dominance policy, we have $\textit{Consequent}(i, \uparrow \textit{Dominated}(pi, pi'))$ and $\textit{Consequent}(i, \uparrow \textit{Dominated}(pi))$. The usage of *Consequent* will become clearer later on.

$\textit{GovernedBy}(r, p)$ means request r is directly or indirectly governed by a policy with name p, which can be a regular policy or a dominance policy. Suppose subject s makes request r. One of its applicable (regular) policies with name p_1 decides that r should be authorized, and another applicable (regular) policy with name p_2 decides that r should be prohibited. Then r is governed by both p_1 and p_2, i.e., $\textit{GovernedBy}(r, p_1)$ and $\textit{GovernedBy}(r, p_2)$. If there exists a dominance policy p for p_1 and p_2, then r is governed by p as well, i.e., $\textit{GovernedBy}(r, p)$.

The conceptual constraints for the additional conceptual attributes are listed as follows:

$$\forall i, x.\ \mathit{Consequent}(i, x) \rightarrow \mathit{PolicyInst}(i)$$
$$\wedge \exists i_1, i_2.\ (x = \uparrow \mathit{DominatedBy}(i_1, i_2)) \vee (x = \uparrow \mathit{Dominated}(i_1))$$

$$\forall r, i.\ \mathit{GovernedBy}(r, p) \rightarrow \mathit{Request}(r) \wedge \mathit{Policy}(p)$$

The first parameter of *Consequent* must be a policy instance, and the second parameter is the name of either $\mathit{DominatedBy}(i_1, i_2)$ or $\mathit{Dominated}(i_1)$, where i_1 and i_2 are some policy instances. The first and second parameters of *GovernedBy* are requests and policies, respectively.

We also define a new conceptual action type called *unfollows*. When a policy p becomes applicable for a request r in a system *sys*, and a policy instance i for p is generated, the consequent of i becomes true accordingly, and we say *sys* **follows** i. When *sys* **unfollows** i, certain parts of the consequent—the parts identified using $\mathit{Consequent}(i, x)$—become false. The conceptual transition rule for *unfollows* is defined as

$$\forall i, i_1, i_2, \mathit{sys}, x.\ \mathit{DominatedBy}(i_1, i_2) \wedge \neg \mathit{Dominated}(i_2) \wedge \mathit{InSys}(i_1, \mathit{sys}) \wedge \mathit{InSys}(i_2, \mathit{sys})$$
$$\wedge\ \mathit{Consequent}(i_1, x) \rightarrow [\mathit{unfollows}(\mathit{sys}, i_1)](\neg \downarrow x)$$

Let i_1 and i_2 be two policy instances in a system *sys*. If i_1 is dominated by i_2 but i_2 is **not** dominated by any other policy instances, then *sys* unfollows the dominated policy instance i_1. And afterwards the consequents of i_1 become false. According to the syntax of dominance policies we will be defining in Definition 9.2, the consequents are $\mathit{DominatedBy}(pi_1, pi_2)$ and $\mathit{Dominated}(pi_1)$, where pi_1 and pi_2 are policy instances. In summary, *a system unfollows a policy instance if the instance is dominated by a non-dominated policy instance.*

9.3.2 PCR RefABAC: System Model

The logical structure for a system *sys*'s system model $\mathscr{S}_{sys}$ in PCR RefABAC is the same as one in Coco RefABAC. Specifically, $\mathscr{S}_{sys}$ is identified as a tuple $\langle \mathscr{S}_{Attr}, \mathscr{S}_C, \mathscr{S}_{Act}, \mathscr{S}_T, \mathscr{S}_P, \mathscr{S}_D \rangle$.

First, we modify the definition for (regular) policies as follows:

$$\forall r, \vec{x}.\ \exists pi.\ \mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{sys}) \wedge \mathit{PolicyInst}(pi) \wedge \mathit{IsInstOf}(pi, \texttt{pol}) \wedge \phi$$
$$\rightarrow [\mathit{authorizesRequest}(pi, r)](\mathit{Authorized}(r) \wedge \mathit{AuthorizedBy}(r, pi) \wedge \mathit{GovernedBy}(r, \texttt{pol}))$$

$$\forall r, \vec{x}.\ \exists pi.\ \mathit{Active}(r) \wedge \mathit{InSys}(r, \texttt{sys}) \wedge \mathit{PolicyInst}(pi) \wedge \mathit{IsInstOf}(pi, \texttt{pol}) \wedge \phi$$
$$\rightarrow [\mathit{prohibitsRequest}(pi, r)](\mathit{Prohibited}(r) \wedge \mathit{ProhibitedBy}(r, pi) \wedge \mathit{GovernedBy}(r, \texttt{pol})),$$

where ϕ is a Boolean combination of attribute instances and of their negations. For a policy with name `pol`, we add $\mathit{GovernedBy}(r, \texttt{pol})$ to indicate r is governed by the policy. The *GovernedBy* attribute is used in dominance policies, defined as follows.

Definition 9.2 (Dominance Policy)**.** Given two policies with names $\mathtt{p}_1$ and $\mathtt{p}_2$, a **dominance policy** domP between the policy instances of the two policies is a DL$^+$ formula of the following form:

$$\begin{aligned}\forall i_1, i_2, r.\ &\mathit{IsInstOf}(i_1, \mathtt{p}_1) \wedge \mathit{IsInstOf}(i_2, \mathtt{p}_2) \wedge \mathit{GovernedBy}(r, \mathtt{p}_1) \wedge \mathit{GovernedBy}(r, \mathtt{p}_2) \\ &\wedge \exists di.\ \mathit{IsInstOf}(di, \mathtt{domP}) \wedge \mathit{GovernedBy}(r, \mathtt{domP}) \wedge \phi \\ &\rightarrow \mathit{DominatedBy}(i_1, i_2) \wedge \mathit{Dominated}(i_1) \\ &\quad \wedge \mathit{Consequent}(di, \uparrow \mathit{DominatedBy}(i_1, i_2)) \wedge \mathit{Consequent}(di, \uparrow \mathit{Dominated}(i_1)),\end{aligned}$$

where ϕ is a Boolean combination of attribute instances and of their negations.

The formula ϕ in a dominance policy is a DL$^+$ formula that defines the conditions in which the two policy instances i_1 and i_2 are in conflict. When $\mathtt{p}_1$ and $\mathtt{p}_2$ are regular (not dominance) policies, the conflicting condition in ϕ specifies that the same request is authorized by one of them but prohibited by the other, similar to what we defined in Coco RefABAC. On the other hand, when $\mathtt{p}_1$ and $\mathtt{p}_2$ are dominance policies, the conflicting condition is that i_1 says a policy instance x dominates another policy instance y but i_2 says y dominates x.

