A Uniform Formal Approach to Business and Access Control Models, Policies and their Combinations

by

Vahid Reza Karimi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2012

© Vahid Reza Karimi 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Access control represents an important part of security in software systems, since access control policies determine which users of a software system have access to what objects and operations and under what constraints. One can view access control models as providing the basis for access control rules. Further, an access control policy can be seen as a combination of one or more rules, and one or more policies can be combined into a set of access control policies that control access to an entire system. The rules and resulting policies can be combined in many different ways, and the combination of rules and policies are included in policy languages.

Approaches to access control (AC) policy languages, such as XACML, do not provide a formal representation for specifying rule- and policy-combining algorithms or for classifying and verifying properties of AC policies. In addition, there is no connection between the rules that form a policy and the general access control and business models on which those rules are based.

Some authors propose formal representations for rule- and policy-combining algorithms. However, the proposed models are not expressive enough to represent formally classes of algorithms related to history of policy outcomes including ordered-permit-overrides, ordered-deny-overrides, and only-one-applicable. In fact, they are not able to express formally any algorithm that involves history including the class related to consensus such as weak-consensus, weak-majority, strong-consensus, strong-majority, and super-majority-permit. In addition, some other authors propose a formal representation but do not present an approach and automated support for the formal verification of any classes of combining algorithms.

The work presented in this thesis provides a uniform formal approach to business and access control models, policies and their combinations. The research involves a new formal representation for access control rules, policies, and their combination and supports formal verification. In addition, the approach explicitly connects the rules to the underlying access control model. Specifically, the approach

- provides a common representation for systematically describing and integrating business processes, access control models, their rules and policies,
- expresses access control rules using an underlying access control model based on an existing augmented business modeling notation,
- can express and verify formally all known policy- and rule-combining algorithms, a result not seen in the literature,

- supports a classification of relevant access control properties that can be verified against policies and their combinations, and
- supports automated formal verification of single policies and combined policy sets based on model checking.

Finally, the approach is applied to an augmented version of the conference management system, a well-known example from the literature. Several properties, whose verification was not possible by prior approaches, such as ones involving history of policy outcomes, are verified in this thesis.

Acknowledgements

I would like to thank my supervisor Professor Donald Cowan for his guidance and financial support during my studies. I also acknowledge my Ph.D. committee members Professors Howard Armitage, Daniel Berry, Michael Godfrey, William McCarthy, and Grant Weddell for their time and efforts to be part of this committee. I would like to thank Research Professor Paulo Alencar for many discussions and his input at the later stage of this study.

I am also thankful for financial support from the Ontario Graduate Scholarship in Science and Technology (OGSST) program.

Table of Contents

List of Tables xiii

List of Figures xv

1 Introduction 1

1.1 Contributions . . . 2

1.2 Research Approach . . . 2

1.3 Thesis Organization . . . 6

2 Related Work 9

2.1 Access Control . . . 9

2.1.1 Access Control Models . . . 10

2.1.2 Access Control Policies and Policy Languages . . . 13

2.2 Business Patterns and Business Processes . . . 16

2.3 Formal Verification . . . 21

2.3.1 Formal Verification of Access Control Policies . . . 23

2.3.2 Formal Verification of Business Processes . . . 28

3 Access Control Models, Rules and Policies and their Combinations 31

3.1 Overview of Representing Classes of AC Models . . . 34

3.2 Representing Classes of Access Control Models . . . 36

3.2.1 Modeling Roles and User Assignments 38
3.2.2 Modeling Permissions 44
3.2.3 A Core Access Control Model Formed by Combining Patterns . . . 46
3.2.4 Advantages of the Core Access Control Model 48
3.2.5 Modeling Constraints, Role Hierarchies, and Mutually Exclusive Roles 52
3.2.6 Extending the Approach to Describe DAC and MAC 56
3.3 Defining Access Control Rules based on Models 58
3.3.1 Access Control Rule Syntax 59
3.3.2 Description of Access Control Rule Syntax 64
3.3.3 Access Control Rule Examples 70
3.3.4 Translation from EBNF to Predicate Logic 73
3.4 Creating Access Control (AC) Policies from AC Rules 75
3.4.1 The Use of Algorithmic Forms 76
3.4.2 The Use of State Machines 76
3.5 Policy-combining Algorithms 80
3.6 An Advantage of the Presented Approach 81

4 Specification of Properties for Access Control and Categorization 87
4.1 Background 87
4.2 AC Property Specification and Categories 88
4.2.1 Any Primitives of Agents, Events, and Resources Individually, and in Connection with AC Results 92
4.2.2 Any Primitives of Agents, Events, and Resources Individually, and their Attributes, and in Connection with AC Results 94
4.2.3 A Combination of Agents, Events, Resources, and their Relationships, and in Connection with AC Results 96
4.2.4 A Combination of Agents, Events, Resources, their Attributes, and their Relationships, and in Connection with AC Results 98
4.3 General Form of AC Property Specification 98
4.4 Related Work on Property Specifications 101

5 Evaluation: Conference Management Case Study **103**

5.1 CONTINUE, Policies, and Properties 105

5.2 Business and AC Models, Rules and Combination 106

5.2.1 Business and AC Models 106

5.2.2 Access Control Rule 106

5.2.3 AC Rule Combination by Algorithmic Form and State Machine . . 109

5.2.4 An Advantage of the Thesis's Approach 116

5.3 Formal Analysis 117

5.3.1 Formal Specification of AC Policies in PROMELA 117

5.3.2 Formal Specification of AC Properties in LTL 119

5.3.3 Verification Results and Expressive Advantage 120

5.4 A Note on the Use of SPIN 123

6 Conclusion **125**

6.1 Summary of Contributions 125

6.2 Limitations 126

6.3 Future Work 127

6.3.1 Other Access Control Models and their Extensions 127

6.3.2 Rights, Delegations, and Obligations 128

6.3.3 Analysis and Formal Methods 128

6.3.4 Privacy 129

6.3.5 Different Domains 129

APPENDICES **131**

A An Overview of REA **133**

A.1 REA as an Ontology 135

A.2 REA Patterns for Policy-level Specification 137

B BNF and EBNF Definitions **139**
B.1 Access Control Rule in BNF 139
B.2 Other BNF and EBNF Definitions 143

C A Brief Background on Logic **159**
C.1 Propositional Logic 159
C.2 Predicate Logic 161
C.3 Linear Temporal Logic 163

D An Overview of SPIN and Alloy **165**
D.1 SPIN 165
D.2 Alloy and Example 168

E The CONTINUE Policies and Properties **179**
E.1 The CONTINUE Policies 179
E.2 The CONTINUE Properties 183

F Other Combining Access Control Algorithms **185**

References **197**

List of Tables

3.1 Entities and attributes of the *role modeling and user assignments* pattern in a table format 43

3.2 Entities and attributes of *permission modeling* pattern 46

3.3 A table representation of policies for the banking example 48

3.4 Additional elements added to access control models 53

3.5 A static separation of duties representation for the banking example 54

3.6 An example of the elements of an access control (AC) rule (a singleton policy) 59

3.7 Extended BNF (EBNF) 61

5.1 Rule-combining algorithms in the context of their use with formal verification 116

5.2 Rule-combining algorithms in the context of using with formal verification 116

5.3 Some LTL and SPIN operators 119

5.4 State space, memory, and verification time, New, First Applicable 121

5.5 State space, memory, and verification time with ordered-permit-overrides rule-combining algorithm 122

A.1 A categorization of REA ontology 136

B.1 The original BNF 139

D.1 SPIN's temporal operators and their notations 167

D.2 Alloy's relational and logical operators 168

List of Figures

1.1 A high-level overview 4

2.1 The RBAC model 12
2.2 An XACML example 15
2.3 (a) An exchange pattern, (b) a pattern to model a policy 17
2.4 Two Exchanges in a Value Chain: (a) renting a car, (b) repairing a car . . 18
2.5 IDEF0 20
2.6 Two rules and their combinations using MTBDDs 25
2.7 The SecureUML metamodel 27

3.1 Access control models, their realizations, and rule definitions 32
3.2 Access control policies as a combination of rules and a rule-combining algorithm 33
3.3 First-applicable Rule-combining Algorithm 34
3.4 The existence of a general policy can be modeled by a *Mirror* pattern. . . 35
3.5 RBAC with permissions as operations on objects 37
3.6 Analysis and design versions of a class 38
3.7 A banking example 39
3.8 The *Basic* pattern, its specialization and enhancement, and example 40
3.9 (a) The *Root* pattern and (b) the specialized *Root* pattern 41
3.10 (a) Pattern for modeling roles and user assignments, (b) an example 42

3.11 (a) Pattern for modeling permissions, (b) an example 45
3.12 (a) A combination of two presented patterns, (b) an example 47
3.13 Two levels of policy descriptions 49
3.14 The Job Classification class is the agent type class. 50
3.15 Aggregation and Group 51
3.16 Implicit versus Explicit Modeling of an Aggregate [93] 52
3.17 The specialized and enhanced Basic pattern, its variation to model role hierarchies, and an example 55
3.18 Role hierarchies representations [34, 87] 56
3.19 The specialized and enhanced Basic pattern, its variation to model mutually exclusive roles, and an example 57
3.20 An example of an access control model 59
3.21 Plane, plane type, flight, flight type, and fleet 60
3.22 Access control rule definition in Extended BNF 64
3.23 Rule Combination using the AC rule definitions for first-applicable algorithm 77
3.24 Combining rules with the AC rule definitions for permit-unless-deny algorithm 78
3.25 A UML state machine using AC rule definitions for first-applicable algorithm 79
3.26 A UML state machine that uses the definitions of AC rules for permit-unless-deny algorithm 80
3.27 An algorithmic description for first-applicable policy-combining algorithm . 81
3.28 A UML state machine for first-applicable policy-combining algorithm . . . 82
3.29 The algorithmic form for weak-consensus policy-combining algorithm . . . 83
3.30 A UML state machine representing weak-consensus policy-combining algorithm 84
3.31 The algorithmic form for weak-majority policy-combining algorithm 85
3.32 A UML state machine for weak-majority policy-combining algorithm . . . 86

4.1 Predicate and propositional versions of expressions 88
4.2 Propositional versions of predicates 89

4.3 Category 1, sub-categories, and sub-category variations 93
4.4 Category 2, sub-categories, and sub-category variations 95
4.5 Temporal implications [47] 96
4.6 Category 3, sub-categories, and sub-category variations 97
4.7 Category 4, sub-categories, and sub-category variations 99
4.8 A general form of AC Property 100

5.1 A summary of this chapter for access control policy specification 104
5.2 A summary of this chapter for property description and verification 104
5.3 A REA model of submitting and reviewing papers 107
5.4 A core access control model, based on this thesis's presentation 107
5.5 An example of access control model for reviewing papers 108
5.6 The defined AC rules and states for ordered-permit-overrides 110
5.7 A UML state machine using the defined AC rules for ordered-permit-overrides 111
5.8 The defined AC rules and states for ordered-deny-overrides 112
5.9 A UML state machine using the defined AC rules for ordered-deny-overrides 113
5.10 The defined AC rules and states for only-one-applicable 114
5.11 A UML state machine using the defined AC rules for only-one-applicable . 115

A.1 A REA model 134
A.2 Resource and its sub-categories 134
A.3 A state diagram of a resource 135
A.4 Patterns for policy-level specifications 137
A.5 A flight example for policy-level specifications 138

B.1 Access control rule definition in BNF 143
B.2 Agent-related definitions in Extended BNF 146
B.3 Agent-related definitions in BNF 148
B.4 Resource-related definitions in Extended BNF 151

B.5 Resource-related definitions in BNF 153
B.6 Event definitions in Extended BNF 155
B.7 Event-related definitions in BNF 157

D.1 A verification of policies that are not based on a model. 171
D.2 Visualizing of a REA rent business model using Alloy 172
D.3 A counter-example indicating an error 174
D.4 A counterexample of separation of duties 176
D.5 Employees and Customers 176

F.1 The algorithmic form for strong-consensus policy-combining algorithm . . . 186
F.2 A UML state machine for strong-consensus policy-combining algorithm . . 187
F.3 The algorithmic form for strong-majority policy-combining algorithm . . . 188
F.4 A UML state machine for strong-majority policy-combining algorithm . . . 189
F.5 The algorithmic form for super-majority-permit policy-combining algorithm 190
F.6 A UML state machine for super-majority-permit policy-combining algorithm 191
F.7 The defined AC rules and states for weak-consensus rule-combining algorithm 192
F.8 A UML state machine that uses AC rule definitions for weak-consensus rule-combining algorithm 193
F.9 The defined AC rules and states for weak-majority rule-combining algorithm 194
F.10 A UML state machine that uses the AC rule definitions for weak-majority rule-combining algorithm 195

Chapter 1

Introduction

Access control is an important part of business processes and is a significant part of security, as "security depends on authentication, authorization, and auditing: the gold standard [65]." Access control constitutes an important component of operating systems, database management systems (DBMS), and applications. Access control policies define which users have access to what objects and operations and describe any existing constraints. Several incidents of information leaks in real systems owing to the implementation of incorrect access control policies (e.g., [109], [17]) have been reported. These incidents indicate the need for thorough analysis of access control policies and their properties. Although testing reveals many existing errors in software systems, errors still remain undetected even in safety-critical and economically vital systems [29].

One can view access control models as providing the basis for access control rules. Further, an access control policy can be seen as a combination of one or more rules, and one or more policies can be combined into a set of access control policies that control access to an entire system. The rules and resulting policies can be combined in many different ways, and the combination of rules and policies are included in policy languages.

Approaches to access control (AC) policy languages, such as XACML, do not provide a formal representation for specifying rule- and policy-combining algorithms or for classifying and verifying properties of AC policies. In addition, there is no connection between the rules that form a policy and the general access control and business models on which those rules are based.

Some authors propose formal representations for rule- and policy-combining algorithms ([36], [59]). However, the proposed models are not expressive enough to represent formally classes of algorithms related to history of policy outcomes including ordered-permit-

overrides, ordered-deny-overrides, and only-one-applicable. In fact, they are not able to express formally any algorithm that involves history including the class related to consensus such as weak-consensus, weak-majority, strong-consensus, strong-majority, and super-majority-permit. In addition, some other authors (e.g., [66]) propose a formal representation but do not present an approach and automated support for the formal verification of any classes of combining algorithms.

1.1 Contributions

The work presented in this thesis provides a uniform formal approach to business and access control models, policies and their combinations. The research involves a new formal representation for access control rules, policies, and their combination and supports formal verification. In addition, the approach explicitly connects the rules to the underlying access control model. Specifically, the approach

- provides a common representation for systematically describing and integrating business processes, access control models, their rules and policies,
- expresses access control rules using an underlying access control model based on an existing augmented business modeling notation,
- can express and verify formally all known policy- and rule-combining algorithms, a result not seen in the literature,
- supports a classification of relevant access control properties that can be verified against policies and their combinations, and
- supports automated formal verification of single policies and combined policy sets based on model checking.

1.2 Research Approach

This section provides a bird's eye view of the research approach and then subsequently a more detailed presentation of this work.

An Overview: Access control (AC) is an essential part of any business process. This thesis proposes a new common representation for business models and classes of access

control models based on the resources, events, agents (REA) modeling approach to business processes. The common modeling representation is defined incrementally. First, access control primitives are mapped onto novel REA-based access control patterns. Then, REA-based access control patterns are combined to define access control models. Based on these models, access control rules are defined. General access control policies, which are combinations of rules, are formed by augmenting the common representation with state machines. This approach allows the specification and verification of more expressive access control policies than are reported in the literature.

Once a model is constructed, it is formalized and then mapped to a language to facilitate model checking analysis. Specifically, access control policies and related business processes are specified in PROMELA, the language of the SPIN model checker. Finally, formal AC properties to be verified are specified in linear temporal logic (LTL) and checked against the access control policies. A conference management case study [36] is augmented with these new access control policies to illustrate the proposed approach.

A property categorization for the specification of access control properties based on property patterns [28, 27, 62] and using REA primitives is introduced. This categorization involves four classes based on REA primitives, and each class has five subclasses (absence, existence, universality, precedence, and response) based on property patterns. In addition, this classification provides guidelines for reusing formal property specification. Currently, there is no such categorization of access control properties reported in the literature.

The Detailed View: This thesis presents a common modeling representation for describing both business and access control models, and their associated rules and policies [55]. The AC policy model is then formalized using state machines, and the properties of the model are specified in linear temporal logic (LTL). To mechanize the verification of the properties against the model, the formal specifications of the model and properties are translated into PROMELA (the language of the SPIN model checker) and LTL, which is used by SPIN, respectively. The entire approach is shown in Figure 1.1, where Box A shows the model, Box B shows the formalization, and Box C shows the automated support for formal verification. Each box contains several constituents that are described in the next few labeled paragraphs.

A. The Common Modeling Representation: Boxes A1, A2, and A3, which are components of this model, are described next.

A1. REA and Access Control Models: This thesis uses the REA model to represent business models for two reasons. First, REA contains rules that define relationships among resources, events, and agents to express business processes. Second, REA has patterns that provide descriptions of generic business processes. REA uses an object-oriented approach

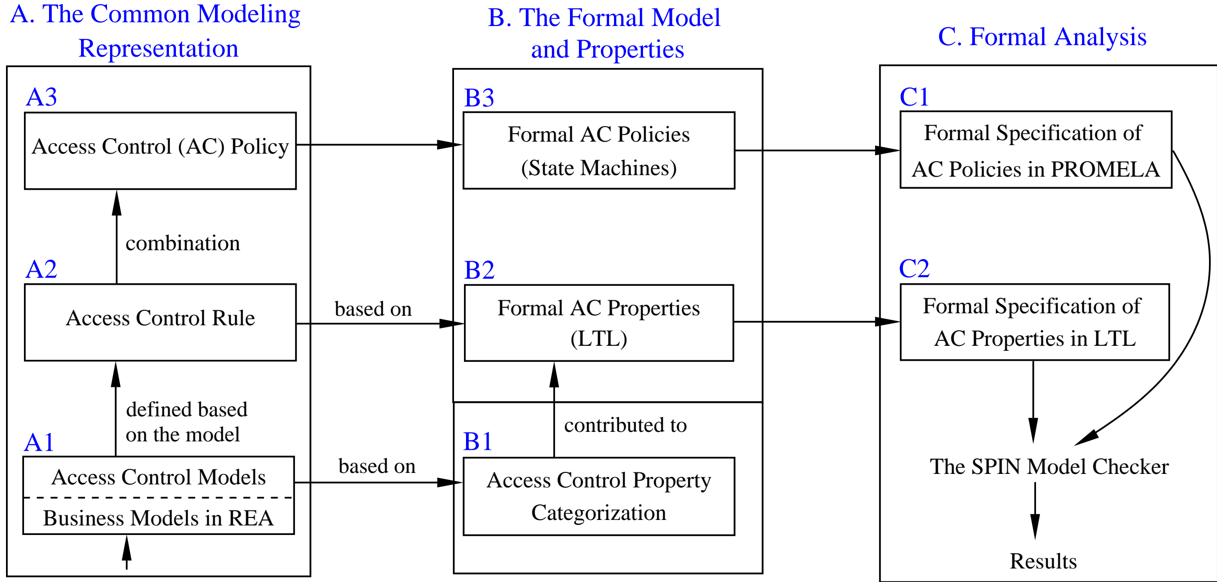


Figure 1.1: A high-level overview

and UML diagrams to represent business models. Access control models are then built based on REA primitives and patterns. Box A1 of Figure 1.1 shows this approach.

A first contribution shows how the common representation can be used to describe both business and access control models. According to Ferraiolo et al. [33], an access control model can be described by the five elements *users*, *objects*, *subjects*, *operations*, *permissions*, and the relationships among them. In Chapter 3, as a second contribution, these five elements are mapped to resources, events, and agents, leading to three generic patterns that support the representation of the main categories of access control (i.e., role-based access control, discretionary access control, and mandatory access control). However, the approach can be used to support any access control model that depends on these five elements (users, objects, subjects, operations, and permissions). Chapter 3 discusses these topics in detail.

A2. Access Control Rule: Moving from Box A1 to Box A2 of Figure 1.1 portrays the idea that access control (AC) rules are defined based on AC models. The grammar of AC rules is declared in Extended Backus-Naur Form (EBNF). The contributions with regard to AC rules are twofold: a definition of AC rules is based on the elements of the underlying AC model, and the syntax in EBNF for the AC rules is provided. Chapter 3 presents AC rules and their definitions in EBNF. In contrast, there is no connection between the AC rules and an underlying AC model.

A3. Access Control Policy: In its simplest form, an access control policy consists of a single access control rule, but several such rules are generally combined to make a policy. As a contribution, this thesis uses state machines to describe all-known policies. To summarize, the novelty of this research includes a) a rigorous systematic approach using general forms of state machines (algorithmic forms) to define AC policies and b) policies based on AC rules, which are based on AC models. Box A3 of Figure 1.1 shows access control policy as a combination of AC rules. Chapters 3 and 5 provide detailed discussion of this topic.

B. The Formal Model and Properties: Boxes B1, B2, and B3 are related to the categorization of properties, formal representation of AC policies, their combination and their properties.

B1. Access Control Property Categorization: Box B1 of Figure 1.1 shows a novel categorization for describing access control properties. This categorization uses property patterns and REA primitives and their combinations. This classification involves four categories based on REA primitives, and each category has five subcategories (absence, existence, universality, precedence, and response) based on property patterns. This topic is described in Chapter 4.

B2. Formal AC Property: Formal AC properties in linear temporal logic (LTL) are specified based on AC rules in EBNF. Chapter 4 describes the AC properties and their syntax in EBNF.

B3. Formal AC Policies: Box B3 of Figure 1.1 is the formalization of AC policies using AC rules and state machines where the AC rules govern the transitions between states. The state machine representation allows for the description and verification of more complex policies than are currently reported in the literature. For example, policies relying on history of policy outcomes and policies relying on consensus can be represented and verified in this formalism. This topic is discussed in Chapters 3 and 5.

C. Formal Analysis: This box shows formal analysis of access control policies and their properties in the context of business processes. Analysis of this type is needed because multiple access control policies can interact to produce undesirable behaviour.

C1. Formal Specification of AC Policies: This step consists of the specification of state machines in the specification language of a model checker. Since this thesis uses the SPIN model checker, state machines are encoded in PROMELA.

C2. Formal Specification of AC Properties: Properties are specified using the general form of properties from Box B2 and the specific notation of LTL for the SPIN model checker.

Finally, the properties are verified against the policy model using SPIN, and results are obtained.

A Case Study: An augmented version of the conference management case study called CONTINUE that uses Boxes A, B, and C is described in Chapter 5 to show the method presented in this thesis. This case study is enhanced with rule- and policy-combining algorithms that involve history of policy outcomes, which have not been formally specified and verified in the literature.

The notation used in this thesis makes it possible to specify and verify formally all known access control policy combinations, thereby significantly extending the coverage and correctness of AC decision-making in general and in the conference management system in particular. For instance, the use of only-one-applicable combining algorithm, which has not been formally represented before, makes it possible to detect whether access control policies have rules with contradictory results (i.e., one rule permits a case, and the other denies the same case). In this situation, the final decision according to only-one-applicable is indeterminate. By knowing that the result is indeterminate, the inconsistency can be resolved by removing the conflicting rule. As another example, the strong-majority combining algorithm can be represented and verified formally, allowing policies that take into consideration voting decisions under different majority conditions.

This case study uses access control models, rules, and policies, and their formalization that are presented in Chapter 3. In addition, the definitions, categories, and general forms of LTL AC properties from Chapter 4 are used to represent this case study's properties. The case study is represented and verified using the SPIN model checker and its specification languages PROMELA and LTL.

AC properties of CONTINUE are verified including several properties where verification was not possible by prior approaches (as are attested by these approaches [36] and [59]) if certain rule-combining algorithms, such as properties involving history of policy outcomes, e.g., ordered-permit-overrides and only-one-applicable. The time and memory usage of the verification results are presented to indicate that the approach described in this thesis is practical.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents the related work, including prior research on access control models and policies, business patterns and business process models, and formal verification of both access control policies and business processes. Using

a running example, Chapter 3 describes the common modeling representation that consists of access control models, access control rules and policies, and their combinations. This chapter covers Boxes A1, A2, A3, and B3 of Figure 1.1. Chapter 4 discusses a categorization and specification of properties for business-related access control policies and corresponds to Boxes B1 and B2 of Figure 1.1. Building upon Chapters 3 and 4, Chapter 5 uses a conference management case study to evaluate the approach presented in this thesis. This chapter revisits Boxes A and B and is related to Box C of Figure 1.1. This chapter also describes the expressiveness of this thesis's approach and compares the verification results with the outcomes of two previous works that use the same case study but apply different analysis approaches. Finally, Chapter 6 concludes by discussing contributions, limitations, and future extensions of this work.

Chapter 2

Related Work

Summary: This thesis discusses access control (AC) models, access control rules and policies (in addition to the combination of AC rules and policies) and uses a business process modeling notation. It also discusses the expression of access control properties in addition to formal specification and verification of these properties. Therefore, the related areas include access control models, access control policies, business processes, and formal verification of business processes and access control policies.

Section 2.1 discusses related access control work, and in two subsections, describes access control models, access control policies and policy languages. Section 2.2 provides an overview of business processes. Finally, related work on the formal verification of access control policies and business processes is discussed in two subsections of Section 2.3.

2.1 Access Control

Authentication, access control, and audit form the foundation of information and system security [89], as previously mentioned in Chapter 1. The focus of this thesis is on one aspect of security: access control or authorization.

An early work on access control in computer systems includes Lampson's research [64] in which an access control or "protection" is defined as a "general term for all the mechanisms that control the access of a program to other things in the system."

Section 2.1.1 presents access control models, and Section 2.1.2 describes access control policies and policy languages. These topics are related areas because this work describes

the addition of a class of access control models and a policy language to a business model in Chapter 3.

2.1.1 Access Control Models

This thesis presents access control models based on the five elements, as described in Chapter 1, that are needed by any access control model. There are numerous access control models (and policies) that are variously classified. In one classification [98, 12], three main categories of access control are Mandatory Access Control, Discretionary Access Control, and Role-based Access Control. These three categories are described next.

Mandatory Access Control (MAC): In the 1980s, the Trusted Computer System Evaluation Criteria (TCSEC) document (first version 1983; second version 1985), commonly referred to as *the orange book* because of the color of this document's cover, described Mandatory Access Control (MAC). MAC is concerned with confidentiality, a need that evolved in a military information setting. Access is granted or denied based on comparing the security label of an object with the security clearance of a subject. The tight control of MAC is not suitable in all cases, but this type of control is needed in some specific situations.

In the 1970s, David Bell and Leonard La Padula described an example of MAC. Their description of MAC had two main entities: a subject and an object; each had also a *security class*. A *security clearance* describes a subject's *security class*, whereas a *security classification* represents an object's *security class* [98]. Bell and La Padula described two rules for MAC [98]: 1) simple security property (ss-property): a subject can read an object only if the subject's security class is equal or higher than the object's security class; hence, *no read up* represents an alternative name for ss-property, 2) star property ($*$-property): a subject can write an object only if the security level of an object is equal or higher than the subject's class; hence, *no write down* is another name for $*$-property. If a subject can write at the level lower than his/her security class, then a subsequent read by a subject at this lower level can compromise confidentiality. BLP also includes an additional property, called the discretionary property (*ds-property*) that enables users to grant others access to an object; this discretionary access is constrained by the first two rules [98].

The goal of MAC is to provide confidentiality of information, and this main focus on confidentiality makes MAC rigid. MAC contrasts with the needs of commercial systems, including business processes. As a result, MAC and its specific variations or examples are not emphasized in this thesis since the AC model presentation in Chapter 3 does not start

off with MAC. However, Chapter 3 shows how the AC model that is presented in that chapter can be modified to represent MAC.

Discretionary Access Control (DAC): The TCSEC's document also described DAC. As the name indicates, the owners of an object use their own discretion to grant others access to that object. DAC is also called *Identity-based Access Control* (IBAC) because of its emphasis on a user's identity in allowing or denying access to an object [15]. The notion of control is more applicable than the concept of ownership for cases such as that of an organization that owns objects and has employees that control these objects [32]. Nevertheless, an individual or an identity holds the ownership or control of objects. The BLP *ds-property* represents the basis of DAC by providing for discretionary access control [98]. One disadvantage of discretionary access control, for which it has been continuously criticized, is the lack of assurance. For instance, individuals with ownership or control over objects can grant an object access to other individuals who should not have access.

Although DAC is used in commercial systems, it is known for its weakness in granting access [33]: if a person A grants access to B, then B can grant access to C who is unknown to A. Discretionary Access Control is not emphasized in this thesis because of the formal verification aspect of this thesis and its emphasis on correctness of policies. Even though DAC is not emphasized, since the presentation of AC models does not start with it in the next chapter, Chapter 3 shows the way in which the AC model presented in that chapter can be adjusted to represent DAC.

Role-based Access Control (RBAC): The RBAC model [32] was introduced in 1992 as a generalized model of access control by adapting the existing role-based access control approaches. RBAC represents a conceptually simple model in which an object's access is determined by a subject's role. RBAC [34] introduces roles between users and permissions, and permissions are assigned to roles instead of to users. This arrangement makes permission assignment easier because permission of roles changes less frequently than the user roles (i.e., people change jobs or are assigned to various roles more often than permissions are changed). In addition, an estimate indicates that the number of roles is about 3-4% of the number of users [90]; in other words, each individual usually takes three to four roles in an organization, that is, the ratio of users to roles is about three to four. As another advantage, introducing roles simplifies the assignment of permissions because the number of permissions in an enterprise is larger than the number of roles. Figure 2.1 [87] shows RBAC.

RBAC was extended and described as four models [87]: RBAC_0 is the base or the core model. The core model includes neither hierarchy of roles nor any constraints (e.g., separation of duties). $RBAC_1$ is the core model with role hierarchies. $RBAC_2$ is the core

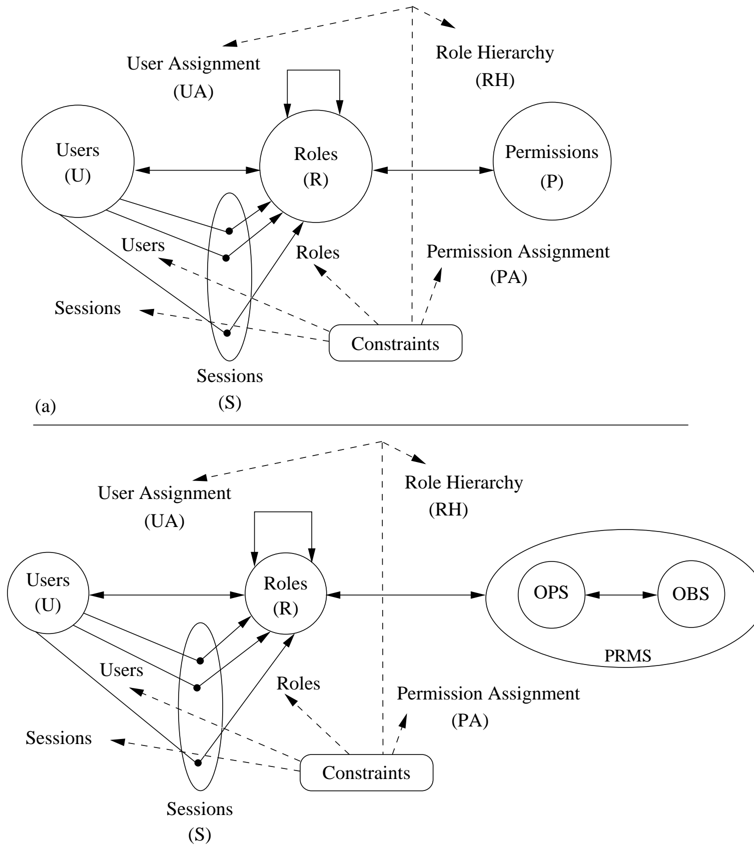


Figure 2.1: The RBAC model

model with constraints such as restrictions (e.g., a purchasing manager cannot be a paying manager). $RBAC_3$ is the comprehensive model and consists of the core, role hierarchy, and constraint models. Figure 2.1(a) shows role hierarchies and constraints; constraints are a significant element of RBAC. The arrowheads on both ends of the solid lines indicate many-to-many relations. Figure 2.1(b) [34] is a representation similar to Figure 2.1(a) and shows permissions (PRMS) as many-to-many relationships of operations (OPS) on objects (OBS).

RBAC has been used frequently in commercial systems and does not have the drawbacks of MAC and DAC, as described previously; therefore, RBAC is chosen as the model for this thesis.

One extension to RBAC is Rule-based RBAC (RB-RBAC). RB-RBAC [1] assigns users to roles on the basis of users' attributes because RBAC does not describe how to assign

users to roles. As described in Chapter 3, RB-RBAC can be represented by the model that is presented in the next chapter because of the use of attributes in the discussed model.

Bertino et al. [14] propose temporal-RBAC (TRBAC), an extension to RBAC, to enable periodical activation and deactivation of roles; as a result, roles can be either active or inactive within a certain time. The authors provide the syntax and semantics of TRBAC and use an example. Although Bertino et al. [14] describe the features of TRBAC in detail, it has to be determined whether such an extensive extension to RBAC hinders its practicality.

RBAC is not concerned with the sequence of events, and layering other models, such as Task-based Authorization Controls (TBAC), on top of it is recommended [87].

An RBAC Extension: Task-based Authorization Controls. Thomas and Sandhu [102], in describing the need for a task-based authorization control paradigm, defined a task or an activity a) being long lived, b) as perhaps including subtasks to be authorized individually or collectively, c) as possibly having multiple individuals to perform each subtask, and d) as being distributed in space and time. Subsequently, Thomas and Sandhu [103] introduced the Task-based Authorization Controls (TBAC) paradigm, somewhat different from their initial work. In addition to subject, object, and action sets, TBAC includes two additional sets: "usage and validity count" and "authorization step" [103]. Each authorization step contains its own subjects, objects, and actions sets. In addition, TBAC includes the notion of "usage" and "validity" with a permission: no permission is unrestricted; each is limited and has an expiration period.

Similar to RBAC, TBAC is described as one of four models [103]: TBAC_0 (the core or base model), TBAC_1 (the model with composite authorization), TBAC_2 (the model with constraints), and TBAC_3 (the consolidated model). A TBAC_1 composite authorization includes more than one authorization step; for instance, transferring money from one account to another account consists of two steps. One step consists of an authorization to deduct money from a source account, and the addition of the money to a destination account represents the second authorization step.

TBAC and RBAC are not in contention, but TBAC is mainly a research work, and the extent of the use of this model in commercial systems, if any, is not discussed in the literature. For this reason, TBAC is not further discussed in this thesis.

2.1.2 Access Control Policies and Policy Languages

For the specification of access control policies, several research works present different policy languages that are not explicitly based on an access control model. In addition,

current approaches to policy languages are not able to specify and verify formally all forms of rule- and policy-combination. In contrast, this thesis describes the process of I) creating access control models and basing rules on access control policies and policy sets explicitly on these models and II) further provides rule- and policy-combining representation that supports formal specification and verification. A brief overview of policy languages in the literature is provided next as a contrast to this thesis's approach.

A policy language is a language that describes security policies. These languages are classified into two categories [15]: 1) high-level policy languages describe conditions and constraints on entities independent of implementations and 2) low-level policy languages represent constraints as inputs or arguments to commands (e.g., the UNIX-based X-Windows provides a language to control access to the console such that typing *xhost +hostnamea -hostnameb* makes the system allow access from *hostnamea* but disallows access from *hostnameb*.) In this thesis, whenever the term policy language is used, the term means a high-level one.

Various such languages exist; some are, implicitly or explicitly, improvements and extensions of prior ones. eXtensible Access Control Markup Language (XACML) is the standard and general-purpose XML-based access control language. In it, a *rule* constitutes the most elementary part of a *policy* and is composed of three components [79]: 1) a *target* that consists of *subject*, *resource*, *action*, and *environment* elements, 2) a *condition* that is an optional element, 3) an *effect* that can be either *deny* or *permit*.

Figure 2.2 [79] (with some abbreviations) represents an access control policy stating that"any user with an e-mail name in the 'med.example.com' namespace is allowed to perform any *action* on any *resource*." Line one of this figure identifies a rule named Rule1 as an abbreviated id; a value of "Permit" for the *effect* is also shown. A *condition* does not exist for this example. In addition, *resource*, *action*, and *environment*, as three components of targets, have not been specified in Figure 2.2 because the access control policy in English states that any *action* and *resource* is allowed, and no description of an *environment* exists. Lines five to twelve use a matching function with the "med.example.com" literal value and a pointer using *SubjectAtributeDesignator* to identify a specific subject attribute.

The XACML specification [79] includes the *core and hierarchical role-based access control (RBAC) profile* within XACML v2.0, but separation of duties is currently beyond the scope of this profile. XACML describes rule- and policy-combining algorithms in English and provides pseudo-codes for the application of these algorithms. In contrast, Chapters 3 and 5 of this thesis use these algorithms and provide their descriptions in algorithmic forms and state machines. In addition, the transitions between the states are governed by the elements of access control rules, which are defined in extended BNF (EBNF).

```
<Rule RuleId=''...Rule1'' Effect=''Permit''>
 <Target>
  <Subjects>
   <Subject>
    <SubjectMatch MatchId=''...function:...Name-match''>
      <AttributeValue DataType=''...#string''>
          med.example.com
      </AttributeValue>
      <SubjectAttributeDesignator
           AttributeId=''...subject-id''
           DataType=''...Name''/>
    </SubjectMatch>
   </Subject>
  </Subjects>
 </Target>
</Rule>
```

Figure 2.2: An XACML example

Finin et al. [35] presented ROWLBAC for the integration of RBAC with OWL. They used OWL to represent policies that can be represented by the RBAC model. Policy languages in the literature with the exception of ROWLBAC and to a certain extent the XACML profile of RBAC are not model-based. Despite the use of a language to describe a model (RBAC), ROWLBAC does not describe AC rule and policy combinations that exist in many policy languages. The description of ROWLBAC is limited to RBAC, whereas Chapters 3 and 5 and Appendix F clarify an approach for the combinations of AC policies and rules, which are based on the RBAC model in REA. These combinations are beyond the capability of RBAC and ROWLBAC.

Knowledgable Agent-oriented System (KAoS) [104] enables specifying and resolving conflicts of policies; policies are specified as an ontology. KAoS, originally a framework for software agents, can include policies describing positive or negative authorizations. KAoS includes the Policy Adminstration Tool (KPAT), a graphical user interface for specifying and revising policies, and an analysis framework that relies on algorithms written as an extension of Java Theorem Prover (JTV) for detecting policy conflicts [104]. The analysis of access control policies is also a part of this thesis and is described later. Nevertheless, similar to many policy languages, access control policies defined by KAoS have no connection to an explicit access control model. In addition, KAoS does not include the combination of AC rules and policies that currently exist in policy languages. Therefore, the last two points, i.e., AC rules based on a model and the inclusion of rule and policy

combinations, differentiate KAoS and the work presented in this thesis.