One may notice that the similarity between the formalism of PCR RefABAC and non-monotonic logics. In our formalism of PCR RefABAC, the essential idea is that the system first computes all the possible dominance relations among the (regular and dominance) policies. Then the system *unfollows* the dominated dominance policies. In terms of Reiter's default logic [Rei80], we could write a dominance policy as a *default rule* of the form

$$\frac{\psi : \mathtt{M} \neg \mathit{Dominated}(di)}{\mathit{DominatedBy}(i_1, i_2) \wedge \mathit{Dominated}(i_1),}$$

where ψ is the precondition of a dominance policy, i.e.

$$\begin{aligned}\psi \equiv \forall i_1, i_2, r.\ &\mathit{IsInstOf}(i_1, \mathtt{p}_1) \wedge \mathit{IsInstOf}(i_2, \mathtt{p}_2) \wedge \mathit{GovernedBy}(r, \mathtt{p}_1) \wedge \mathit{GovernedBy}(r, \mathtt{p}_2) \\ &\wedge \exists di.\ \mathit{IsInstOf}(di, \mathtt{domP}) \wedge \mathit{GovernedBy}(r, \mathtt{domP}) \wedge \phi,\end{aligned}$$

and M can be read as "it is consistent to assume." In other words, when the preconditions of a dominance policy domP are satisfied, and it is consistent to assume that an instance *di* generated from domP is not generated, then the policy instance i_1 is dominated by i_2 and i_1 is dominated. When using non-monotonic logics, we do not need to include action type *unfollows* and its transition rule.

9.3.3 Example

Recall that in Example 9.2, we defined a policy with name exPolPro, which says volunteers are prohibited to read the daily logs of residents. In Example 9.3, we defined a policy with name exPolAut, which says volunteers who help nurses with daily checkups are authorized to read the daily logs.

The dominance policy with name domPolAut defined in Example 9.4 says that when the re-

quester is a volunteer responsible for daily checkups, the policy instance of exPolAut dominates the one of exPolPro. The dominance policy is modified into the PCR syntax as follows:

$$\begin{aligned}\forall i, i', r.\ & \mathit{IsInstOf}(i, \texttt{exPolPro}) \wedge \mathit{IsInstOf}(i', \texttt{exPolAut}) \\ & \wedge \mathit{GovernedBy}(r, \texttt{exPolPro}) \wedge \mathit{GovernedBy}(r, \texttt{exPolAut}) \\ & \wedge \exists i.\mathit{IsInstOf}(i, \texttt{domPolAut}) \wedge \mathit{GovernedBy}(r, \texttt{domPolAut}) \\ & \wedge \mathit{Prohibited}(r, i) \wedge \mathit{Authorized}(r, i') \\ & \wedge \mathit{Requester}(r, s) \wedge \mathit{VolunteerForDailyCheckup}(s) \\ & \rightarrow \mathit{DominatedBy}(i, i') \wedge \mathit{Dominated}(i) \\ & \quad \wedge \mathit{Consequent}(i, \uparrow \mathit{DominatedBy}(i, i')) \wedge \mathit{Consequent}(i, \uparrow \mathit{Dominated}(i))\end{aligned}$$

Suppose some residents at the care facility receive special care and their daily checkups are only done by senior nurses. So volunteers, no matter who, are not allowed to read their daily logs. We formalize this principle as the following dominance policy, denoted by domPolPro, between exPolPro and exPolAut:

$$\begin{aligned}\forall i, i', r, s, o.\ & \mathit{IsInstOf}(i, \texttt{exPolPro}) \wedge \mathit{IsInstOf}(i', \texttt{exPolAut}) \\ & \wedge \mathit{GovernedBy}(r, \texttt{exPolPro}) \wedge \mathit{GovernedBy}(r, \texttt{exPolAut}) \\ & \wedge \exists i.\mathit{IsInstOf}(d\,p\,i, \texttt{domPolPro}) \wedge \mathit{GovernedBy}(r, \texttt{domPolPro}) \\ & \wedge \mathit{Prohibited}(r, i) \wedge \mathit{Authorized}(r, i') \\ & \wedge \mathit{RequestedAct}(r, \uparrow \mathit{readObj}(s, o)) \wedge \mathit{OwnerOf}(x, o) \wedge \mathit{SpecialCare}(x) \\ & \rightarrow \wedge \mathit{DominatedBy}(i', i) \wedge \mathit{Dominated}(i') \\ & \quad \wedge \mathit{Consequent}(i, \uparrow \mathit{DominatedBy}(i', i)) \wedge \mathit{Consequent}(i, \uparrow \mathit{Dominated}(i'))\end{aligned}$$

When a volunteer who is responsible for helping the nurses with daily checkups makes a request to read the daily logs of a resident who is under special care, the two (regular) policies exPolPro and exPolAut and the two dominance policies domPolPro and domPolAut all become applicable. Let req be such a request and exPolProInst, exPolAutInst, domPolProInst, domPolAutInst be the four generated policy instances respectively. According to domPolPro and its policy instance domPolProInst, the following facts are true:

$$\begin{aligned}& \mathit{GovernedBy}(\texttt{req}, \texttt{domPolPro}), \\ & \mathit{DominatedBy}(\texttt{exPolAut}, \texttt{exPolPro}), \\ & \mathit{Dominated}(\texttt{exPolAut}), \\ & \mathit{Consequent}(\texttt{domPolProInst}, \uparrow \mathit{DominatedBy}(\texttt{exPolAut}, \texttt{exPolPro}))) \\ & \mathit{Consequent}(\texttt{domPolProInst}, \uparrow \mathit{Dominated}(\texttt{exPolAut})).\end{aligned}$$

Similarly, according to domPolAut and its policy instance domPolAutInst, the followings are true:

$$\begin{aligned}
&\mathit{GovernedBy}(\texttt{req}, \texttt{domPolAut}),\\
&\mathit{DominatedBy}(\texttt{exPolPro}, \texttt{exPolAut}),\\
&\mathit{Dominated}(\texttt{exPolPro}),\\
&\mathit{Consequent}(\texttt{domPolAutInst}, \uparrow \mathit{DominatedBy}(\texttt{exPolPro}, \texttt{exPolAut})))\\
&\mathit{Consequent}(\texttt{domPolAutInst}, \uparrow \mathit{Dominated}(\texttt{exPolPro})).
\end{aligned}$$