Rei [54] exemplifies a policy language and is capable of specifying a wide range of policies, such as access control policies. These policies are intended not to be tied to a specific application/domain. Rei's policy representation can include rights and prohibitions corresponding to KAoS's positive authorization and negative authorization, respectively. An entity, a, can perform an action, b, if certain *conditions* are met, a statement expressed as *has(a,right(b,conditions))*. Similar to KAoS, there is a core ontology that can be extended or retracted to fulfill specific needs of various applications. Rei relies on a Prolog engine for policy analysis. If conflicts exist, such as overlapping of subject, object, and target, then *metapolicies*, such as those created by specifying orders for policies, can be defined to resolve these conflicts. Similarly, the analysis of access control policies is also a part of this thesis and is described later. Nevertheless, Rei is a language that does not define rules based on an explicit model and does not discuss rule and policy combinations; therefore, Rei's approach differs from the approach that is presented in this thesis.

2.2 Business Patterns and Business Processes

This thesis uses REA, which is presented as business patterns, for describing business processes. The common representation (Chapter 3) is based on the Resource-Event-Agent (REA) model, new combinations of access control and business patterns, and state machines. Therefore, a brief overview of business patterns and REA is provided next. In addition, other modeling notations for business processes are described.

Although the idea of patterns in software engineering was popularized by design patterns, which are still "the most popular and influential pattern work" [20], there are other types of patterns. Business patterns are one type of pattern, related to business domains, and are used to create business models that are simplified views of a business. The term "business" in business patterns describes an organization that uses resources and has goals [31].

The Resource-Event-Agent (REA) model is a collection of business patterns based on the notions of Resources, Events, and Agents. These patterns are collectively called REA. REA was initially introduced by McCarthy [68] and has been extended over the years. Hruby et al. [48] presented REA as a group of business patterns. (An overview of REA is provided in Appendix A.) Figures 2.3(a) and 2.3(b) show an exchange pattern and another pattern to model a policy, respectively [48]. These two patterns can be combined by merging their common elements.

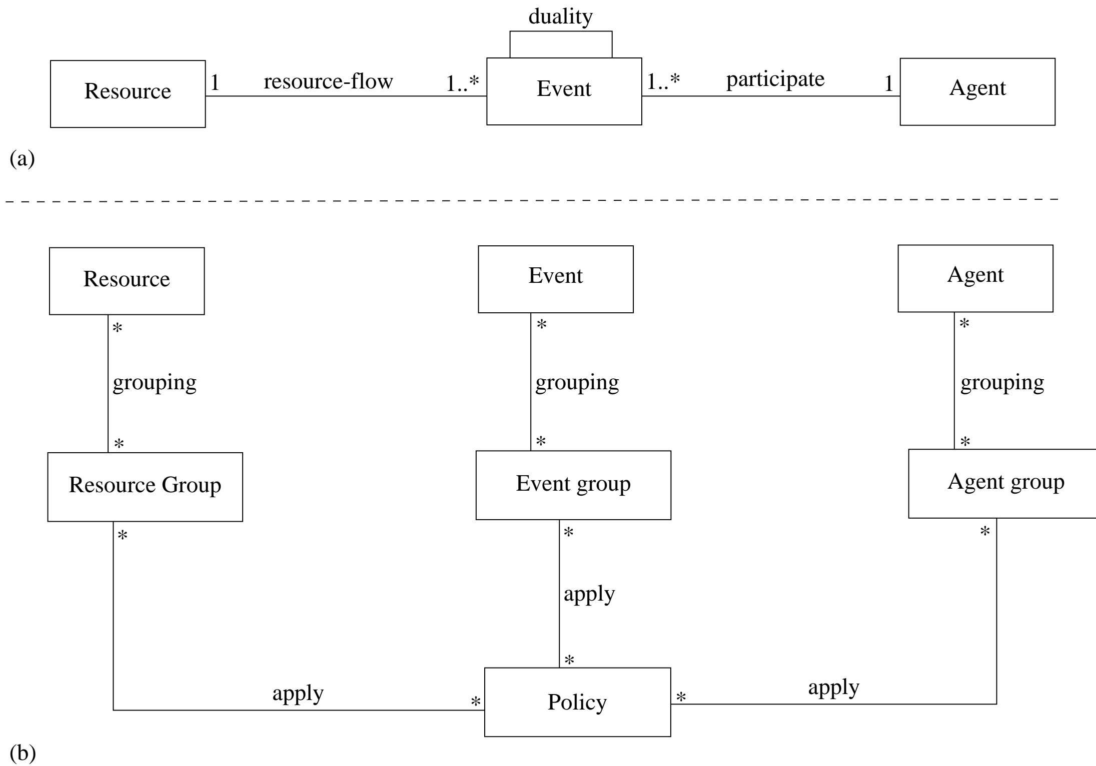


Figure 2.3: (a) An exchange pattern, (b) a pattern to model a policy

Figure 2.4 [56] shows two exchange patterns connected together such that an output of one exchange is an input to the other one. This figure shows two simple REA exchanges in Figure 2.4(a), renting a car, during which an employee receives a client's cash and provides a car, and the customer provides cash and receives a car. In Figure 2.4(b), repairing a car, an inspector provides cash to a mechanic and receives the repaired car (the reverse also holds: a mechanic provides a repaired car and receives cash). These two models are connected to make a value chain: cash is the output resource of the car-renting process and is the input in the car-repairing process; similarly, a repaired car is an output resource of the car-repairing process and is an input in the car-renting process.

As an advantage, REA includes rules for constructing business models; as a result, this process is more orderly than others. As another advantage, REA uses an object-oriented approach and UML diagrams to represent business processes.

Eriksson and Penker [31] provide a collection of business patterns classified into three categories: resource and rule patterns, goal patterns, and process patterns. Each of these

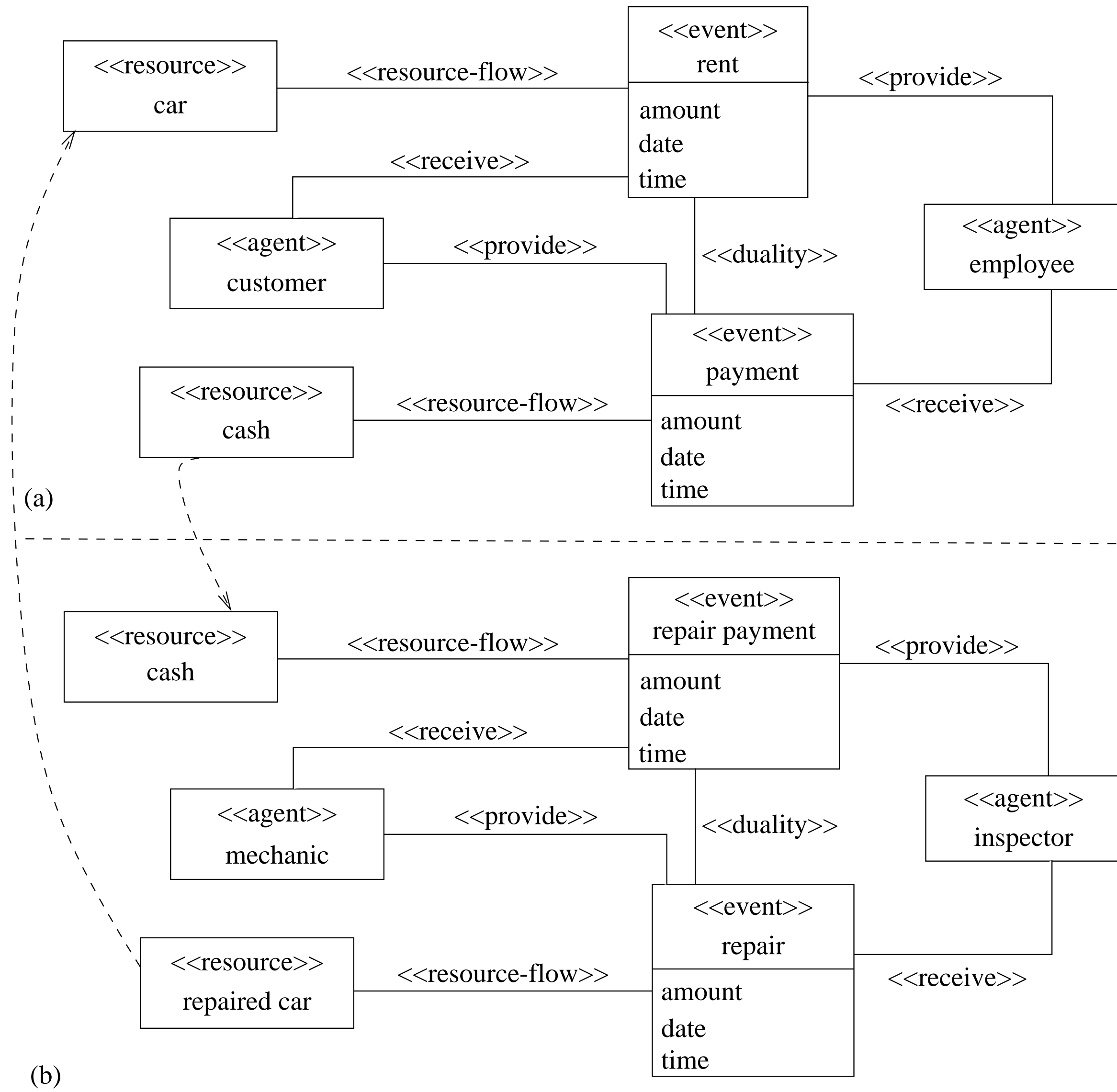


Figure 2.4: Two Exchanges in a Value Chain: (a) renting a car, (b) repairing a car

categories contains several patterns. Arlow and Neustadt [4] also present a set of business patterns, but their patterns, such as customer and money, are related to the detailed modeling of individual concepts present in business; probably for this reason, they call their patterns "business archetype patterns."

Process Modeling and Enterprise Modeling: Next, a brief description of a line of research, identified as enterprise modeling, is provided because the selected business patterns of this work are represented as an ontology and emphasize the semantic of entities. An enterprise model "is a computational representation of the structure, activities, processes, information, resources, people, behavior, goals, and constraints of a business, government, or other enterprise. It can be both descriptive and definitional–spanning what is and what should be [40]." TOronto Virtual Enterprise (TOVE) [39] exemplifies an effort on enterprise modeling. The OMG Semantics of Business Vocabulary and Business Rules (SBVR) describes a business modeling approach based on the semantics of business vocabularies, facts, and business rules.

Several modeling notations for specifying business processes exist. Because of the existence of various such languages and difficulties in translation from one language to another, some authors compare this situation with the Tower of Babel [21]. A few popular, currently or at some point in time, of these languages for modeling business processes are described next.

Business Process Modeling Notation (BPMN): The following description of BPMN graphical notation is mainly based on an article by Dijkman et al. [25]. BPMN, an OMG standard, includes numerous graphical elements or constructs divided into three groups: *objects*, *sequence flows*, and *object flows*. *Object* is subsequently classified into three categories: *event*, *activity*, and *gateway*; each category includes several elements. For instance, a *start event* graphical construct represents the beginning of a process, and an *end event* notation shows the end of a process. In addition, *terminate*, *message*, and *error* events represent other types of events. A *sequence flow* links two objects and describes an ordering within a process, whereas a *message flow* describes the interaction between processes as a method to connect them.

Despite BPMN being similar to the UML activity diagrams and part of OMG, BPMN has more constructs than activity diagrams; in addition, BPMN is not an approach related to object-oriented modeling.

Business Process Execution Language (BPEL): This XML-based specification language represents business processes in a declarative manner. BPEL can specify both abstract and executable business processes [77]: an abstract BPEL partially specifies a business process without describing all the detail of execution, whereas an executable one describes the process in detail. Barreto et al. [77] regard a BPEL specification as a container that starts with a *process* element. Similar to BPMN, BPEL is not related to object-oriented modeling. In addition, BPEL does not discuss business processes in general but considers them in the context of the Web.

Finally, the use of activity diagrams, Data Flow Diagrams (DFDs), flow charts, and Integration DEFinition (IDEF) models for process modeling is mentioned next.

Activity Diagram: These diagrams are the current UML standards for showing steps that make a process. An activity diagram resembles a sequence diagram, but the former focuses on operations, and the latter emphasizes objects [16]. (Although an activity diagram can also show objects by the addition of implicit and explicit pins and data objects, these additional elements are not necessarily included in every activity diagram.) An activity diagram is a variation of a flow chart: for instance, an activity diagram can show both sequential and concurrent processes, whereas a traditional flowchart cannot [16]. Activity diagrams comprise a few notations to describe processes, such as business processes; nevertheless, an activity diagram is a general representation and does not represent a modeling approach.

Data Flow Diagram (DFD) and Flow Chart: DFD was a popular diagram within the structured analysis paradigm, but its use has declined. As the name of this diagram indicates, DFD shows the flow of data within processes and sub-processes. Control of flow is not explicitly modeled, and what causes this flow is determined using other structured analysis techniques, such as decision tables [21]. The UML activity diagrams provide a means to represent DFDs by the addition of object nodes and pins to activity diagrams. *Flow charts*, variations of UML activity diagrams, can also describe control of flow.

IDEF: Integration DEFinition (IDEF) is a family of modeling notation; in it, IDEF0 and IDEF3 are related to business process modeling. The following IDEF descriptions are largely based on the IDEF web site, www.idef.com, unless stated otherwise.

IDEF0/IDEF3: The primary use of IDEF0 is modeling functions. IDEF0 shows a function as a box with *inputs*, *mechanisms* (i.e., "the means used to perform a function"), and *controls* (i.e., "conditions required to produce correct output") as inflows to the box, and *outputs* as outflows from the box. Figure 2.5 [76] depicts IDEF0.

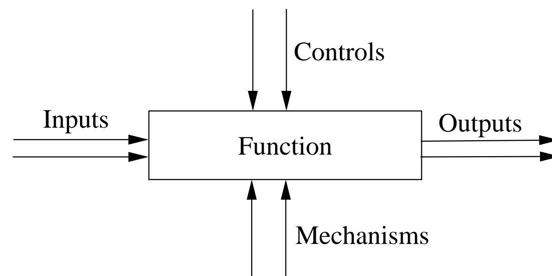


Figure 2.5: IDEF0

A function can be gradually decomposed to show sub-functions. IDEF0 recommends function decomposition up to six levels. Functions can be organized in a hierarchical representation, for instance from left to right, such that an output from one function is an input to another. Therefore, IDEF0 can show a sequence of activities without time considerations. IDEF0 identifies what functions are performed, and what is needed to perform these functions. Some authors note the lack of guidance in function decomposition as a drawback of this approach [21]. Although IDEF0 can represent a sequence of activities as described, IDEF0 is not intended for this procedure. IDEF3 is designed for process description and includes two diagrams: process flow and object state transition network. A process flow diagram shows the sequence of activities within a process, and an object state transition network presents an object-centered view.

REA is compatible with other approaches, such as IDEF0 [48]: IDEF0 *inputs* correspond to REA resources consumed, and IDEF0 *outputs* correspond to REA resources produced. IDEF0's *mechanisms* (i.e., "the means used to perform a function" [76]) represent people and machines and therefore are equivalent to REA agents and resources; IDEF0 *controls* (i.e., "conditions required to produce correct output" [76]) are comparable to REA policies. The sub-functions of an IDEF0 can also be compared to the REA value chain model, described previously, in which an input from a module is the output to another one.

2.3 Formal Verification

This thesis uses model checking, a formal method approach, also used in the industry, with weaknesses and strengths. Baier et al. [8] list characteristics of model checking and note its strengths as follows: a) this general approach is applicable to software and hardware; b) it allows properties to be validated individually; c) when a property is not valid, the approach creates a counter-example for the invalidity of the property; d) the approach is rapidly being adopted by the industry, and new techniques and tools are constantly being introduced. The authors express the following weaknesses of model checking: a) the approach, in most cases, is suitable for "control-intensive" applications rather than "data-intensive" ones; b) model checking is subject to theoretical limitations of decidability and effective computations; c) this approach verifies a model and not an actual system; d) the state-space explosion is still a problem to overcome.

Model checkers use temporal logic to express various properties; therefore, an overview of temporal logic follows. Although several types of temporal logic for property specifi-

cations exist, Linear time Temporal Logic (LTL)[1] and Propositional Computational Tree Logic (CTL), a branching-time temporal logic, are used most often by model checkers. In LTL, time is a set of paths, and each path is a sequence that consists of time instances; therefore, LTL is called linear because of this view of time. In CTL, time is viewed as a tree, and each moment can have various possible futures. CTL cannot express *fairness* properties because this type of property describes that something will happen or fail to happen, with or without certain conditions, infinitely often (read repeated *liveness*) [13]. The "infinitely often" of the above definition translates into GF, where G and F stand for "always" and "eventually," respectively. A property that includes GF cannot be expressed in CTL because the temporal operators, such as G and F, must follow either the universal quantifier ($\forall$) or existential quantifier ($\exists$).

LTL does not have universal and existential quantifiers ($\forall$ and $\exists$). The nonexistence of the universal quantifier ($\forall$) does not impose any problem in property specification because LTL implicitly quantifies over all paths. In addition, the nonexistence of existential quantifier ($\exists$) can be remedied using the dual negation relation between $\forall$ and $\exists$ [49]: to check whether there is a path from a state s that satisfies a formula ϕ, one can check its equivalent to determine whether all paths from the state s satisfy $\neg\phi$. However, a property statement that includes both existential and universal quantifiers cannot be described by the negation of the original statement because its negation also contains the existential quantifier. For instance, "For every computation it is always possible to return to the initial state" (i.e., $\forall G \exists F$ *start*) cannot be expressed by LTL [8].

Alloy [50] represents a formal specification and analysis mainly based on predicate logic with some similarities and differences to the model checking approach. Alloy Analyzer is a compiler that translates a problem into a Boolean formula; the formula is subsequently handed to a SAT solver. Alloy Analyzer also translates the SAT solver's solution into Alloy's language. Therefore, the Alloy analysis relies on SAT solvers.

Theorem provers are also used in the verification of access control and/or business processes. Theorem proving uses axioms and proofs to show the correctness of a system [23]. In addition, Description Logics (DLs) are formal languages for representing and reasoning about knowledge. In general, DLs represent subsets of first order logic, but some are supersets of predicate logic. The expressivity of DLs depends on the constructors they use in specification [7, 6]. Various DL reasoners such as FaCT++, Pellet, and Racerpro with different reasoning capability are available.

[1]Berard et al. [13] use the term Propositional Linear Temporal Logic (PLTL) instead of LTL. PLTL is more descriptive and accurate, but because of the common use in literature of LTL to mean PLTL, the term LTL is also used here.

2.3.1 Formal Verification of Access Control Policies

Formal verification of access control policies is an active research area. This thesis also discusses the verification of these policies, but the policies are based on access control models. In addition, the combination of the policies is also represented by state machines. This section describes prior works on the verification of access control policies.

Various formal method approaches, such as logic programming, model checking, and theorem proving, are used for the verification of access control policies. Before describing a few examples, it may be beneficial to mention the theoretical limitation in analysis and to elaborate on the scope of this limitation.

The theoretical limitation always applies in that these types of analysis for the most difficult problems are probably intractable. Role-based Access Control (RBAC) analysis is shown to have PSPACE-complete complexity for general cases, and for certain restricted cases, the complexity changes to NP-complete; for some others, running time is reduced to polynomial [52]. It is known that $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME}$, although it has not been discovered whether any of the above containment relationships is an equality [95]. Some strict containment between the above non-adjacent classes, such as $\mathrm{P} \subset \mathrm{EXPTIME}$, is known, but it has not been determined whether $\mathrm{P} \subset \mathrm{PSPACE}$ holds [83]. Nevertheless, the intractability indicates the existence of possible exponential running time for difficult instances, but this possibility does not apply to every instance. Jha et al. [52] performed several experiments to determine how realistic the analysis of RBAC access control is. They included cases in which the number of roles varied from 12 to 100, and the number of rules were in the range of 31 to 250. The results were very positive because these nontrivial cases have been analyzed in an expected amount of time. In addition, an exponential running time such as 2^n indicates a worst-case measure and the existence of at least one problem of size n to require such a running time; however, most cases require much less time than 2^n [41].

Harrison et al. [45] also describe the general theoretical limitation in analysis but are in favor of analysis. They study the safety analysis problem in an access control matrix and conclude that it is undecidable: no algorithm can determine whether, in a general case, an arbitrary configuration of an access matrix is safe. In other words, if users can grant rights for accessing objects or operations to other users, determining whether someone can gain a prohibited access when the access control system has no controls to limit users' access is impossible [33]. Nevertheless, Harrison et al. [45] suggest that the undecidability of the general case should not prevent analysis of restricted versions. For instance, they show that analysis of the safety problem within an access control matrix is decidable if no subjects or objects are allowed to be created.

Next, a few representations of access control policy analysis are provided. Jha et al. [52] use model checking and logic programming to analyze access control policies and compare these two approaches. They perform two experiments and conclude that even a problem with a probably intractable running time (i.e., probably an exponential running time) for the most difficult cases, can be solved in a reasonable time for some realistic instances. They use both model checking and a logic programming approach and conclude that logic programming using XSB (the Stony Brook University Extended Prolog) performs better for small instances, while model checking using New Symbolic Model Verifier (NuSMV) performs better for larger cases. Their experiments show that the number of rules is a determining factor in runtime; even data sets with a larger number of roles and fewer rules run faster than data sets with fewer roles and more rules. The authors' work provides insights in terms of using two different analysis approaches, but the scope of their work is limited to RBAC: Their work is not concerned with rule- and policy-combining algorithms that exist in policy languages.

Jürjens et al. [53] introduce UMLsec, extensions to UML using stereotypes to enable security specifications (both authorizations and authentications). In this approach, UML diagrams are annotated using a language with formal semantics. For instance, they use a permission-based access control (i.e., associating permissions to entities) and annotate UML class and sequence diagrams with these permissions to describe static and dynamic aspects. Then, the authors translate these permissions into the language of a first order theorem prover, such as SPASS, for the purpose of analysis. Despite the large scope of UMLsec (i.e., includes authorization and authentication), UMLsec annotates UML diagrams to define precise security definitions. Some may argue that this approach defeats the entire purpose and advantage of UML as a visual modeling language. Similarly, their work does not apply to policy languages and rule and policy combinations.

Several authors describe different approaches in which they initially use UML and OCL and then add analysis capabilities. HOL-OCL represents a tool for analyzing UML class models annotated with OCL constraints. HOL-OCL mainly relies on the Higher Order Logic (HOL) theorem prover for analysis and consists of several components [19]: a) a data repository that can accept UML and OCL specifications and translate them into XML Metalanguage Interchange (XMI). The data repository can also accept directly XMI; in addition, there are other tools to perform this translation from UML to XMI, b) a datatype package in HOL that includes object-oriented data structures of UML/OCL models, c) a library that consists of over 10,000 UML/OCL definitions such as those of *integer*, *real*, *bag*, and *sequence*, d) a collection of proof procedures for HOL that provides necessary modules of analysis. HOL-OCL describes an analysis approach for UML class diagrams, but the scope of this research is limited to only one type of UML diagram.

The current standard language for specifying access control policies is eXtensible Access Control Markup Language (XACML). Some research on formalizing XACML exists, and one representation is discussed next.

Fisler et al. [36] present an approach and a tool called Margrave[2] for the verification of access-control policies written in an XACML subset, which will eventually be extended. For instance, some XACML features, such as conditions with nested non-boolean functions and multi-subject requests, are not currently handled by Margrave. To analyze access control properties, Margrave translates these XACML properties into *Multi-Terminal Binary Decision Diagrams* (MTBDDs), variations and general forms of *Binary Decision Diagrams* (BDDs). An MTBDD has multiple terminal nodes. Margrave builds an MTBDD for each rule; then, these MTBDDs representing rules are combined. For instance, Figure 2.6(a) shows an MTBDD representing a rule in which a project manager (p) is permitted to book (b) a room (r), and Figure 2.6(b) describes another rule in which a member (m) is not allowed to book a room. Figure 2.6(c) shows the combination of these two rules.

In addition, the authors' approach also includes a change-impact analysis component that compares two polices using a decision diagram called a *change-analysis decision diagram*. The authors use the policies of conference management to evaluate their approach.

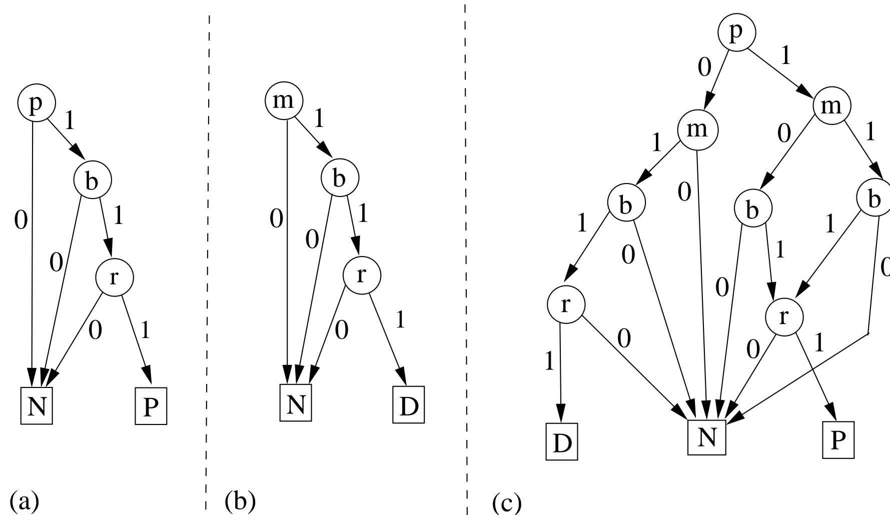


Figure 2.6: Two rules and their combinations using MTBDDs

Fisler et al. present their research for access control policy analysis in addition to providing a change impact analysis. Their approach is not capable of expressing some

[2] "a margrave is a lord or keeper of borders: that is, a medieval access control manager [36]."

rule-combining algorithms, such as the ordered-permit-overrides algorithm. This topic is discussed in detail in Chapter 5.

Enterprise Policy Authorization Language (EPAL) resembles XACML. For instance, a policy in both XACML and EPAL consists of one or several rules with similar three-part structures [3]: a) an applicability that determines whether a rule applies to a request using attribute values, b) a condition that describes existing constrains, c) an effect with a value of either permit or deny that determines when a rule is applicable and conditions are met. Therefore, because of similarities between XACML and EPAL, the research on formalizing XACML can be applicable to EPAL. Fisler et al. [36] believe that Margrave "can, with minor changes, handle EPAL just as well as it does XACML."

Finally, some research starts with UML and OCL and performs analysis using additional tools. Some representatives of this line of research for access control policy analysis are discussed next. UML-based Specification Environment (USE) enables the analysis of both structure (i.e., classes, associations, attributes, and invariants of these elements) and behaviour (i.e., operations and their pre- and post-conditions) of models [44]. It appears that the current scope of OCL specification contains only class diagrams, but the USE project also includes UML sequence diagrams. Gogolla et al.'s future work consists of more support for sequence diagrams and the import and export of diagrams to XMI [44].

Sohr et al. [96] present two approaches for the analysis of access control policies. In their first approach, USE receives both a textual format of UML models and OCL (i.e., a textual representation of class names, attributes, and associations) as input. After syntax checking, USE draws UML diagrams. The authors mention that USE may find errors in certain situations and recommend the use of theorem provers to find other errors [96]. Despite the stated limitation, USE has been capable of detecting bugs such as initialization errors within the first variant of Alloy [50]. In their second approach, the authors use the theorem prover Isabelle and first order temporal logic. They also use Temporal Object Constraint Language (TOCL), first introduced by Ziemann and Gogolla, to specify a type of dynamic separation of duties. The scope of this work is also limited to RBAC.

Basin et al. [10] present the SecureUML metamodel, which uses UML to visualize and extend RBAC. They also use OCL expressions, e.g., for expressing class invariants. Figure 2.7 [9] shows the updated SecureUML metamodel from a subsequent publication.

The RBAC model (Figure 2.1) shows *permission*, which is generally viewed as *operations* on *objects*. Figure 2.7 presents *permission* connected to the class of *action* (i.e., *operation*) and associated with the class of *resource* (i.e., *objects*), but permission is generally expressed as operations on objects. Therefore, the existence of class "permission" in addition to "action" and "resource" is not justified. Despite their choice, their model (Figure 2.7) can

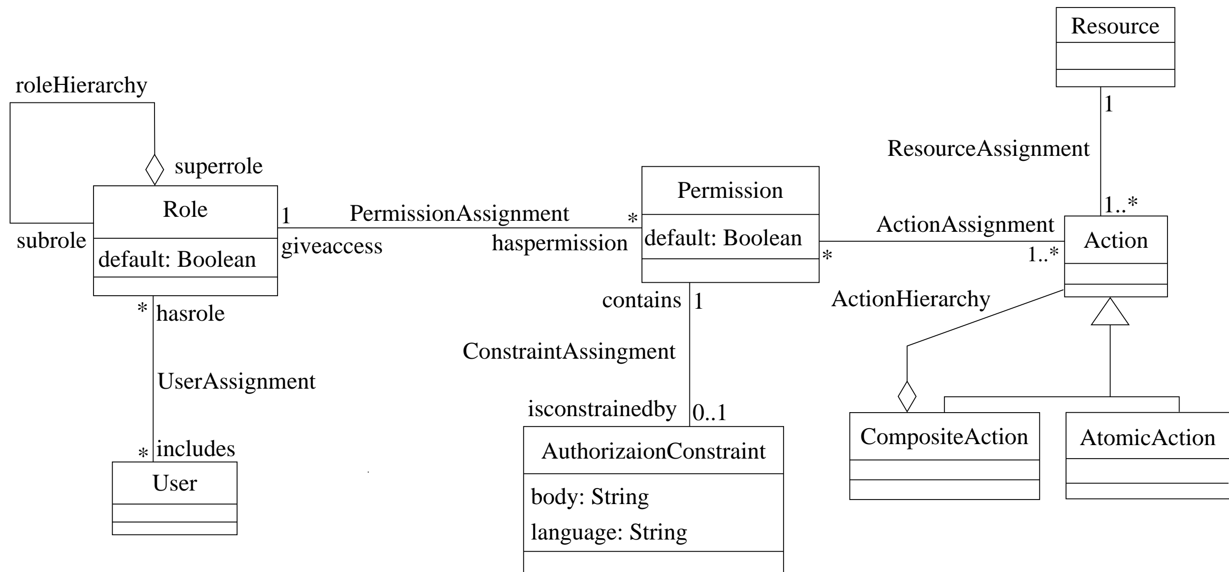


Figure 2.7: The SecureUML metamodel

be considered an extension of RBAC because it is not substantially different from RBAC.

Basin et al. [9] add querying capabilities to answer certain questions, such as whether two roles exist such that one includes the set of all actions of the other in addition to not being related to each other in a hierarchical manner in terms of privileges. The authors introduce SecureMOVA, an extension of MOVA, to provide query capabilities. (MOVA enables drawing UML class and object diagrams plus writing and evaluating OCL constraints.)

Finally, it is worth mentioning an effort called the Common Criteria for Information Technology and Security Evaluation, commonly referred to as CC. The description of this effort also compares common criteria and this thesis' scope.

The Common Criteria started in the mid to late 1990s through the evolution, merging, and extension of three existing evaluation criteria: The Trusted Computer System Evaluation Criteria (TCSEC) of the 1980s, the Canadian Trusted Computer Product Evaluation Criteria (CTCPEC) of the 1990s, and Europe's Information Technology Security Evaluation Criteria (ITSEC) of the 1990s [15].

The Common Criteria consists of two categories [98]: a) *functional* requirements that describe desired security behaviour and consist of eleven classes, including *audit* and *identification and authentication*, and b) *assurance* requirements that specify the existing in-

tended security measures and include several classes. The Security Assurance requirements use an Evaluation Assurance Level (EAL) classification, which currently consists of seven levels, but the last three levels are hardly applied in practice.

The scope of CC includes all aspects of security, including access control, authentication, and auditing. Several licensed laboratories exist around the world to evaluate these criteria; in addition, annual conferences entitled *International Common Criteria Conference*s are dedicated to this subject. Despite continuously updated ISO (ISO 15048) and extensive classification of security requirements, the common criteria represents a general approach.

2.3.2 Formal Verification of Business Processes

This thesis describes the verification of business and access models and policies. Therefore, at this point it may help to outline some prior works on formal verification of business processes.

Similarly, and in a parallel effort to access control policy verification, the desired properties for business processes are verified using different analysis techniques. This formal analysis includes some properties, such as whether an action is reachable in a business process. A few representatives are discussed next.

Janssen et al. [51] present Architectural Modelling Box for Enterprise Redesign (AMBER) for the specification of business processes. AMBER has graphical constructs to describe actions, hierarchical compositions, and causality relations, among others. These graphical representations are automatically translated to the language of SPIN. The authors view their automatic translation as a bridge between informal and formal models. They define GARAGE, a process for repairing a car after an accident, and PRO-FIT, a process for evaluating a claim sent from a customer to an insurance company. Using Liner Temporal Logic (LTL), Janssen et al. define and evaluate five properties, e.g., *is the car always repaired when delivered?* AMBER does not perform an automatic translation of properties, and users have to write properties in LTL. The work in this thesis (Chapter 4) also specifies properties in LTL, but these properties are categorized and have specific forms.

Some research papers for verification of business processes start with Business Process Execution Language (BPEL) descriptions and then apply various formal verification tools to analyze certain properties. A few representations of this line of research follow.

VERification for BUSiness processes (VERBUS) [37] represents an approach and a tool, based on the model checking approach, for verification of business process properties.

Currently, VERBUS receives a BPEL description and translates it automatically to a common formal specification that is subsequently translated to the languages of SPIN and Symbolic Model Verifier (SMV) model checkers. The authors state that VERBUS differs from other similar approaches because it can be extended by the inclusion of other business process languages and formal methods techniques. According to the authors, the extendibility is achieved through use of a common intermediate language. As an advantage of this approach, VERBUS can examine both Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) properties. VERBUS can use both SPIN and SMV model checkers.

Salaün et al. [86] state that BPEL lacks well-defined semantics and advocate the use of languages with formal semantics such as Calculus of Communicating Systems (CCS), a process algebra language. With a small test case written in both BPEL and CCS, they describe a sanitary agency model that can accept requests from elderly citizens and provide responses accordingly. To analyze certain properties, Salaün et al. use Concurrency Workbench of the New Century (CWB-NC), which can accept a CCS specification. The authors use Calculus of Communicating Systems to describe their test case; nevertheless, their test case is a small one.

Ouyang et al. [81] take advantage of extensive research using Petri nets to describe and analyze business processes and present their approach using BPEL2PNML and WofBPEL tools. Using Petri Net Markup Language (PNML), BPEL2PNML accepts BPELs and translates them into Petri nets. (Other existing Petri net graphical visualizers, such as the PIPE editor, can also transform PNML layout descriptions into Petri net diagrams.) WofBPEL receives PNML layout information and performs three different types of analysis, including the detection of unreachable actions within business processes. In addition, WofBPEL creates an XML file that contains the results of an analysis, including description of errors. The authors provide small examples and show three different error messages captured by their analysis.

Some authors, e.g., Dijkman et al. [25], argue that the Business Process Modeling Notation (BPMN) standard may possibly introduce errors in modeling business processes because BPMN has many constructs that lack formal semantics. The authors translate BNMPs to Petri nets because Petri nets are formal models based on the concept of flow, either the flow of an object or the flow of control. The authors translate some BNMP models, collected from different sources, into Petri nets; their analysis detects dead tasks and livelock errors. Their work neither covers the BNMP existing capabilities of multiple concurrent executions of subprocesses for handling exceptions completely nor BNMP OR-join gateways.

Chapter 3

Access Control Models, Rules and Policies and their Combinations

Summary: This chapter describes a common representation for business and access control models, rules, and policies based on the Resource-Event-Agent (REA) model, REA patterns, and state machines (SMs).

Access control models provide the basis for access control rules. An access control policy can be seen as a combination of one or more rules, and one or more policies can be combined into a set of access control policies that control access to an entire system. The rules and resulting policies can be combined in many different ways, and the combination of rules and policies are included in policy languages.

The top two boxes of Figure 3.1 correspond to Box A1 of Figure 1.1 and show the approach of using a common representation to describe both business processes and access control models. Business models are described using REA and business patterns (e.g., [42, 48, 68, 100]). Section 2.2 and Appendix A also describe REA.

As the top right-hand box of Figure 3.1 presents, the role-based access control (RBAC) model is selected to show the approach. As explained in Chapter 1 and described in detail later in this chapter, this modeling approach applies not only to RBAC but to any general access control model that is based on the five constituents: users, objects, permissions, operations, and subjects [33]. In general, access control models can be expressed in REA through three REA AC patterns that map resources, events, and agents to the previously mentioned five constituents of access control. Extension of the model to two other main access control categories, discretionary access control (DAC) and mandatory access control (MAC), is illustrated.

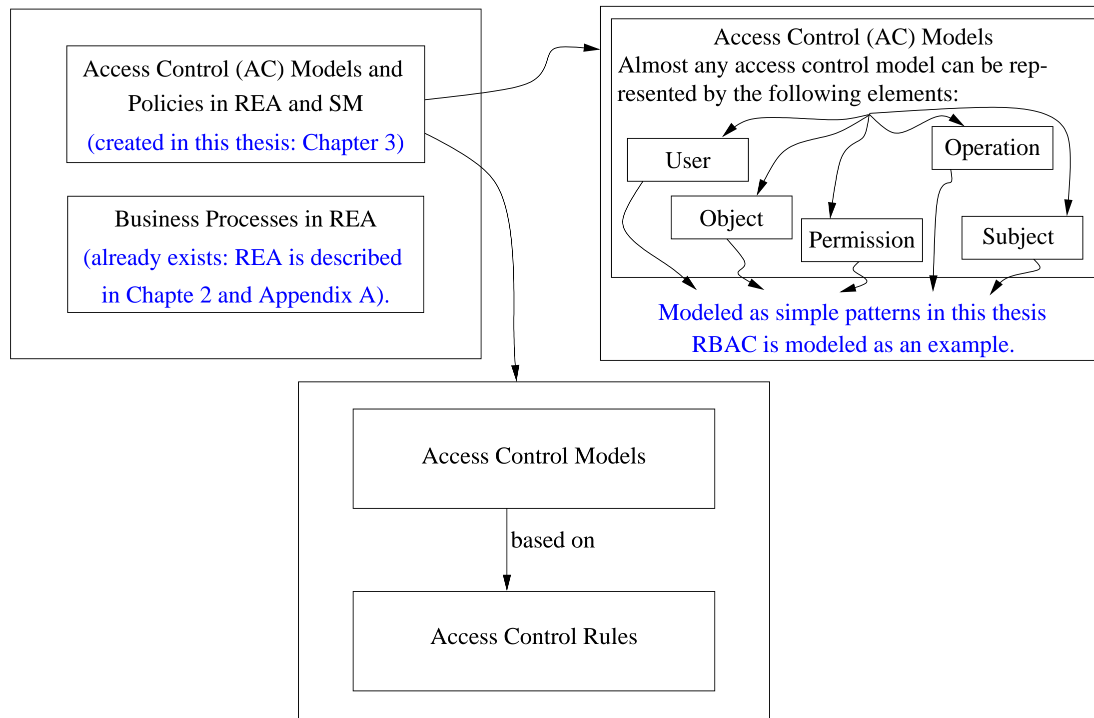


Figure 3.1: Access control models, their realizations, and rule definitions

The box at the bottom of Figure 3.1 represents access control rules based on access control models and corresponds to A2 of Figure 1.1. The grammar of AC rules, based on AC models, is described in BNF.