Thus, domPolProInst and domPolAutInst are in conflict. To resolve the conflict, we define a *second level* dominance policy domTwo between domPolPro and domPolAut as follows:

$$\begin{aligned}
\forall pi_1, pi_2, r. &\mathit{IsInstOf}(pi_1, \texttt{domPolPro}) \wedge \mathit{IsInstOf}(pi_2, \texttt{domPolAut})\\
&\wedge \mathit{GovernedBy}(r, \texttt{domPolPro}) \wedge \mathit{GovernedBy}(r, \texttt{domPolAut})\\
&\wedge \exists dpi. \mathit{IsInstOf}(dpi, \texttt{domTwo}) \wedge \mathit{GovernedBy}(r, \texttt{domTwo})\\
&\rightarrow \wedge \mathit{DominatedBy}(pi_2, pi_1) \wedge \mathit{Dominated}(pi_2)\\
&\quad \wedge \mathit{Consequent}(dpi, \uparrow \mathit{DominatedBy}(pi_2, pi_1)) \wedge \mathit{Consequent}(dpi, \uparrow \mathit{Dominated}(pi_2))
\end{aligned}$$

Let domTwoInst be the policy instance generated from domTwo here. Then we have

$$\begin{aligned}
&\mathit{DominatedBy}(\texttt{domPolAutInst}, \texttt{domPolProInst}),\\
&\mathit{Dominated}(\texttt{domPolAutInst}).
\end{aligned}$$

We omit the rest of the facts generated from the consequents of domTwoInst for simplicity. Since *Dominated*(domPolProInst) does not hold, then based on the transition rule for *unfollows*, the consequents of domPolAutInst become false, i.e.,

$$\begin{aligned}
&\neg \mathit{DominatedBy}(\texttt{exPolPro}, \texttt{exPolAut}),\\
&\neg \mathit{Dominated}(\texttt{exPolPro}).
\end{aligned}$$

Therefore, exPolPro is no longer dominated but exPolAut still is. The system will grant the request req.

9.4 Conclusion

We defined two RefABAC frameworks in this chapter, Coco RefABAC and PCR RefABAC, to address the issue when policies conflict with one another. Coco RefABAC incorporates the conflict resolution mechanism in our previous work [Ajm16]. The basic idea is to define a dominance policy for two potentially conflicting policies so that the instances of one policy dominate those of the other under certain conditions. The system grants a request if the request is authorized by a non-dominated

policy instance, and denies it if the request is prohibited by a non-dominated policy instance. PCR RefABAC generalizes Coco RefABAC by defining multi-level dominance policies. If the instances of two dominance policies conflict with each other, then another dominance policy at a higher level can resolve the conflict.

CHAPTER

10

RELATED WORK

In this chapter, we discuss the related work for our dissertation. We first list the works that discuss attribute management (Section 10.1), then policy management (Section 10.2). Table 10.1 summarizes the key works. We also discuss current ABAC models by dividing them into three categories based on their representation (Section 10.3).

10.1 Attribute Management

Attribute management consists of defining policies and mechanisms for operations that disclose, create, delete and modify an attribute value and an attribute itself. We divide our discussion on attribute management into two categories: one focuses on *attribute disclosure* which does not change anything, the other focuses on *attribute modification* which changes an attribute value or an attribute.

10.1.1 Attribute Disclosure

Attribute disclosure is mainly concerned with disclosing the value of an attribute, and whether an entity has a value for an attribute. The major area where protected attribute disclosure is of concern is when two parties trying to establish trust before any authorization decision is made. As mentioned earlier, ABAC is especially suitable for applications where the identity of someone is unknown, and thus authorization decisions are based on authenticating attributes instead of identities. During the authentication, two parties who are unknown to one another need to exchange attributes or credentials that contain sensitive attributes.

Table 10.1 Summary of Related Work

Citation	Attribute Disclosure	Attribute Modification	Policy Disclosure	Policy Modification
Card requirements language [Cam10]	✓	✗	✗	✗
ATN [Yu00; WL00]	✓	✗	✗	✗
Ack Policies [WL02b]	✓	✗	✗	✗
Policy Database [IY05]	✓	✗	✗	✗
Trust Level [Mew07]	✓	✗	✗	✗
Attribute Sensitivity [Zha13]	✓	✗	✗	✗
Multiple PDP [Kol07]	✓	✗	✗	✗
$ABAC_{\alpha}$ [Jin12]	✓	✓	✗	✗
$ABAC_{\beta}$ [Jin14]	✓	✓	✗	✗
Crisis Management [Sma14]	✓	✓	✗	✗
$UCON_{ABC}$ [PS04]	✓	✓	✗	✗
$UCON_{LEGAL}$ [Gop12]	✓	✓	✗	✗
Policy Graph [Sea01]	✗	✗	✓	✗
Policy Indistinguishability [Bra04]	✗	✗	✓	✗
Three-tier System [Ant07]	✗	✗	✓	✗
PBNM System [Ant07]	✗	✗	✓	✗
Causality Graph [Bar07]	✗	✗	✗	✗
Our Reflective ABAC	✓	✓	✓	✓

A card requirements language [Cam10] is such a language that allows for restrictive disclosure of attributes. When a subject requests to use a service provided by a server, the server first tells the subject the applicable policies for evaluating whether the subject can use the service. The subject then reveals the required attributes to the server so that the server can evaluate the policies based on the revealed attributes. The language has a formal semantics and explicitly specifies which attributes must be revealed. It is also technology independent so that any ABAC model may use it for attribute disclosure. The card requirements language is mainly useful for authenticating a subject and it relies on the disclosure of applicable policies.

Automated trust negotiation (ATN) uses an iterative and bilateral approach to establish trust between two parties. Two parties gradually establish trust by exchanging attributes or credentials in multiple passes. In [Yu00; WL00], AC policies are used to protect objects as well as credentials. When a subject requests a resource, the access mediator sends the subject the AC policies of the resource to ask the subject for its attributes or credentials. The subject's attributes are protected by AC policies as well. So the subject sends a counter query, asking the access mediator to satisfy these policies first. However, the act of sending a counter query reveals the fact that the subject indeed has the needed attributes, which would disclose sensitive information.

Acknowledgment policies (Ack policies) [WL02b] are proposed to handle the above problem where a party may infer whether someone holds a credential based on its response of sending

counter queries. Suppose Alice considers an attribute *attr* sensitive. No matter if she has a value for *attr*, she establishes acknowledge policies for *attr*. For each credential that contains *attr*, AC policies may be established to protect the disclosure of the credential. When a requester asks Alice for a credential that contains the sensitive attribute, the AC policies for the credential are sent back to the requester. The requester must satisfy these AC policies first before Alice reveals her credentials. In this way, Alice behaves the same no matter she has the credential or not. [WL02a] gives a more thorough picture of how to enforce Ack policies and how to prevent unauthorized inference of sensitive attributes.

As an alternative to Ack policies, [IY05] proposes *policy databases* to protect sensitive attributes and to conceal the fact that someone may or may not have an attribute. Compared with Ack policies [WL02b; WL02a], this approach focuses on the actual information gain instead of the exchanged messages during trust negotiation. Suppose Alice requests a resource and she needs to provide further attributes. For each of the sensitive attributes she has, she can choose her own policy that governs the attribute and submit the policy anonymously to the policy database for that attribute. Then she pulls the corresponding policies for that attribute from the policy database. On the other hand, for each of the sensitive attributes she does not have, she pulls policies at random from the policy database to use as her own.

[Mew07] defines an AC policy language that incorporates trust levels that a subject has on an object. The trust level of an object must meet the subject's requirements for the subject to disclose its attributes. The work in [Zha13] improves upon [Mew07] by adding attribute sensitivity of subject attributes and dividing trust into direct trust and recommended trust. It also conducts an experiment that demonstrates the accuracy of the method. The approach is independent of the underlying ABAC model.

[Kol07] discusses an ABAC architecture for protecting sensitive attributes. Instead of having one PDP in the ACM that acts in a centralized way, the ACM has multiple PDP's and each subject has a PDP selector. When the subject is asked to disclose an attribute with a service provider, the subject's PDP selector selects which PDP to use based on the subject's privacy preferences. Therefore, each subject can define its individual attribute disclosure rules.

10.1.2 Attribute Modification

Attribute modification is often regarded as an administrative duty that is regulated by the AAP in an ACM. The policy language in $ABAC_\alpha$ [Jin12] offers a way of letting non-administrative subjects add and modify an attribute value. $ABAC_\alpha$ is an ABAC model designed for minimally covering DAC, MAC and RBAC. Its policy language defines four sub-languages: (1) for accessing an object; (2) for assigning and modifying subject attribute values; (3) for assigning object attribute values at the time of creation; and (4) for modifying object attribute values after creation. However, the language is only restricted to adding, modifying or deleting an attribute value of some entity. It assumes that creating and deleting new attributes remain as administrative work. $ABAC_\beta$ [Jin14] develops upon $ABAC_\alpha$ by separating the sub-language (2) two languages, one for assigning subject attribute values

at the time of creation, and the other for modifying subject attribute values after creation.

The ABAC model designed for crisis management systems [Sma14] handles both disclosure and modification of attribute values. It also incorporates trust and purpose when authorizing a request. A *permission assignment function* uses the requester's and target's attributes, the context and the type of requested operation to decide if the requested operation should be authorized. A *req_trust* function specifies the minimum required trust level to grant an access. Finally, a purpose assignment function specifies what purposes the request should have to access the target. When a subject requests to access an attribute, the conditions specified in all of the three functions must be satisfied to authorize the request.

$UCON_{ABC}$ [PS04] is an expressive usage control model. Usage control is a generalization of access control to cover authorizations, obligations, conditions, continuity (ongoing controls) and mutability. $UCON_{ABC}$ has six core models. The pre-authorizations model $UCON_{preA}$ is similar as the core of an AC model, where authorizations are made before a requested operation is exercised. The ongoing-Authorization model $UCON_{onA}$ handles authorizations that need to be made after an operation is exercised, such as revoking an access right for a subject. The pre-obligations model $UCON_{preB}$ introduces pre-obligations that have to be fulfilled when a request is made but before the request is authorized. The ongoing-obligations model $UCON_{onB}$ are obligations that must be fulfilled while an operation is exercises. The pre-conditions $UCON_{preC}$ model define environmental conditions that must be satisfied before an operation is exercises and the ongoing-conditions $UCON_{onC}$ model is for the environment conditions during the execution of an operation. Each of the six core models in $UCON_{ABC}$ is roughly divided into three sub-models, depending on whether attributes are **mutable**, i.e., whether attribute values can be modified. The first kind of sub-model disallows mutable attributes, i.e., no attribute can be modified during authorization or obligation. The second kind is for the attributes that need to be updated before an authorization or obligation is made, and the third is for mutable attributes after an authorization or obligation is made.

10.2 Policy Management

Policy graphs [Sea01] are proposed to protect the disclosure of policies. A policy graph is a directed graph with a source node and a sink node. Each policy graph represents a set of policies that all decide whether a requester can access an entity, where negotiation may be necessary to establish trust between the two parties. The source node is the requester and the sink node is the target. All the other nodes represent attributes that the two negotiation parties need to provide. A directed edge from node A to node B means that a negotiation participant must provide credentials that satisfy the part of a policy represented by node A before it can learn about the part of the policy represented by node B. Trust is iteratively established by finding a path from the requester to the target. [Sea01] also proposes a policy language based on first-order logic without quantifiers or negations. The policy language can be easily represented in policy graphs for restrictive policy disclosure. [Yu03] further builds policy graphs into negotiation protocols, strategies and inter-operations.

Cryptography is also used to conceal policy definitions. [Bra04] defines the notion of *policy indistinguishability* to determine if an adversary can learn about the definition of a policy. A system has full policy indistinguishability if a polynomial-time bound adversary cannot gain a non-negligible advantage in winning the disinguishability game for any two policies for which the adversary does not possess a complete satisfying set of credentials. [Bra04] also defines a secret splitting scheme that improves concealment of policies from non-satisfying recipients.

[Ant07] discusses the issues of managing policies in a fully functional AC system. It specifies that three tiers of policies are needed to serve different purposes. The top tier is for privacy preferences specified in natural or formal languages that are intended for general users. The middle tier includes traditional security policies, such as access control and auditing policies. The bottom tier is for machine representation of policies. The AC and auditing policies must be materialized so that they can be stored properly in an information repository. However, policy management has been largely regarded as an administrative work so far. How to automatically compiling the policies and checking for necessary safety properties remains a challenging issue.

A prototype of *dynamic policy-based network management (PBNM) system* [Pér06] is proposed to handle policy creation and management, policy negotiation, and dynamic policy provisioning. The PBNM system has the following components: a policy decision point (PDP), a policy repository, a policy editor, an operator console, a policy negotiation proxy (PNP), and one or more policy enforcement point (PEP) devices. The authorized subjects create and modify policy documents using the policy editor. When a policy document is ready, the policy editor submits the policy document to the PDP. The PDP then processes the policy document by checking if it conforms to the appropriate XML schema definition and if it adheres to the associated policy specification. If the policy document contains a policy for new remote domains, PDP also negotiates with the new remote domains.

10.3 ABAC Models

We divide current ABAC models into three categories based on the way they are represented. The first category is for ABAC models that use a matrix (Section 10.3.1) as the main representation. The second uses a set notation (Section 10.3.2). The third uses a logical formalism (Section 10.3.3).

10.3.1 ABAC Models Using Matrix

Matrices are mainly used in DAC models, as mentioned in Section 2.2.1. Each row represents a subject and each column an object. The entry at row i and column j is a set of symbols, each of which represents an access right or ownership. Though matrices offer a simple and readable representation of AC models, it may not be suitable for systems with large numbers of subjects and objects. Every time a subjects requests to access an object, the rows and columns need to be iterated once to find the corresponding row and column.