This chapter also discusses the combination of rules into policies using state machines and explains different possible ordering combinations. This approach is shown in the two top boxes of Figure 3.2, and it maps into Box A3 in Figure 1.1. These combinations can be described by various algorithms, such as the first-applicable algorithm (the top right-hand box of Figure 3.2 and also described in detail in Figure 3.3). The bottom box of Figure 3.2 shows a state machine that describes the formal model for the first-applicable rule-combining algorithm and corresponds to Box B3 in Figure 1.1. As presented in this box, one of the authorizations of *permit*, *deny*, or *not applicable (NA)* is decided based on the existing AC rules and this algorithm. The state machines for these combinations are discussed in detail later. The categorization and formalization of properties from Boxes B1 and B2 of Figure 1.1 are described in Chapter 4.

Section 3.1 provides an overview of the general approach. Section 3.2 describes the

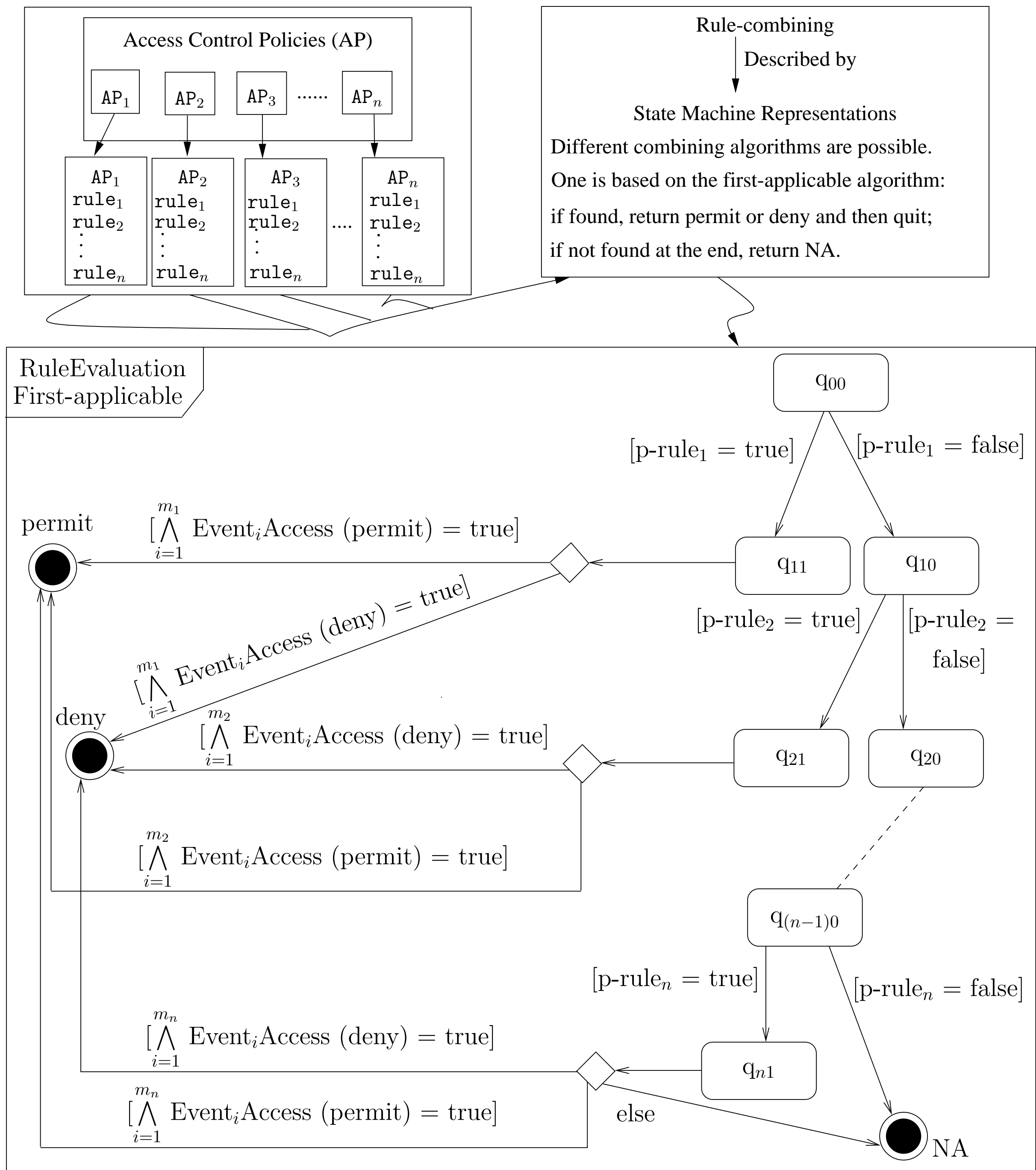


Figure 3.2: Access control policies as a combination of rules and a rule-combining algorithm

The description of the first-applicable rule-combining algorithm follows [79]: the evaluation of rules within a policy is in the same order that rules are listed in a policy. If a rule applies, then the rule's result, i.e., *permit* or *deny*, applies, and the evaluation of the rest of rules halts. Otherwise, the procedure continues to the end. If none of the rules applies, then the result will be *not applicable*.

Figure 3.3: First-applicable Rule-combining Algorithm

generic patterns of this thesis and uses a running banking example, taken from the literature, to illustrate examples of these patterns. This section provides a detailed description of the patterns used to construct classes of access control models and explains Box A1 of Figure 1.1. Section 3.3 explains the specification of a general form of access control rules based on access control models. This section expands on the material in Box A2 of Figure 1.1. Section 3.4 shows the creation of access control policies using general algorithmic forms, state machines, and access control rules. This section presents Boxes A3 and B3 of Figure 1.1.

3.1 Overview of Representing Classes of AC Models

Geerts and McCarthy [43] created a notation to represent policies related to business models using REA and a small number of patterns. Since models such as RBAC represent access control policies, this thesis has adopted a similar approach as a starting point for the access control domain. Specifically, Geerts and McCarthy's notation is enhanced to include the classes, attributes, and relationships to represent access control models.

This section provides an overview of the task of representing access control (AC) models using patterns. First, an overview of general policy patterns as described by Geerts and McCarthy [43] is provided. These patterns are the starting point for modeling access control. Then, a motivation for using patterns and an outline of the pattern approach are provided.

General Policy Patterns: A general policy applies to groups and types, such as groups of airplanes and types of flights, rather than an individual entity. For example, commuter aircraft usually fly between cities a few miles apart, but jumbo jets fly between continents. Such a policy can be modeled graphically as patterns.[1] Detailed descriptions of business models and general policies using REA patterns are provided in Appendix A. Figure 3.4

[1] A pattern is here used in the same sense as described by Fowler [38].

shows an example of a REA pattern called *Mirror* for modeling the existence of a general policy. In this figure, a policy can be expressed by the association between FlightType and PlaneType describing what types of planes can be used for what types of flights.

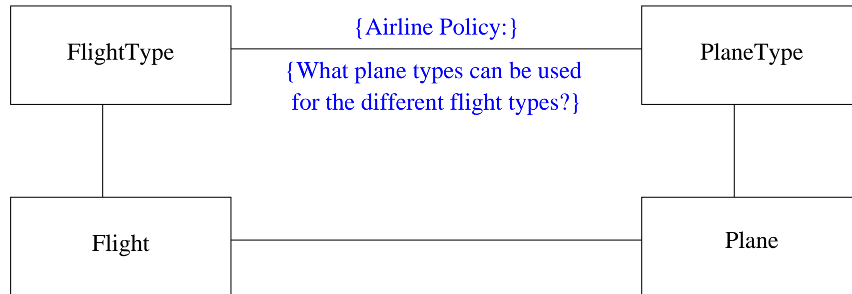


Figure 3.4: The existence of a general policy can be modeled by a *Mirror* pattern.

Motivation for Using Patterns: Patterns are used for representing access control models for the following reasons:

- REA provides some patterns for modeling general policies. However, their patterns are deficient in the sense that they do not include the classes, attributes, and relationships to represent access control models. However patterns, such as the *Mirror* pattern, can serve as an excellent starting point in access control modeling.
- The problem can be solved incrementally as each pattern can solve part of a problem. For instance, one pattern can model a building block (e.g., permission) of an access control model, and another pattern can model a different element. Then, the combination of patterns can build an access control model.
- The benefits of patterns are commonly acknowledged, as patterns can enable reuse and prevent the reinvention the same solutions over and over. In addition, one can take advantage of the guidelines of patterns (i.e., the context, problem, solution, and example of a pattern) in determining when and how to use them [16].

An Outline of the Pattern Approach: A mapping between resources, events, and agents and the primitives of access control, namely subjects, objects, users, permissions, operations and their relationships, is performed using the patterns in Appendix A as a starting point. This mapping is performed using the following steps:

a) The patterns are specialized to deal with only resources, events, and agents.

b) These patterns are further enhanced with attributes to map REA to the primitive building blocks of access control models such as users and permissions. Specialized *Basic* and *Root* patterns model the users (and roles) and role hierarchies (if they exist) of access control models. A specialized *Mirror* pattern models permissions of access control models.

c) Specific combinations of these three specialized patterns based on the *Root* and *Mirror* patterns are used to construct classes of access control models.

3.2 Representing Classes of Access Control Models

Classes of access control models, in general, can be described by five elements [33], namely *users*, *objects*, *subjects*, *operations*, *permissions*, and their relationships. A *user* represents an individual interacting with a computer system. An *object* represents any resource, such as a file, that can be accessed, and therefore is assumed to be passive. An *operation* is an active process such as write, when a user writes to a file, and a *subject* refers to a computer process, such as a program consisting of several *operations*.[2] Finally, a *permission* describes a set of tuples relating operations and objects such that if a tuple contains an operation and an object, then the operation on that object is permitted.

A mapping between these five elements and the three elements (i.e., resources, events, agents) of the REA model is described next. This mapping will be structured using patterns. First, *users* and *objects* correspond to agents and resources, respectively. An event corresponds to the completion of an *operation*, and can be considered as a different view of the same concept: an event includes a distinct change of state, whereas an operation makes the change happen [67]. Since a *permission* describes a set of tuples relating operations and objects, then based on the mapping between events and operations, permissions can also be modeled as tuples of events on resources. In addition, the subject corresponds to a process, which is set of operations that now map to *events*.

Using patterns and this mapping, access control models are constructed. RBAC, Figure 3.5 [34], is used to show the approach. RBAC is a well-known access control model and presented in detail in Section 2.1.1.

[2]The early access control models used the term *subject* for an active process, whereas in some recent access control models, such as role-based access control (RBAC), an *operation* and a *subject* are distinguished between [33]: a *subject* refers to a process possibly invoking several *operations*.

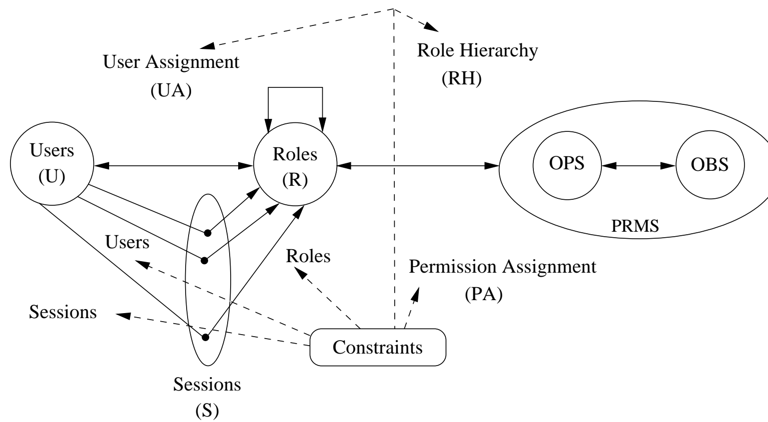


Figure 3.5: RBAC with permissions as operations on objects

This mapping is demonstrated by modeling the elements of Figure 3.5. Section 3.2.1 describes the modeling of roles and user assignments using a pattern and also discusses constraints on users and roles. Section 3.2.2 presents the modeling of permission using a different pattern; this section also discusses constraints on permissions. Section 3.2.3 explains a core RBAC model that is constructed by the combination of patterns of the previous two sections. Section 3.2.4 explains the advantages of the core access control model constructed in this thesis. Section 3.2.5 presents role hierarchies and separation of duties using a variation of the pattern described in Section 3.2.1. Section 3.2.6 illustrates the extension of this modeling to represent discretionary access control (DAC) and mandatory access control (MAC), two other main categories of access control.

Convention: This chapter follows the readability convention proposed by Ambler [2], namely that capital letters or underscores are not used in class names or their attributes (Figure 3.6). For example, *loan officer* and *day of week* are used instead of loanOfficer and dayOfWeek.

Description of Patterns: The pattern representation presented here is consistent with the general presentation of analysis patterns in the literature (e.g., [38]), in which *context*, *problem*, and *solution* are the three main elements. The *resulting context* is adopted from Hruby's form [48]. The pattern description consists of the following elements:

Name: A short descriptive name for a pattern

Context: A description of the situation to which a pattern applies

Problem: A brief explanation of the problem that a pattern attempts to solve

Solution: An explanation of the way a pattern solves a described problem. This chapter

Analysis

Order

Placement Date
Delivery Date
Order Number

Calculate Total
Calculate Taxes

Design

Order

- deliveryDate:Date
- orderNumber: int
- placementDate: Date
- taxes: Currency
- total: Currency

calculateTaxes(Country, State): Currency
calculateTotal(): Currency

Figure 3.6: Analysis and design versions of a class

intends to describe the solution in enough detail to be coherent, but also to be general enough to allow the solution to apply broadly.

Example: An example of the solution

Resulting Context: A clarification and the consequences of the solution in a broader aspect

Pattern descriptions [57] for access control models, which appear in the next sections, are presented both in general and in relation to a specific banking example. Figure 3.7 shows this banking application, provided by Chandramouli [22].

3.2.1 Modeling Roles and User Assignments

The pattern described next models the user, role, and the assignment of a user to a role (user assignment) of an access control model. For instance, for RBAC, this pattern uses REA to describe the left hand-side of Figure 3.5, which consists of circles called users and roles, and their connections.

Name: *Role Modeling and User Assignments* Pattern

Context: This work assumes that the roles are already known and can be described using various existing entities (classes) and their attributes.

Problem: The concept of roles in organizations is significant because activities and tasks are associated with roles. More specifically, how can one model users and roles using REA primitives? What kind of roles can be included in teams or organizational units (i.e., teams

Banking Case Study [22]

The banking application is used by tellers, customer service reps and loan officers, accountants, and accounting managers.

Policies: The existing policies are as follows:
P1) A teller can modify deposit accounts.
P2) A customer service rep can create and delete deposit accounts and also has a teller's permissions.
P3) A loan officer can create and modify loan accounts.
P4) An accountant can create general ledger reports.
P5) An accounting manager can modify ledger posting rules and also has the permissions of an accountant.

Hierarchical Role Relationships:
H1) A customer service rep role ranks higher than a teller role.
H2) An accounting manager role ranks higher than an accountant role.
H3) Customer service rep, loan officer, and accounting manager roles rank at the same level.
H4) A branch manager ranks the highest.

Static Separation of Duties: A single user cannot hold the following pair of roles:
1) customer service rep and accounting manager
2) loan officer and accounting manager
3) teller and accountant
4) teller and loan officer
5) accountant and loan officer

Dynamic Separation of Duties: One individual cannot hold the following two roles at the same time: customer service rep and loan officer

Figure 3.7: A banking example

or organizational units are aggregation of users.)? How can different roles of a user, if they arise in relation to teams or organizational units, be modeled? In addition, how are these roles assigned to users?

Solution: The solution is constructed in two steps using the *Basic* and *Root* patterns.

Step One: This step models users and roles of access control using REA. Users and roles correspond to individuals and their types respectively, and can be specialized to agents and agent types of REA. The *Basic* pattern (Appendix A) can model the constraint relationship

of access control between agent and agent type and so is used as a starting point.

Figure 3.8(a) shows an example of the *Basic* pattern. The *Basic* pattern can represent a statement (i.e., a general policy) using a classType and its attribute; e.g., in Figure 3.8(a), the attribute of the FlightType class can be used to state *the scheduled departure time of a flight type is at a specific time.*

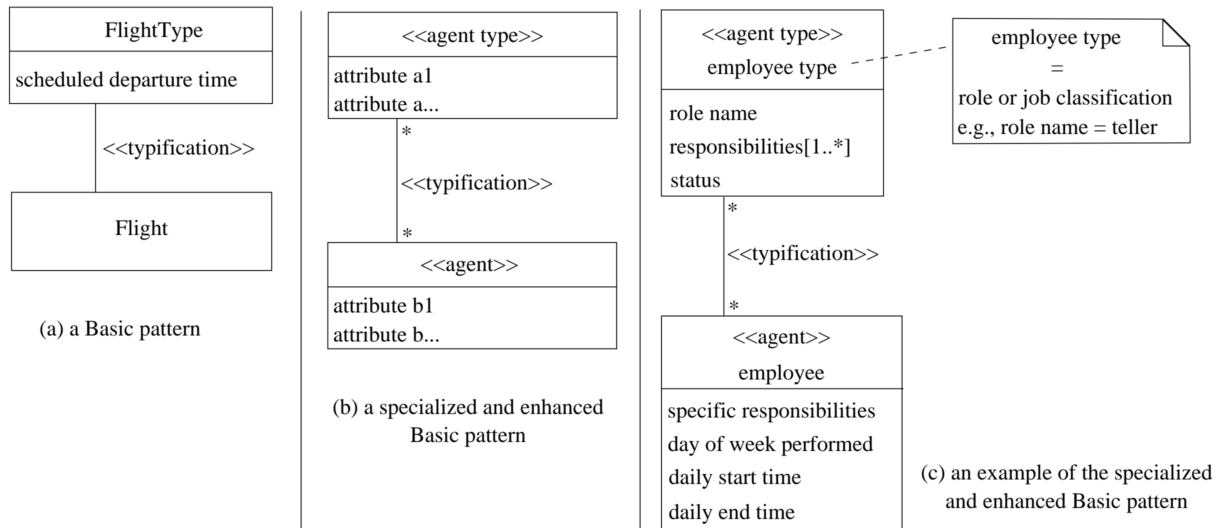


Figure 3.8: The *Basic* pattern, its specialization and enhancement, and example

First, the *Basic* pattern is specialized to deal only with agent and agent types, and then the pattern is enhanced with attributes of agent and agent types, as shown in Figure 3.8(b). For instance, in Figure 3.8(c), the agent type is enhanced with attributes *role name*, and *status*, and *responsibilities*. Thus, if a *role name* is *teller* and the *status* is *temporary*, the teller *responsibilities* can have values such as *deposit cheque* or *cash* and *update passbook*. Agents can also have attributes such as *start time*. The multiplicities of association *many* (*, zero or more) next to agent and agent type (role) allow one agent to have many roles, and one role to have many agents.

The use of *type* (e.g., FlightType, agent type) is consistent with the work of Geerts and McCarthy [43] as described in Appendix A.

Step Two: This step first adds to step one the modeling of kinds of roles that can be part of teams or organization units. This step also models different roles of a user, if they arise, in relation to teams or organization units. Thus, this step uses agents (e.g., users), agent type or role, and agent group (e.g., teams or organization units). The *Root* pattern

(Appendix A) can represent access control constraints between agent type (role) and agent group (e.g., teams), and between agent and agent group.

Figure 3.9(a) shows an example of the *Root* pattern in which Plane, PlaneType, and Fleet (i.e., plane group) are associated with one another. Based on the association between PlaneType and Fleet, the *Root* pattern in Figure 3.9(a) represents a statement (i.e., a general policy): *specific types of planes (i.e., PlaneType) can be part of a Fleet.*

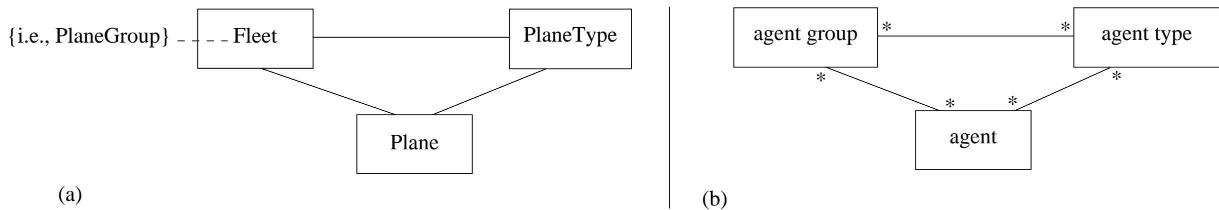


Figure 3.9: (a) The *Root* pattern and (b) the specialized *Root* pattern

Figure 3.9(b) shows the specialized *Root* pattern that has only agent, agent type, and agent group classes. Based on Figure 3.9(b), one can state *a specific agent type can be part of a group*; for instance, specific agent type (e.g., managers) can be part a group such as a bank branch. The multiplicities of association of *many* (*, zero or more), next to agent type (role) and agent group allow one agent type (role) to be part of *many* (zero or more) groups, and one agent group (e.g., team) to have *many* (zero or more) roles. Similarly, the multiplicities of association *many* (*, zero or more) next to agent and agent group allow one agent to be part of many groups (i.e., zero or more), and one agent group (e.g., team) to have *many* (zero or more) agents.

As a result of the two steps described, Figure 3.10 shows the *role modeling and user assignments* pattern that uses the *Basic* and *Root* patterns. This figure also shows the attributes of agent, agent type, and agent group. By using various combinations of these attributes, one will be able to describe details of the functions of agents, agent types or roles, and agent groups that are related to access control. For instance, a specific teller can have access to cash at certain hours, such as 9 am. to 5 pm., based on the attributes shown in Figure 3.10.

User Assignments: This pattern not only applies to RBAC but also to extensions of RBAC such as rule-based RBAC [1]. Rule-based RBAC suggests the assignment of users to roles based on the attribute values of individual users. This assignment can be described in the form of a rule and therefore the naming rule-based RBAC. For instance, Kern and Walhorn [58] provide an example: *if the costCenter of a user is AB2500 (costCenter is an attribute of users), then the user has the role of a cashier.* The pattern in this section is

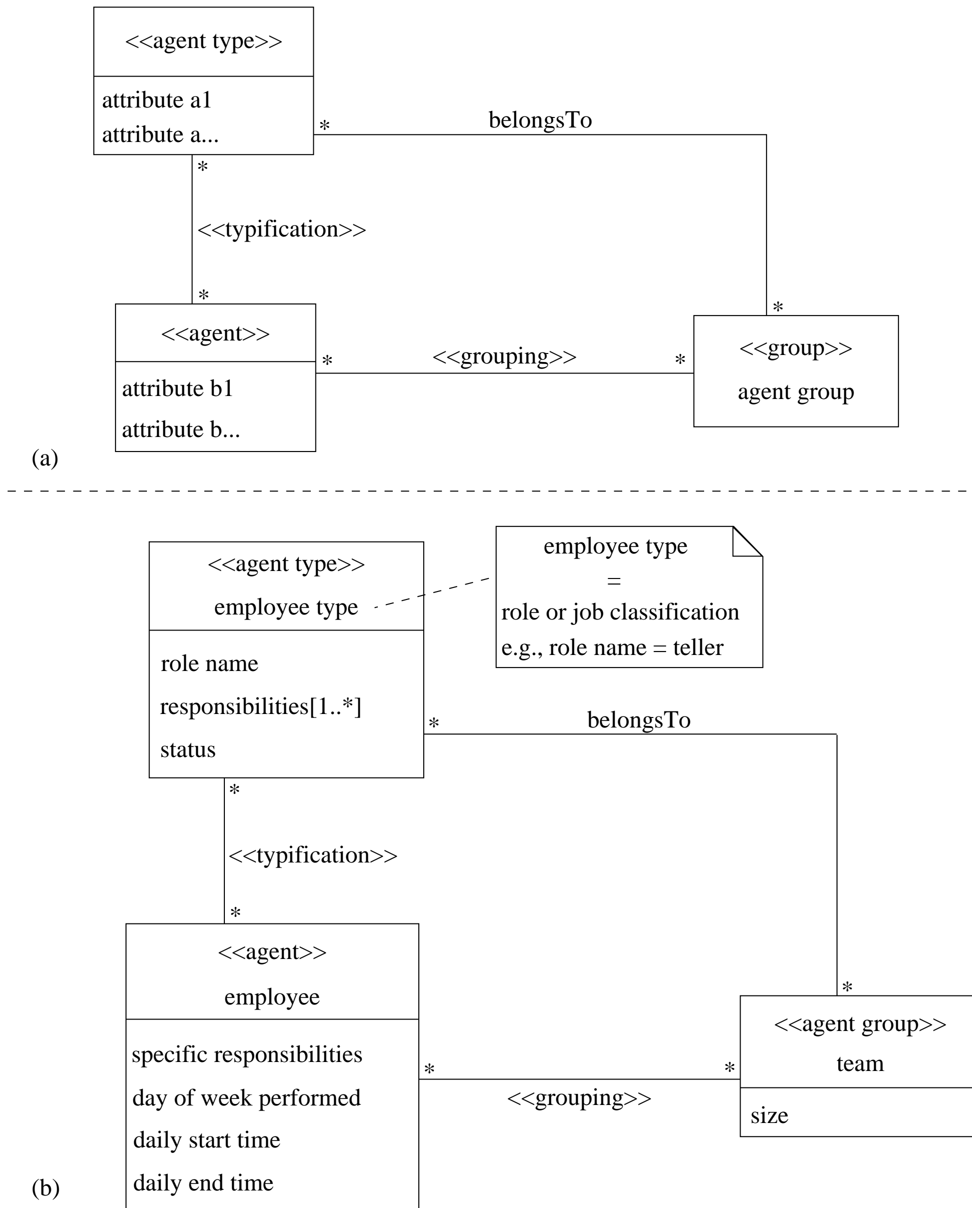


Figure 3.10: (a) Pattern for modeling roles and user assignments, (b) an example

also capable of describing this approach (whenever it is applicable) because the pattern includes attributes. Therefore, in this case, the association called typification in Figure 3.10 always exists at the class level, but at the instance level based on the attribute values of

an agent, a link (i.e., an association at the object level) may or may not exist between an agent (i.e., user) and agent type (i.e., role).

Example: Figure 3.10(b) shows an example of the *Role Modeling and User Assignments* pattern. Table 3.1 is related to Figure 3.10(b) as the table includes the agent, agent type, agent group, and attributes of the figure.

As Table 3.1 shows, a detailed description of roles and their functions is possible using only the attributes of agents, agent types, and agent groups. For instance, the last row of this table can use each element of the columns of this table and provide a detailed representation: e.g., an *employee* with a *role name* of *teller* in a *team* of a specific *size*. Similarly, using the columns of the second row of the table, an *employee* in a *temporary teller* role with certain responsibilities can be portrayed using *role name*, *status*, and *responsibilities* attributes. The third row illustrates the possibility of describing an *employee* with a *teller* role to be valid only for a permitted time period (e.g., Monday to Friday and 9:00 am to 5:00 pm) using *day of week performed*, *daily start time*, and *daily end time* attributes.

user (agent)	**agent/agent type attribute**	**agent type or role**	**agent group**	**agent group attribute**
employee	role name, responsibilities	teller	-	-
employee	role name, status, responsibilities	temporary teller	-	-
employee	role name, day of week performed	teller	-	-
employee	role name	teller	team	size

Table 3.1: Entities and attributes of the *role modeling and user assignments* pattern in a table format

Resulting Context: To summarize, this pattern models users, roles, their aggregations (e.g., organization units and teams), and their relationships. The main purpose of this pattern is to model the foundational building blocks (i.e., users and roles) that are needed to construct access control models.

In addition, using attributes of this pattern, it is possible to present some features of the extended RBAC models such as rule-based RBAC (as previously described) or temporal RBAC (TRBAC) [14]. With TRBAC, some roles are active at certain times (e.g., a teller role may be valid only for a permitted time period).

3.2.2 Modeling Permissions

The pattern described next models the permission of an access control model. For instance, for RBAC, this pattern uses REA to model the right hand-side of Figure 3.5. Permissions in this figure are shown by the symbol PRMS, and include OPS (operations) and OBJS (objects) circles and their connections.

Name: *Permission Modeling* Pattern

Context: Permission is one of the main components of any access control model and constitutes an element of access control policies. Describing the processes of an organization not only includes an explanation of what needs to be performed but also includes permissions and restrictions in performing these activities.

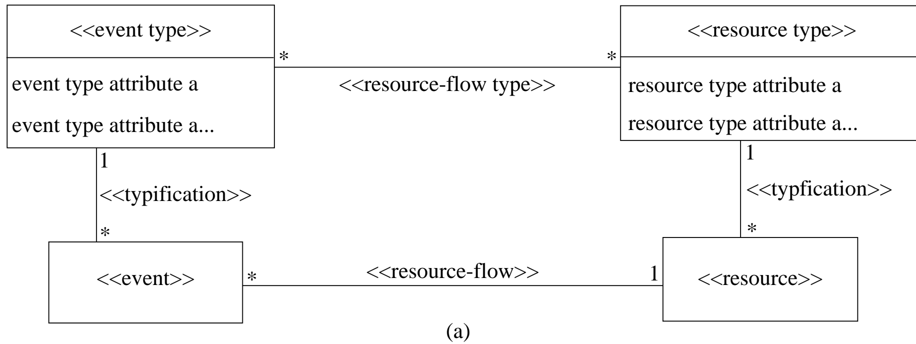
Problem: How can permission be represented? More specifically, how can permission as an element of access control be modeled?

Solution: Permissions can be viewed as a set of tuples relating operations and objects such that if a tuple contains an operation and an object, then the operation on that object is permitted. Operations translate to REA events, and objects are equivalent to REA resources. The following brief description of events and operations describes the mapping of events to operations.

Events and Operations: These two terms are essentially two different views of the same concept; an event includes a distinct change of state, whereas an operation is the agent of this change [67]. In other words, the operation view considers the mechanism for this change, and the event view is concerned about the result of the operation [67, 75]. For instance, *verify order* represents an operation, whereas *order verified* describes an event occurrence.

Figure 3.4 has previously shown an example of the *Mirror* pattern (Appendix A). The *Mirror* pattern can be specialized to represent the permission of access control using operations and objects. The *permission modeling* pattern specializes the *Mirror* pattern by the identification of event, event type, resource, and resource type as entities of this pattern, as shown in Figure 3.11(a). An operation (OP), an object (OB), two typification associations (one between event and event type and the other between resource and resource type), and two other associations (one connecting event type and resource type and the other connecting event and resource) are also presented in this figure.

An enhancement to this pattern includes the use of attributes of event types or resource types, or both. Attributes can be used with this pattern to describe constraints on operations and objects, as shown next.



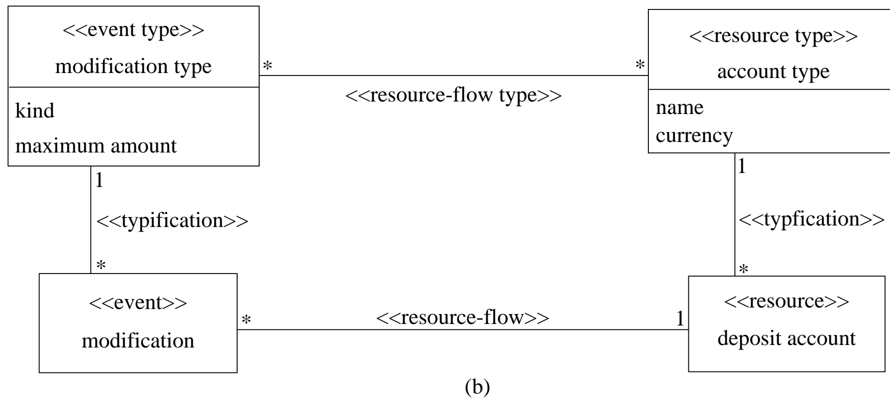


Figure 3.11: (a) Pattern for modeling permissions, (b) an example

Example: An example of permission can be the maximum amount of modification such as withdrawal from a foreign currency deposit account. Figure 3.11(b) models such a permission and also uses *currency* and *maximum amount* attributes.

Table 3.2 is related to Figure 3.11(b) because the table uses the entities and attributes of the figure. For instance, the last row of this table can be described using the columns of this table as a modification with a maximum amount for an account type with a currency (e.g., foreign currency) of a deposit account. The first row is similar to the third row, but the first row does not use attributes. In the second row, *create* is the value of attribute *kind* of the *modification type* class.

Resulting Context: To summarize, this pattern models permission, which is a foundational building block of access control. In addition, using various attribute combinations of event types and resource types, one will be able to provide constraints on describing permissions.

event/event type	event type attribute	resource type	resource	resource type attribute
modification	-	account type	deposit account	-
create	-	account type	loan account	-
modification	maximum amount	account type	deposit account	currency

Table 3.2: Entities and attributes of *permission modeling* pattern

3.2.3 A Core Access Control Model Formed by Combining Patterns

This section describes how to combine the *users and role assignments* pattern and the *permission* pattern (Sections 3.2.1 and 3.2.2) to construct a core access control model. To recap, the *users and role assignments* pattern models the left hand-side of Figure 3.5 consisting of circles called users and roles, and their connections. The *permission* pattern models the same figure's right hand-side consisting of OPS (operations) and OBJ (objects) circles and their connections.

Name: *Core Access Control Model* Pattern

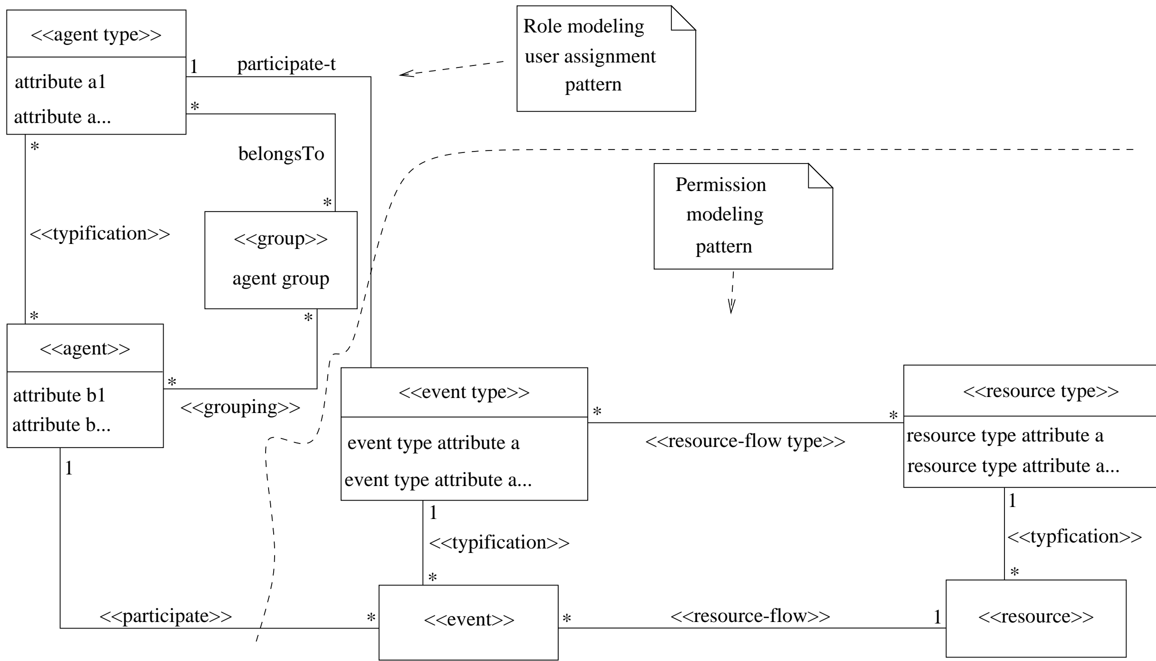
Context: A core access control model represents the basic functions of an access control model. This core access control model can represent who can perform what operations on which objects.

Problem: How can a core access control model for business models be obtained using the same entities used in describing business models? How can such a core access control model portray access control policies?

Solution: The combination of *role modeling and user assignments* and *permission* patterns creates a core model of authorization. Figure 3.12(a) shows the combination of these two patterns, where the dotted line separates the two patterns and identifies the two associations (i.e., participate-t and participate) that connect them.

Example: Figure 3.12(b) shows an example of the combination of these two patterns. This combination presents an example of a core RBAC model in which an employee role (i.e., teller) is shown in conjunction with one permission (e.g., modification of deposit accounts) of this role.

Table 3.3 is related to Figure 3.12 because the columns of this table are the classes of the figure. This table shows the policies of the banking running example. The word



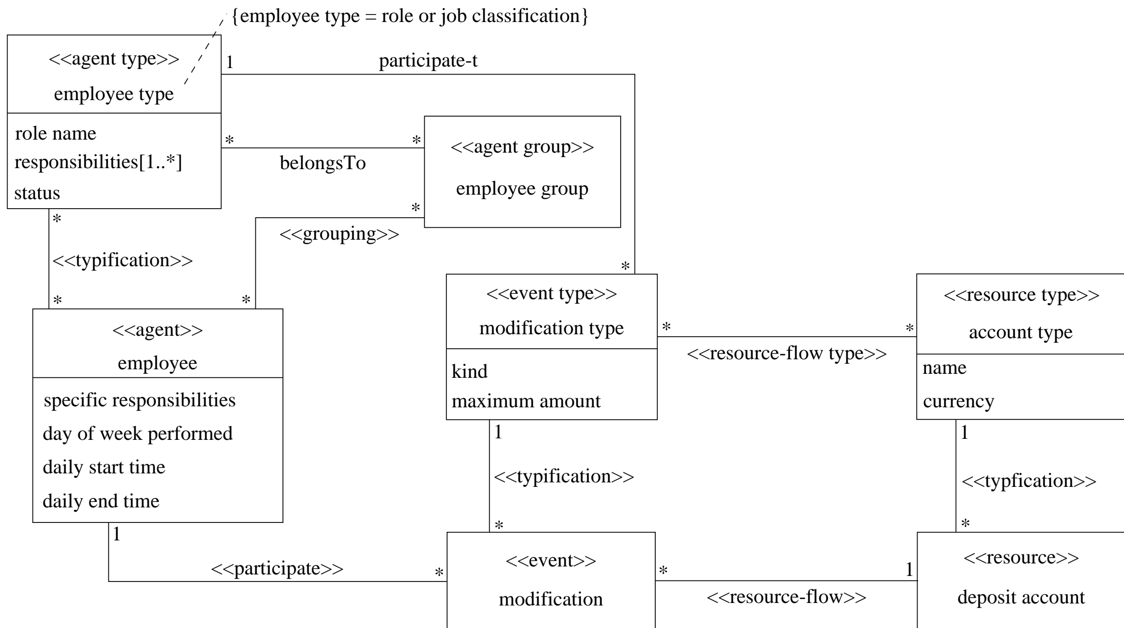


Figure 3.12: (a) A combination of two presented patterns, (b) an example

"policies" in this table and this section is the same as the use of this word in the running banking example. The columns of this table represent event and event type, resource and resource type, agent and agent type that are used to portray access control policies. Based on the first row of Table 3.3, an employee with a teller role can modify account types of deposit accounts. The words *create* and *delete* in this table are the values of attribute *kind* of the modification type class. The remaining rows of this table are similar to the first one.