The *typed access matrix model (TAM)* [San92] is defined by introducing strong typing into the

regular DAC model, i.e., every subject and object is assigned a type when it is created and the type does not change thereafter. *Monotonic TAM (MTAM)* is based on TAM and it has strong safety properties. *Ternary MTAM* is also defined in [San92] and it has polynomial safety for its decidable case while retaining the full expressive power of MTAM. The *Dynamic-Typed Access Matrix Model (DTAM)* improves upon TAM by allowing dynamic modification of objects' types. DTAM is also able to describe non-monotonic protection systems for which the safety problem is decidable.

The *attribute-based access matrix model (ABAM)* [Zha05] defines an ABAC model using a matrix representation. ABAC extends the traditional DAC matrix by adding attribute tuples as parameters for each row and column. Each row represents a subject with its attribute value tuple, and each column represents an object with its attribute value tuple. Suppose a subject s has attributes $(a_1, a_2, ..., a_n)$, where $a_1, a_2, ..., a_n$ takes values from domains $V_1, V_2, ..., V_n$ respectively. An *attribute value tuple* $ATT(s)$ for the subject s is $(a_1 = v_1, a_2 = v_2, ..., a_n = v_n)$, where $v_i \in V_i$ for $1 \leq i \leq n$. Table 10.2 shows an example access matrix for two subjects s_1 and s_2 and two objects o_1 and o_2 (we assume subjects and objects are distinct here for simplicity).

Table 10.2 An Example Access Matrix in ABAM

	$o_1 : ATT(o_1)$	$o_2 : ATT(o_2)$
$s_1 : ATT(s_1)$	{*read*, *write*}	
$s_2 : ATT(s_2)$		{*read*}

10.3.2 ABAC Models Using Set Notation

RBAC [San96] is based on sets and binary relations and functions between sets, as mentioned in Section 2.2.3. An RBAC model includes 1) sets of users, subjects, objects, roles and permissions; 2) relations *PA* between the set of permissions and the set of roles, and *UA* between the set of users and the set of roles; and 3) a function *creator* that indicates the creating user of a subject. Many ABAC models follow the tradition of RBAC of using sets, relations and functions as the main way of representing types of components.

One of the first ABAC models [YT05] defines sets of subjects, resources, and environments. Each attribute is a function taking a subject, a resource, or an environment and returning the corresponding attribute value, such as $Role(s) =$"*ServiceConsumer*". The ABAC model does not explicitly define a policy language. From the examples provided in the paper, the policy language is quantifier-free first order logic extended with natural numbers, equality, and predicates from set theory, such as "is an element of" ($\in$) and "is a subset of" ($\subseteq$). set constructors. For instance, the following policy

$$can\_access(s, m, e) \leftarrow (Age(s) \geq 21) \wedge (Rating(m) \in \{\text{R,PG-13,G}\}) \tag{10.1}$$

says that a subject s can rate a movie m if s is no younger than 21 years old and the rating of m is R, PG-13, or G. The ABAC model does not distinguish different types of access rights.

A combination of ABAC and RBAC is introduced in [Wei10]. The combined model uses attributes to assign roles to subjects. The model focuses on the usage of ABAC in web services. A many-to-many attribute condition to service role assignment relation $CSRA \subseteq C \times SR$, where C is the set of attributes and SR a set of service roles, indicates the attribute conditions that a subject must satisfy when it is assigned to the service role. A many-to-many user to service role assignment relation $USRA \subseteq U \times SR$, where U is the set of users, indicates the service roles a user has. Many other relations are also defined in the model to assign users to business roles, users to security domains, and so on.

The representation of ABAC_α [Jin12] is similar as the ABAC model defined in [YT05]. ABAC_α defines sets of users, subjects, objects, attributes and access rights. An attribute is a function that is either atomic or set valued. An atomic-valued attribute returns a single value, such as the age of a user. A set-valued attribute returns a set of values, such as the roles of a user. Each ABAC_α policy consists of a condition expressed in first-order syntax with natural numbers, equality, and predicates from set theory. The above example policy in Equation 10.1 can be expressed as follows in ABAC_α when the access right is specified to *watch*:

$$Authorization_{watch}(s,o) \equiv (Age(s) \geq 21) \wedge (Rating(o) \in \{\text{R,PG-13,G}\}),$$

where *Age* and *Rating* are atomic-valued attributes.

The ABAC model for crisis management systems [Sma14] discussed in Section 10.1.2 also follows the set representation. The conditions in an authorization policy are defined by comparing attribute values using equality and predicates from set theory. The policy described in Equation 10.1 can be formalized in the ABAC model as in the following:

$pa_1(s.ATTR, o.ATTR, CTXT, OP)$:
  if $((s.ATTR[Age] \geq 21) \wedge (o.ATTR[Rating] \in \{\text{R,PG-13,G}\}) \wedge$
    $(o.ATTR = \{Age, Rating, Content\}) \wedge (OP = \{watch\}))$
  return allow
else return deny

We use the *Content* attribute to indicate the content of the movie. The ABAC model [Sma14] is designed for accessing attributes instead of an object itself. It does have a natural representation for reading an object, or watching a movie as shown in the above example. Moreover, the attributes that are not being requested are disclosed as well, such as the *Age* and *Rating* attributes in the example.

10.3.3 ABAC Models Using Logical Formalism

Logical formalisms have been widely used to formally define ABAC models. Wang et al. [Wan04] present a logic-programming formalization of ABAC that uses predicates for sets, elements, and

subsets, as in ABAC$_\alpha$, but a different vocabulary to specify authorization conditions. The authorization conditions are defined using three pre-defined functions: *cando*, *dercando*, and *do*. The functions use attributes of the requester and target to *recursively* decide if a request should be authorized or prohibited. The value of an attribute is represented as a hereditarily finite set of sets. An attribute with all its possible values can be visually represented as a tree. For instance, $\{payment, \{payment, dollar\}\}$ means the attribute *payment* takes the value *dolor*, and the payment attribute is a tree with root *payment* whose two children are *dollar* and *euro*. The idea of using trees, or hierarchies to represent attributes is adopted from [BS00], where credentials and services are organized into hierarchies.

Description logics have also been used in formalizing ABAC. [JFC04] uses description logic $\mathscr{ALC}(\mathscr{D})$ to formalize an ABAC model. TBox axioms represent policies and ABox axioms represent the facts in the system. It can also define policies involving clock/calendar unit representations and duration representations. However, the policies do not explicitly state whether a requested operation is authorized or prohibited. Rather, the policies are used to assign a subject to a role (in the sense of RBAC), which is then implicitly assigned some permissions. For example, the paper claims that the following policy

$$VideoShowServiceUser \equiv \exists domainName.\{abc.com\} \sqcap \geq_3 (userLevel)$$

states that "the user from domain *abc.com* with $userLevel \geq 3$ can access the *VideoShowService*." Using TBox axioms to represent policies do not fully utilize the power of description logics. For instance, even though the fact that an attribute is a sub-attribute of another can be represented using TBox axioms, it does not make sense to formalize it as a policy.