	Permission				
	A		**E**	**R**	
policies	**user (agent)**	**agent type or role**	**event/ event type**	**object (resource)**	**object type (resource type)**
P1	employee	teller	modification	deposit accounts	account type
P2	employee	customer service rep	modification, create, delete	deposit accounts	account type
P3	employee	accountant	create	general ledger reports	account type
P4	employee	accounting manager	modification	ledger posting rules	account type
P5	employee	loan officer	create, modification	loan accounts	account type

Table 3.3: A table representation of policies for the banking example

Figure 3.12(b) uses a UML class diagram and portrays the first row of Table 3.3; the other rows can be portrayed similarly. The policies of this table can be described by an access control model, which is an extension to RBAC.

Resulting Context: To summarize, the combination of a *role modeling and user assignments* pattern with a *permission modeling* pattern creates a core access control model. This combination (i.e., a core access control model) can portray a large range of access control models and policies.

3.2.4 Advantages of the Core Access Control Model

Figures 3.10 and 3.12 use classes that have *type* or *group* as part of their names, such as a *resource type* class or an *agent group* class. The advantages of using these classes are described next.

First Advantage: REA uses *resource type*, *event type*, and *agent type* classes that basically keep information and detailed descriptions about resources, events, and agents,

respectively. Figure 3.14 illustrates the purpose and advantage of using such classes with an example of an *agent type* class called Job Classification. REA uses these classes for the modeling of general policies, and this thesis uses these classes for presenting access control models and their portrayed access control policies. An example of using such classes for the representation of general policies and its advantage is illustrated by Figure 3.13.

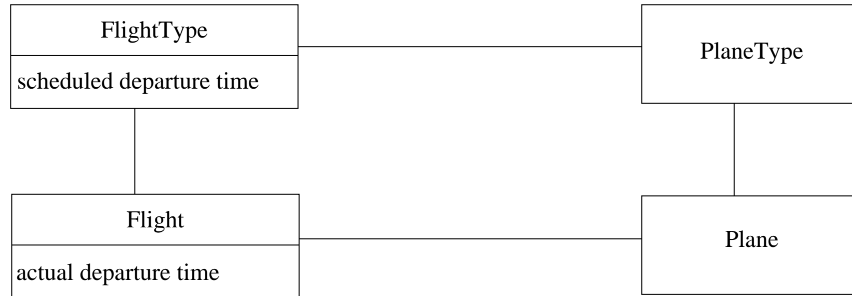


Figure 3.13: Two levels of policy descriptions

The top level of Figure 3.13 can be used for defining policies, because the instances of FlightType and PlaneType describe the common characteristics of flights. For instance, based on the top level of this figure, one can state that *flights between Toronto and Vancouver leave at 10 am based on the value of scheduled departure time* (*departure city* and *arrival city* attributes are not shown in this figure). The instances of the lower level of this figure can be actual flights that take place. For instance, based on the lower level, one can state that *the flight between Toronto and Vancouver actually left at 10:15 am.* An alternative of not using the FlightType class can be the inclusion of the FlightType's *scheduled departure time* attribute in the Flight class and deleting the FlightType class. But this alternative is not concise and thus creates redundancy, because *scheduled departure time* must be included in every instance of the Flight Class. Therefore, the use of classes such as FlighType reduces redundancy as policies can be defined using the common characteristics of classes and be applicable for all the instantiations of the concept.

Second Advantage: Aggregation and group are discussed in the literature [107, 85, 67, 72, 71, 92]. Figure 3.15 provides a brief overview. REA uses the notion of *group*, a subcategory of aggregation, to be specific on the kind of aggregation it uses. Similarly, this thesis uses *group* in modeling access control.

In addition, REA uses explicit aggregates, and this thesis also presents access control models using the explicit representation of aggregates. The advantages of explicit aggregates versus implicit modeling of aggregates is discussed in the literature. Figure 3.16

The Use of Classes with Type as a Part of Their Names

The following left hand-side figure has a class called Job Classification that is an agent type class, or in this case, the Employee type, in REA. The Job Classification class can hold detailed information, such as *role name* values and *base pay rate* values for each Employee or agent. The right hand-side figure shows an alternative approach in which the *role name* and *base pay rate* attributes of the Job Classification class are included in the Employee class, and the Job Classification class, or the employee type class, does not exist.

The disadvantage of the right-hand side of the figure is its lack of conciseness and consequent redundancy in keeping information. For instance, the *base pay rate* attribute of the Employee class must be duplicated for every instance of the Employee class. In addition, if the value of *base pay rate* changes, this change must be reflected for every instance of the Employee class. Similarly, if an attribute and its value that applies to all role names is added or deleted, then for the model on the right, this change must be reflected for every instance of the Employee class. For the model on the left, only one entry is added to or removed from the Job Classification class.

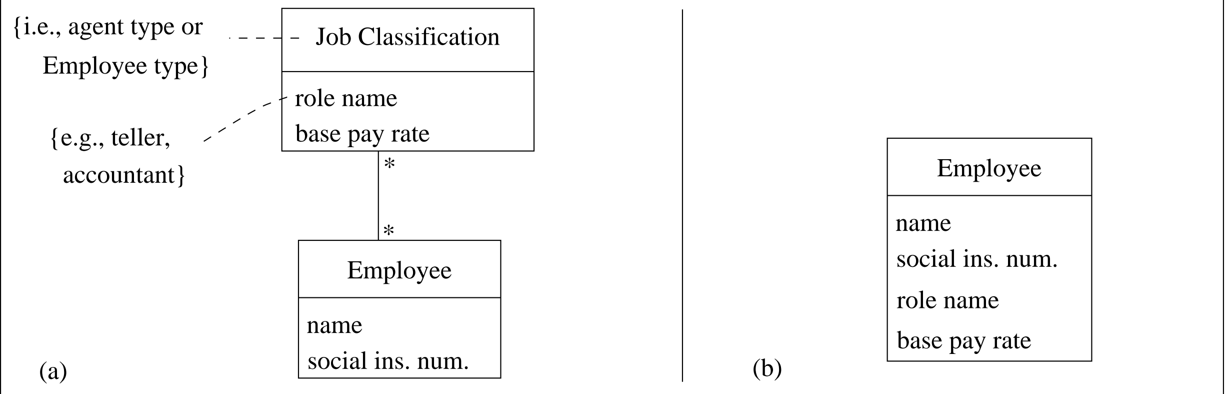


Figure 3.14: The Job Classification class is the agent type class.

provides an overview of this topic. As a result of using explicit aggregates, the model presented in this chapter is more expressive than general access control models.

The following examples show this benefit and present cases that cannot be expressed by general access control models because explicit aggregates are not used.

Agent Group Example: Can a teller within a team with size three (size is an attribute of the team) have a certain permission on a resource (e.g., a large withdrawal from a savings account with foreign currency)? An implicit representation of a team as a relationship

Aggregation and *group* are widely discussed in the literature (e.g., [107, 85, 67, 72, 71, 92]). *Group* is the member-bunch sub-category of aggregation, e.g., a ship is part of (i.e., member of) a fleet. The member-bunch is an aggregation with two specific characteristics [107, 67]: 1) parts are not "in a specific spatial/temporal position" in relation to each other and to the whole (the non-configurational attribute) and 2) parts are not similar in relation to each other and to the whole (the nonhomoeomerous attribute).

Figure 3.15: Aggregation and Group

cannot describe the size (i.e., an emergent property of a team) and the mutual property between a team and the rest of the model.

Agent Group Example: Can a teller belong to a team with no leader have/have not certain permissions? This expression cannot be described by an implicit representation of aggregates. The explanation provided for example 1 in Figure 3.16 applies here.

Agent Group Example: Can a loan officer be a member of more than one team and, as a result, get permissions he/she should not have (e.g., first and second approvers of a loan)? This expression cannot also be described by an implicit representation of aggregates. The explanation provided for example 2 in Figure 3.16 applies here.

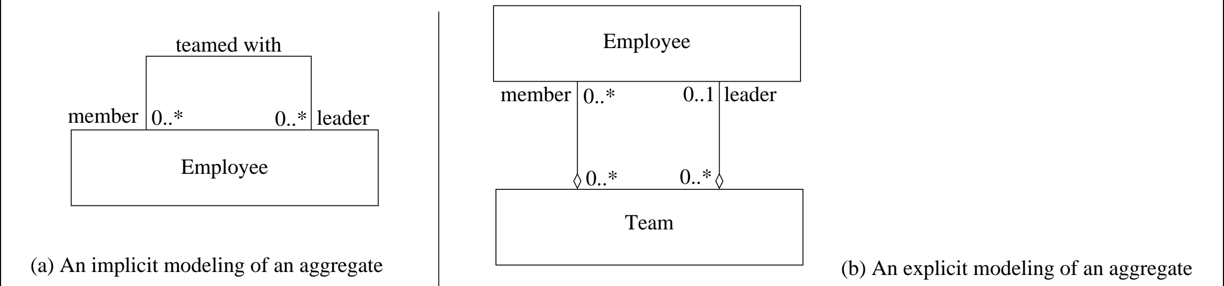
Resource Group Example: A broker has a permission to sell a minimum and maximum number of stocks within a portfolio. A portfolio is a grouping of resources, and minimum and maximum number of stocks are two attributes of a portfolio.

Resource Group Example: A teller has a permission to modify a portfolio that has a certain number of accounts. A portfolio represents a grouping of certain accounts, and number of accounts is an attribute of a portfolio.

Event Group Example: A head teller has an authorization to perform closing events (a grouping of events) at the closing time of a bank. The explanation provided for example 2 in Figure 3.16 applies here. The prefix *head* in head teller is similar to the member in that example, and more than one team in that example is similar to more than one event here.

In summary, the model presented in this chapter is more expressive than any general access control models because of the inclusion of several related additional group entities, attributes, and their relationships. Table 3.4 shows the additional elements that are introduced in this chapter and added to access control models.

An aggregate can be modeled implicitly using associations, whereas the explicit approach models an aggregate using a class [5, 93]. The following figure [93] on the left models Team as an association, whereas the figure on the right models Team as a class.



(a) An implicit modeling of an aggregate

(b) An explicit modeling of an aggregate

An explicit modeling of an aggregate has two advantages [109]: 1) emergent properties of the model can be expressed, e.g., the size (i.e., an emergent property) of a group or team (i.e., an aggregate); for the model on the right, Team can have an attribute of *size*, and 2) the mutual properties between an aggregate and its components can be expressed. The latter case is described by the following two examples that also show that the model on the left cannot model some cases.
Example 1: Can a team have no leader? The figure on the left shows a *team* as an association between member(s) and leader(s). By this choice, if this association exists, then both association-ends must be in place; i.e., the implicit aggregation cannot model a team that has no leader.
Example 2: Can an employee be a member of more than one team? The answer to this question by the model on the left is indeterminate because this model associates an employee, as a member, with leaders. If an employee, as a member, is associated with many leaders, then the association does not explicitly indicate that these leaders belong to different teams, i.e., the leaders may lead one team. As a result, the answer to the question of whether an employee can be a member of more than one team is indeterminate by the model on the left.

Figure 3.16: Implicit versus Explicit Modeling of an Aggregate [93]

3.2.5 Modeling Constraints, Role Hierarchies, and Mutually Exclusive Roles

In general, constraints are included in access control because many such constraints among users (agents), roles (agent types), and permissions occur in reality. For example, it may be the case that the role of teller needs authorization (constraints) to deposit cheques

Addition of groups	Addition of attributes	Addition of relationships
resource group	resource group attributes	among resource group and the rest of AC model
agent group	agent group attributes	among agent group and the rest of AC model
event group	event group attributes	among event group and the rest of AC model

Table 3.4: Additional elements added to access control models

(permission) over 1000 dollars. Constraints are part of RBAC and other access control models and are the principle motivation behind them [87]. In effect, constraints define access control restrictions.

Constraints on Agents, Roles, and Permissions: The *role modeling and user assignments* pattern (Section 3.2.1) can model constraints on agents and roles (i.e., agent types) and their attributes and agent-role assignments as previously described. For instance, a constraint can be that an employee does not have a role of teller on Sundays. Similarly, as described in Section 3.2.2, the *permission modeling* pattern can model constraints on permissions using an event type and its attributes (e.g., a *maximum amount* attribute), a resource type and its attributes (e.g., the *currency* attribute of Figure 3.11), and the relationship between event type and resource type. Another example of using constraints in permission modeling can be that a pilot (i.e., a role), based on *skill* or other attributes, is authorized to fly (an event) certain type of planes (resources).

Role Hierarchies and Mutually Exclusive Roles: Both role hierarchies and mutually exclusive roles can be viewed as cases in which constraints on roles exist. For instance, for role hierarchies, one can mention that a manager is higher than a teller in terms of authorization. A variation of the *step one* of *role modeling and user assignments* pattern (Section 3.2.1) is used next to describe both the modeling of role hierarchies and mutually exclusive roles.

Name: A variation of the *step one* of *role modeling and user assignments* pattern

Context for Role Hierarchies: Role hierarchies in organizations express the line of authorities or responsibilities and exemplify constraints and relationships among roles in terms of authorization.

Context for Separation of Duties: Separation of duties is a well-known principle and

	teller	customer service rep	accountant	accounting manager	loan officer
teller	-	-	+	-	+
customer service rep	-	-	-	+	-
accountant	+	-	-	-	+
accounting manager	-	+	-	-	+
loan officer	+	-	+	+	-

Table 3.5: A static separation of duties representation for the banking example

has two broad categories: static and dynamic; the dynamic one contains several variations [94]. The static separation of duty represents constraints on roles as a strong exclusion of roles in which no user can ever take both roles A and B if these two roles are exclusive (e.g., that a purchaser and an approver of an order must be different people is a common example provided frequently in the literature). The static separation of duties of the banking example (Figure 3.7) is shown in Table 3.5, where the plus sign in a cell of this table indicates the existence of a static separation of duty between two roles associated with that cell.

Problem: How are role hierarchies modeled? In addition, how can the static separation of duties or mutually exclusive roles be modeled? Because of the relationship between roles (i.e., agent type) and agents, they both must be included in the presentation of role hierarchies and mutually exclusive roles.

Solution for Representing Role Hierarchies: The pattern for modeling role hierarchies is a variation of the *step one* of *role modeling and user assignments* pattern (first pattern) (Section 3.2.1). Therefore, the same explanation provided for the first pattern also holds here. Figure 3.17(a) shows the specialized and enhanced Basic pattern from the *step one* of first pattern, previously shown in Section 3.2.1. Figure 3.17(b) is a variation in which a relation called *role hierarchy* with multiplicities is added. Based on the description provided for role hierarchies representations in Figure 3.18, this pattern can represent role hierarchies for the general case. Figure 3.17(c) shows an example of this variation for representing role hierarchies.

Solution for Representing Mutually Exclusive Roles: Mutually exclusive roles can be represented as relationships among roles, as previously shown [84]. The pattern for modeling mutually exclusive roles is also a variation of the *step one* of the first pattern (Section 3.2.1). The same explanation provided for the first pattern also holds. Figure 3.19(a) shows the specialized and enhanced Basic pattern from the *step one* of the first

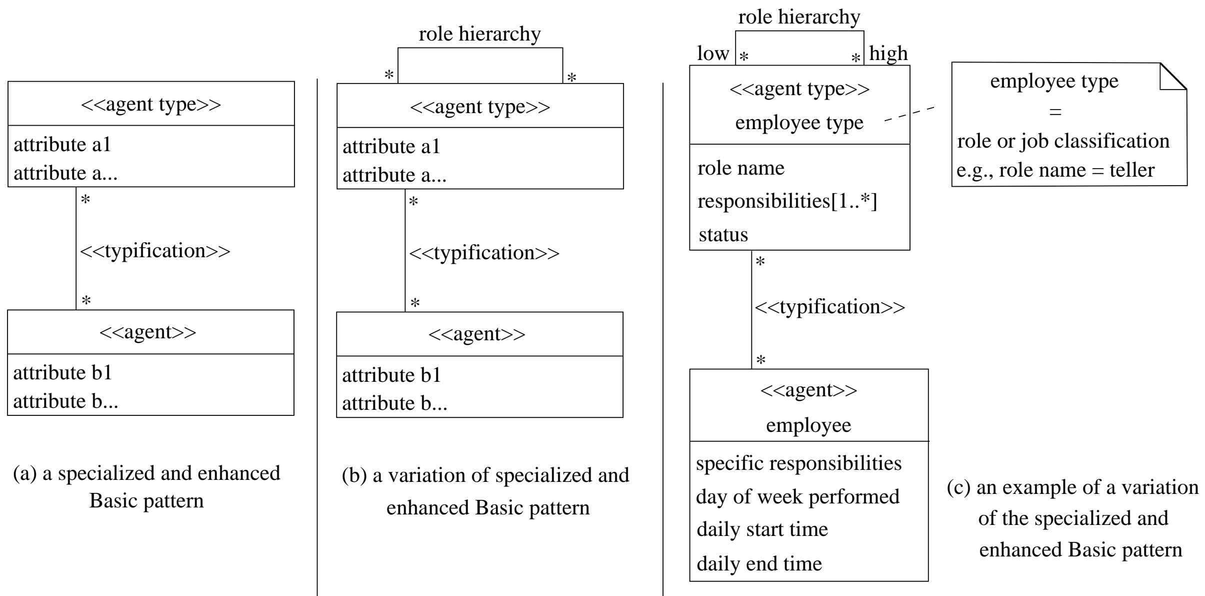


Figure 3.17: The specialized and enhanced Basic pattern, its variation to model role hierarchies, and an example

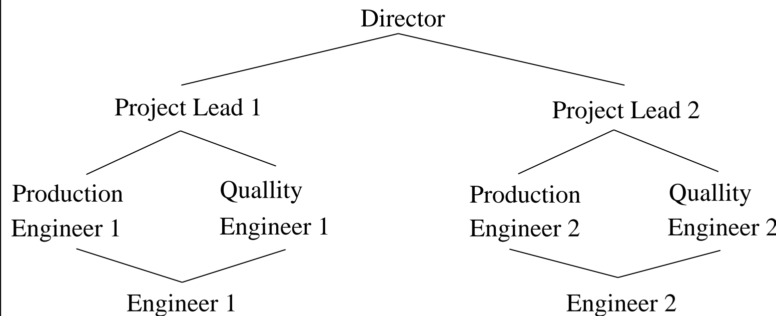
pattern. A variation of this pattern is shown in Figure 3.19(b) and (c), where a relation called *mutually exclusive* with multiplicities is shown to represent mutually exclusive roles.

Example: In Figure 3.17(c), if the *role hierarchy* relation has a teller role on the side identified as *low*, then a *customer service rep* is on the side identified as *high*.

Similarly, in Figure 3.19, for mutually exclusive roles, whenever a role is of a teller, then one instance of this relation may indicate that a teller is not an accountant, and another instance is that a teller is not a loan officer.

Resulting Context: This pattern models role hierarchies and mutually exclusively roles of access control. This pattern is a variation of the *step one* of the first pattern presented in this chapter, and it shows an agent and agent type (role). RBAC mentions dynamic separation of duties whose support needs a dynamic representation. Next chapter (Chapter 4) provides a dynamic representation.

Role hierarchies are represented either by *partial orders* for the general case or by *tree* structures for the restricted case as described next.
A partial order is a binary relation between certain elements (therefore the word partial) of a set where the elements of sets in this case are roles. For instance, the following figure shows a partial order for the *role hierarchy* relation, which is depicted as a line connecting two roles. This figure shows roles, such as Director and Production Engineer 1, and their positions in this hierarchy. The higher a role in this hierarchy indicates the higher level of authorization.



A restriction on roles can be imposed such that roles may have one or more immediate descendants but can only have a single immediate ascendant. With this restriction on roles, role hierarchies can be represented using tree structures such as the one shown in the following figure.

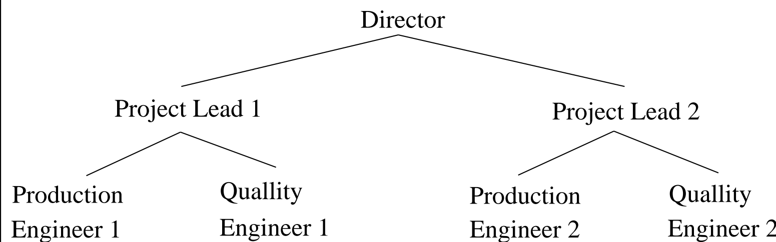


Figure 3.18: Role hierarchies representations [34, 87]

3.2.6 Extending the Approach to Describe DAC and MAC

As previously mentioned in Chapter 2, three main categories of access control are Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role-based Access Control (RBAC) [98, 12]. This chapter selected RBAC to show the approach, but this method can be applied to more general access control approaches, as will be illustrated through application to DAC and MAC.

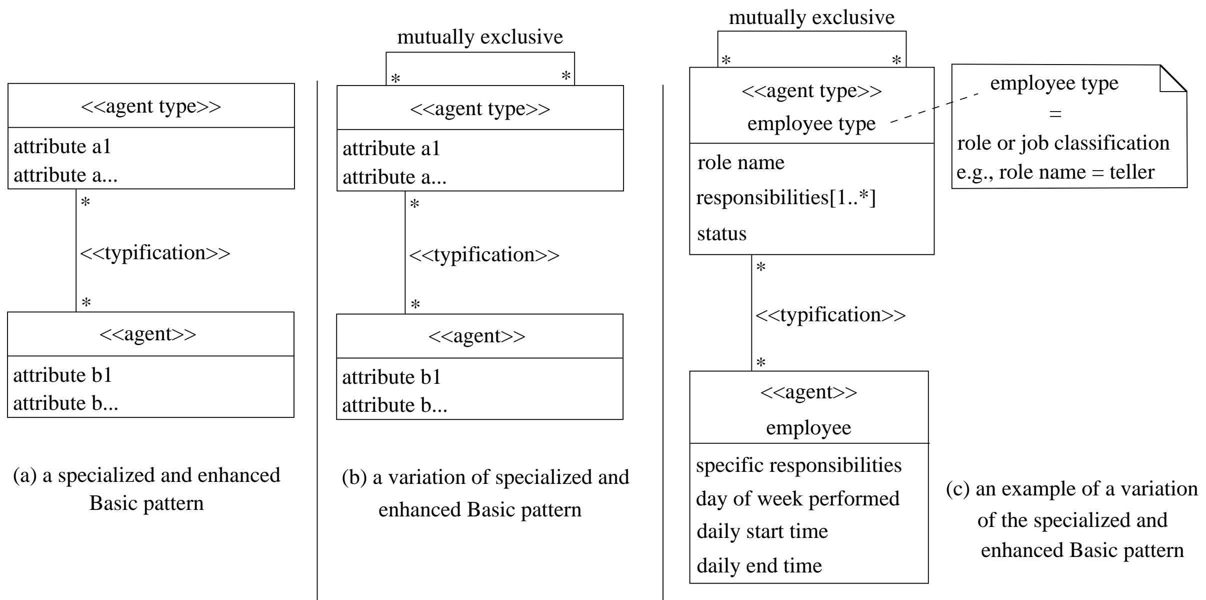


Figure 3.19: The specialized and enhanced Basic pattern, its variation to model mutually exclusive roles, and an example

Describing DAC: The literature [88, 80] discusses the use of RBAC to represent Discretionary Access Control (DAC). Similar discussion applies here. The main idea of DAC is based on the principle that the owner of that object, usually the creator, has the authority to determine who gets or loses access to an object. Several DAC variations, which mainly differ from each other in terms of rules for granting or revoking access, are available. To describe DAC using RBAC, one can proceed as follows.

To define a variation of DAC in which an owner of a resource is the only one who can grant or revoke an individual's write access to the owner's resource, one needs to add two more elements to the model described in this chapter. The first element is an *administrative role*, e.g., *own-resource*, which is in addition to the role entity that already exists in RBAC. The second addition is an *administrative permission* entity with two permissions: One permission allows the addition of users to have write access to the owner's resource, and the other allows the removal of users from having write access to the owner's resource. Similarly, for adding read access, two permissions for the purpose of reading are required: one to add a user, and the other to remove the user.

Describing MAC: Similarly, Mandatory Access Control (MAC) can be represented by RBAC as discussed in the literature [80]. As previously mentioned (Chapter 2), security

labels are assigned to both users and objects in MAC. The security labels for users are called *security clearance*, and the labels for objects are named *security classification*. Examples of such labels are *top-secret*, *secret*, *confidential*, and *unclassified*. MAC can describe a user's access to an object by using two rules: *no read up* and *no write down*. *No read up* means that a user can read objects if the security labels of objects are equal or lower than the security labels of the user; *no write down* indicates that a user can write objects if the security labels of objects are higher or equal than the security clearance of the user.

The model described in this chapter can be adjusted to describe MAC as shown next. Permission is represented in this chapter as an operation on a resource; therefore, in this case, an operation is either *read* or *write* on a resource. The roles of RBAC to represent MAC can be identified with security labels for both read and write operations. For instance, two roles can be top-secret-read and top-secret-write, where top-secret is a security label. Therefore, the number of roles is determined by read, write, and the number of security labels. For instance, a user maps to two roles: security-label-read (e.g., top-secret-read) and security-label-write (e.g., top-secret-write). The effect of no read up and no write down can be included by the inclusion of two constraints: 1) a user who is assigned to a role can read a resource if and only if the user is also assigned to the lowest level of writing that resource and 2) a user is assigned only to one role (e.g., top-secret-read). Finally, two hierarchies are needed: one to represent the security-label-read hierarchy and the other to describe theq security-label-write hierarchy.

3.3 Defining Access Control Rules based on Models

Given an agent and a resource, an access control rule states whether the agent is permitted to perform an operation on the resource. Performing an operation is equivalent to an event as described previously. These rules can apply to individual agents, events, and resources, or to types of agents (roles), resources, and events. Further, a rule can be generalized and can contain combinations of agents and agent types, combinations of resources and resource types, and combinations of events and event types.

For example, tellers and managers but not loan officers are permitted to modify but neither create nor delete deposit accounts with foreign currency. In this case, tellers, managers, and loan officers are used in an agent expression, deposit accounts with foreign currency are used in a resource expression, and modify, create, and delete are components of an event expression. In general, an access control rule can be viewed as an expression that combines an agent expression, a resource expression, and an event expression and their relationships.

Table 3.6, which is the first row of Table 3.3, presents an example of the entities of an AC rule that is based on the model shown in Figure 3.20.

		Permission		
A		**E**	**R**	
user (agent)	**agent type or role**	**event/ event type**	**object (resource)**	**object type (resource type)**
employee	teller	modification (modify)	deposit accounts	account type

Table 3.6: An example of the elements of an access control (AC) rule (a singleton policy)

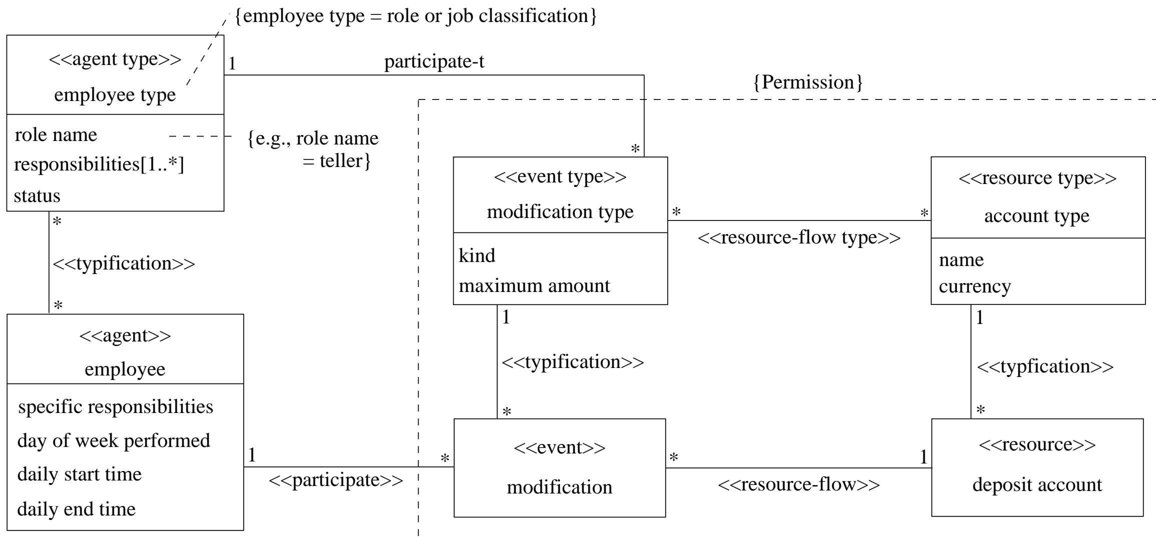


Figure 3.20: An example of an access control model

The columns of this table are classes of an access control model. Figure 3.20, a slight simplification of Figure 3.12(b), shows an example of such an access control model corresponding to Table 3.6. The general format of an access control model for this example was also previously presented in Figure 3.12(a).

3.3.1 Access Control Rule Syntax

As access control rule syntax uses types, groups, and their instances, the following example can be beneficial to provide another overview of these terms.

Figure 3.21 shows plane, plane type, and fleet. Examples (instances) of planes and plane types represent completely different information. Examples (instances) of plane types are Boeing 737 and Boeing 787 that can provide general characteristics of plane type such as their number of engines. On the other hand, examples (instances) of planes represent actual planes with specific identification numbers and their actual plane age. Similarly, in this figure, examples of flight type captures general information such as the *scheduled departure time* of flight types, whereas the examples (instances) of flights provide the information about *actual departure time* of flights that took place. Finally, a fleet (a group) can be organized based on plane types. Then, general examples of a fleet can consist of a specific plane type.

Note that the access control rule syntax does not include the instances of agents, resources, or events but instead the instances of types and groups are used in the sense described. The intention for inclusion of instances at the type and group levels is to use the general characteristics and general information provided by these instances, as described.

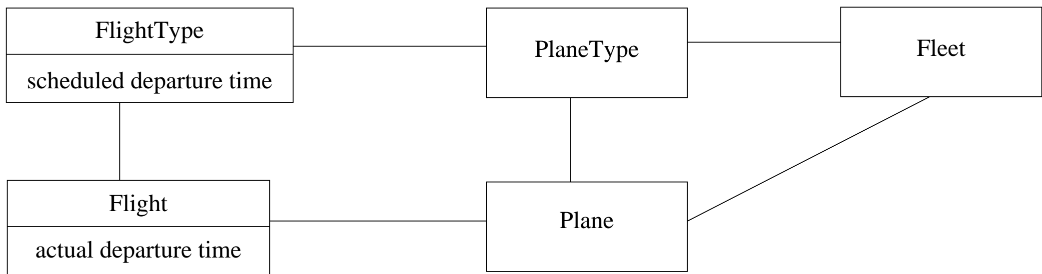


Figure 3.21: Plane, plane type, flight, flight type, and fleet

An access control (AC) rule can be stated as

ACRule ::= (AgentExp and ResourceExp and EventExp and AgeEveRel and ResEveRel) implies EventResult

Figure 3.22 presents the detailed syntax of ACRule in Extended Backus-Naur Form (EBNF). Table 3.7 (ISO 14977) [99] shows the EBNF elements and their meanings.

Appendix B provides the corresponding definitions of AC rules in Backus-Naur Form (BNF). These definitions (i.e., the AC rule grammar) in BNF are checked with a tool called Gold Parser [24]. Appendix B also provides definitions related to the descriptions of agents, resources, and events in BNF and EBNF. All BNF definitions in this appendix are checked with the Gold Parser tool to ensure the grammars are well formed.

Extended BNF	Meaning
unquoted words	Non-terminal symbol
"..."	Terminal symbol
(...)	Grouping
[...]	Optional symbols
{...}	Symbols repeated zero or more times
=	Defining symbol
|	Alternative
;	Rule termination
,	Concatenation

Table 3.7: Extended BNF (EBNF)

Access Control Rule Definition in Extended BNF

ACRule = (AgentExp "and" ResourceExp "and" EventExp "and" AgeEveRel "and" ResEveRel) "implies" EventResult;
AgentExp = ([uop] equA equArep);
uop = "not";
equA = ATG | attrATG;
equArep = {bop [uop] equA};
bop = ("and" | "or");
ATG = Agent "=" className | AgentTG "=" identifier
attrATG = AgeAttIde"."attrNameValue {"," AgeAttIde"."attrNameValue};
AgentDesignation = Agent | AgentType | AgentGroup;
AgentTG = AgentType | AgentGroup;
AgeAttIde = ATG | ATClassName | AClassName | AGClassName | AgentDesignation
className = "string";
identifier = "instance" "string";
instanceName = "string";

Access Control Rule Definition in Extended BNF (contd.)

```
Agent =  "var" "string" identifierA;
AgentType =  "var" "string"  identifierAT;
AgentGroup =  "var" "string"  identifierAG;
attrNameValue =  attrName  attrValue;
identiferA =  "Agent" ;
identiferAT =  "AgentType";
identiferAG =  "AgentGroup";
attrName =  "string";
attrValue =  relationN  valueNum  |  relationC  valueC;
relationN =  "<"  |  ">"  |  "≥"  |  "≤"  |  "="  |  "≠";
relationC =  "equals"  | "notEquals"
valueNum =  number;
number =  "integer"  |  "real";
valueC =  "character"  |  "string"  |  bool;
bool =  "True"  | "False";
ATClassName =  identifier  "in"  (AgentType "=" identifier);
AClassName =  className  "in"  (Agent "=" className);
AGClassName =  identifier  "in"  (AgentGroup "=" identifier);
ResourceExp =  ([uop]  equR   equRrep);
equR =  RTG  |  attrRTG;
equRrep = {bop   [uop]  equR};
RTG =  Resource "="className  |  ResourceTG "="identifier ;
ResourceDesignation =  Resource  |  ResourceType  |  ResourceGroup;
ResourceTG =  ResourceType  |  ResourceGroup;
Resource =  "var" "string"  idenitiferR;
ResourceType =  "var" "string"  identifierRT;
ResourceGroup =  "var" "string"  identifierRG;
attrRTG =  ResAttIde"."attrNameValue  {"," ResAttIde"."attrNameValue};
identifierR =  "Resource";
identifierRT =  "ResourceType";
```

Access Control Rule Definition in Extended BNF (contd.)

```
identifierRG =   "ResourceGroup";
ResAttIde =   RTG  |  RTClassName  |  RClassname  |  RGClassname  |
              ResourceDesignation
RTClassName =   identifier   "in" (ResourceType "=" identifier);
RClassname =   className   "in" (Resource "=" className);
RGClassname =   identifier   "in" (ResourceGroup "=" identifier);
EventExpr =     ([uop]   ETG    ETGrep);
ETG =   Event "=" className  |  EventTG "=" identifier;
ETGrep =  {bop    [uop]   ETG};
EventDesignation =   Event    |   EventType    |   EventGroup;
EventTG =   EventType    |   EventGroup;
Event =   "var"  "string"   identifierE;
EventType = "var"  "string"   identifierET;
EventGroup = "var"  "string"   identifierEG;
identifierE =   "Event";
identifierET =   "EventType";
identifierEG =   "EventGroup";
AgeEveRel =   ([uop]   AgeERel    AgeERelrep);
AgeERel =   "RelATET"(ATClassName "," ETClassName)   |
            "RelAE"(AClassName "," EClassName)   |
            "RelATG"(ATClassName "," AGClassName)   |
            "RelAT"(AClassName "," ATClassName)   |
            "RelAG"(AClassName "," AGClassName);
AgeERelrep = {bop    [uop]   AgeERel}
ResEveRel =   ([uop]   ResERel    ResERelrep);
ResERel =   "RelRTET"(RTClassName "," ETClassName)  |
            "RelRE"(RClassName "," EClassName)  |
            "RelRTG"(RTClassName "," RGClassName)   |
            "RelRT"(RClassName "," RTClassName)   |
            "RelRG"(RClassName "," RGClassName);
```

Access Control Rule Definition in Extended BNF (contd.)

```
ResERelrep = {bop [uop] ResERel};
EventResult = ([uop] accETG accETGrep);
accETG = accessETG "=" result;
accETGrep = {bop [uop] accETG};
result = "permit" | "deny";
accessETG = (ETClassName | EClassName) "Access"
ETClassName = identifier "in" (EventType "=" identifier);
EClassName = className "in" (Event "=" className);
```

Figure 3.22: Access control rule definition in Extended BNF

3.3.2 Description of Access Control Rule Syntax

EBNF is used to define the syntax of access control rules in Figure 3.22. This figure can be described based on the definitions of notations used by EBNF as shown in Tables 3.7.

Figure 3.22 is explained by identifying the starting lines of each definition. For instance, if an item of this description mentions *the expression starting with ACRule*, it indicates that the definition starts with a line commencing with ACRule in Figure 3.22.

The first expression starts with ACRule: An access control (AC) rule is defined as an agent expression (*AgentExp*), a resource expression (*ResourceExp*), an event expression (*EventExp*), relations related to agents and events (*AgeEveRel*), relations related to resources and events (*ResEveRel*). The conjunctions of these expressions imply an event result (*EventResult*).

The expressions in Figure 3.22 are divided into six sections based on these six elements: *AgentExp*, *ResourceExp*, *EventExp*, *AgeEveRel*, *ResEveRel*, and *EventResult*

I. AgentExp The following descriptions are mainly related to AgentExp.

- The expression starting with AgentExp: An agent expression, *AgentExp*, is identified by a form called *equA* that may be followed by another form called *equArep* (*equA* and *equArep* are defined shortly). An optional *not* can also precede *equA*.
- The expression starting with uop: *uop* is "not".

- The expression starting with equA: *euqA* can be expressions either about agents, or agent types, or agent groups (ATG), which are called agent-related expressions, or about the attributes of agents, or agent types, or agent groups (attrATG).
- The expression starting with equArep: Zero or more agent-related expression(s) is (are) possible using ‘"and" and "or" connectives. An optional "not" can also precede *equArep* .
- The expression starting with bop: *bop* is either "and" or "or".
- The expression starting with ATG: This expression identifies an agent, or agent type, or agent group in conjunction with a class name or an identifier.
- The expression starting with attrATG: This expression identifies an attribute name, and its value in conjunction with an identification of *AgeAttIde*, which is defined shortly.
- The expression starting with AgentDesignation: An *AgentDesignation* can be any of the following three elements: agent, agent type, or agent group.
- The expression starting with AgentTG: An *AgentTG* can be either an agent type or an agent group.
- The expression starting with AgeAttIde: An agent attribute identification can be either an agent designation, the class name of an agent, agent type, or agent group, or can be an agent designation along with the class name of the agent designation.
- The expression starting with className: A class name is a string.
- The expression starting with identifier: An identifier is a string that distinguishes an instance.
- The expression starting with Agent: An agent is defined as a variable that holds a value of a string type within identifierA.
- The expression starting with AgentType: An agent type is defined as a variable that holds a value of a string type within identifierAT.
- The expression starting with AgentGroup: An agent group is defined as a variable that holds a value of a string type within identifierAG.
- The expression starting with attrNameValue: An attribute has a name and a value.