Web Ontology Language (OWL) [Dea04] implements various description logics. [Pri06; Pri07] combines an ontology, represented using Web Ontology Language (OWL), with XACML. The architecture for the ABAC model is extended with an ontology and an inference engine, which can be used for retrieving and inferring missing attribute values from the original attribute repository. For example, suppose the ontology defines a rule stating that if a subject is greater than or equal to 18, then the subject is of full age. If the attribute repository stores an attribute indicating Bob's age is 21, then the inference engine is able to infer that Bob is of full age. [Cir07] develops a RBAC model in which the attributes of subjects and objects are used to dynamically assigns roles and access rights. R*OWL*BAC [Fin08] implements the RBAC model defined in [San96] using OWL, and [SJ16] implements ABAC$_\alpha$ [Jin12] in OWL.

CHAPTER

11

CONCLUSION

11.1 Summary

In Chapter 1, we identified and addressed three issues with the current Attribute-based Access Control (ABAC) models, regarding the management of administrative actions, the relationships among attributes, and the protection of sensitive information. To address these issues, we proposed a reflective ABAC paradigm, called RefABAC. To better understand the formalism of RefABAC, we introduced and clarified some terminology often used in access control (AC) literature, and discussed informally what reflection is (Chapter 2).

Before formally defining RefABAC, we first informally discussed the basic structure of and the main concepts in RefABAC (Chapter 3). Given a set of systems, a RefABAC framework is composed of one conceptual model and one system model for each of the systems. The conceptual model defines the common and essential entities shared among the systems. Each system model defines the domain-dependent entities of its corresponding system. The building blocks in a RefABAC model consist of attributes, attribute instances, constraints, action types, actions, transition rules, and policies. Informally, attribute instances, or attributions, are instances of attributes. Constraints describe the relationships among attributes. Actions are instances of action types, and transition rules describe the pre- and post-conditions of action types. Policies either authorize or prohibit actions. The general workflow of a RefABAC framework is that a subject first makes a request to a system to perform some action. Next, the applicable policies indicate whether the request should be authorized or prohibited. The system to which the request is made then makes the final decision on whether the request should be granted or denied. Finally, if the request is granted, the requesting subject may perform the requested action.

Chapter 4 defines the logical language DL^+ we use to formalize RefABAC models. DL^+ is a variation of dynamic logic (DL) with operators for referencing and dereferencing. Given an entity in the logical language, such as a formula ϕ, the reference of ϕ is written as $\uparrow\phi$, and the reference can be dereferenced via $\downarrow$ to return the original entity so that $\downarrow\uparrow\phi = \phi$. DL^+ allows one to refer to any entity in the same way as an individual. Therefore, we can define attributes on any entities as we do for subjects and objects.

In Chapter 5, we defined a baseline RefABAC framework called Core RefABAC. Core RefABAC uses a simple denial-by-default approach. If a request is not authorized by any policies or if it is prohibited by any policies, it is denied. Otherwise, it is granted. We defined the attributes, constraints, action types and transition rules in the conceptual model and system models for Core RefABAC. We formalized Scenario 1.1 using Core RefABAC (Chapter 6), and implemented examples in MySQL 3.7 (Chapter 7).

We also define three advanced versions of Core RefABAC. DC (Disclosure Control) RefABAC (Chapter 8) improves upon Core RefABAC by keeping track of the disclosed information. After an action is performed, a system records what information has been disclosed during the action. A system can also use the record of the disclosed information to control what further information can be disclosed. Coco RefABAC and PCR RefABAC (Chapter 9) improve upon Core RefABAC by providing a policy-conflict-resolution mechanism. Coco RefABAC incorporates the mechanism we proposed in [Ajm16] by defining (single-level) dominance policies. If two policies are in conflict, a dominance policy may resolve the conflict by stating which policy takes priority. PCR RefABAC generalizes the methodology in Coco RefABAC by defining (multi-level) dominance policies. If two dominance policies are in conflict, a dominance policy at a higher level may resolve the conflict.

11.2 Principal Contributions

We list the principal contributions of the dissertation, following the order of the problems we listed in Chapter 1.

1. In a RefABAC framework, the administrative and non-administrative actions are defined using transition rules and regulated using policies. The representations and the policies for administrative and non-administrative actions are formalized in the same way using the logical language DL^+.

2. We defined the notion of constrains to account for the relationships among attributes.

3. RefABAC can define actions on reading an attribute instance, a policy, a relationship among attributes. The actions are regulated using policies.

4. RefABAC can accommodate distributed systems. Constraints, transition rules and policies may involve attributes from different systems, thus allowing communications among systems.

11.3 Directions for Future Research

We implemented some examples in a SQL language. It would also be worth implementing a RefABAC model using a standard access control language, such as XACML [Oas13]. The policy-conflict-resolution mechanism [Ajm16] we used in defining PCR RefABAC was implemented in Answer Set Programming (ASP). Adapting RefABAC to non-monotonic logics and implementing it in ASP would be interesting too.

We considered first-order logic and temporal logics before settling on dynamic logics. However, they do not explicitly express the fact that an action is performed. Using a combination of temporal and dynamic logics, such as [HT99], would be useful. By adding temporal logics, one may represent and reason about the effective period of a request.

In DC RefABAC, we discussed how to control information closure. From a system's point of view, the system discloses a piece of information to a subject. From a subject's point of view, the subject possesses the knowledge contained in the piece of information. Future research may formalize DC RefABAC using epistemic logic and allow a system to represent and reason about a subject's knowledge and to define policies about knowledge.

We have been focusing on the area of access control. Our policies are only authorization and prohibition policies. One may adapt RefABAC to other areas, such as social norms [Sin15], and add obligations using deontic logic.

BIBLIOGRAPHY

[Ajm16] Ajmeri, N. et al. "Coco: Runtime Reasoning about Conflicting Commitments". *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. Ed. by Kambhampati, S. IJCAI/AAAI Press, 2016, pp. 17–23.

[All96] Allis, V. et al. "Meta-level selection techniques for the control of default reasoning". *Future Generation Computer Systems* **12**.2-3 (1996), pp. 189–201.

[Alv12] Alviano, M. et al. "Disjunctive datalog with existential quantifiers: Semantics, decidability, and complexity issues". *Theory Pract. Log. Program.* **12**.4-5 (2012), pp. 701–718.

[Ant07] Antón, A. I. et al. "A roadmap for comprehensive online privacy policy management". *Commun. ACM* **50**.7 (2007), 109–116.

[Bar00] Barklund, J. et al. "Reflection principles in computational logic". *Journal of Logic and Computation* **10**.6 (2000), pp. 743–786. eprint: `/oup/backfile/content_public/journal/logcom/10/6/10.1093/logcom/10.6.743/2/100743.pdf`.

[Bar07] Barth, A. et al. "Privacy and Utility in Business Processes". *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*. IEEE Computer Society, 2007, pp. 279–294.

[BL73] Bell, D & LaPadula, L. J. *Secure Computer Systems: Mathematical Foundations*. Tech. rep. Mitre Corporation, 1973, pp. 513–523.

[Ben03] Benferhat, S. et al. "A stratification-based approach for handling conflicts in access control". *8th ACM Symposium on Access Control Models and Technologies, SACMAT 2003, Villa Gallia, Como, Italy, June 2-3, 2003, Proceedings*. Ed. by Ferrari, E. & Ferraiolo, D. F. ACM, 2003, pp. 189–195.

[BS00] Bonatti, P. & Samarati, P. "Regulating Service Access and Information Release on the Web". *Proceedings of the 7th ACM Conference on Computer and Communications Security*. CCS '00. Athens, Greece: ACM, 2000, 134–143.

[Bra04] Bradshaw, R. W. et al. "Concealing complex policies with hidden credentials". *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*. Ed. by Atluri, V. et al. ACM, 2004, 146–157.

[Cam10] Camenisch, J. et al. "A card requirements language enabling privacy-preserving access control". *15th ACM Symposium on Access Control Models and Technologies, SACMAT 2010, Pittsburgh, Pennsylvania, USA, June 9-11, 2010, Proceedings*. Ed. by Joshi, J. B. D. & Carminati, B. ACM, 2010, pp. 119–128.

[Cir07] Cirio, L. et al. "A Role and Attribute Based Access Control System Using Semantic Web Technologies". *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, OTM Academy Doctoral Consortium, MONET, OnToContent, ORM, PerSys, PPN, RDDS,*

SSWS, and SWWS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part II. 2007, 1256–1266.

[Cox05] Cox, M. T. "Metacognition in computation: A selected research review". *Artif. Intell.* **169**.2 (2005), pp. 104–141.

[CK08] Crampton, J. & Khambhammettu, H. "Delegation in role-based access control". *Int. J. Inf. Sec.* **7**.2 (2008), pp. 123–136.

[Dea04] Dean, M. et al. *OWL Web Ontology Language Reference*. Cambridge, MA, USA: World Wide Web Consortium (W3C), 2004.

[Def85] Defense, D. of. *Department of Defense Trusted Computer System Evaluation Criteria*. Tech. rep. Department of Defense, 1985, pp. 1–116.

[Elk15] Elkandoussi, A. et al. "Toward resolving access control policy conflict in inter-organizational workflows". *12th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2015, Marrakech, Morocco, November 17-20, 2015*. IEEE Computer Society, 2015, pp. 1–4.

[EB04] Evered, M. & Bögeholz, S. "A Case Study in Access Control Requirements for a Health Information System". *Proceedings of the Australasian Information Security Workshop 2004 (AISW 2004)*. Ed. by James Hogan Paul Montague, M. P. & Steketee, C. Vol. 32. Conferences in Research and Practice in Information Technology. Australian Computer Society. Sydney, NSW, Australia: Australian Computer Society, 2004.

[Fef62] Feferman, S. "Transfinite Recursive Progressions of Axiomatic Theories". *The Journal of Symbolic Logic* **27**.3 (1962), pp. 259–316.

[Fef91] Feferman, S. "Reflecting on Incompleteness". *Journal of Symbolic Logic* **56**.1 (1991), pp. 1–49.

[FK07] Ferraiolo, D. F. & Kuhn, R. D. *Role-based access control*. 2007.

[Fin08] Finin, T. W. et al. "R*OWL*BAC: representing role based access control in *OWL*". *13th ACM Symposium on Access Control Models and Technologies, SACMAT 2008, Estes Park, CO, USA, June 11-13, 2008, Proceedings*. Ed. by Ray, I. & Li, N. ACM, 2008, pp. 73–82.

[Fra] *Framework*. URL: `https://www.merriam-webster.com/dictionary/framework` (visited on 05/18/2020).

[Gop12] Gopalan, R. et al. "$\text{UCON}_{\text{LEGAL}}$: a usage control model for HIPAA". *ACM International Health Informatics Symposium, IHI '12, Miami, FL, USA, January 28-30, 2012*. Ed. by Luo, G. et al. ACM, 2012, pp. 227–236.

[Har76] Harrison, M. A. et al. "Protection in Operating Systems". *Communications of the ACM* **19**.8 (1976), pp. 461–471.

[HT99] Henriksen, J. G. & Thiagarajan, P. S. "Dynamic Linear Time Temporal Logic". *Ann. Pure Appl. Log.* **96**.1-3 (1999), pp. 187–207.

[Hu13] Hu, V. C. et al. “Guide to attribute based access control (ABAC) definition and considerations (draft)”. *NIST special publication* **800**.162 (2013).

[Ini13] Initiative, J. T.F. T. *Security and Privacy Controls for Federal Information Systems and Organizations*. Tech. rep. National Institute of Standards and Technology, 2013.

[IY05] Irwin, K. & Yu, T. “Preventing attribute information leakage in automated trust negotiation”. *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*. Ed. by Atluri, V. et al. ACM, 2005, 36–45.

[Jha15] Jha, S. et al. “Enforcing Separation of Duty in Attribute Based Access Control Systems”. *Proceedings of the 11th International Conference on Information Systems Security*. ICISS 2015. Kolkata, India: Springer, 2015, pp. 61–78.

[JFC04] Jin, P. & Fang-Chun, Y. “Description Logic Modeling of Temporal Attribute-Based Access Control”. *2006 First International Conference on Communications and Electronics*. Hanoi, Vietnam: IEEE, 2004, pp. 414–418.

[Jin14] Jin, X. “Attribute-based Access Control Models and Implementation in Cloud Infrastructure as a Service”. PhD thesis. The University of Texas at San Antonio, 2014.

[Jin12] Jin, X. et al. “A unified attribute-based access control model covering DAC, MAC and RBAC”. *Data and Applications Security and Privacy XXVI*. Ed. by Cuppens-Boulahia, N. Berlin, Heidelberg: Springer, 2012, pp. 41–55.

[Koc02] Koch, M. et al. “Conflict Detection and Resolution in Access Control Policy Specifications”. *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by Nielsen, M. & Engberg, U. Vol. 2303. Lecture Notes in Computer Science. Springer, 2002, pp. 223–237.

[Kol07] Kolter, J. et al. “A Privacy-Enhanced Attribute-Based Access Control System”. *Data and Applications Security XXI, 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Redondo Beach, CA, USA, July 8-11, 2007, Proceedings*. Ed. by Barker, S. & Ahn, G. Vol. 4602. Lecture Notes in Computer Science. Springer, 2007, pp. 129–143.