- The expression starting with identifierA: An agent identifier is recognized by the word agent.
- The expression starting with identifierAT: An agent type identifier is recognized by the word agent type.
- The expression starting with identifierAG: An agent group is recognized by the word agent group.
- The expression starting with attrName: An attribute name is a string.
- The expression starting with attrValue: Attribute values are either numbers or strings. Relational symbols (*relationN* or *relationC*) are also included for both numbers and strings.
- The expression starting with relationN: relationN is short for *relation for numbers* and is one of $<$, $>$, $<$, $\geq$, $\leq$, and $=$ symbols for comparing numbers.
- The expression starting with relationC: relationC is short for *relation for characters* and is for comparing equalities of characters and strings.
- The expression starting with valueNum: *ValueNum* is a number.
- The expression starting with valueC: *valueC* is any of character, string, or bool value.
- The expression starting with bool: bool can be either True or False.
- The expression starting with ATClassName: *ATClassName* is defined to be an identifier, previously defined, of an agent type.
- The expression starting with AClassName: *AClassName* is defined to be the class name of an agent.
- The expression starting with AGClassName: *AGClassName* is defined to be an identifier, previously defined, of an agent group.

II. ResourceExp: The next explanations are related to ResourceExp.

- The expression starting with ResourceExp: A resource expression, *ResourceExp*, is identified by a form called *equR* that may be followed by another form called *equRrep* (*equR* and *equRrep* are defined shortly). An optional `"not"` can also precede the form *equR*.

- The expression starting with equR: *euqR* can be expressions either about resources, or resource types, or resource groups (RTG), which are called resource-related expressions, or are about the attributes of resources, or resource types, or resource groups (attrRTG).
- The expression starting with equRrep: Zero or more resource-related expression(s) is (are) possible using "and" and "or". An optional "not" can also precede *equRrep*.
- The expression starting with RTG: This expression identifies a resource, or resource type, or resource group in conjunction with either a class name or an identifier.
- The expression starting with ResourceDesignation: A *ResourceDesignation* can be any of these three: resource, resource type, or resource group.
- The expression starting with ResourceTG: A *ResourceTG* can be either a resource type or a resource group.
- The expression starting with Resource: A resource is defined as a variable that holds a value of a string type within identifierR.
- The expression starting with ResourceType: A resource type is defined as a variable that holds a value of a string type within identifierRT.
- The expression starting with ResourceGroup: A resource group is defined as a variable that holds a value of a string type within identifierRG.
- The expression starting with attrRTG: This expression identifies an attribute name, and its value with the class name of either a resource, or resource type, or resource group.
- The expression starting with identifierR: A resource identifier is recognized by the word resource.
- The expression starting with identifierRT: A resource type identifier is recognized by the word resource type.
- The expression starting with identifierRG: A resource group is recognized by the word resource group.
- The expression starting with ResAttIde: A resource attribute identification can be either a resource designation, the class name of a resource, resource type, or resource group, or can be a resource designation along with the class name of a resource designation.

- The expression starting with RTClassName: *RTClassName* is defined to be an identifier, previously defined, of a resource type.
- The expression starting with RClassName: *RClassName* is defined to be a class name of a resource.
- The expression starting with RGClassName: *RGClassName* is defined to be an identifier, previously defined, of a resource group.

III. EventExp: The following statements are related to EventExp.

- The expression starting with EventExp: An event expression, *EventExp*, is identified by a form called *ETG* that may be followed by another form called *ETGrep* (*ETG* and *ETGrep* are defined shortly). An optional `"not"` can also precede the form *ETG*.
- The expression starting with ETG: This expression describes the designation of an event, event type, or event group along with their class names or their identifiers.
- The expression starting with ETGrep: *ETGrep* describes zero or more event(s), or event type(s), or event group(s) designations along with their class names. The conjunction and negation of these events are possible.
- The expression starting with EventDesignation: An *EventDesignation* can have one of three: event, event type, or event group.
- The expression starting with EventTG: An *EventTG* can be either an event type or an event group.
- The expression starting with Event: An event is defined as a variable that holds a value of a string type within identifierE.
- The expression starting with EventType: An event type is defined as a variable that holds a value of a string type within identifierET.
- The expression starting with EventGroup: An event group is defined as a variable that holds a value of a string type within identifierEG.
- The expression starting with identifierE: An event identifier is recognized by the word event.

- The expression starting with identifierET: An event type identifier is recognized by the word event type.
- The expression starting with identifierEG: An event group is recognized by the word event group.

IV. AgeEveRel: This section explains different agent and event relationships.

- The expression starting with AgeEveRel: The expression identifies the existence of one or more relationships involving agents and events. An optional negation is possible to indicate such a relationship does not exist.
- The expression starting with AgeERel: The relationships involving agents and events can be between one of the following elements: agent type and event type, agent and event, agent type and agent group, agent type and agent, agent and agent group.
- The expression starting with AgeERelrep: This expression describes zero or more repetition(s) of relationships involving agents and events, as just described.

V. ResEveRel: This section describes different resource and event relationships.

- The expression starting with ResEveRel: This expression describes the existence of one or more relationships involving resources and events. An optional negation is possible to indicate such a relationship does not exist.
- The expression starting with ResERel: The relationships involving resources and events can be between one of the following elements: resource type and event type, resource and event, resource type and resource group, resource type and resource, resource and resource group.
- The expression starting with ResERelrep: The possibility of zero or more repetition(s) of relationships involving resources and events is described by this expression.

VI. EventResult: The following explanations are related to EventResult.

- The expression starting with EventResult: This expression describes the format of an event and its result. One or more events are possible.

- The expression starting with accETG: This expression describes the result of an event, or event type, or event group along with its class name.
- The expression starting with accETGrep: *accETGrep* describes zero or more repetition of accETG as just described.
- The expression starting with result: *result* can be either a *permit* or *deny.*
- The expression starting with accessETG: This expression defines the class name of an event type or event along with the word Access.
- The expression starting with ETClassName: *ETClassName* distinguishes an identifier, previously defined, of an event type.
- The expression starting with EClassName: This expression describes the class name of an event.

3.3.3 Access Control Rule Examples

The following examples are related to a banking situation. Employees in this situation can have possible roles such as tellers, managers, and loan officers and can have full time status. Different working teams are possible in the banking case. Account types can have an attribute of type of currency (e.g., Canadian or US), and examples of accounts are deposit and loan accounts. Modification, deletion, and addition of accounts are some possible events. To summarize, the following syntactical elements exist:

Agent: employee

Agent type or role: teller, manager, loan officer

Agent type attribute: status (full-time can be a value for status)

Agent Group: teamA

Resource/ResourceType: account type, deposit account, loan account

Event/EventType: modification (modify), creation (create), and deletion (delete)

Using these elements and EBNF definitions, three examples of access control rules are provided.

AC Rule Example 1: Tellers or managers are permitted to modify deposit accounts.

The AC rule format perviously defined follows:

ACRule = (AgentExp and ResourceExp and EventExp and AgeEveRel

and ResEveRel) implies EventResult;

Next, each component of the above expression, such as AgentExp, is described. The combinations of these expressions using *and* and *implies* create an access control rule as shown next.

AgentExp: (AgentType = Teller or AgentType = Manager)

and

ResourceExp: (ResourceType = DepositAccount)

and

EventExp: (EventType = Modify)

and

AgeEveRel: (RelATET(Teller, Modify) and RelATET(Manager, Modify))

and

ResEveRel: (RelRTET(DepositAccount, Modify))

implies

EventResult: (Modify Access = permit)

AC Rule Example 2: Tellers can modify but can neither create nor delete deposit accounts.

This rule can be expressed using the general format, shown in Figure 3.22, as follows:

ACRule = (AgentExp and ResourceExp and EventExp and AgeEveRel

and ResEveRel) implies EventResult;

The components of the above expression follow. The combinations of these components using *and* and *implies* create an access control rule, as shown next.

AgentExp: (AgentType = Teller)

and

ResourceExp: (ResourceType = DepositAccount)

and

EventExp: (EventType = Modify or EventType = Create or EventType = Delete)

and

AgeEveRel: (RelATET(Teller, Modify) and not RelATET(Teller, Create) and not RelATET(Teller, Delete))

and

ResEveRel: (RelRTET(DepositAccount, Modify) and not RelRTET(DepositAccount, Create) and not RelRTET(DepositAccount, Delete))

implies

EventResult: (Modify Access = permit and Create Access = deny and Delete Access = deny)

AC Rule Example 3: Loan officers who are full-time and are in TeamA can create or delete loan accounts. This rule can be expressed as

The components of this expression and their combinations are shown next.

AgentExp: (AgentType = LoanOfficer, AgentType.Status = FullTime and AgentGroup = TeamA)

and

ResourceExp: (Resource = LoanAccount)

and

EventExp: (EventType = Create or EventType = Delete)

and

AgeEveRel: (RelATET(LoanOfficer, Create) and RelATET(LoanOfficer, Delete) and RelATG(LoanOfficer, TeamA))

and

ResEveRel: (RelRTET(LoanAccount, Create) and RelRTET(LoanAccount, Delete))

implies

EventResult: (Create Access = permit and Delete Access = permit)

3.3.4 Translation from EBNF to Predicate Logic

The grammar in Figure 3.22 uses "and", "or", and "implies", which become $\wedge$(and), $\vee$(or), and $\rightarrow$(implication), respectively, in predicate logic.

For instance, an expression

(AgentExp and ResourceExp and EventExp and AgeEveRel and ResEveRel)

implies EventResult, (in Figure 3.22) translates in predicate logic as

(AgentExp $\wedge$ ResourceExp $\wedge$ EventExp $\wedge$ AgeEveRel $\wedge$ ResEveExp) $\rightarrow$ EventResult

The term "not" in Figure 3.22, becomes $\neg$ in predicate logic. For instance, an expression such as "not" equA translates to $\neg$equA in predicate logic.

Predicates: The definitions provided in the grammar (Figure 3.22) use the symbols of equalities ("="), greater than (">"), less than ("<"), greater than or equal ("$\geq$"), and less than or equal ("$\leq$"). These symbols translate to the corresponding logical predicates $=$, $>$, $<$, $\geq$, and $\leq$.

For instance, attrName "=" attrValue in this figure translates to the expression using the equality predicate attrName = attrValue in infix notation, such as employmentStatus = FullTime.

The three examples of the previous section are described again.

AC Rule Example 1: Tellers or managers are permitted to modify deposit accounts.

The AC rule format can be shown as

ACRule = (AgentExp $\wedge$ ResourceExp $\wedge$ EventExp $\wedge$ AgeEveRel $\wedge$ ResEveRel)
$\rightarrow$ EventResult

Next, each component of the above expression, such as AgentExp, is described. The combinations of these expressions using *and* and *implies* create an access control rule as shown next.

$\forall$AgentType $\forall$ResourceType $\forall$EventType |

(AgentType = Teller $\vee$ AgentType = Manager) $\hookleftarrow$ AgentExp

$\wedge$

(ResourceType = DepositAccount) $\hookleftarrow$ ResourceExp

$\wedge$

(EventType = Modify) $\hookleftarrow$ EventExp

$\wedge$

(RelATET(Teller, Modify) $\wedge$ RelATET(Manager, Modify)) $\hookleftarrow$ AgeEveRel

$\wedge$

(RelRTET(DepositAccount, Modify)) $\hookleftarrow$ ResEveRel

$\rightarrow$

ModifyAccess(permit) $\hookleftarrow$ EventResult

AC Rule Example 2: Tellers can modify but can neither create nor delete deposit accounts.

This rule can be expressed using the general format, shown in Figure 3.22, as follows:

ACRule = (AgentExp $\wedge$ ResourceExp $\wedge$ EventExp $\wedge$ AgeEveRel $\wedge$ ResEveRel)

$\rightarrow$ EventResult

The components of the above expression follow. The combinations of these components using *and* and *implies* create an access control rule as shown next.

$\forall$AgentType $\forall$ResourceType $\forall$EventType |

(AgentType = Teller) $\hookleftarrow$ AgentExp

$\wedge$

(ResourceType = DepositAccount) $\hookleftarrow$ ResourceExp

$\wedge$

(EventType = Modify $\vee$ EventType = Create $\vee$ EventType = Delete) $\hookleftarrow$ EventExp

$\wedge$

(RelATET(Teller, Modify) $\wedge$ $\neg$RelationATET(Teller, Create) $\wedge$

$\neg$RelationATET(Teller, Delete)) $\hookleftarrow$ AgeEveRel

$\wedge$

(RelRTET(DepositAccount, Modify) $\wedge$ $\neg$RelRTET(DepositAccount, Create) $\wedge$

$\neg$RelRTET(DepositAccount, Delete)) $\hookleftarrow$ ResEveRel

$\rightarrow$

ModifyAccess(permit) $\wedge$ CreateAccess(deny) $\wedge$ DeleteAccess(deny) $\hookleftarrow$ EventResult

AC Rule Example 3: Loan officers who are full-time and are in TeamA can create or delete loan accounts. This rule can be expressed as

The components of this expression and their combination are shown next.

$\forall$AgentType $\forall$AgentType.Status $\forall$AgentGroup $\forall$ResourceType $\forall$EventType |

(AgentType = LoanOfficer $\wedge$ AgentType.Status = FullTime $\wedge$
AgentGroup = TeamA) $\hookleftarrow$ AgentExp

$\wedge$

(ResourceType = LoanAccount) $\hookleftarrow$ ResourceExp

$\wedge$

(EventType = Create $\vee$ EventType = Delete) $\hookleftarrow$ EventExp

$\wedge$

(RelATET(LoanOfficer, Create) $\wedge$ RelATET(LoanOfficer, Delete) $\wedge$ $\hookleftarrow$ AgeEveRel
RelATG(LoanOfficer, TeamA))

$\wedge$

(RelRTET(LoanAccount, Create) $\wedge$ RelRTET(LoanAccount, Delete)) $\hookleftarrow$ ResEveRel

$\rightarrow$

CreateAccess(permit) $\wedge$ DeleteAccess(permit) $\hookleftarrow$ EventResult

3.4 Creating Access Control (AC) Policies from AC Rules

An access control policy is defined as a combination of one or more AC rules. As an AC policy usually consists of several rules, many policy languages describe combining algorithms to provide different strategies for making decisions about this combination. For instance, a combining algorithm may permit a request if a rule in the collection of rules allows such a request, regardless of the existence or not existence of another rule within the collection that denies such a request. Conversely, a combining algorithm may deny a request if one rule denies such a request and another rule within the collection permits the request.

These combining algorithms are usually more detailed as just described. Two such algorithms for combining rules into policies are described in detail in Section 3.4.1.

This section (Section 3.4) presents the combination of AC rules to create AC policies in two steps: first, an informal description of state machines in algorithmic forms is shown to describe this combination (Section 3.4.1); this step corresponds to Box A3 of Figure 1.1. Then, a formal representation of state machines is presented to describe the AC rule combinations (Section 3.4.2); this step corresponds to Box B3 of Figure 1.1.

3.4.1 The Use of Algorithmic Forms

The first-applicable rule-combining algorithm, described earlier in this chapter, follows [79]: the evaluation of rules within a policy is in the same order that rules are listed in a policy. If a rule applies, then the rule's result, i.e., *permit* or *deny*, applies and the evaluation of the rest of the rules halts. Otherwise, the procedure continues to the end. If none of the rules applies, then the result will be *not applicable*.

Figure 3.23 shows an algorithmic form for creating policies from rules for the first-applicable rule-combining algorithm.

The description in this figure uses the notion of states in conjunction with the AC rule definitions provided earlier in this chapter.

An AC rule has the following format, previously described in detail:

(AgentExp $\wedge$ ResourceExp $\wedge$ EventExp $\wedge$ AgeEveRel $\wedge$ ResEveRel) $\rightarrow$ EventResult

Therefore, the *premise* of a rule in this figure includes the following part:

(AgentExp $\wedge$ ResourceExp $\wedge$ EventExp $\wedge$ AgeEveRel $\wedge$ ResEveRel)

The *consequence* of a rule is the EventResult portion.

Another algorithm is permit-unless-deny [78], which can be described as follows: if any decision is deny, then the result will be deny; otherwise, the result will be permit.

Figure 3.24 shows an algorithmic form for the evaluation of rules for the permit-unless-deny rule-combining algorithm. Similarly, this description uses the AC rule definitions, provided earlier, and also includes the notion of states.

3.4.2 The Use of State Machines

The section describes the use of formal state machines to represent algorithmic forms for the combination of rules to create policies. Before showing these state machines, the

```
initial state = state_00;
for i = 1 to n do
    // n = the number of rules in a policy
    if premise-rule_i = false then
        move to state_i0;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            move to state permit;
            exit loop;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state deny;
            exit loop;
        end
    end
    if i = n then
        move to state NA;
    end
end
```

Figure 3.23: Rule Combination using the AC rule definitions for first-applicable algorithm

associated states and their meanings are defined.

State Naming Convention and Meaning: The initial state is q_{00}. With the exception of the initial state, the initial digit(s) of a state name is 1 or greater and indicate(s) the rule number. The last digit indicates whether the assumption of that rule holds (1) or does not hold (0); e.g., q_{11}: the state in which rule 1's assumption holds (i.e., true = 1).

q_{10}: the state in which rule 1's assumption does not hold (i.e., false = 0).

Each rule is described as follows:

(AgentExp $\wedge$ ResourceExp $\wedge$ EventExp $\wedge$ AgeEveRel $\wedge$ ResEveRel) $\rightarrow$ EventResult

This expression can be shorten and written as $p \rightarrow q$, where p is an *assumption* and consists of the following part: (AgentExp $\wedge$ ResourceExp $\wedge$ EventExp $\wedge$ AgeEveRel $\wedge$ ResEveRel). q is a *conclusion* and consists of the EventResult part. The assumptions and conclusions for rules *one* to n can be shown as $p_1 \ldots p_n$, and $q_1 \ldots q_n$, respectively. Each assumption is not necessarily an atomic statement, but atomic statements are usually combined using conjunction, disjunction, or negation; this combination was previously provided

```
initial state = state_00;
for i = 1 to n do
    // n = the number of rules in a policy
    if premise-rule_i = false then
        move to state_i0;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            continue;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state deny;
            exit loop;
        end
    end
    if i = n then
        move to state permit;
    end
end
```

Figure 3.24: Combining rules with the AC rule definitions for permit-unless-deny algorithm

in EBNF definitions. Similarly, each conclusion can include conjunction, disjunction, and negation; the EBNF definitions also show these various possibilities.

Similarly, when the state machine is used to describe policy combinations instead of rule combinations, the word *rule* changes to *policy* in the provided description. For instance, q_{11} and q_{10} for policy-combining state machines have the following meanings:

q_{11}: the state in which policy 1's has the outcome in which the assumption holds (i.e., true = 1).

q_{10}: the state in which policy 1's has the outcome in which the assumption does not hold (i.e., false = 0).

Transition within State Meaning: The transitions between states are governed by examining the **p-rule** and **q-rule**. As mentioned previously, the **p-rule**s on state machines are identical to the assumption part of a rule:

- (AgentExp $\wedge$ ResourceExp $\wedge$ EventExp $\wedge$ AgeEveRel $\wedge$ ResEveRel)

The **q-rule** on state machines is identical to the conclusion part of a rule:

- EventResult

The conclusion of a rule or EvenResult on state machines is shown as follows:

$$[\bigwedge_{i=1}^{m_1} \text{Event}_i\text{Access (permit)} = \text{true}]$$

The superscript of $\bigwedge_{i=1}^{m_1}$ indicates the number of elements in the EventResult expression of a rule. For instance, m_1 is the number of elements in the EventResult of rule 1, and m_2 is the number of elements in the EventResult of rule 2.

Figure 3.25 shows the UML state machine for the first-applicable algorithm.

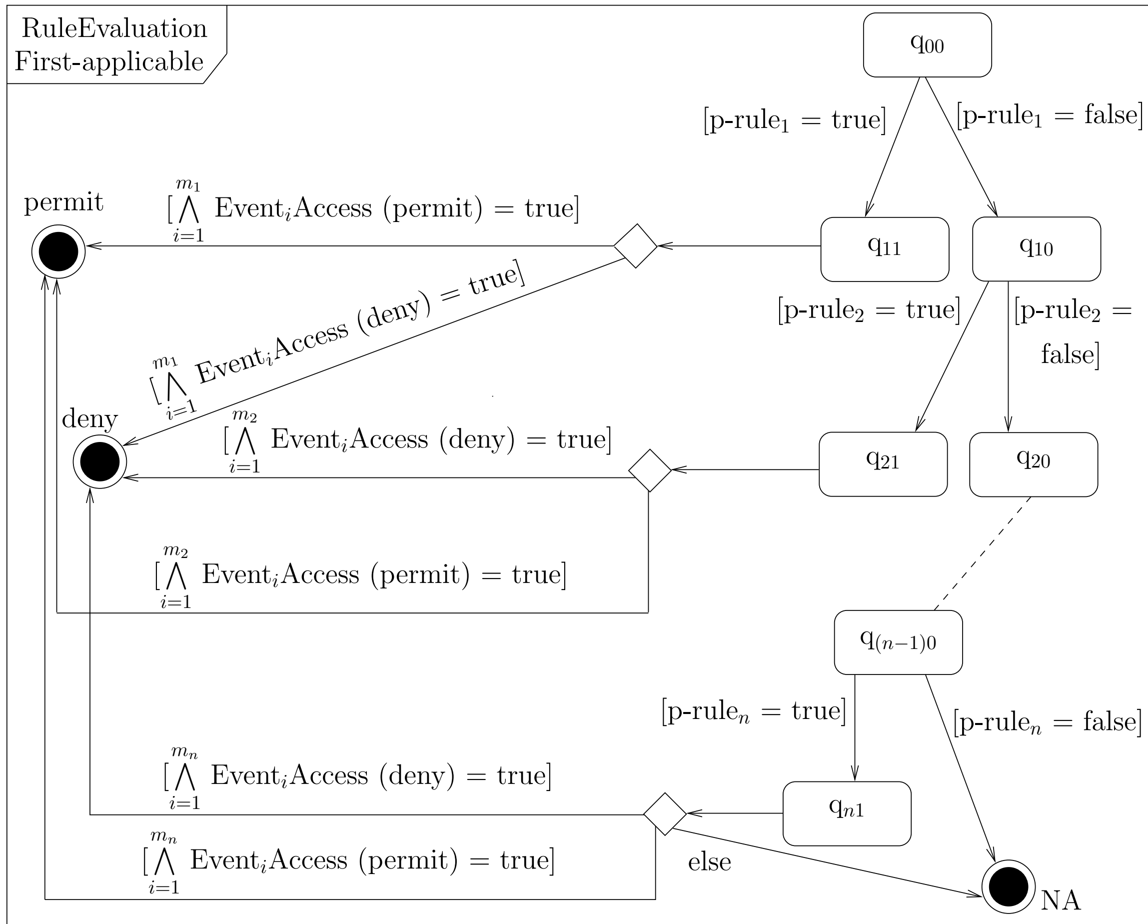


Figure 3.25: A UML state machine using AC rule definitions for first-applicable algorithm

Figure 3.26 shows the UML state machine for the permit-unless-deny rule-combining algorithm.

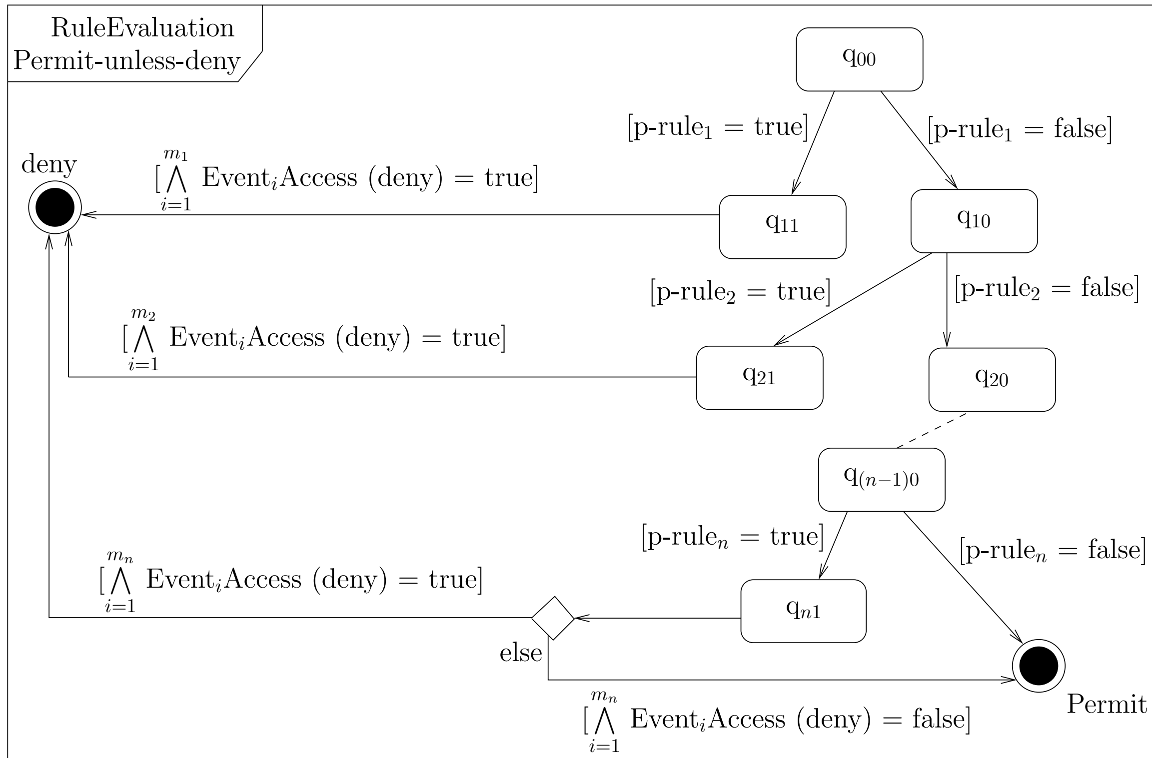


Figure 3.26: A UML state machine that uses the definitions of AC rules for permit-unless-deny algorithm

3.5 Policy-combining Algorithms

Policy-combining algorithms are similar to the rule-combining algorithms, which are described in Section 3.4.

For instance, the first-applicable policy-combining algorithm can be described as follows [79]: the evaluation of policies within a policy set is in the same order that policies are listed in a policy set. If a policy applies and its result is *permit* or *deny*, then the result (i.e., *permit* or *deny*) applies and the evaluation of the rest of the policies halts. If a policy

does not apply or its result is *not applicable*, then the procedure continues. If no other policy exists, then the result will be *not applicable*.

An algorithmic form for describing the first-applicable policy-combining algorithm is shown in Figure 3.27.

```
initial state = state00;
for i = 1 to n do
    // n = the number of policies in a policy set
    if premise-policy_i = false then
        move to state_i0;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            move to state permit;
            exit loop;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state deny;
            exit loop;
        end
    end
    if i = n then
        move to state NA;
    end
end
```

Figure 3.27: An algorithmic description for first-applicable policy-combining algorithm

Figure 3.28 shows a UML state machine that corresponds to the first-applicable policy-combining algorithm.

3.6 An Advantage of the Presented Approach

Many policy languages allow the combination of rules and policies. As Li et al. [66] explain, XACML provides more flexible approaches of these combinations among policy languages, but even XACML cannot express formally several possible combinations. Li et al. [66] provide a few possible approaches, such as weak-consensus, that XACML cannot express formally. According to the description provided in this chapter, this section

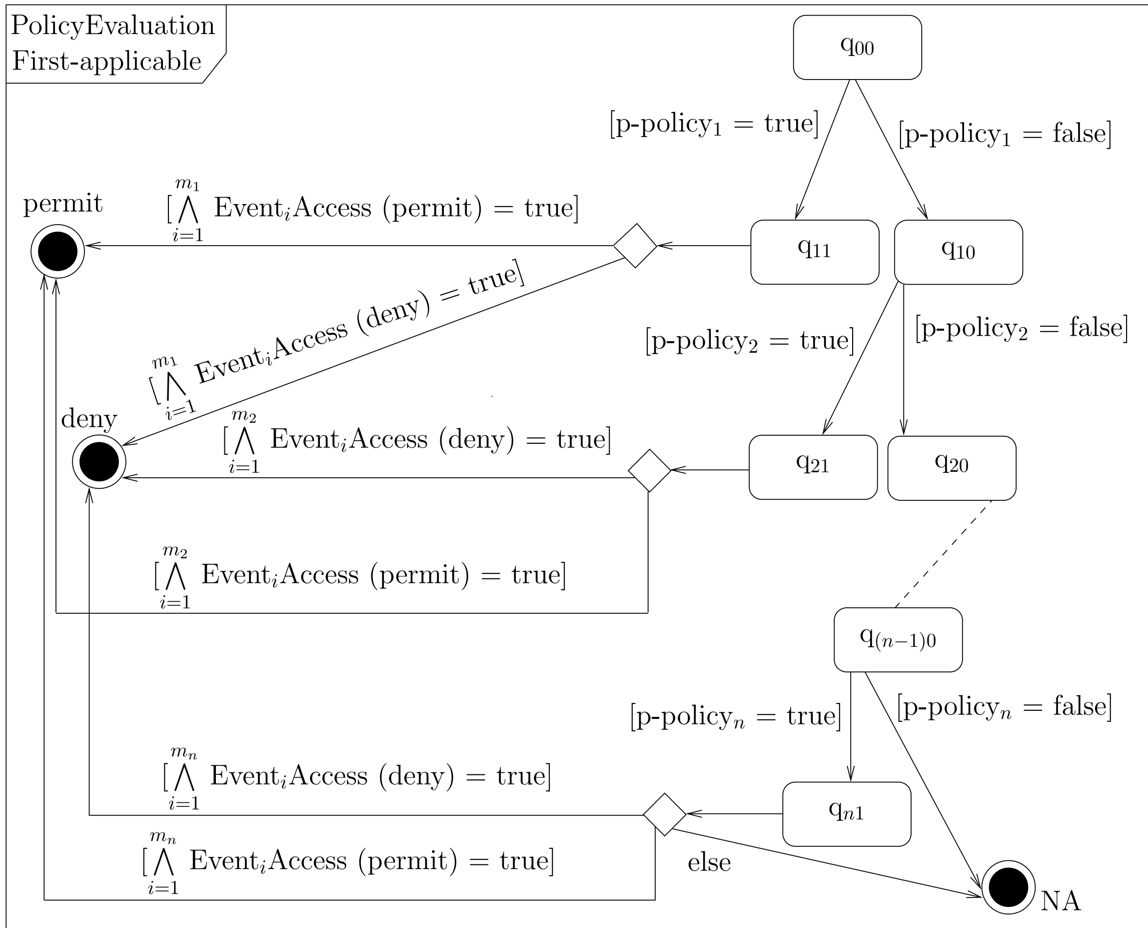


Figure 3.28: A UML state machine for first-applicable policy-combining algorithm

presents weak-consensus and weak-majority policy-combining algorithms. Appendix F shows other possible approaches such as the strong-consensus policy-combining algorithm, strong-majority policy-combining algorithm, and super-majority-permit policy-combining algorithm.

Weak-consensus [66]: "Sub-policies should not conflict with each other: Permit a request if some sub-policies permit a request, and no sub-policy denies it. Deny a request if some sub-policies deny a request, and no sub-policy permits it. Yield a value indicating conflict if some permit and some deny."

Figure 3.29 shows the weak-consensus policy-combining algorithm according to the approach shown in this chapter.

```
initial state = state_00;
set PermitRes to false;
set DenyRes to false;
set ConflictRes to false;
for i = 1 to n do
    // n = the number of policies in a policy set
    if premise-policy_i = false then
        move to state_io ;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            set PermitRes to true;
            move to state permitRes;
        else if Event_i Access(deny) = true for every element of EventResult then
            set DenyRes to true;
            move to state denyRes;
        end
    end
end
if i = n and PermitRes = true and DenyRes = false then
    move to state permit;
else if i = n and DenyRes = true and PermitRes = false then
    move to state deny;
else if i = n and PermitRes = true and DenyRes = true then
    set ConflictRes to true;
    move to state conflict;
end
```

Figure 3.29: The algorithmic form for weak-consensus policy-combining algorithm

Figure 3.30 represents the state machine representation for the weak-consensus policy-combining algorithm that corresponds to the presented algorithmic form.

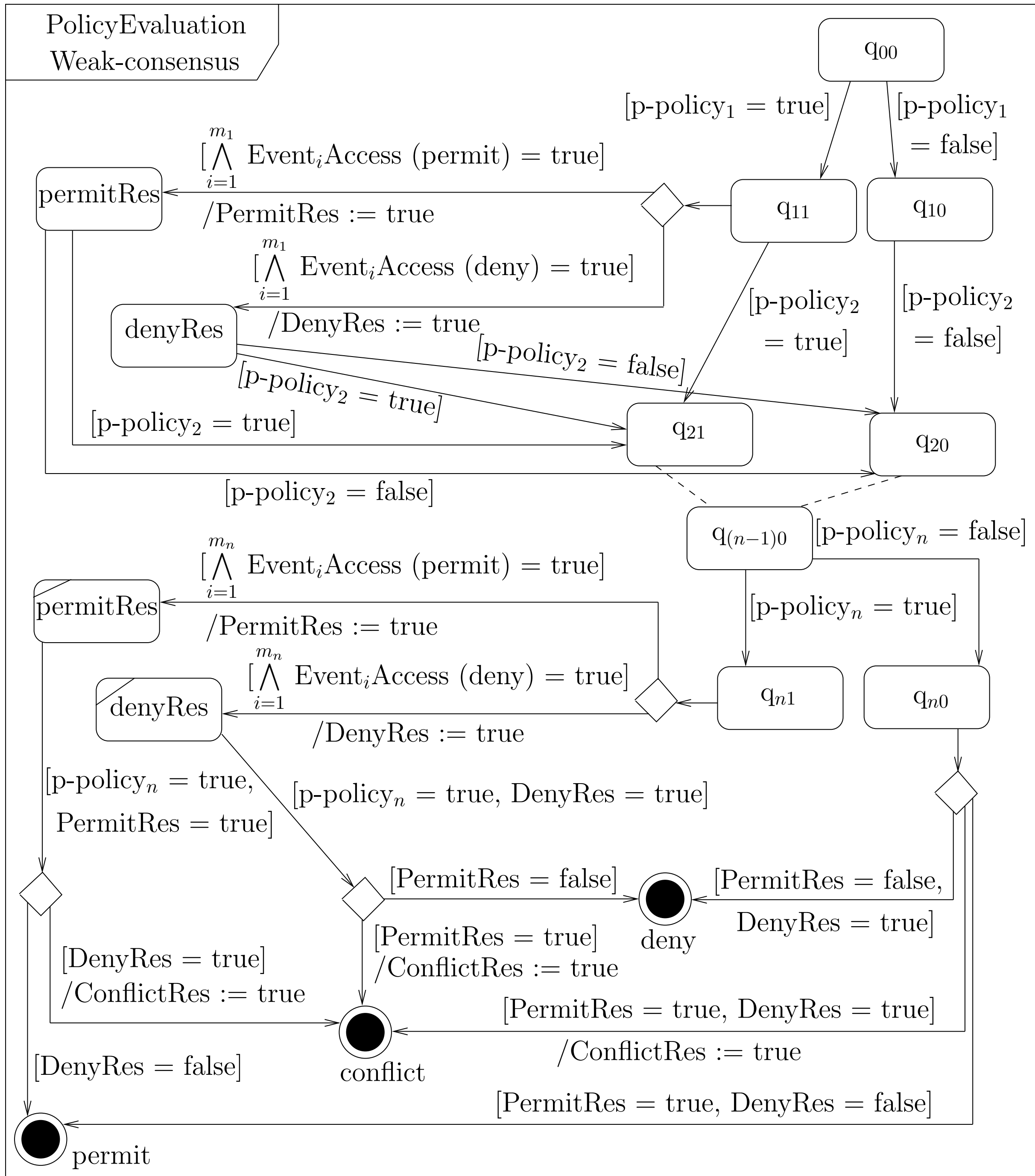


Figure 3.30: A UML state machine representing weak-consensus policy-combining algorithm

Weak-majority[66]: "A decision (permit or deny) wins if it has more votes than the opposite. Permit (deny, resp.) a request if the number of sub-policies permitting (denying, resp.) the request is greater than the number of sub-policies denying (permitting, resp.)."

Figure 3.31 shows the weak-majority policy-combining algorithm, and the corresponding state machine is presented in Figure 3.32.

```
initial state = state_00;
set NumPermit to zero;
set NumDeny to zero;
for i = 1 to n do
    // n = the number of policies in a policy set
    if premise-policy_i = false then
        move to state_io ;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            add one to NumPermit;
            move to state permitRes;
        else if Event_i Access(deny) = true for every element of EventResult then
            add one to NumDeny;
            move to state denyRes;
        end
    end
end
if i = n and NumPermit > NumDeny then
    move to state permit;
else if i = n and NumDeny > NumPermit then
    move to state deny;
end
```

Figure 3.31: The algorithmic form for weak-majority policy-combining algorithm

As mentioned previously, Appendix F provide the algorithmic forms and state machines of the strong-consensus policy-combining algorithm, strong-majority policy-combining algorithm, and super-majority-permit policy-combining algorithm.

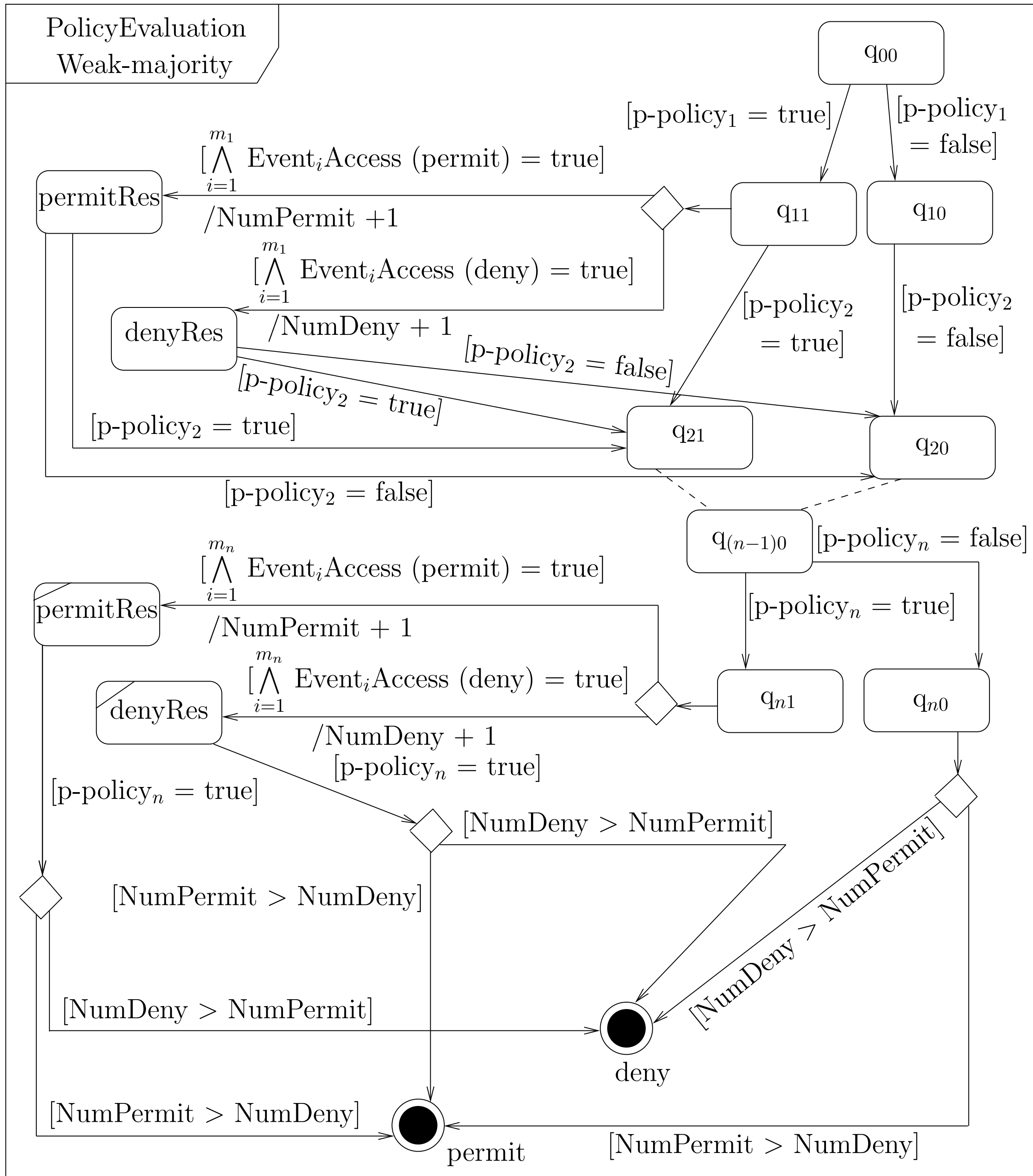


Figure 3.32: A UML state machine for weak-majority policy-combining algorithm

Chapter 4

Specification of Properties for Access Control and Categorization

Summary: The previous chapter, Chapter 3, described the common representation model, which is based on REA notations, REA patterns, and state machines. In general, properties are specified and checked to determine whether these properties are satisfied in a model.

In this chapter, properties for access control (AC) are specified and possible categories of properties, using the AC rule definitions provided in Chapter 3, are identified. The benefits of categorization have been acknowledged in many fields. Categorization provides organization of understanding and can enable the discovery of missing categories and elements. A main advantage of this categorization is the possibility of reusing it.

This chapter also provides a general form of AC property specification that can also represent the AC properties related to categorization.

Section 4.1 provides a short background on Linear Temporal Logic (LTL), used here to specify AC properties. Section 4.2 explains AC property specification for access control and categories using the AC rules previously defined; this section describes Box B1 of Figure 1.1. Section 4.3 explains a general form of property specification and describes Box B2 of Figure 1.1. Section 4.4 explains some related work on property specifications.

4.1 Background

Linear Temporal Logic (LTL) is used in this work and is briefly reviewed next. Appendix C provides a more detailed description of LTL.

LTL: The syntax of LTL follows [49]:

$$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi)$$
$$\mid (\Box\,\phi) \mid (\Diamond\,\phi) \mid (X\,\phi) \mid (\phi\ \mathrm{U}\ \phi) \mid (\phi\ \mathrm{W}\ \phi) \mid \phi\ \mathrm{R}\ \phi$$

p is a propositional atom. The temporal operators $\Box$, $\Diamond$, X, U, R, W mean "always," "eventually," "neXt," "Until," "Release," and "Weak-until," respectively. Alternative notations for temporal operators $\Box$ and $\Diamond$ are G and F, respectively. Appendix C describes the semantics of linear temporal logic, and some proof examples using the semantics of LTL, in addition to a short description of other forms of logic.

Convention: Similar to the common convention on binding priorities (e.g., [49]), the following precedence applies: the unary connectives that include negation ($\neg$), always or Globally (G or $\Box$), eventually or some Future states (F or $\Diamond$), and neXt (X or $\bigcirc$) bind most tightly. Next, in order, come Until (U or $\mathcal{U}$), Release (R or $\mathcal{R}$), Weak-until (W or $\mathcal{W}$), followed by the logical and ($\wedge$), and or ($\vee$), and finally, the implication ($\rightarrow$).

4.2 AC Property Specification and Categories

This section specifies and categorizes properties based on the components of AC rules, defined in EBNF in Chapter 3. Properties are specified in LTL, which is a propositional temporal logic. First, Figures 4.1 and 4.2 provide the general idea for the translation from the EBNF elements and its predicates, such as equalities and relationships, to propositions. The propositional versions are identified with a subscript of *prop* in these figures. Then, PROMELA, the language of SPIN, is used to describe this translation at the code level. Figure 4.1 shows three elements AGT, RGT, and ETG, defined in EBNF in Chapter 3. Their propositional versions are identified as AGT_{prop}, RGT_{prop}, and EGT_{prop}, respectively.

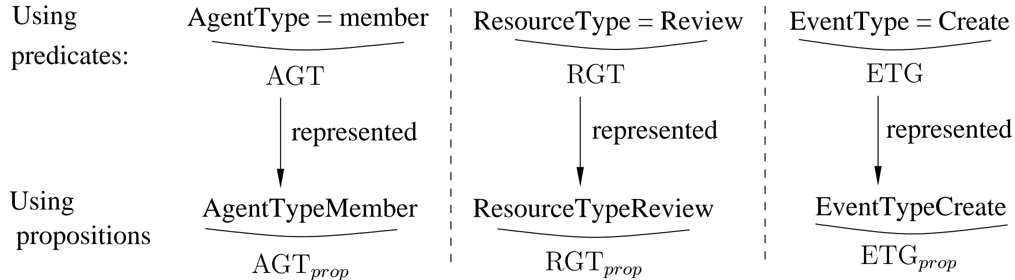


Figure 4.1: Predicate and propositional versions of expressions

The propositional elements of Figure 4.1 can be described by the language of a model checker. For instance, if PROMELA is used, then the propositional versions in this figure (i.e., AgentTypeMember, ResourceTypeReview, and EventTypeCreate) can be defined as follows. In the following expressions, the symbol "==" means equal. Appendix D provides an overview of PROMELA.

```
#define AgentTypeMember (AgentType == Member)
#define ResourceTypeReview (ResourceType == Review)
#define EventTypeCreate (EventType == Create)
```

Similarly, predicates, such as the ones that represent relationships, can be described as one unit–propositions–as shown in Figure 4.2.

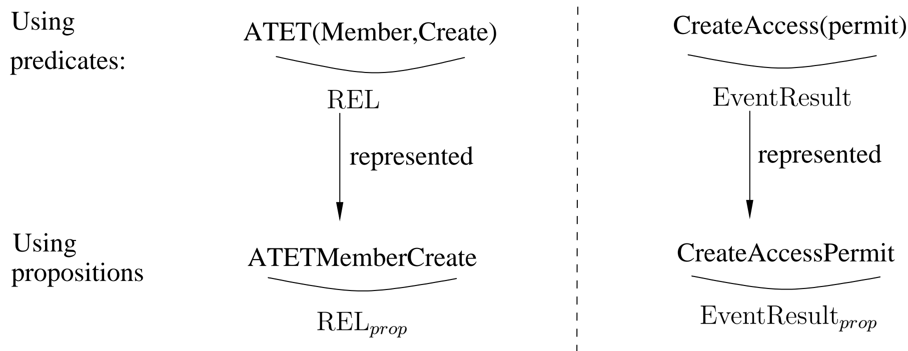


Figure 4.2: Propositional versions of predicates

The predicates that can represent existing relationships, such as between agent types and event types ATET(Member, Create), can be represented by the language of a model checker. For instance, using PROMELA (i.e., the language of SPIN), this relationship can be expressed using an array called RelA, which is defined by the keyword *typedef*. The elements of this array are AgeN and Act, which can hold information about agent types and event types.

```
typedef RelA {
        byte AgeN;
        byte Act;
        }
```

Then, a specific array of the required size can be defined; e.g., RelA memR[1];

For instance, the elements of this specific array can hold information about a relationship between an agent type of member and an event type of *create* as follows. In the following expressions, the symbol "=" is for assignment.

memR[0].AgeN = member; memR[0].Act = create;

Similar to the previously provided description to represent predicates by propositions, the propositional version of the relationship between an agent type of member and an event type of *create* can be represented using PROMELA's keyword *define*, as shown next. ETMemberCreate is the propositional representation of the information within the array just described.

#define ETMemberCreate (memR[0].AT == member && memR[0].ET == create)

The predicates, such as CreateAccess(permit) or its equivalent representation in EBNF form, *Create Access = permit*, can be defined as CreateAccess == Permit. The later expression can be defined as a proposition called CreateAccesPermit, as shown next.

#define CreateAccessPermit (CreateAccess == Permit)

Based on this explanation about the propositional and predicate versions of expressions, the categories of properties are described next.

Definition 4.1. Property Description: Properties are categorized into four high-level categories that are later divided into subcategories:

1. This category holds any primitives of agents, events, and resources individually, and in connection with AC results. The base form of this category can have one of the following forms:

    a) $\text{AGT}_{prop} \wedge \text{EventResult}_{prop}$

    b) $\text{RGT}_{prop} \wedge \text{EventResult}_{prop}$

    c) $\text{EGT}_{prop} \wedge \text{EventResult}_{prop}$

    AGT_{prop} and the rest of these terms are propositional expressions corresponding to previously defined terms as follows:

    AGT_{prop}: propositional expression of AGT, defined in Chapter 3

    $\text{EventResult}_{prop}$: propositional expression of EventResult, defined in Chapter 3

    RTG_{prop}: propositional expression of RTG, defined in Chapter 3

    ETG_{prop}: propositional expression of ETG, defined in Chapter 3

2. This category holds any primitives of agents, events, resources, and their attributes individually, and in connection with AC results. The base form of this category can have one of the following forms:

   a) $\text{AgentExp}_{prop} \land \text{EventResult}_{prop}$
   b) $\text{ResourceExp}_{prop} \land \text{EventResult}_{prop}$
   c) $\text{EventExp}_{prop} \land \text{EventResult}_{prop}$

   AgentExsp_{prop} and the rest of these terms are propositional expressions corresponding to previously defined terms as follows:

   AgentExp_{prop}: propositional expression of AgentExp, defined in Chapter 3

   $\text{EventResult}_{prop}$: propositional expression of EventResult, defined in Chapter 3

   $\text{ResourceExp}_{prop}$: propositional expression of ResourceExp, defined in Chapter 3

   EventExp_{prop}: propositional expression of EventExp, defined in Chapter 3

3. This category holds a combination of agents, events, resources, and their relationships, and in connection with AC results. The base form of this property is

   $$\Box(\text{AGT}_{prop} \land \text{RGT}_{prop} \land \text{ETG}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow X\Diamond\text{EventResult}_{prop})$$

   AgeEveRel_{prop}: propositional expression of AgeEveRel defined in Chapter 3

   ResEveExp_{prop}: propositional expression of ResEveExp defined in Chapter 3

4. This category holds a combination of agents, events, resources, their attributes, and their relationships, and in connection with AC results. The base form of this property follows:

   $$\Box(\text{AgentExp}_{prop} \land \text{ResourceExp}_{prop} \land \text{EventExp}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow X\Diamond\text{EventResult}_{prop})$$

The type and group of entities (e.g., as shown in Chapter 3), such as a resource type or group, also apply whenever agent, event, and resource entities are used. For brevity, their types and groups are not mentioned in the prose description of these categories.

Each category is further divided into five subcategories, and each subcategory has three variations. The subcategories and their variations use the specification patterns and their LTL specifications as suggested by Dwyer et al. [28, 27, 62].

The next four subsections describe these four categories and provide detailed examples for property specification for each category.

4.2.1 Any Primitives of Agents, Events, and Resources Individually, and in Connection with AC Results

As mentioned in Section 4.2, the base form of this category can be one of the following:

a) $\text{AGT}_{prop} \land \text{EventResult}_{prop}$

b) $\text{RGT}_{prop} \land \text{EventResult}_{prop}$

c) $\text{EGT}_{prop} \land \text{EventResult}_{prop}$

The base forms, in addition to temporal and logical operators, are used to create category 1, its subcategories, and variations of subcategories.

An example of the first base form ($\text{AGT}_{prop} \land \text{EventResult}_{prop}$) is *a state cannot be reached in which a teller has a permission of create*. This expression can be presented as a general form of $\square(\neg(\text{AGT}_{prop} \land \text{EventResult}_{prop}))$. *A teller* is a specific example of the term AGT_{prop}, and *a permission to create* is a specific instance of the term $\text{EventResult}_{prop}$.

An example of the second base form (i.e., $\text{RGT}_{prop} \land \text{EventResult}_{prop}$) is as follows: *it is impossible to reach a state in which a loan account is deleted*. This expression can be presented as a general form of $\square(\neg(\text{RGT}_{prop} \land \text{EventResult}_{prop}))$. *A loan account* is a specific example of the term RGT_{prop}, and *a permission to delete* is a specific instance of the term $\text{EventResult}_{prop}$.

An example of the third base form (i.e., $\text{EGT}_{prop} \land \text{EventResult}_{prop}$) is *it is always the case that a deposit is allowed*. A general form of this expression can be $\square(\text{EGT}_{prop} \land \text{EventResult}_{prop})$. *A deposit* is a specific example of EGT_{prop}, and *the deposit action is permitted* is an instance of $\text{EventResult}_{prop}$).

Using the first base form ($\text{AGT}_{prop} \land \text{EventResult}_{prop}$), Figure 4.3 shows category 1, its sub-categories, and the variations of each sub-category.

Two other variations of this figure are possible simply by replacing the first base form ($\text{AGT}_{prop} \land \text{EventResult}_{prop}$) with the second base form ($\text{RGT}_{prop} \land \text{EventResult}_{prop}$) or the third base form ($\text{EGT}_{prop} \land \text{EventResult}_{prop}$).

Category 1: The base form of this category follows: ($AGT_{prop} \wedge EventResult_{prop}$). The five subcategories, in which p, q, r, and s have this category's base form, are listed below.

1. **absence sub-category:** p, q, and r have the base form $AGT_{prop} \wedge EventResult_{prop}$.
   a) absence globally (p is false globally): $\Box\neg(AGT_{prop} \wedge EventResult_{prop})$
   b) absence before r (p is false before r): $\Diamond r \rightarrow (\neg p \; U \; r)$
   c) absence after q (p is false after q): $\Box(q \rightarrow \Box(\neg p))$
2. **universality sub-category:**
   a) universality globally (p is true globally): $\Box(AGT_{prop} \wedge EventResult_{prop})$
   b) universality before r (p is true before r): $\Diamond r \rightarrow (p \; U \; r)$
   c) universality after q (p is true after q): $\Box(q \rightarrow \Box p)$
3. **existence sub-category:**
   a) existence (p becomes true): $\Diamond(AGT_{prop} \wedge EventResult_{prop})$
   b) existence before r (p becomes true before r): $\neg r \; W \; (p \wedge \neg r)$
   c) existence after q (p becomes true after q): $\Box((\neg q) \vee \Diamond(q \wedge \Diamond p)$
4. **response sub-category:**
   a) globally (s responds to p globally): $\Box((AGT_{prop} \wedge EventResult_{prop}) \rightarrow \Diamond s)$
   b) before r (s responds to p before r): $\Diamond r \rightarrow (p \rightarrow (\neg r \; U \; (s \wedge \neg r))) \; U \; r$
   c) after q (s responds to p after q): $\Box(q \rightarrow \Box(p \rightarrow \Diamond s))$
5. **precedence sub-category:**
   a) globally (s precedes p globally): $\neg(AGT_{prop} \wedge EventResult_{prop}) \; W \; s$
   b) before r (s precedes p before r): $\Diamond r \rightarrow (\neg p \; U \; (s \vee r))$
   c) after q (s precedes p after q): $\Box\neg q \vee \Diamond(q \wedge (\neg p \; W \; s))$

Figure 4.3: Category 1, sub-categories, and sub-category variations

4.2.2 Any Primitives of Agents, Events, and Resources Individually, and their Attributes, and in Connection with AC Results

These properties are described by any primitives of agents, events, and resources individually, and their attributes, and in connection with AC results. The description of attributes can use one or both of the following: a) values of attributes or constraints on these values, b) derived attribute values.

The base form of this category can take one of the following forms:

a) $\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop}$

b) $\text{ResourceExp}_{prop} \wedge \text{EventResult}_{prop}$

c) $\text{EventExp}_{prop} \wedge \text{EventResult}_{prop}$

An example of the first base form is the following: *a state cannot be reached in which a junior teller is permitted to create (e.g., an account).* This expression can be described as $\Box(\neg(\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop}))$, in which AgentExp_{prop} represents a *junior teller* and $\text{EventResult}_{prop}$ can describe an access of permit to create.

An example of the second base form (i.e., $\text{ResourceExp}_{prop} \wedge \text{EventResult}_{prop}$) is *it is impossible to reach a state in which a savings account with an opening balance of less than 100 is permitted to be created*. This expression can be presented as a general form of $\Box(\neg(\text{ResourceExp}_{prop} \wedge \text{EventResult}_{prop}))$. *A savings account with an opening balance of 100* is a specific example of the term $\text{ResourceExp}_{prop}$, and *a permission to create* is a specific instance of the term $\text{EventResult}_{prop}$.

An example using the third base form (i.e., $\text{EventExp}_{prop} \wedge \text{EventResult}_{prop}$) is *it is impossible to allow an invalid (e.g., larger than a designated amount) withdrawal*. A general form of this expression would be $\Box(\neg(\text{EventExp}_{prop} \wedge \text{EventResult}_{prop}))$. *An invalid withdrawal* is a specific example of EventExp_{prop}, and *the withdraw action is permitted* is an instance of $\text{EventResult}_{prop}$).

Using the first base form ($\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop}$), Figure 4.4 shows category 2, its sub-categories, and the variations of each sub-category.

Two other variations of this figure are possible simply by replacing the first base form ($\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop}$) with the second base form ($\text{ResourceExp}_{prop} \wedge \text{EventResult}_{prop}$) or the third base form ($\text{EventExp}_{prop} \wedge \text{EventResult}_{prop}$).

Category 2: The base form of this category follows: ($\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop}$). The five subcategories, in which p, q, r, and s have this category's base form, are listed below.

1. **absence sub-category:**

   a) absence globally (p is false globally): $\Box\neg(\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop})$

   b) and c) see Note 1 below

2. **universality sub-category:**

   a) universality globally (p is true globally): $\Box(\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop})$

   b) and c) see Note 1 below

3. **existence sub-category:**

   a) existence (p becomes true): $\Diamond(\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop})$

   b) and c) see Note 1 below

   Note 1: b) and c) in all the above three items are similar to the corresponding LTL expressions in Figure 4.3, but p, r, and q are in the base form of this category.

4. **response sub-category:**

   a) globally (s responds to p globally): $\Box((\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop}) \rightarrow \Diamond \text{s})$

   b) and c) see Note 2 below

5. **precedence sub-category:**

   a) globally (s precedes p globally): $\neg(\text{AgentExp}_{prop} \wedge \text{EventResult}_{prop})\ \text{W s}$

   b) and c) see note 2 below

   Note 2: b) and c) in the above two items are similar to the corresponding LTL expressions in Figure 4.3, but p, r, q, and s are in the base form of this category.

Figure 4.4: Category 2, sub-categories, and sub-category variations

4.2.3 A Combination of Agents, Events, Resources, and their Relationships, and in Connection with AC Results

The base form of this property uses temporal implication as described in Figure 4.5. This figure explains the reasons for using various temporal operators, as are used in the expression. As a result of this use, the base form of this property follows:

$$\Box(\text{AGT}_{prop} \land \text{RGT}_{prop} \land \text{ETG}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow \text{X}\Diamond\text{EventResult}_{prop})$$

AGT_{prop}: propositional expression of AGT, defined in Chapter 3

RTG_{prop}: propositional expression of RTG, defined in Chapter 3

ETG_{prop}: propositional expression of ETG, defined in Chapter 3

AgeEveRel_{prop}: propositional expression of AgeEveRel, defined in Chapter 3

ResEveExp_{prop}: propositional expression of ResEveExp, defined in Chapter 3

$\text{EventResult}_{prop}$: propositional expression of EventResult, defined in Chapter 3

To describe p implies q, one can write a simple expression, $p \rightarrow q$

As $p \rightarrow q$ is equivalent to $(!p) \lor q$, this implication holds, for instance, in the first state of a run in which p is false or q is true. In order to make the implication true within each step of a run, the implication must be written as $\Box(p \rightarrow q)$

This expression is still not correct because it has no notion of temporal implication (e.g., the evaluation of a rule reaches a result at some point.). This expression must be written as $\Box(p \rightarrow \Diamond q)$

The latest expression still holds in a case in which p and q hold at the same state. To describe that q holds in the next state, one can change the description to

$$\Box(p \rightarrow X\Diamond q)$$

Finally, the last expression holds if p never becomes true because of the implication ($\rightarrow$). The addition of $\land \Diamond p$ at the end of the previous expression ensures that p is expected to hold at some point of time. This addition prevents the expression from being vacuously true. The final description is $\Box(p \rightarrow X\Diamond q) \land \Diamond p$

Figure 4.5: Temporal implications [47]

Figure 4.6 shows category 3, its subcategories, and their variations.

Category 3: The base form of this category follows:

$$\Box(\text{AGT}_{prop} \land \text{RGT}_{prop} \land \text{ETG}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow \text{X}\Diamond\text{EventResult}_{prop})$$

Note 1: Subcategories 1, 2, and 3, below, have variations b) and c) that are similar to the corresponding LTL expressions in Figure 4.3, but p, r, and q have the above base form.

Note 2: Subcategories 4 and 5, below, have variations b) and c) that are both similar to the corresponding LTL expressions in Figure 4.3, but p, r, q, and s are in this category's base form.

1. **absence sub-category:**

   a) absence globally (p is false globally): $\Box\neg(\Box(\text{AGT}_{prop} \land \text{RGT}_{prop} \land \text{ETG}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow \text{X}\Diamond\text{EventResult}_{prop}))$

2. **universality sub-category:**

   a) universality globally (p is true globally): $\Box(\text{AGT}_{prop} \land \text{RGT}_{prop} \land \text{ETG}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow \text{X}\Diamond\text{EventResult}_{prop})$

3. **existence sub-category:**

   a) existence (p becomes true): $\Diamond(\Box(\text{AGT}_{prop} \land \text{RGT}_{prop} \land \text{ETG}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow \text{X}\Diamond\text{EventResult}_{prop}))$

4. **response sub-category:**

   a) globally (s responds to p globally): $\Box(\Box(\text{AGT}_{prop} \land \text{RGT}_{prop} \land \text{ETG}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow \text{X}\Diamond\text{EventResult}_{prop}) \rightarrow \Diamond\text{s})$

5. **precedence sub-category:**

   a) globally (s precedes p globally): $\neg(\Box(\text{AGT}_{prop} \land \text{RGT}_{prop} \land \text{ETG}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow \text{X}\Diamond\text{EventResult}_{prop}))\ \text{W s}$

Figure 4.6: Category 3, sub-categories, and sub-category variations

4.2.4 A Combination of Agents, Events, Resources, their Attributes, and their Relationships, and in Connection with AC Results

Similar to and based on the description provided in Figure 4.5, the base form of this property follows:

$$\Box(\text{AgentExp}_{prop} \land \text{ResourceExp}_{prop} \land \text{EventExp}_{prop} \land \text{AgeEveRel}_{prop} \land \text{ResEveExp}_{prop} \rightarrow X\Diamond\text{EventResult}_{prop})$$

AgentExp_{prop}: propositional statement of AgentExp, previously defined in Chapter 3.

$\text{ResourceExp}_{prop}$: propositional expression of ResourceExp, defined in Chapter 3

EventExp_{prop}: propositional expression of EventExp, defined in Chapter 3

AgeEveRel_{prop}: propositional expression of AgeEveRel defined in Chapter 3

ResEveExp_{prop}: propositional expression of ResEveExp defined in Chapter 3

$\text{EventResult}_{prop}$: propositional expression of EventResult, defined in Chapter 3

Figure 4.7 shows category 4, its subcategories, and their variations.

4.3 General Form of AC Property Specification

This section provides a general form of AC property specification. This general form uses both the AC rules presented in Chapter 3 and the categorization specification described in Section 4.2.

Figure 4.8 shows a general form of AC property. This definition includes the temporal operators of LTL and the connective of propositional logic. An atomic proposition is represented as "p", and the elements identified as AgentExp_{prop}, $\text{ResourceExp}_{prop}$, EventExp_{prop}, AgeEveRel_{prop}, ResEveExp_{prop}, $\text{EventResult}_{prop}$, AGT_{prop}, RGT_{prop}, and ETG_{prop} are propositions equivalent to the previously defined (Chapter 3, Figure 3.22) AgentExp, ResourceExp, EventExp, AgeEveRel, ResEveExp, EventResult, AGT, RGT, and ETG, respectively.

Next, an example is provided to describe the general form of AC property specification in the context of specific cases.

Example: It is always the case that a program committee (PC) member who owns reviews can eventually edit his/her reviews:

Category 4: The base form of this category follows:

$$\Box(\text{AgentExp}_{prop} \wedge \text{ResourceExp}_{prop} \wedge \text{EventExp}_{prop} \wedge \text{AgeEveRel}_{prop} \wedge \text{ResEveExp}_{prop} \rightarrow X\Diamond\text{EventResult}_{prop})$$

Note 1: Subcategories 1, 2, and 3, below, have variations b) and c) that are similar to the corresponding LTL expressions in Figure 4.3, but p, r, and q have the above base form.

Note 2: Subcategories 4 and 5, below, have variations b) and c) that are both similar to the corresponding LTL expressions in Figure 4.3, but p, r, q, and s are in this category's base form.

1. **absence sub-category:**

   a) absence globally (p is false globally): $\Box\neg(\Box(\text{AgentExp}_{prop} \wedge \text{ResourceExp}_{prop} \wedge \text{EventExp}_{prop} \wedge \text{AgeEveRel}_{prop} \wedge \text{ResEveExp}_{prop} \rightarrow X\Diamond\text{EventResult}_{prop}))$

2. **universality sub-category:**

   a) universality globally (p is true globally): $\Box(\text{AgentExp}_{prop} \wedge \text{ResourceExp}_{prop} \wedge \text{EventExp}_{prop} \wedge \text{AgeEveRel}_{prop} \wedge \text{ResEveExp}_{prop} \rightarrow X\Diamond\text{EventResult}_{prop})$

3. **existence sub-category:**

   a) existence (p becomes true): $\Diamond(\Box(\text{AgentExsp}_{prop} \wedge \text{ResourceExp}_{prop} \wedge \text{EventExp}_{prop} \wedge \text{AgeEveRel}_{prop} \wedge \text{ResEveExp}_{prop} \rightarrow X\Diamond\text{EventResult}_{prop}))$

4. **response sub-category:**

   a) globally (s responds to p globally): $\Box(\Box(\text{AgentExp}_{prop} \wedge \text{ResourceExp}_{prop} \wedge \text{EventExp}_{prop} \wedge \text{AgeEveRel}_{prop} \wedge \text{ResEveExp}_{prop} \rightarrow X\Diamond\text{EventResult}_{prop}) \rightarrow \Diamond s)$

5. **precedence sub-category:**

   a) globally (s precedes p globally): $\neg(\Box(\text{AgentExp}_{prop} \wedge \text{ResourceExp}_{prop} \wedge \text{EventExp}_{prop} \wedge \text{AgeEveRel}_{prop} \wedge \text{ResEveExp}_{prop} \rightarrow X\Diamond\text{EventResult}_{prop}))\ W\ s$

Figure 4.7: Category 4, sub-categories, and sub-category variations

P = uop (P) | (P bop P) | ultl (P) | (P bltl P) | AgentExp_{prop} | $\text{ResourceExp}_{prop}$ | EventExp_{prop} | AgeEveRel_{prop} | ResEveExp_{prop} | $\text{EventResult}_{prop}$ | ATG_{prop} | RTG_{prop} | ETG_{prop};
uop = "not";
bop = "and" | "or" | "implies";
ultl = "always" | "eventually" | "next";
bltl = "until" | "release" | "weak until";

Figure 4.8: A general form of AC Property

$$\Box(\text{pcMember} \wedge \text{ownedReview} \wedge \text{eventIsUpdate} \wedge \text{pcMemberUpdateRel} \wedge \text{UpdateReviewRel} \rightarrow X\Diamond\text{permitUpdate})$$

This property can be described using the general AC property as follows:

Note: The extra parentheses are removed in the following expressions, using the convention provided in section 4.1.

- pcMember corresponds to AgentExp_{prop} of the general property definition.
- ownedReview corresponds to $\text{ResourceExp}_{prop}$ of the general property definition.
- The use of option (P bop P) creates (AgentExp_{prop} "and" $\text{ResourceExp}_{prop}$), where bop is the option "and" ($\wedge$); the new P is now (AgentExp_{prop} "and" $\text{ResourceExp}_{prop}$).
- eventIsUpdate corresponds to EventExp_{prop}.
- Another use of option (P bop P) where (AgentExp_{prop} "and" $\text{ResourceExp}_{prop}$) is the first P, and the second P is eventIsUpdate, and bop is "and" creates the new P expression (AgentExp_{prop} "and" $\text{ResourceExp}_{prop}$ "and" eventIsUpdate)
- pcMemberUpdateRel corresponds to AgeEveRel_{prop} in the general form.
- Another use of option (P bop P) where the first P is

  (AgentExp_{prop} "and" $\text{ResourceExp}_{prop}$ "and" eventIsUpdate), and the second P is pcMemberUpdateRel, and bop is "and" creates the new P expression

  (AgentExp_{prop} "and" $\text{ResourceExp}_{prop}$ "and" eventIsUpdate "and" pcMemberUpdateRel)

- UpdateReviewRel corresponds to ResEveExp_{prop}. Similar to the explanation of the previous item, another use of (P bop P) creates the new P, called A, as follows:

  (AgentExp_{prop} “and” $\text{ResourceExp}_{prop}$ “and” eventIsUpdate “and” pcMemberUpdateRel “and” UpdateReviewRel)

- permitUpdate corresponds to $\text{EventResult}_{prop}$. The use of ultl (P), where P is prmitUpdate, and ultl is “next”, creates “next”(permitUpdate). Another application of ultl (P), where P is this latest expression, and ultl is “eventually”, creates an expression, called B, as follows: “eventually”“next”(permitUpdate)

- Another use of option (P bop P), where the first P is what is called A, and the second P is what is called B, and bop is “implies”, creates the following expression:

  (AgentExp_{prop} “and” $\text{ResourceExp}_{prop}$ “and” eventIsUpdate “and” pcMemberUpdateRel “and” UpdateReviewRel “implies” “eventually”“next”permitUpdate)

- Finally, the use of ultl (P), where P is the last expression, and ultl is “always”, creates the expression

  “always”(AgentExp_{prop} “and” $\text{ResourceExp}_{prop}$ “and” eventIsUpdate “and” pcMemberUpdateRel “and” UpdateReviewRel “implies” “eventually”“next”permitUpdate)

  Or

  $$\square(\text{pcMember} \wedge \text{ownedReview} \wedge \text{eventIsUpdate} \wedge \text{pcMemberUpdateRel} \wedge \text{UpdateReviewRel} \rightarrow \text{X}\Diamond\text{permitUpdate})$$

4.4 Related Work on Property Specifications

The Object Management Group’s (OMG) “Semantics of Business Vocabulary and Business Rules (SBVR)” [101] uses predicate logic with a small extension of modal logic. The SBVR document uses limited modal logic operators without committing to any particular modal logic. SBVR defines a rule as a “proposition that is a claim of obligation or of necessity,” and a business rule is expressed as a rule under a business jurisdiction. SBVR classifies business rules as structural (or definitional) or operative. A definitional rule is a statement that represents a claim of necessity, whereas an operative rule is a claim of obligation. In general, a special case of modal logic is temporal logic [49]. In temporal logic, a statement can be true at some point in time but false at others. Similarly, modal logic labels statements as “true by necessity” and “true by possibility,” and therefore,

statements that are true now and those that are always true are differentiated [108]. Both modal and temporal logic creates various worlds by interpretations that can correspond to each other. Woodcock and Loomes provide a comparison of these two interpretations [108]: in a basic modal logic, $\Box\varphi$ denotes that φ is true by necessity, and $\Diamond\varphi$ states that φ may possibly be true; if time represents the ordering of these various worlds, and assuming that there is no past or that time is non-branching, then $\Box\varphi$ means that φ is always true, and $\Diamond\varphi$ means that φ will be true at some future (or eventual) stage.

Martin and Odell [67] classify rules into two categories: constraint and derivation.[1] Constraint rules restrict either structure or operation of an object and are categorized into stimulus/response, operation constraint, and structure constraint rules; conversely, derivation rules are divided into inference and computation.

The Business Rules Group (BRG) [18] classifies business rules into three categories: structural assertions, action assertions, and derivations. Structural assertions describe static representations, such as those modeled by entity-relationship diagrams. As they state, their structural assertions usually describe "possibilities"; for instance, "A model may be requested by a customer from a rental branch" [18]. BRG defines dynamic assertions as constraints on the results of actions; as a result, these types of assertions use the words "must or must not" and "should or should not." The BRG's dynamic assertions include assertions for conditions (e.g., "is a car registered?"), for integrity constraints (e.g., "a car must be registered"), or for authorization (e.g., only certain individuals perform certain tasks). Finally, the derivations of the Business Rules Group consist of either "inference" or "mathematical calculation" using facts, other derivations, or action assertions.

A small running example that describes the specification of access control properties in Alloy and demonstrates the benefit of access control formal analysis [56] is presented in Section D.2.1. Dwyer et al. [28] collected 555 specifications from 35 different sources within several domains. Some had only temporal logic specifications, and others had both informal prose and temporal logic specifications. The authors classified these specifications into two categories: *occurrence* and *order*. *Occurrence* is grouped into *absence*, *universality*, *existence*, and *bounded existence*. Furthermore, *order* is split into four classes: *precedence*, *response*, *chain precedence*, and *chain precedence*. For instance, to describe an order of precedence such that the occurrence of one event/state, S, is a necessary pre-condition for the occurrence of a second one, P, requires the following LTL specification: $\Diamond P \rightarrow (\neg P\, U(S \wedge \neg P))$

[1] Martin and Odell attribute many of their stated ideas about rules to Frans Van Assche and mentions an internal publication: Frans Van Assche, *Rule-Based IEM*, James Martin and Co., internal paper, December 1991.

Chapter 5

Evaluation: Conference Management Case Study

Summary: This chapter describes a conference management case study, called CONTINUE, and its realistic access control policies and properties. This detailed illustration demonstrates the approach explained in Chapters 3 and 4 of this thesis.

As Figure 5.1 shows, this chapter uses the materials on access control models, rules and policies, and their combinations from Chapter 3 (corresponding to Boxes A1, A2, A3, and B3 of Figure 1.1) to build access control policies in CONTINUE. This chapter also mentions the approach for encoding CONTINUE's access control policies using PROMELA (the language of SPIN) corresponding of Box C1 of Figure 1.1).

CONTINUE's policy and property descriptions use the approach of Chapters 3 and 4 as shown in Figure 5.2 that corresponds to Box C of Figure 1.1. The properties are specified in Linear Temporal Logic (LTL). The LTL specifications are followed by the formal verification of these properties using the SPIN model checker.

Properties are then verified, and the results are expressed and compared/contrasted with the outcomes of two prior works that use the same case study but apply different verification techniques. Similar to the two prior works, the first-applicable combining algorithm is first used, and the verification time and state space, obtained by using SPIN, are provided. Then, the expressiveness advantage of this thesis's approach over other prior works is described. Specifically, three combining algorithms–*ordered-permit-overrides*, *ordered-deny-overrides*, and *only-one-applicable* that prior works are not capable of describing–are presented based on this thesis's approach in two formats including state machines. Using the ordered-permit-overrides algorithm, the verification of properties for the same case

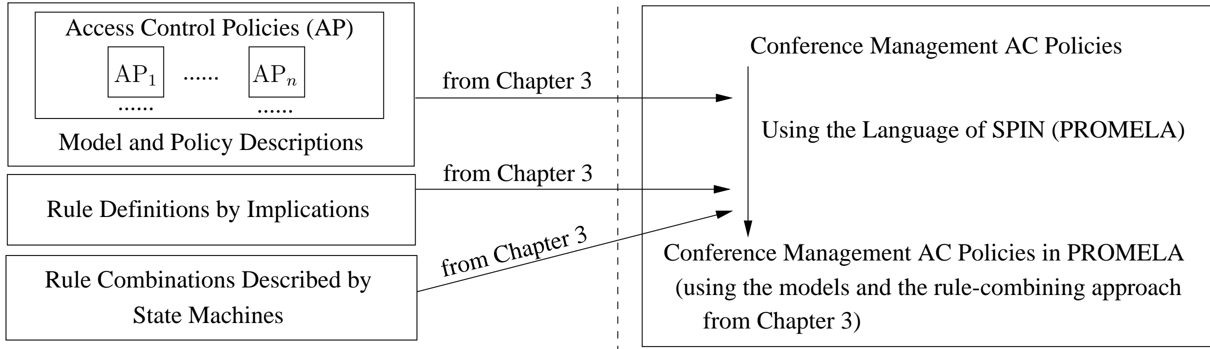


Figure 5.1: A summary of this chapter for access control policy specification

study is also performed, and verification time and state space are expressed.