[KL68] Kreisel, G. & Lévy, A. “Reflection Principles and Their Use for Establishing the Complexity of Axiomatic Systems”. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik* **14**.7-12 (1968), pp. 97–142.

[Lee90] Leeuwen, J. van, ed. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 1990.

[Mew07] Mewar, V. S. et al. “Access Control Model for Web Services with Attribute Disclosure Restriction”. *Proceedings of the The Second International Conference on Availability, Reliability and Security, ARES 2007, The International Dependability Conference - Bridg-*

ing Theory and Practice, April 10-13 2007, Vienna, Austria. IEEE Computer Society, 2007, pp. 524–531.

[Mod] *Model*. URL: https://www.merriam-webster.com/dictionary/model (visited on 05/18/2020).

[Oas13] Oasis. *eXtensible Access Control Markup Language (XACML) Version 3.0*. Tech. rep. 22 February. OASIS, 2013.

[PS04] Park, J. & Sandhu, R. S. "The UCON_{ABC} usage control model". *ACM Trans. Inf. Syst. Secur.* **7**.1 (2004), pp. 128–174.

[Pér06] Pérez, G. M. et al. "Dynamic Policy-Based Network Management for a Secure Coalition Environment". *IEEE Communications Magazine* **44**.11 (2006), 58–64.

[Pra76] Pratt, V. R. "Semantical Considerations on Floyd-Hoare Logic". *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*. IEEE Computer Society, 1976, pp. 109–121.

[Pri06] Priebe, T. et al. "Supporting Attribute-based Access Control with Ontologies". *Proceedings of the The First International Conference on Availability, Reliability and Security*. IEEE Computer Society, 2006, 465–472.

[Pri07] Priebe, T. et al. "Supporting Attribute-based Access Control in Authorization and Authentication Infrastructures with Ontologies". *JSW* **2**.1 (2007), 27–38.

[PO17] Pussewalage, H. S. G. & Oleshchuk, V. A. "Attribute based access control scheme with controlled access delegation for collaborative E-health environments". *J. Inf. Secur. Appl.* **37** (2017), pp. 50–64.

[Rei80] Reiter, R. "A Logic for Default Reasoning". *Artif. Intell.* **13**.1-2 (1980), pp. 81–132.

[San73] Sandhu, R. S. "Lattice-Based Access Control Models". *Computer* 11 (1973), pp. 9–19.

[San92] Sandhu, R. "The typed access matrix model". *Proceedings of IEEE Computer Society Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 1992, 122–136.

[San96] Sandhu, R. S. et al. "Role-based access control models". *Computer* **29**.2 (1996), pp. 38–47.

[Sea01] Seamons, K. E. et al. "Limiting the Disclosure of Access Control Policies during Automated Trust Negotiation". *Proceedings of the Network and Distributed System Security Symposium, NDSS 2001, San Diego, California, USA*. The Internet Society, 2001.

[SJ16] Sharma, N. K. & Joshi, A. "Representing Attribute Based Access Control Policies in OWL". *Tenth IEEE International Conference on Semantic Computing, ICSC 2016, Laguna Hills, CA, USA, February 4-6, 2016*. IEEE Computer Society, 2016, pp. 333–336.

[Sin15] Singh, M. P. "Norms as a Basis for Governing Sociotechnical Systems: Extended Abstract". *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelli-*

gence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. Ed. by Yang, Q. & Wooldridge, M. J. AAAI Press, 2015, pp. 4207–4211.

[Sma14] Smari, W. W. et al. "An extended attribute based access control model with trust and privacy: Application to a collaborative crisis management system". *Future Generation Computer Systems* **31** (2014). Special Section: Advances in Computer Supported Collaboration: Systems and Technologies, 147–168.

[SF16] St-Martin, M. & Felty, A. P. "A verified algorithm for detecting conflicts in XACML access control rules". *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*. Ed. by Avigad, J. & Chlipala, A. ACM, 2016, pp. 166–175.

[Wan04] Wang, L. et al. "A Logic-based Framework for Attribute Based Access Control". *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*. FMSE '04. Washington DC, USA: ACM, 2004, 45–55.

[Wei10] Wei, Y. et al. "An attribute and role based access control model for service-oriented environment". *2010 Chinese Control and Decision Conference*. 2010, pp. 4451–4455.

[Wik20a] Wikipedia contributors. *Conjunctive normal form — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Conjunctive_normal_form&oldid=945093526`. [Online; accessed 19-May-2020]. 2020.

[Wik20b] Wikipedia contributors. *Disjunctive normal form — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Disjunctive_normal_form&oldid=943273816`. [Online; accessed 19-May-2020]. 2020.

[WL00] Winsborough, W. H. & Li, N. "Automated trust negotiation". *In DARPA Information Survivability Conference and Exposition, volume I*. IEEE Press, 2000, pp. 88–102.

[WL02a] Winsborough, W. H. & Li, N. "Protecting sensitive attributes in automated trust negotiation". *Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society, WPES 2002, Washington, DC, USA, November 21, 2002*. Ed. by Jajodia, S. & Samarati, P. ACM, 2002, pp. 41–51.

[WL02b] Winsborough, W. H. & Li, N. "Towards Practical Automated Trust Negotiation". *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002), 5-7 June 2002, Monterey, CA, USA*. IEEE Computer Society, 2002, pp. 92–103.

[Yu00] Yu, T. et al. "PRUNES: an efficient and complete strategy for automated trust negotiation over the Internet". *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 1-4, 2000.* Ed. by Gritzalis, D. et al. ACM, 2000, pp. 210–219.

[Yu03] Yu, T. et al. "Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation". *ACM Trans. Inf. Syst. Secur.* **6**.1 (2003), pp. 1–42.

[YT05] Yuan, E. & Tong, J. “Attribute Based Access Control (ABAC) for Web Services”. *IEEE International Conference on Web Services (ICWS'05)*. IEEE, 2005.

[Zha13] Zhang, G. et al. “Protecting Sensitive Attributes in Attribute Based Access Control”. *Service-Oriented Computing - ICSOC 2012 Workshops - ICSOC 2012, International Workshops ASC, DISA, PAASC, SCEB, SeMaPS, WESOA, and Satellite Events, Shanghai, China, November 12-15, 2012, Revised Selected Papers*. Ed. by Ghose, A. K. et al. Vol. 7759. Lecture Notes in Computer Science. Springer, 2013, pp. 294–305.

[Zha05] Zhang, X. et al. “An Attribute-based Access Matrix Model”. *Proceedings of the 2005 ACM Symposium on Applied Computing*. SAC '05. Santa Fe, New Mexico: ACM, 2005, 359–363.