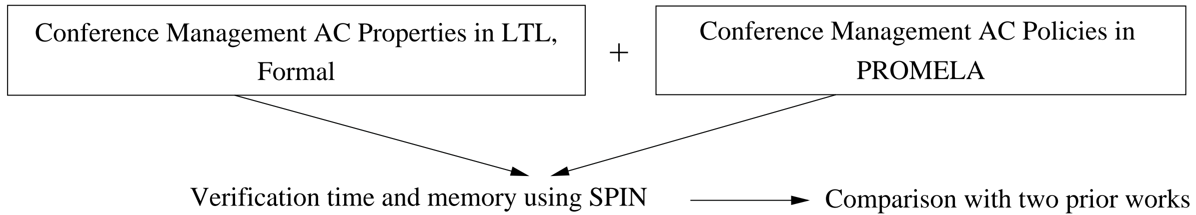


Figure 5.2: A summary of this chapter for property description and verification

Section 5.1 is a summary of the CONTINUE conference management case study and the scope of this application. Section 5.2 explains business and access control models, AC rules and their combinations using algorithmic forms and state machines for the case study. This section describes Boxes A1, A2, A3, and B3 of Figure 1.1 for the case study. Section 5.3 points to the materials of Boxes B1 and B2 of Figure 1.1 for the case study and is mainly about Box C (C1 and C2) of Figure 1.1. This section details the results in terms of verification time and memory, and the state space of the problem.[1] This section also describes the expressiveness advantage of this thesis's approach over those of prior formal verification works' approaches. Section 5.4 provides a short note on selecting the SPIN model checker to specify and verify properties.

[1]The *state* of a program is a set consisting of the values of the program variables and location counters of the program, and the *state space* of a program represents the set of states that can possibly exist within a computation [11].

5.1 CONTINUE, Policies, and Properties

CONTINUE [60] is a free conference management application supporting the submission, review, discussion, and notification phases of conferences. Shriram Krishnamurthi initially wrote CONTINUE. A broad description of CONTINUE's behaviour follows:

- During the initial stage, individuals can view the conference information.
- During the submission phase, authors, including program committee (PC) members, but not PC chairs, can submit papers.
- PC chairs assign papers to non-conflicted PC members (i.e., PC members cannot be assigned to review their own papers).
- Only those who are assigned to review papers can submit reviews.
- No PC members can view other PC members' reviews unless the former have submitted their own reviews. The purpose of preventing PC members from accidentally accessing other member reviews, before submitting their own, is to reduce bias in their reviews.
- PC chairs can see all decisions, whereas PC members do not possess this authorization. PC members who have submitted papers should not be able to determine who reviewed their papers.
- Paper reviews are read during the discussion phase and are the basis for decisions on which papers are accepted.

Many conferences do not allow their own chairs to submit research papers; based on this practice, CONTINUE allows PC chairs to read/write all decisions [60]. CONTINUE has been used by several conferences, such as the International Symposium on Software Testing and Analysis. In addition, other works also used this case study, making comparison possible between the verification results obtained by applying the approach described in this work and those of other authors.

The original conference management access control policies are described by eXtensible Access Control Markup Language (XACML). The CONTINUE policies are available in XACML format on the following web site.[2] For brevity, a prose description of the XACML

[2] http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/continue/CodeB/

policies are provided in Appendix E because in their original XACML format they occupy more space than is available here. The XACML format and also the prose translation demonstrate the difficulty of defining access control policies correctly because of the numerous rules and their nested referrals. Furthermore, this application provides properties that are provided in Appendix E. This thesis specifies properties according to the approach provided in Chapter 4.

5.2 Business and AC Models, Rules and Combination

This section uses the CONTINUE case study to describe the modeling of access control on top of business models, the specification of AC rules based on models using the provided AC rules in Extended BNF, and the description of AC rule combinations using algorithmic forms and state machines.

5.2.1 Business and AC Models

Figure 5.3 shows a REA exchange model. This figure views submitting and reviewing (papers) as dual events and is presented. An agent or author (or a role) submits a paper, and a PC member updates (e.g., reads, writes, modifies, or deletes) reviews. The two events of *submit* and *update* are viewed as REA dual events.

An access control model can be built on top of Figure 5.3. Figure 5.4 shows such a generic model that is a slight modification of Figure 3.12(a) in Chapter 3.

An example of the generic model of Figure 5.4 can be constructed. This access control example model (Figure 5.5) can be combined with Figure 5.3. In this combination, the identical classes and associations are merged. These classes consist of *ConfMember*, *Update*, and *Review*. The associations consist of one between PCMember and Update, and another one between Update and Review.

Similarly, examples of access control models for submission of papers can be defined using the generic form in Figure 5.4, which can subsequently be merged with Figure 5.3.

5.2.2 Access Control Rule

An access control rule of the case study is described in relation to the extended BNF (EBNF) form previously defined.

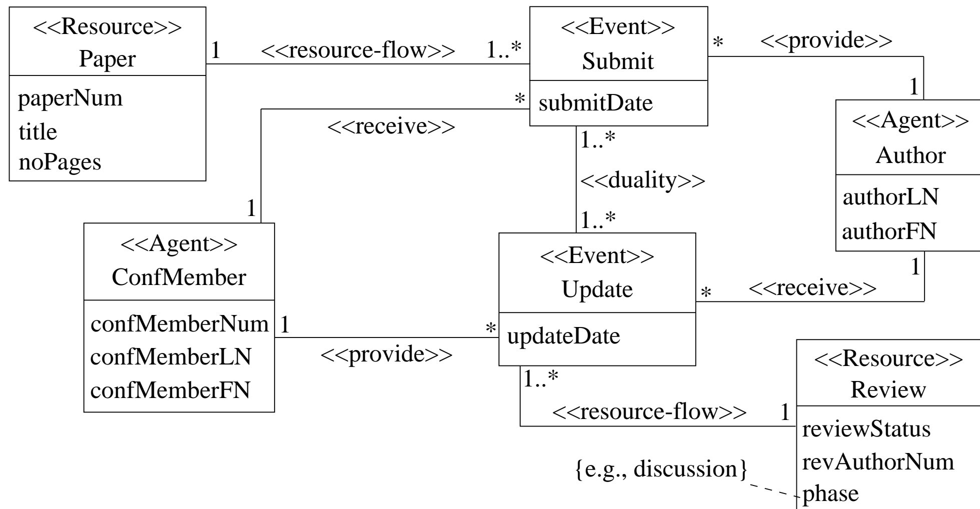


Figure 5.3: A REA model of submitting and reviewing papers

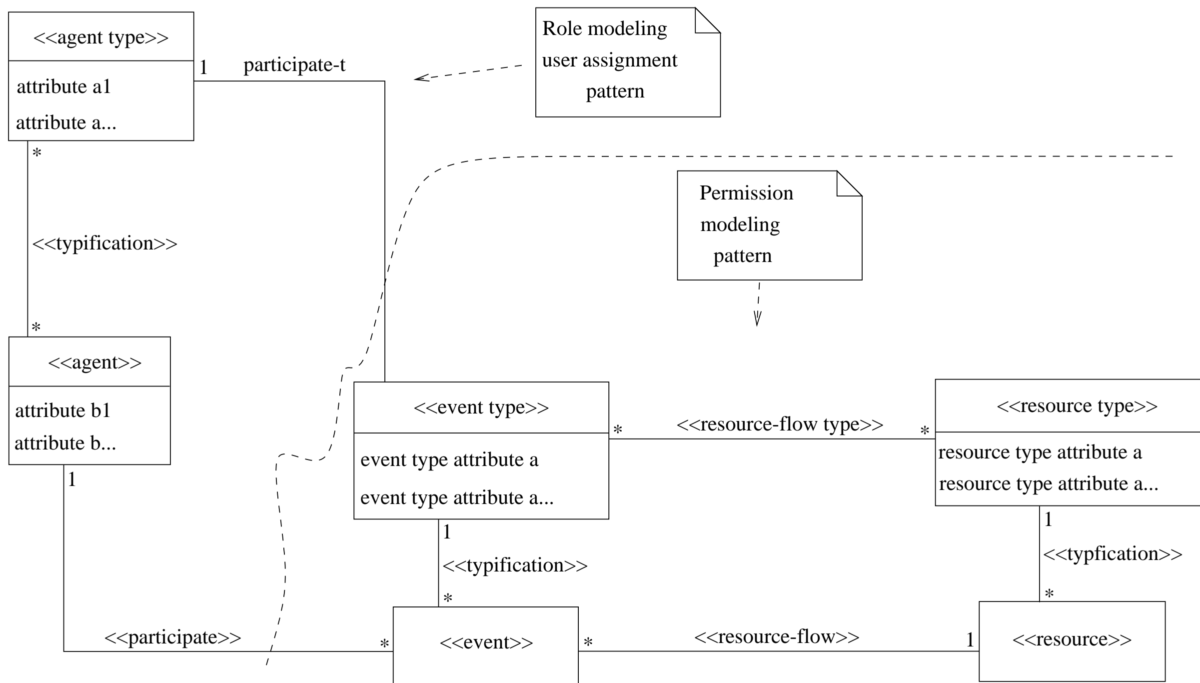


Figure 5.4: A core access control model, based on this thesis's presentation

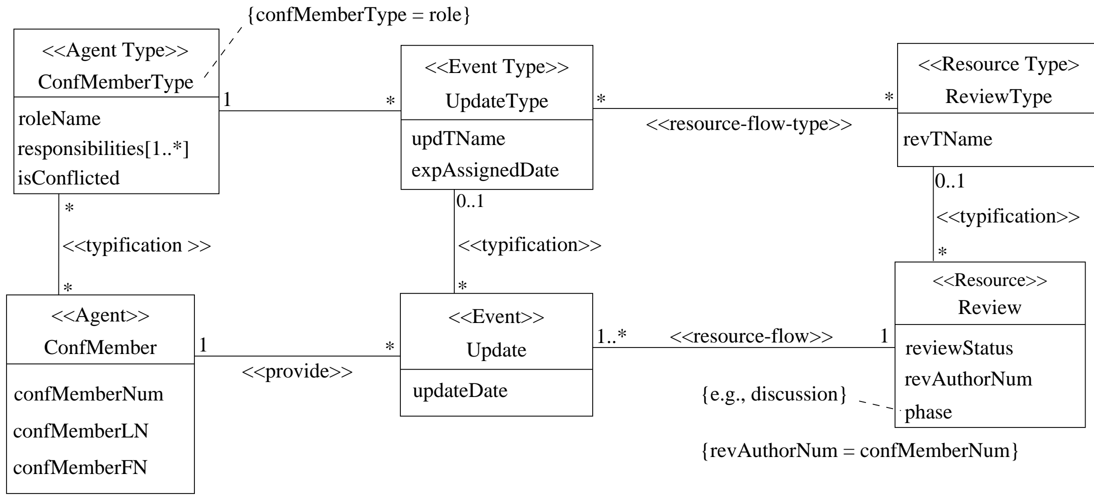


Figure 5.5: An example of access control model for reviewing papers

Example 1: A *pc chair* is permitted to create or delete *review* resources.

This rule is based on Figure 5.5. In this figure, the *roleName* attribute of ConfMemberType can have values of PCChair and PCMember. This rule can be described by the EBNF definitions (Figure 3.22) in Chapter 3. According to the provided definitions, an AC rule can be defined as follows:

ACRule = (AgentExp and ResourceExp and EventExp and AgeEveRel
and ResEveRel) implies EventResult;

Next, each component of the above expression, such as AgentExp, is described. The combinations of these expressions using *and* and *implies* create an access control rule, as shown next.

AgentExp: (AgentType = PCChair)

and

ResourceExp: (ResourceType = Review)

and

EventExp: (EventType = Create or EventType = Delete)

and

AgeEveRel: (RelATET(PCChair, Create) and RelATET(PCChair, Delete)

and

ResEveRel: (RelRTET(Review, Create) and RelRTET(Review, Delete))

implies

EventResult: (Create Access = permit and Delete Access = permit)

5.2.3 AC Rule Combination by Algorithmic Form and State Machine

The approach shown in this chapter can describe various ordering combination algorithms, such as *ordered-permit-overrides*, *ordered-deny-overrides* and *only-one-applicable* for the purpose of formal verification, whereas prior works are not capable of expressing these combinations. These three combining algorithms and their specifications according to this thesis's approach follow.

The Ordered-permit-overrides Rule-combining Algorithm: This algorithm can be described as follows [79]: The evaluation of rules within a policy is in the same order that these rules are listed in a policy. If any rule evaluates to *permit*, then the result is *permit*. If none of the rules evaluates to *permit* and the result of at least one evaluation is *deny* and the results of the rest are *not-applicable*, then the result is *deny*. If none of the rules applies, then the final evaluation result is *not applicable*.

Figure 5.6 shows a description of ordered-permit-overrides in an algorithmic form.

Similarly, Figure 5.7 also shows the UML state machine corresponding to the ordered-permit-overrides rule-combining algorithm. The same convention from Chapter 3, restated below, still holds.

State Naming Convention and Meaning: The first digit of a state name (e.g., q_{10} in Figure 5.7) indicates the rule number, and the second digit indicates whether the assumption of that rule holds (1) or does not hold (0); e.g.,

q_{11}: the state in which rule 1's assumption holds (i.e., true = 1).

q_{10}: the state in which rule 1's assumption does not hold (i.e., false = 0).

Permit-overrides, another algorithm, is similar to ordered-permit-overrides with one difference: in permit-overrides, rules can be evaluated in any orders within a policy. This algorithm can be implemented using the non-deterministic if-statement of SPIN (Appendix D).

The Ordered-deny-overrides Rule-combining Algorithm: This algorithm can be described as follows [79]: The evaluation of rules within a policy is in the same order that

```
set seen-deny to false;
initial state = state00;
for i = 1 to n do
    // n = the number of rules in a policy
    if premise-rule_i = false then
        move to state_i0;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            move to state permit;
            exit loop;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state deny-seen;
            set seen-deny to true;
        end
    end
    if i = n then
        if seen-deny = true then
            move to deny state;
        else
            move to state NA;
        end
    end
end
```

Figure 5.6: The defined AC rules and states for ordered-permit-overrides

these rules are listed in a policy. If any rule evaluates to *deny*, then the result is *deny*. If none of the rules evaluates to *deny* and the result of at least one evaluation is *permit* and the results of the rest are *not-applicable*, then the result is *permit*. If none of the rules applies, then the final evaluation result is *not applicable*. Figure 5.8 shows the algorithmic form representation of this algorithm.

Similarly, Figure 5.9 shows the UML state machine for this algorithm.

Similar to the previous case, another rule-combining algorithm is *deny-overrides* in which rules can be evaluated in any order within a policy. Similarly, the use of the non-deterministic if-statement of SPIN, as previously explained, applies in this case too.

The Only-one-applicable Rule-combining Algorithm: The XCAML standard de-

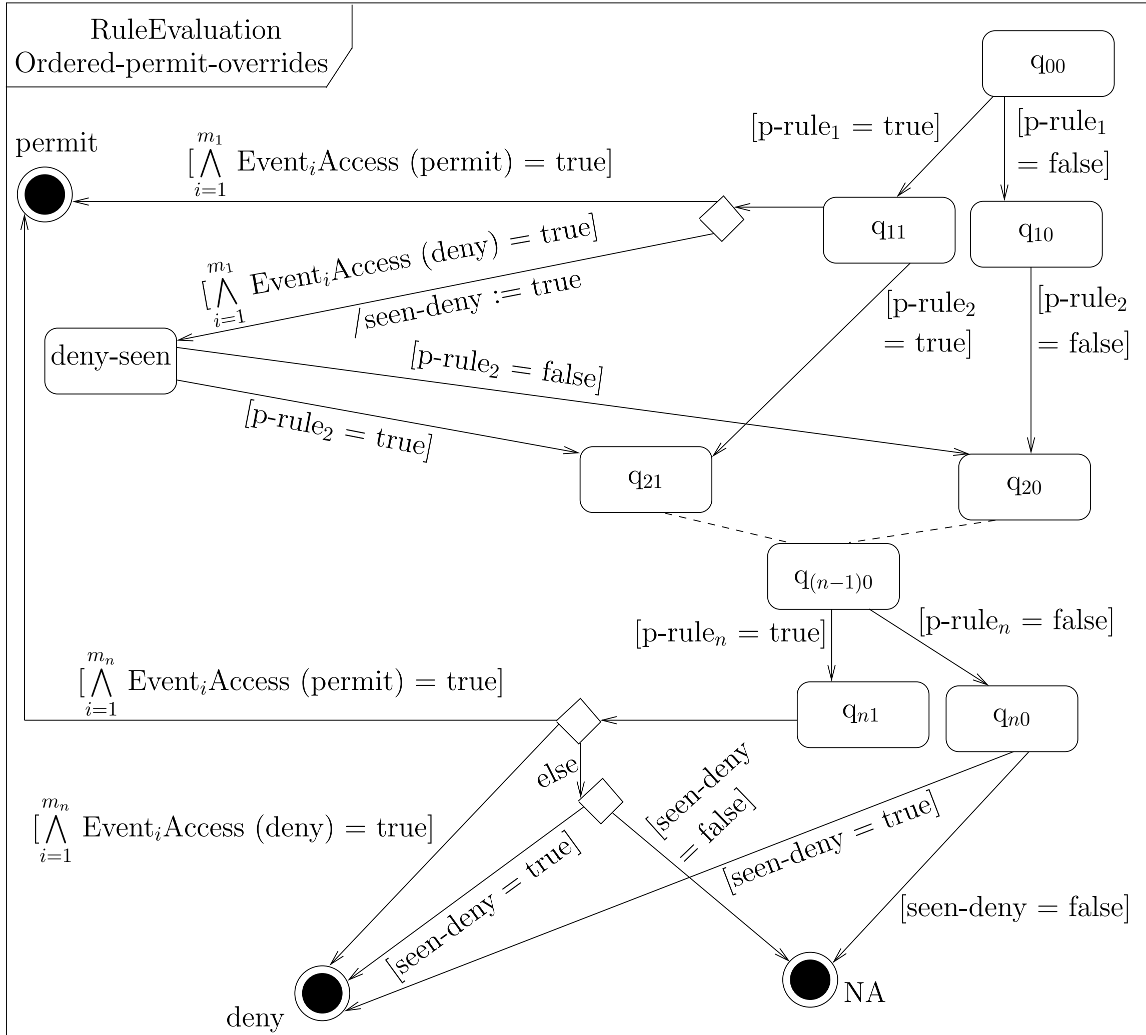


Figure 5.7: A UML state machine using the defined AC rules for ordered-permit-overrides

fines the only-one-applicable combining algorithm for policies not rules. XCAML defines the only-one-applicable algorithm for policies and not for rules. Kolovski et al. [59] describes the only-one-applicable rule-combining algorithm as the same rule- and policy-combining algorithms of XACML are similar. This algorithm can be described as follows: if more than one rule is applicable, then the result is *indeterminate*. If none of the rules is applicable, then the result is *not applicable*. If only one rule applies, then the result of that rule applies. Figures 5.10 and 5.11 describe this algorithm.

```
set seen-permit to false;
initial state = state_00;
for i = 1 to n do
    // n = the number of rules in a policy
    if premise-rule_i = false then
        move to state_i0;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            move to state permit-seen;
            set seen-permit to true;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state deny;
            exit loop;
        end
    end
    if i = n then
        if seen-permit = true then
            move to state permit;
        else
            move to state NA;
        end
    end
end
```

Figure 5.8: The defined AC rules and states for ordered-deny-overrides

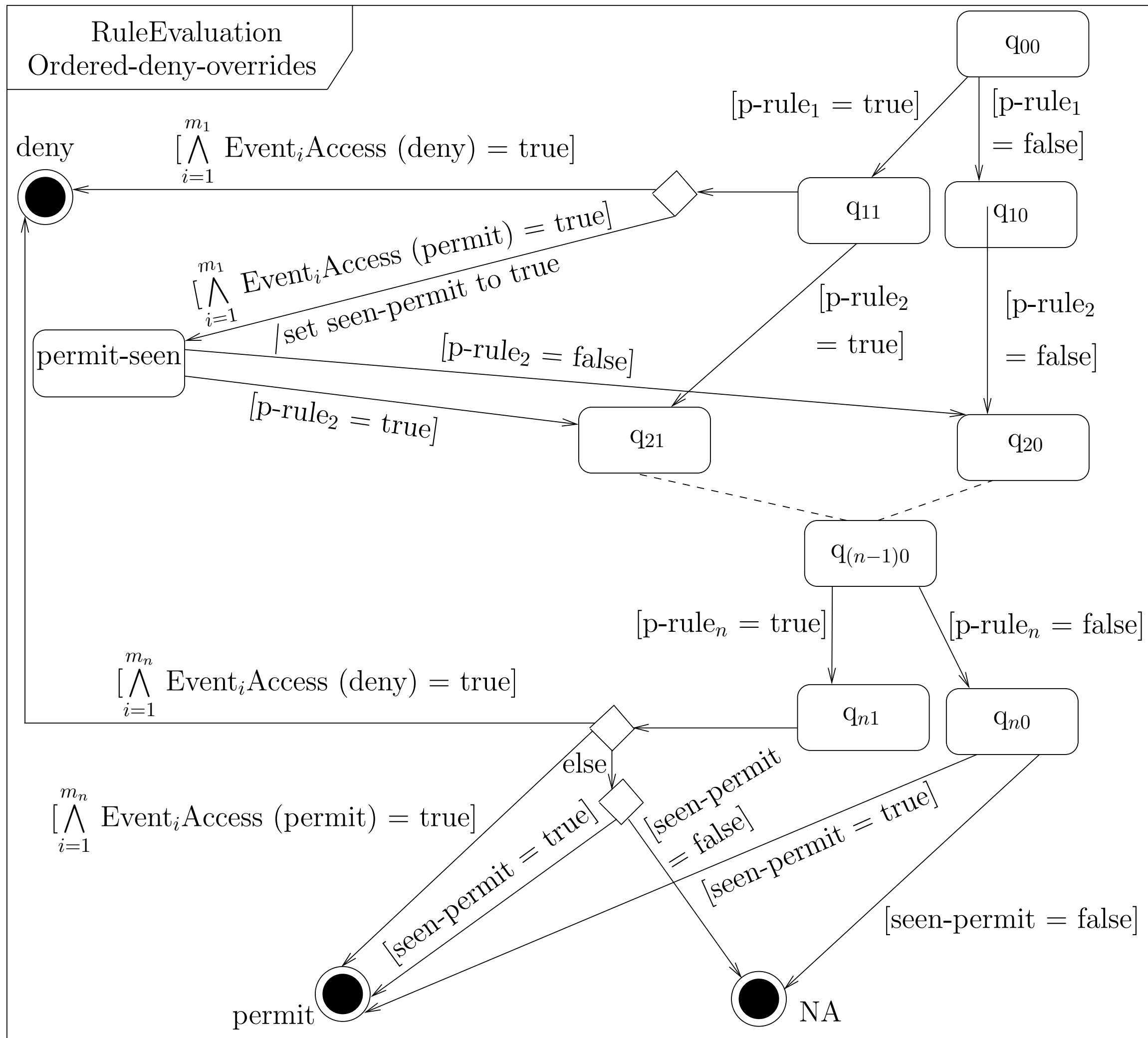


Figure 5.9: A UML state machine using the defined AC rules for ordered-deny-overrides

```
initial state = state_00;  set num-seen to zero;  set result to false;
for i = 1 to n do
    // n = the number of rules in a policy
    if premise-rule_i = false then
        move to state_io ;
        if i = n and num-seen > 1 then
            move to state indeterminate;
            exit loop;
        else if i = n and num-seen = 0 then
            move to state NA;
            exit loop;
        else if i = n and num-seen = 1 and result = permit then
            move to state permit;
            exit loop;
        else if i = n and num-seen = 1 and result = deny then
            move to state deny;
            exit loop;
        end
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            set result to permit;
            add one to num-seen; move to state seen;
        else if Event_i Access(deny) = true for every element of EventResult then
            set result to deny;
            add one to num-seen; move to state seen;
        end
        if i = n and num-seen > 1 then
            move to state indeterminate;
            exit loop;
        else if i = n and num-seen = 1 and result = permit then
            move to state permit;
            exit loop;
        else if i = n and num-seen = 1 and result = deny then
            move to state deny;
            exit loop;
        end
    end
end
```

Figure 5.10: The defined AC rules and states for only-one-applicable

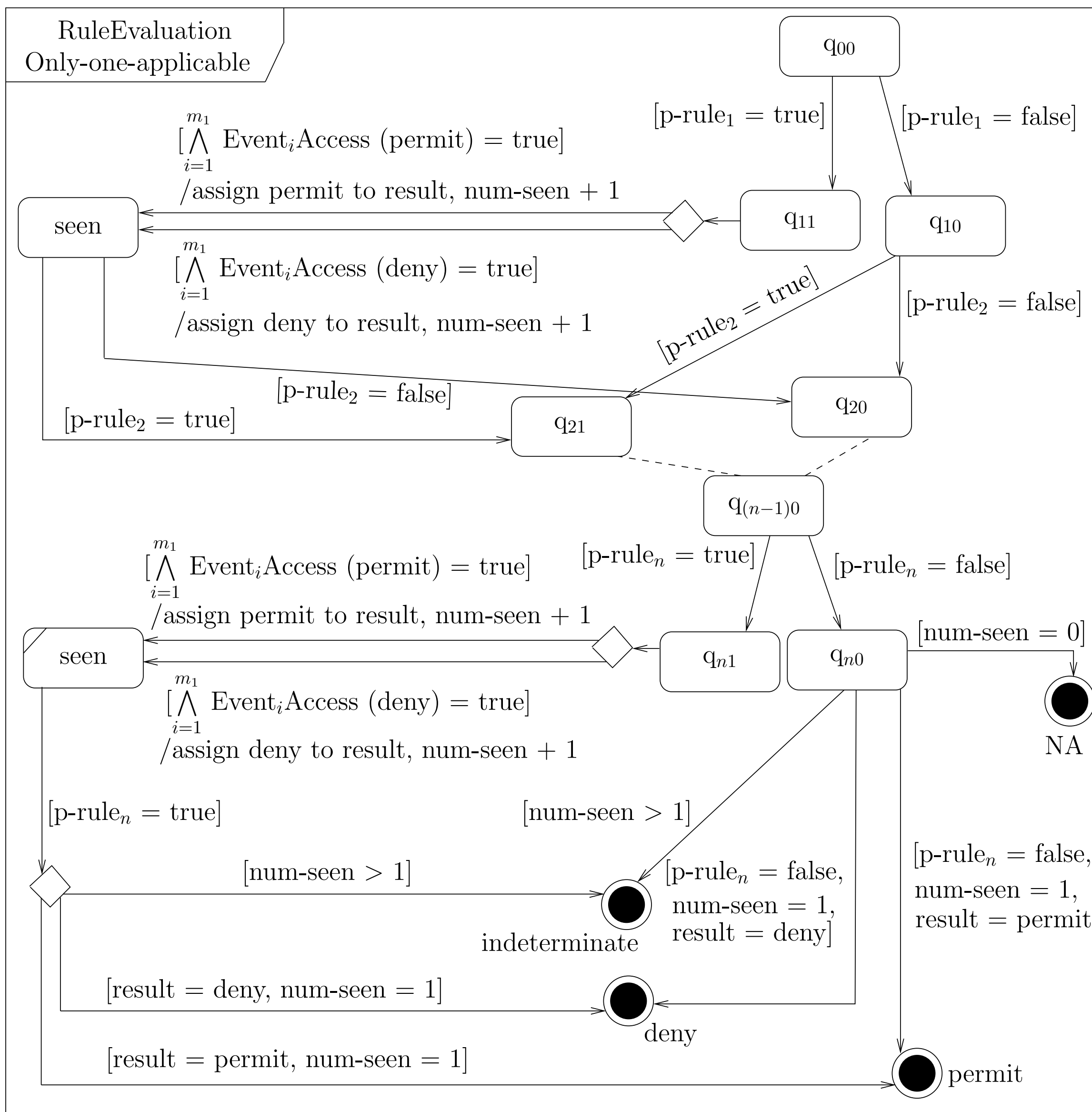


Figure 5.11: A UML state machine using the defined AC rules for only-one-applicable

5.2.4 An Advantage of the Thesis's Approach

Table 5.1 shows a summary and comparison of this thesis and prior approaches in terms of using rule-combining algorithms in the context of formal verification of properties. The plus sign indicates the capability of their expressions, whereas the minus sign shows the inability of their expressions. XCAML defines the only-one-applicable algorithm for policies and not for rules. Kolovski et al. [59] describes the only-one-applicable rule-combining algorithm as the same rule- and policy-combining algorithms of XACML are similar.

rule-combining algorithms	**Fisler et al.**	**Kolovski et al.**	**this thesis**
first-applicable	+	+	+
permit-overrides	+	+	+
deny-overrides	+	+	+
ordered-permit-overrides	-	-	+
ordered-deny-overrides	-	-	+
only-one-applicable	-	-	+

Table 5.1: Rule-combining algorithms in the context of their use with formal verification

Similarly, an expression of the combination of ordered-permit-overrides, ordered-deny-overrides, and only-one-applicable rule-combining algorithms with others is not possible. As a result, if a policy uses the first-applicable algorithm, and another applies ordered-permit-overrides, then their combinations cannot be expressed in conjunction with the previously presented formal verification. Table 5.2 shows the various possible combinations.

rule-combining algorithms	**Fisler et al.**	**Kolovski et al.**	**this thesis**
first-applicable + ordered-permit-overrides	-	-	+
first-applicable + ordered-deny-overrides	-	-	+
first-applicable + only-one-applicable	-	-	+
permit-overrides + ordered-permit-overrides	-	-	+
permit-overrides + ordered-deny-overrides	-	-	+
permit-overrides + only-one-applicable	-	-	+
deny-overrides + ordered-permit-overrides	-	-	+
deny-overrides + ordered-deny-overrides	-	-	+
deny-overrides + only-one-applicable	-	-	+

Table 5.2: Rule-combining algorithms in the context of using with formal verification

5.3 Formal Analysis

This section uses the SPIN model checker to assist in expressing CONTINUE's policies and properties in addition to the analysis of properties. SPIN accepts specifications written in a C-like language called PROMELA. Correctness properties can be written as assertion statements or specified as Linear Temporal Logic (LTL) formulas. An overview of SPIN is provided in Appendix D.1.

Access control was initially defined using a matrix to specify who has access to what, but the growth of an organization adds to the maintenance problems of such a matrix; instead, access control is now defined using rules to describe the information within that matrix [36]. Despite the benefits of using rules, some disadvantages still exist [58]: 1) large organizations have a lot of rules; therefore, application of these rules makes it problematic to determine who has access to what, and 2) the addition and modification of new rules add to the problem of maintaining these rules. (Although the disadvantages are described only in the context of Rule-based RBAC, they apply generally). In addition, it is well known that what one specifies is what one gets but not necessarily what one wants. Therefore, the analysis of specifications still constitutes a significant step no matter how carefully specification is described.

5.3.1 Formal Specification of AC Policies in PROMELA

The existing CONTINUE policies, described in XACML, are specified in PROMELA. The formats of AC rules have also previously been defined in extended BNF (EBNF). These rules are encoded in PROMELA.

In addition, this specification of AC rules follows the algorithmic forms or state machines to encode policies and their combinations. The specific AC rules of CONTINUE are specified in their place-holders, identified as rule-numbers, in algorithmic forms or state machines.

The following example shows one possible combination of policies. The following if-statement shows the premise of rules, such as premiseRule1 and premiseRule2, and the conclusion of rules, such as conclusionRule1 and conclusionRule2. The premises and conclusions of rules are defined in PROMELA as shown in the top portion of Figure 4.1, using equalities and arrays as described on Page 89. Initially, the state is q_{00}, as shown in the state machines of this thesis. If premiseRule1 holds, then the state is q_{11}, and based on conclusionRule1, the state proceeds to a result (*permit* or *deny*). Otherwise, the state is state

q_{10}, and the procedure continues to the state representing premiserule2. The procedure continues the evaluation to the end in the same manner.

```
if
::  premiseRule1 -> conclusionRule1;
else
   if
   :: premiseRule2 -> conclusionRule2;
   fi;
   else
      if
      :: premiseRule3 -> conclusionRule3;
      fi;
fi;
```

Next, premiseRule1, conclusionRule1, premiseRule2, conclusionRule2, permiseRule3, and conclusionRule3 of the if-statement are defined. RT, AT, and ET stand for ResourceType, AgentType, and EventType, respectively.

The definition of `premiseRule1` follows:

```
RT == pcmemberR && AT == member && ET == readEvent && memR[13].AgeN ==
mem && memR[13].RAct == readA && memR[13].ResN == pcmemberR
```

`conclusionRule1` is defined as `readAccess == permit`

`premiseRule2` is defined as

```
RT == pcmemberR && AT == admin && (ET == writeEvent || ET ==
createEvent || ET == deleteEvent) && admR[2].AgeN == adm &&
admR[2].WAct == writeA && admR[2].CAct == createA && admR[2].DAct
== deleteA && admR[2].ResN == pcmemberR
```

The definition of `conclusionRule2` follows:

```
writeAccess == permit; createAccess == permit; deleteAccess == permit
```

`premiseRule3` is defined as follows:

```
RT == pcmemberR && AT == member && AT.userID == RT.resourceID && (ET
```

```
== readEvent || ET == writeEvent || ET == createEvent || ET ==
deleteEvent) && memR[13].AgeN == mem && memR[13].RAct == readA &&
memR[13].ResN == pcmemberR
```

The definition of `conclusionRule3` follows:

```
readAccess == deny; writeAccess == deny; createAccess == deny;
deleteAccess == deny
```

5.3.2 Formal Specification of AC Properties in LTL

CONTINUE provides properties that this thesis expresses in LTL. The SPIN model checker is used to specify properties. Properties can be specified as LTL expressions in SPIN. Appendix D.1 provides an overview of SPIN. Table 5.3 shows the math symbols and the SPIN equivalents used.

Operator	LTL	SPIN
and	$\wedge$	&&
or	$\vee$	||
not	$\neg$	!
implies	$\rightarrow$	->
always	$\square$	[ ]
eventually	$\lozenge$	<>
next	X	X

Table 5.3: Some LTL and SPIN operators

Property Example: For any state, if an individual is neither a PC chair nor an administrator, then he or she cannot eventually set (write) the meeting flag resource.

This property exemplifies category 3 and its subcategory the *universality sub-category*, as described in Chapter 4. This property can be defined in the LTL notation of SPIN as

```
[](pThree -> X<>notSetup)
```

where pThree and notSetup are defined as follows:

```
#define pThree (!(AT == chair || AT == admin) && (RT ==
```

```
   ismeetingflagR) && eventIsW && chaERrel && admERrel)
```

where eventIsW, chaERrel, and admERrel are defined as

```
#define eventIsW (ET == writeEvent)

#define chaERrel (chaR[13].AgeN == cha && chaR[13].RAct == readA
   && chaR[13].WAct == writeA && chaR[13].ResN == ismeetingflagR)

#define admERrel (admR[0].AgeN == adm && admR[0].WAct == writeA
   && admR[0].ResN == pcmember_infoChairFlagR)
```

notSetup is defined as `#define notSetup (writeAccess == deny)`

Property Example: It is always the case that if an individual role is not described (i.e., no roles exist for a subject), then no permit exists for the individual.

This property exemplifies the *absence globally sub-category* of category 1 as described in Chapter 4. This property can be defined in the LTL notation of SPIN as `[]pFour`

where pFour is defined as

```
#define pFour (!(AT == uniden && aPermit))
```

aPermit can be described in PROMELA as

```
#define aPermit (readAccess == permit || writeAccess == permit ||
   createAccess == permit || deleteAccess == permit)
```

The use of eventually instead of always in this property expression places this property into the *existence subcategory* of category 1, as shown in Chapter 4.

Property Example: It is always the case that a program committee (PC) member who owns reviews can eventually edit his/her reviews.

This property exemplifies the *universality sub-category* of category 4 (Chapter 4). This example is similar to the first example of this section but uses the attributes of program committees and reviews, thus placing this property into category 4.

5.3.3 Verification Results and Expressive Advantage

This work has verified the twelve properties provided by CONTINUE. The result of this verification is described and compared with the results of two previous works. These two

Pr_1	Pr_2	Pr_3	Pr_5	Pr_6	Pr_7	Pr_9	Pr_{10}	Pr_{11}	Pr_{12}
1,364,017	496,401	538,257	368,017	368,017	436,497	383,633	438,033	387,089	381,953
12.3	3.06	3.2	2.14	2.14	2.7	2.31	2.74	2.37	2.23
468.298	170.426	184.796	126.349	126.349	149.859	131.710	150.387	132.896	131.133
700,017	380,049	377,233	368,017	368,017	377,745	370,833	372,433	369,521	368,465
3.75	1.91	1.84	1.89	1.85	1.93	1.92	1.93	1.8	1.88
240.332	130.479	129.513	126.349	126.349	129.688	127.315	127.865	126.865	126.502

Table 5.4: State space, memory, and verification time, New, First Applicable

works use different approaches that are not completely comparable with the experiment described in this chapter, but they use CONTINUE policies and properties and use verification as a part of their efforts. The approach in this thesis in terms of the use of a model checker to verify access control policies is completely comparable with the line of work described by Jha et al. [52] in Section 2.3.1.

The state space of a program is the multiplication of the number of statements of each process by the number of values each variable can have [11]. The state space in this work was relatively small because the range of values that each variable could take has been selected to be as small as possible. In addition, the number of statements has been reduced by using PROMELA's *atomic* keyword.

Table 5.4 shows the experimental results for the properties that hold. The second row is the state space; the third row is the verification time in seconds, and the fourth is the memory usage reported when running the program. Similarly, the next three rows represent state space, running time, and memory usage for the same properties, but the assumptions of implications that must eventually hold are not included. For instance, based on the explanation provided in Figure 4.5, for Pr_1, the top three rows report the experiment for the expression `[](p -> X<>q) && <>p`, and the bottom three rows of column Pr_1 use the expression `[](p -> X<>q)`, an acceptable form, in which the assumptions that must eventually hold are not stated.

Finally, Pr_4 and Pr_8 failed as they should. Pr_4 failed with the state space of 293,315, a running time 0.654 seconds, and a memory usage of 101.036. Pr_8 failed with the state space of 16,313, a running time of 0.047 seconds, and a memory usage of 7.969.

A Windows PC with a Pentium (R) Dual core 2.7GHz CPU and 4 Gbytes of memory is used for the experiment.

As an example, the CONTINUE case study is used again with the ordered-permit-overrides rule-combining algorithm. All twelve properties whose verifications under this

Pr_1	Pr_2	Pr_3	Pr_5	Pr_6	Pr_7	Pr_9	Pr_{10}	Pr_{11}	Pr_{12}
1,391,409	503,249	545,105	374,865	374,865	450,833	390,481	444,881	393,937	388,801
12.8	3.17	3.33	2.21	2.21	3.02	2.4	2.87	2.4	2.32
477.702	172.777	187.147	128.700	128.700	154.781	134.61	152.738	135.248	133.484
713,713	386,897	384,081	374,865	374,865	387,089	377,681	379,281	376,369	375,313
3.81	1.95	1.95	1.87	1.9	1.96	1.93	1.94	1.91	1.93
245.034	132.031	131.864	128.700	128.700	132.896	129.666	130.216	129.216	128.853

Table 5.5: State space, memory, and verification time with ordered-permit-overrides rule-combining algorithm

algorithm were not possible by prior approaches are verified. Table 5.5 shows the result of this verification.

Pr_4 and Pr_8 failed as they should. Pr_4 failed with the state space of 274,906, a running time 0.623 seconds, and a memory usage of 94.883. Pr_8 failed with the state space of 16,313, a running time of 0.044 seconds, and a memory usage of 7.969. Similar to the previous case, a Windows PC with a Pentium (R) Dual core 2.7GHz CPU and 4 Gbytes of memory is used for the experiment.

A comparison of this chapter's experiment and those of two other works is described next. This comparison cannot be in terms of state space and verification time directly because the other two works use different approaches. One reports a certain timing for parsing XACML and constraining the representation, and the other describes timing for parsing XACML, converting to description logic, and preprocessing time to convert them to normal form. Neither of these elements applies to the work described in this chapter.

Next, two other works that use the same policies and properties but apply different verification methods are discussed. These works are by a) Fisler et al. [36] and b) Kolovski et al. [59].

Fisler et al. [36] Work: The authors, who created and made CONTINUE policies and properties publicly available, use Multi-Terminal Binary Decision Diagrams (MTBDDs) to represent policies. MTBDDs, variations of Binary Decision Diagrams, have multiple terminal nodes such that the *permit*, *deny*, and *not applicable* of a policy rule can be represented. An MTBDD is built for each rule, then these MTBDDs are combined using provided algorithms. MTBDDs can be manipulated using PLT Scheme (renamed Racket), a programming language also used to query and verify properties. The authors report a time of 2050 milliseconds to parse and convert policies into the MTBDD representations and another 20 milliseconds to constrain this representation. The verification of each

property takes less than 1 millisecond. They seemingly obtained these results using a machine with an Athlon XP 1800+ processor at 1.5 GHz with 512 Mb RAM.

The verification time of Fisler et al.'s experiment is faster than that the results reported in Table 5.4, but Fisler et al.'s 2050 milliseconds for parsing policies and 20 milliseconds for constraining policies are not applicable and have not occurred in the experiment using SPIN because compiling the PROMELA program is fast. Fisler et al. use PLT Scheme to write queries about policies, and LTL expressions are used in this chapter to do so.

Note that Fisler et al. cannot describe any of *ordered-permit-overrides*, *ordered-deny-overrides* and *Only-one-applicable* combining algorithms.

b) The Work by Kolovski et al. [59]: The authors used description logics and implemented a prototype of an XACML analysis tool on top of Pellet, a description logic reasoner. In general, description logics are decidable subsets of first order logic, but some are supersets of predicate logic. The authors mention that one advantage of choosing DL representations is their expressiveness. Therefore, a larger subset of XACML that is more expressive than propositional logic can be represented and verified. Parsing these XACML policies took 2.1 seconds, and converting them to description logic took another 1.7 seconds. Preprocessing concepts and transforming them into normal forms consumed 10.6 seconds. Verification of properties took 0.420 seconds on average. The type of machine used to obtain these results is not mentioned. The authors attribute the faster time reported by Fisler et al.'s work to the optimizations for Pellet that are designed to perform verification for a richer logic than propositional logic (i.e., the logic that describes these policies).

Kolovski et al.'s 0.42 seconds verification time is faster than the one reported in Table 5.4, but their 2.1 seconds for parsing, 1.7 seconds for converting policies to description logic, and 10.6 seconds for transforming concepts to normal formals are not applicable to the experiment using SPIN because the compilation of the PROMELA program is fast.

Note that Kolovski et al. [59] are not able to specify *ordered-permit-overrides* and *ordered-deny-overrides* combining algorithms. Kolovski et al. believe that they will be able to express only-one-applicable combining algorithm in their future work.

5.4 A Note on the Use of SPIN

The SPIN model checker is selected to specify policies and properties and to verify properties. This tool has been used in many projects. Initially, the plan was to use Alloy, but

Alloy was not used for this specific task, for reasons explained after the notion of using *scope* in Alloy's analysis has been described.

Alloy's analysis is based on *scope*: an analysis is valid for the specified scope. If the Alloy analyzer does not find a counterexample, it does not necessarily mean that no counterexample exists. A counterexample may exist in a larger scope. As Jackson [50] notes, an increase in scope decreases the chance of the existence of errors; Jackson's "small scope hypothesis" states that most errors can be found in small scope. Therefore, the scope of analysis increases gradually until it reaches a point at which the analyzer runs out of memory. Appendix D provides an overview of Alloy and an example of access control specification and verification using Alloy.

Alloy was not used for two reasons: a) Fisler et al. [36] report the unsuccessful use of Alloy even for handling one third of the CONTINUE policies, as memory was exhausted after only a few minutes. Alloy is based on first-order logic; therefore, it introduces additional variables into a model, making analysis intractable; b) the notion of scope in Alloy's analysis makes a verification valid for the specified scope. Jackson [50, Pages 184–185] describes the consequence of the existence of scope, and compares his tool with model checkers: "model checkers are generally capable of exhausting an entire state space. In an Alloy trace analysis, only traces of bounded length are considered, and the bound is generally small".

Chapter 6

Conclusion

This chapter provides a summary of contributions, a discussion of limitations, and an outline of possible future work.

6.1 Summary of Contributions

The work presented in this thesis provides a uniform formal approach to business and access control models, policies and their combinations. The research involves a new formal representation for access control rules, policies, and their combination and supports formal verification. In addition, the approach explicitly connects the rules to the underlying access control model. Further, this research approach enables the description and verification of all known access control policies expressed in the literature.

Research reported in the literature does not directly connect access control models to rules using a generic AC model. Typically, the research focuses on one or the other. Further, no formal approach (e.g., [36], [59], [66]) supports formal description and verification of classes of policy-combining algorithms related to history of policy outcomes and consensus.

This approach advances AC in the following ways:

- The approach provides a common representation for describing and integrating business processes, access control models, their rules and policies. This contribution emphasizes a systematic approach

  - starting from a business model,
  - adding AC models to these business models,
  - defining AC rules based on these models,
  - combining AC rules using state machines in which the transitions are governed by elements of the defined AC rules.

  This approach in its entirety has not been seen in the literature. Chapters 3 and 5 discuss this approach, and additional materials are provided in Appendices B and F.

- The approach expresses AC rules using an underlying AC model based on an existing augmented business modeling notation. The underlying AC model is general enough to describe Role-based Access Control (RBAC), Discretionary Access Control (DAC), and Mandatory Access Control (MAC). Chapter 3 presents this topic, and some AC rule definitions are provided in Appendix B.

- The approach can express and verify formally all known policy- and rule-combining algorithms, a result not seen in the literature. The algorithms related to history of policy outcomes that include ordered-permit-overrides, ordered-deny-overrides, and only-one-applicable are expressed in Chapter 5. In addition, the verification using this class of combining algorithms for the case study is described and results are provided in the same chapter. The formal expressions of algorithms related to consensus that include weak-consensus, weak-majority, strong-consensus, strong-majority, and super-majority-permit are shown in Chapter 3 and Appendix F.

- The approach supports a classification of relevant AC properties expressed in LTL that can be verified against policies and their combinations. Chapter 4 discusses this topic.

- Finally, the approach supports automated formal verification of single policies and combined policy sets based on model checking. Chapter 5 presents this topic and provides the verification results.

6.2 Limitations

The modeling of three main classes of AC, RBAC, DAC, and MAC, is described by the approach presented in this thesis in Chapter 3. However, several other AC models, some of them are detailed, exist. It appears that the approach shown in this thesis can be

extended to represent other AC models such as the ones involving time (e.g., temporal-RBAC (TRBAC) as mentioned in Chapter 2) and obligation. These extensions can be a topic of future work.

The computational limitations imposed by the model checking approach are discussed (Chapter 2), and these limitations also apply in general. One main limitation is state-space explosion, which implies that as the model gets larger, the analysis will be impractical. Another limitation of model checking is that the analysis is performed on a model of a system and not on the actual system code.

This thesis uses a conference management case study that has appeared frequently in the literature, and therefore the comparison of results is possible. This case study is extended to include the verification of AC policies that use policy-combining algorithms involving history that have not appeared in the literature. The verification results are shown in Chapter 5. Despite the successful reported verification results, as the case study becomes larger, state-space explosion causes verification to be impractical.

6.3 Future Work

Several future directions can enhance the work presented in this thesis. A few are described next.

6.3.1 Other Access Control Models and their Extensions

As discussed in Chapter 1, the presentation of AC models in this thesis does not rely on a specific AC model and is based on the constituent of any AC model: AC models can be described by the elements users, objects, subjects, operations, permissions, and the relationships among them [33]. Three main classes of AC are discussed to show the approach presented in this thesis. Nevertheless, several other AC models exist; for instance, the Usage CONtrol (UCON) model in which usage means the rights to use digital objects and to delegate these rights. UCON is described as an intersection of traditional AC models, digital rights management, and trust management [82]. Exploring in depth other AC models and their extensions, such as trust management, can enhance the scope of this work.

6.3.2 Rights, Delegations, and Obligations

One possible extension of this work can be the inclusion of rights and delegations. For instance, one categorization [100] provided in Figure A.2 (Appendix A) uses generalization and specialization and present resources as goods, rights, or services. This thesis has not explored resources as rights; no differences appear to exist if resources are rights, but this type of presentation may require additional attention.

Delegation within AC models has also been discussed in the literature because of the presence of this feature in various situations. Therefore, delegation can also be described in related models. For instance, an agent with a role and with permissions to perform some operations may delegate this authority to another role. CONTINUE (Chapter 5) includes two roles: PC members and sub-reviewers. Although, in essence, PC members delegate reviewing papers to sub-reviewers, reviewing papers by sub-reviewers is not presented as an authorization given by PC members to sub-reviewers as delegations—possibly because of difficulties involved if such an approach is taken.

Not only does the existence of delegation add more features to an AC model but also the several possible forms of delegations and their combinations, which have already been defined in the literature. Investigating the addition of delegation, its benefits, and possible disadvantages, and determining the extent of its use within AC models can be extensions to this work.

Obligations have also been discussed in the AC literature. For instance, XACML [79, 78] represents an obligation, specified within a policy, as an operation that must be performed. Including an obligation within the approach presented in this thesis can be another topic of future work.

6.3.3 Analysis and Formal Methods

Several enhancements for policy analysis are possible, for instance, the use of various analysis techniques and formal method approaches to determine what techniques are appropriate for certain situations. Performing such experiments could result in the development of guidelines. For instance, Jha et al. [52] mention that a logic programming tool uses an excessive amount of memory if the number of rules exceeds a certain number, whereas memory use is much lower if a model checker tool analyzes the same case. As mentioned in Chapter 2, one experiment shows that the number of rules is a determining runtime factor; even data sets with a larger number of roles but fewer rules run faster than data sets with fewer roles but more rules. In general, however, an increase in the number of roles can

increase the number of rules. Similarly, determining other factors in an analysis would be worthwhile. Finding out what makes one analysis more memory and time consuming than another will forewarn analysts about the degree of complexity to expect from an analysis.

Similarly, an analysis can include policies related to rights, delegations, and privacy, and as can be expected, the inclusion of these additional elements increases the state space of an analysis. Case studies can be used to examine the effect of multi-role permissions, i.e., permissions that a user can obtain by filling more than one role at the same time. Similar to the inclusion of rights and delegation, the existence of multiple roles can increase the state space of an analysis.

6.3.4 Privacy

An enhancement of this work would be the inclusion of privacy, because AC policies and privacy policies are related. Anderson [3] describes this connection well: an AC policy describes who has what operation accesses on which resources, whereas, in addition, a privacy policy must match the purposes between data collected and data accessed.

AC models are not designed to meet privacy requirements, but—as one reason—the connection between these two types of policies has created the need to extend AC models to describe privacy requirements. For instance, Privacy-aware RBAC (P-RBAC) [74] is one such effort that includes privacy within RBAC. The XACML specification [79] includes a brief privacy profile touching on this topic.

6.3.5 Different Domains

Exploring access control in different domains such as social networking and digital rights, and investigating whether each environment imposes different challenges would make a natural extension of this work. The extent and the combination of various factors such as multi-role permissions, delegations, the requirement of privacy, and the number of rules, roles, and permissions of each domain may present additional dimensions in the presentation and analysis of policies for each domain.

APPENDICES

Appendix A

An Overview of REA

The Resource-Event-Agent (REA) model was introduced by McCarthy [68] and contains two broad groups of business processes: *exchange*s and *conversion*s. A sale (i.e., an exchange of cash for a product) and a loan (i.e., an exchange of cash in lieu of future return of cash and its interest) are two exchange processes [48]. Conversely, creating new products and modifying existing ones exemplify conversion processes. Furthermore, a REA *value chain* represents a model in which any number of exchange and conversion processes are connected such that a resource outflow of a process is a resource inflow to another process [48]. In addition to exchange, conversion, and value chain models, other elements such as *commitment*, *contract*, *policy*, *custody*, and *linkage* are also defined, thus enabling larger models to be described.

Nakamura and Johnson [73] describe REA and state that

> Although the REA model was proposed as a result of the study of accounting theories, it can be applied to many other business domains .... The REA model is a promising modeling technique for developing business applications because it has a solid foundation and it can be applied to nearly all business domains.

Figure A.1 [69, 100] shows a partial REA model in which a value chain represent an aggregation of business processes that are themselves an aggregation of economic events. Several rules exist for constructing REA models. For instance, as Figure A.1 shows, an association between a resource and an event is called *resource-flow*, and an association between an agent and event is named *participates*. A *duality* relationship is an association between two (or more) economic events.

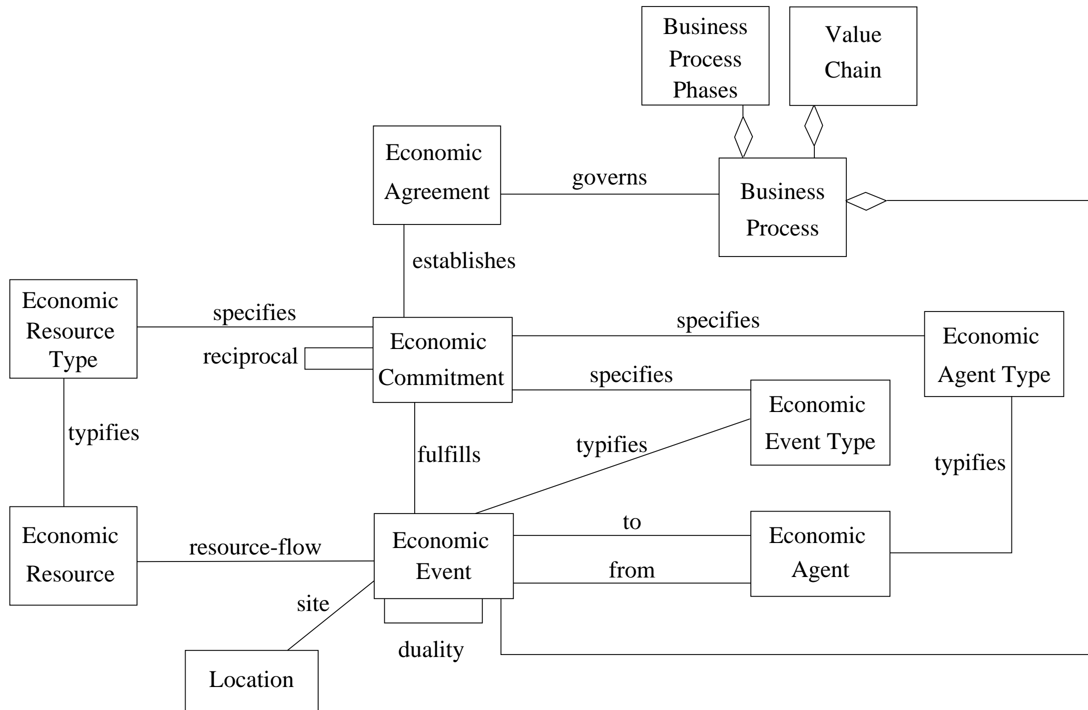


Figure A.1: A REA model

A minimum REA model includes resources, events, and agents and their relationships (i.e., resource-flow, duality, and participates). A promise to execute an event is called a *commitment*; an association between an event and commitment is called a *fulfillment*.

Figure A.2 presents a REA economic resource that can be goods, services, or rights [100]. Further classification beyond this level is also possible and may differ from one industry or business to another. The aggregation symbol in Figure A.2 indicates that a resource can have components.

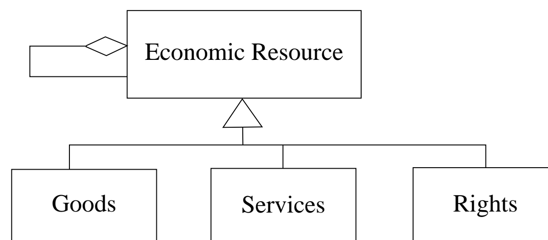


Figure A.2: Resource and its sub-categories

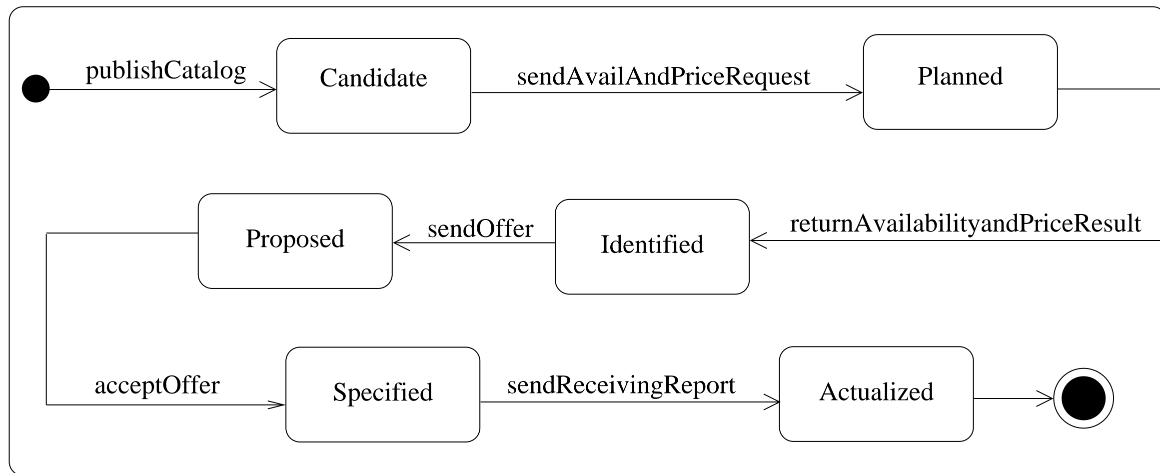


Figure A.3: A state diagram of a resource

Figure A.3 [69] shows states related to business phases *planning*, *identification*, *negotiation*, *actualization*, and *post-actualization*. This figure shows a resource that, during purchase of an item, passes through various business process phases, mentioned previously.

A.1 REA as an Ontology

REA is also presented as an ontology using Sowa's categorization [97]. Ontology is "the study of existence, of all the kinds of entities—abstract and concrete—that make up the world" [97] or similarly "an explicit specification of a conceptualization" [70]. Sowa has classified the components of reality into three categories: "abstract versus physical," "continuant versus occurrent," and "firstness versus secondness versus thirdness"; as a result, he recognizes twelve, $2\times2\times3$, categories.

These categories can be briefly described so [42, 63]: "the category *physical* for anything consisting of matter or energy and the category *abstract* for pure information structures."[1] *Continuant* represents an object and its attributes "that enable its various appearances at different times to be recognized as the same individual,"[2] whereas *occurrent* describes an event or a process identifiable with regard to a certain time and space. *Firstness* represents an entity, x, such that its existence and attributes are independent of anything external to x; for instance, a *woman* can be such an entity. Firstness is also called *first*, *independent*, or

[1]The statement between quotation marks is by Sowa [97].
[2]The statement between quotation marks is by Sowa [97].

predicate $P(x)$. *Secondness* represents entities but in connection with other(s); for instance, a *mother* can describe the same woman but in relation to her children. Relative is also named *second*, *relative*, or $P(x, y)$. Finally, *thirdness* describes a relation, x, mediating two other entities, y and z; *motherhood* can be an example that describes a relation between a mother and her children. Thirdness is also referred to as *third*, *mediating*, or $P(x, y, z)$. As a result of applying Sowa's classification, the REA ontology is organized into the twelve categories shown in Table A.1 [42].

	Physical Categories		**Abstract Categories**	
	Continuant	***Occurrent***	***Continuant***	***Occurrent***
1st ***Or Independent***	EconomicAgent (A) EconomicResource (R)	EconomicEvent (E) Commitment (C)	AgentType (AT) ResourceType (RT)	EventType (ET) CommitmentType (CT)
2nd ***Or Relative***	Association (A-A) Custody (A-R) Linkage (R-R)	Stockflow (E-R) Duality (E-E) Accountability (E-A) Executes (C-E) Involvement (C-A) Reserved (C-R) Reciprocal (C-C)	Typification (A-AT) (R-RT) Characterization (AT-AT) (AT-RT) (RT-RT)	Typification (E-ET) (C-CT) Scenario (ET-RT) (ET-ET) (ET-AT) (CT-ET) (CT-AT) (CT-RT) (CT-CT)
3rd ***Or Mediating***	Responsibility Partnering Configuration	Exchange Conversion Contracting Scheduling	Segmentation Policy Substitutability Complementarity Configuration	Standardization Policy Strategy

Table A.1: A categorization of REA ontology

The benefit of this categorization is its placing all elements into twelve categories, but one problem is the incompleteness of this categorization [63]. A brief explanation of elements of this table follows [42]: a REA commitment can be a hotel reservation, and *segmentation* describes "the rationale for grouping resources and agents into abstract categories like *slow-paying customers*." *Configuration* in the second column can describe a resource represented

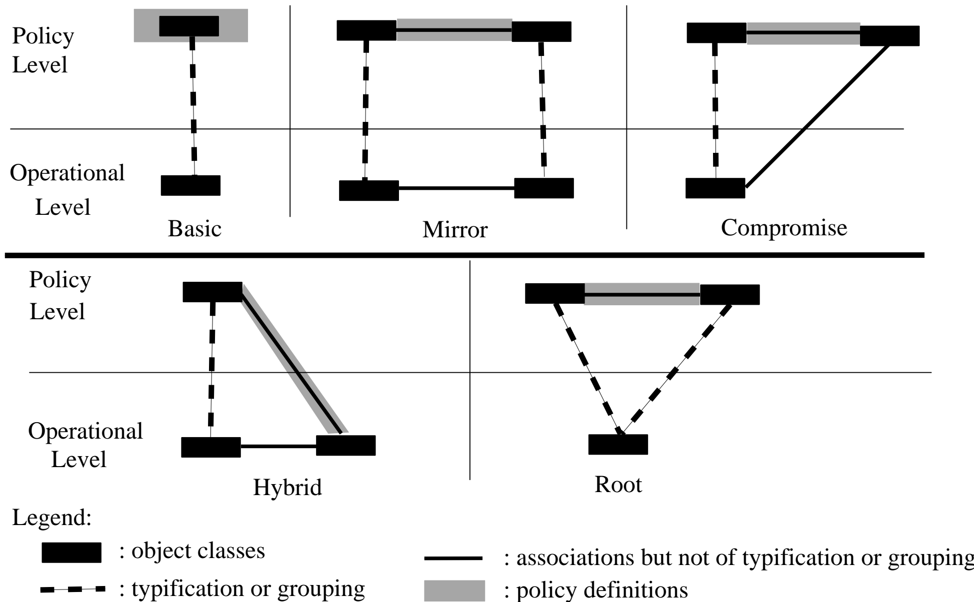


Figure A.4: Patterns for policy-level specifications

as an aggregation of resources (Figure A.2). Similarly, linking REA resource types can constitute a description for *substitutability*, *complementarity*, and *configuration* in column four. Furthermore, Table A.1 and Figure A.1 can be mapped to each other; e.g., *duality* and *typifies (typification)* are shown in both the table and figure.

A.2 REA Patterns for Policy-level Specification

REA patterns for policy-level specification are shown in Figure A.4 [43] and illustrated using a plane and flight scenario in Figure A.5 [43]; these two figures are described next.

Figures A.4 and A.5 can be described so [43]: The *Basic* pattern includes flight, flightType, and a single typification association between them. A policy is defined by a flightType attribute: based on a scheduled departure of a flightType, a policy can state that a flight should take off at a certain time (which can be different from the flight's actual departure time). In these two figures, a typification relationship denotes an "is a-kind-of" association, whereas a grouping association represents an "is a-member-of" association.

The *Mirror* pattern is among planeType, flightType, plane, and flight (i.e., two typification or grouping associations and two other associations). A policy can express the type of

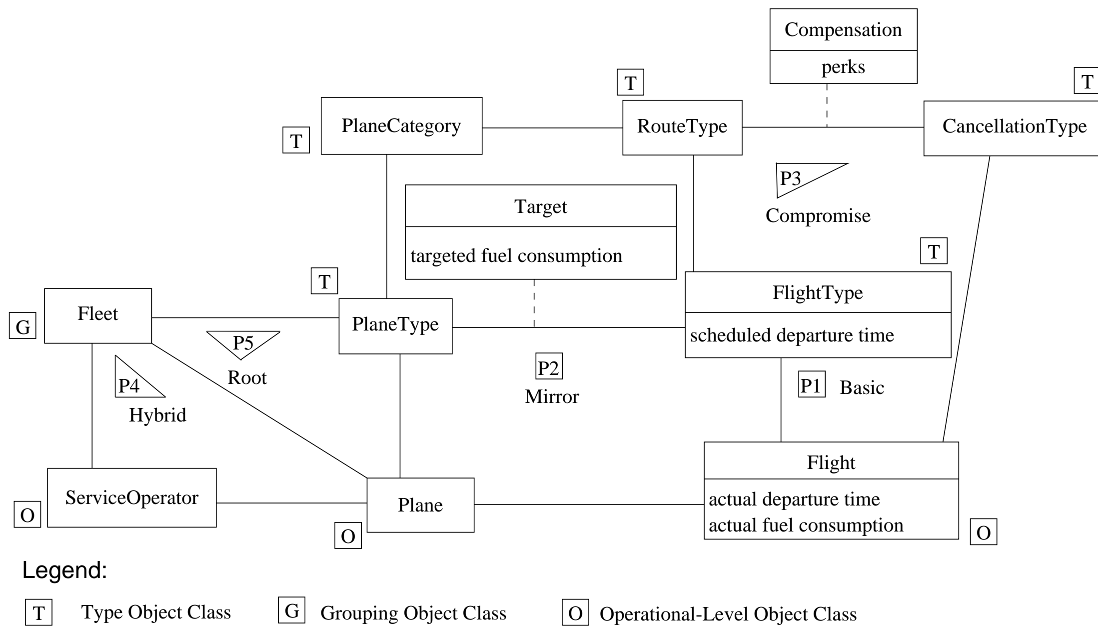


Figure A.5: A flight example for policy-level specifications

plane to be used for the type of flight. The *Compromise* pattern, a variation of Mirror, one typification or grouping is compromised: it consists of one typification or grouping and two other associations—one of which is at the policy level. Compromise includes routeType, cancelationType, and flight; a cancelationType can be defined per routeType.

The *Hybrid* pattern, another variation of Mirror, includes plane, fleet, and serviceOperator and consists of one typification or grouping and two other associations—none of which is at the policy level. This pattern is used when one class has a small number of instances on which the policy is validated (e.g., a list of people who can provide service to a fleet). Operational and policy levels are differentiated in Figures A.4 and A.5 either by their explicit mention or by the use of "T" to denote type.

Finally, the *Root* pattern is defined among fleet, planeType, and fleet and applies to cases in which one operational-level object participates in more than one typification or grouping (e.g., the typification of plane and planeType and the grouping of plane and fleet). This pattern consists of two typification or grouping associations and another association at the policy level (e.g., a plane should be of a specific planeType to be a member of a fleet).

Appendix B

BNF and EBNF Definitions

This appendix first provides the BNF definitions of access control rules in Section B.1. Then, the agent-, resource-, and event-related definitions in BNF and EBNF are presented in Section B.2.

B.1 Access Control Rule in BNF

The original BNF elements and their meanings are shown in Table B.1 [91]. As this table shows, the original BNF has only four elements and does not have specific symbols for *repetitions* (e.g., { }) and *options*(e.g., [ ]). Therefore, a specification in BNF is longer than EBNF. Table B.1 shows the BNF elements and their meanings.

BNF	**Meaning**
⟨unquoted words⟩	Non-terminal symbol
unquoted characters	Terminal symbol
::=	Defining symbol
|	Alternative

Table B.1: The original BNF

The AC rule definitions (i.e., the AC rule grammar) in BNF are checked with a tool called Gold Parser [24]. This tool requires that the symbols such as parenthesis, arithmetic

symbols (e.g., > or =), and commas be enclosed by single quotation marks. Thus, these symbols are enclosed by single quotation marks in the figures identified as BNF. In addition, the Gold Parser requires that definitions start with “start symbol”; therefore, when checking the grammar of AC rule definitions (Figure B.1), the first rule in this figure must be as follows: "Start Symbol" = ⟨ACRule⟩.

Figure B.1 shows the BNF definitions of AC rules, which have been previously presented in EBNF in Chapter 3.

Access Control Rule Definition in BNF

⟨ACRule⟩ ::= '('⟨AgentExp⟩ and ⟨ResourceExp⟩ and ⟨EventExp⟩ and
⟨AgeEveRel⟩ and ⟨ResEveExp⟩')' implies ⟨EventExp⟩
⟨AgentExp⟩ ::= '('⟨equA⟩')' | '('⟨uop⟩ ⟨equA⟩')' | '('⟨equA⟩ ⟨equArep⟩')'
| '('⟨uop⟩ ⟨equA⟩ ⟨equArep⟩')'
⟨uop⟩ ::= not
⟨equA⟩ ::= ⟨ATG⟩ | ⟨attrATG⟩
⟨equArep⟩ ::= ⟨bop⟩ ⟨equA⟩ | ⟨bop⟩ ⟨equA⟩ ⟨equArep⟩ | ⟨bop⟩ ⟨uop⟩ ⟨equA⟩
| ⟨bop⟩ ⟨uop⟩ ⟨equA⟩ ⟨equArep⟩ | empty
⟨bop⟩ ::= '(' and | or ')'
⟨ATG⟩ ::= ⟨Agent⟩ '=' ⟨className⟩ | ⟨AgentTG⟩ '=' ⟨identifier⟩
⟨attrATG⟩ ::= ⟨AgeAttIde⟩'.'⟨attrNameValue⟩ | ⟨AgeAttIde⟩'.'⟨attrNameValue⟩
',' ⟨attrATG⟩
⟨AgentDesignation⟩ ::= ⟨Agent⟩ | ⟨AgentType⟩ | ⟨AgentGroup⟩
⟨AgentTG⟩ ::= ⟨AgentType⟩ | ⟨AgentGroup⟩
⟨AgeAttIde⟩ ::= ⟨ATG⟩ | ⟨ATClassName⟩ | ⟨AClassName⟩ | ⟨AGClassName⟩
| ⟨AgentDesignation⟩
⟨Agent⟩ ::= var string ⟨idenitiferA⟩
⟨AgentType⟩ ::= var string ⟨identifierAT⟩
⟨AgentGroup⟩ ::= var string ⟨identifierAG⟩
⟨className⟩ ::= string
⟨identifier⟩ ::= instance string
⟨attrNameValue⟩ ::= ⟨attrName⟩ ⟨attrValue⟩
⟨attrName⟩ ::= string

Access Control Rule Definition in BNF (contd.)

⟨attrValue⟩ ::= ⟨relationN⟩ ⟨valueNum⟩ | ⟨relationC⟩ ⟨valueC⟩

⟨identifierA⟩ ::= Agent

⟨identifierAT⟩ ::= AgentType

⟨identifierAG⟩ ::= AgentGroup

⟨relationN⟩ ::= '<' | '>' | '≥' | '≤' | '=' | '≠'

⟨relationC⟩ ::= equals | notEquals

⟨valueNum⟩ ::= ⟨number⟩

⟨number⟩ ::= integer | real

⟨valueC⟩ ::= character | string | ⟨bool⟩

⟨bool⟩ ::= True | False

⟨ATClassName⟩ ::= ⟨identifier⟩ in '('⟨AgentType⟩ '=' ⟨identifier⟩')'

⟨AClassName⟩ ::= ⟨className⟩ in '('⟨Agent⟩ '=' ⟨className⟩')'

⟨AGClassName⟩ ::= ⟨identifier⟩ in '('⟨AgentGroup⟩ '=' ⟨identifier⟩')'

⟨ResourceExp⟩ ::= '('⟨equR⟩')' | '('⟨uop⟩ ⟨equR⟩')' | '('⟨equR⟩ ⟨equRrep⟩')'
| '('⟨uop⟩ ⟨equR⟩ ⟨equRrep⟩')'

⟨equR⟩ ::= ⟨RTG⟩ | ⟨attrRTG⟩

⟨equRrep⟩ ::= ⟨bop⟩ ⟨equR⟩ | ⟨bop⟩ ⟨equR⟩ ⟨equRrep⟩ | ⟨bop⟩ ⟨uop⟩ ⟨equR⟩
| ⟨bop⟩ ⟨uop⟩ ⟨equR⟩ ⟨equRrep⟩ | empty

⟨RTG⟩ ::= ⟨Resource⟩ '=' ⟨className⟩ | ⟨ResourceTG⟩ '=' ⟨identifier⟩

⟨ResourceDesignation⟩ ::= ⟨Resource⟩ | ⟨ResourceType⟩ | ⟨ResourceGroup⟩

⟨ResourceTG⟩ ::= ⟨ResourceType⟩ | ⟨ResourceGroup⟩

⟨Resource⟩ ::= var string ⟨identifierR⟩

⟨ResourceType⟩ ::= var string ⟨identifierRT⟩

⟨ResourceGroup⟩ ::= var string ⟨identifierRG⟩

⟨attrRTG⟩ ::= ⟨ResAttIde⟩'.'⟨attrNameValue⟩ | ⟨ResAttIde⟩'.'⟨attrNameValue⟩
',' ⟨attrRTG⟩

⟨identiferR⟩ ::= Resource

⟨identiferRT⟩ ::= ResourceType

⟨identifierRG⟩ ::= ResourceGroup

⟨ResAttIde⟩ ::= ⟨RTG⟩ | ⟨RTClassName⟩ | ⟨RClassName⟩ | ⟨RGClassName⟩ |
⟨ResourceDesignation⟩

Access Control Rule Definition in BNF (contd.)

⟨RTClassName⟩ ::= ⟨identifier⟩ in '('⟨ResourceType⟩ '=' ⟨identifier⟩')'
⟨RClassname⟩ ::= ⟨className⟩ in '('⟨Resource⟩ '=' ⟨className⟩')'
⟨RGClassname⟩ ::= ⟨identifier⟩ in '('⟨ResourceGroup⟩ '=' ⟨identifier⟩')'
⟨EventExp⟩ ::= '('⟨ETG⟩')' | '('⟨uop⟩ ⟨ETG⟩')' | '(⟨ETG⟩ ⟨ETGrep⟩')' |
    '('⟨uop⟩ ⟨ETG⟩ ⟨ETGrep⟩')'
⟨ETG⟩ ::= ⟨Event⟩ '=' ⟨className⟩ | ⟨EventTG⟩ '=' ⟨identifier⟩
⟨ETGrep⟩ ::= ⟨bop⟩ ⟨ETG⟩ | ⟨bop⟩ ⟨ETG⟩ ⟨ETGrep⟩ | ⟨bop⟩ ⟨uop⟩ ⟨ETG⟩ |
    ⟨bop⟩ ⟨uop⟩ ⟨ETG⟩ ⟨ETGrep⟩ | empty
⟨EventDesignation⟩ ::= ⟨Event⟩ | ⟨EventType⟩ | ⟨EventGroup⟩
⟨EventTG⟩ ::= ⟨EventType⟩ | ⟨EventGroup⟩
⟨Event⟩ ::= var string ⟨identifierE⟩
⟨EventType⟩ ::= var string ⟨identifierET⟩
⟨EventGroup⟩ ::= var string ⟨identifierEG⟩
⟨identifierE⟩ ::= Event
⟨identifierET⟩ ::= EventType
⟨identifierEG⟩ ::= EventGroup
⟨AgeEveRel⟩ ::= '('⟨AgeERel⟩ ')' | '('⟨uop⟩ ⟨AgeERel⟩')' | '('⟨AgeERel⟩
    ⟨AgeERelRep⟩')' | '('⟨uop⟩ ⟨AgeERel⟩ ⟨AgeERelRep⟩')'
⟨AgeERel⟩ ::= RelATET '('⟨ATClassName⟩ ',' ⟨ETClassName⟩')'
    | RelAE '('⟨AClassName⟩ ',' ⟨EClassName⟩')'
    | RelATG '('⟨ATClassName⟩ ','⟨AGClassName⟩ ')'
    | RelAT '('⟨AClassName⟩ ',' ⟨ATClassName⟩')'
    | RelAG '('⟨AClassName⟩ ',' ⟨AGClassName⟩ ')'
⟨AgeERelRep⟩ ::= ⟨bop⟩ ⟨AgeERel⟩ | ⟨bop⟩ ⟨AgeERel⟩ ⟨AgeERelrep⟩ | ⟨bop⟩ ⟨uop⟩
    ⟨AgeERel⟩ | ⟨bop⟩ ⟨uop⟩ ⟨AgeERel⟩ ⟨AgeERelRep⟩ | empty
⟨ResEveRel⟩ ::= '('⟨ResERel⟩')' | '('⟨uop⟩ ⟨ResERel⟩')' | '('⟨ResERel⟩
    ⟨ResERelRep⟩')' | '('⟨uop⟩ ⟨ResERel⟩ ⟨ResERelRep⟩')'
⟨ResERel⟩ ::= RelRTET '('⟨RTClassName⟩ ',' ⟨ETClassName⟩')'
    | RelRE '('⟨RClassName⟩ ',' ⟨EClassName⟩')'
    | RelRTG '('⟨RTClassName⟩ ',' ⟨RGClassName⟩')'

Access Control Rule Definition in BNF (contd.)

| RelRT '('⟨RClassName⟩ ',' ⟨RTClassName⟩')'
| RelRG '('⟨RClassName⟩ ',' ⟨RGClassName⟩')'
⟨ResERelRep⟩ ::= ⟨bop⟩ ⟨ResERel⟩ | ⟨bop⟩ ⟨ResERel⟩ ⟨ResERelrep⟩ | ⟨bop⟩ ⟨uop⟩ ⟨ResERel⟩ | ⟨bop⟩ ⟨uop⟩ ⟨ResERel⟩ ⟨ResERelRep⟩ | empty
⟨EventResult⟩ ::= ⟨accETG⟩ | ⟨accETG⟩ ⟨accETGrep⟩ | ⟨uop⟩ ⟨accETG⟩
| ⟨uop⟩ ⟨accETG⟩ ⟨accETGrep⟩
⟨accETG⟩ ::= ⟨accessETG⟩ '=' ⟨result⟩
⟨accETGrep⟩ ::= ⟨bop⟩ ⟨accETG⟩ | ⟨bop⟩ ⟨accETG⟩ ⟨accETGrep⟩
| ⟨bop⟩ ⟨uop⟩ ⟨accETG⟩
| ⟨bop⟩ ⟨uop⟩ ⟨accETG⟩ ⟨accETGrep⟩ | empty
⟨result⟩ ::= permit | deny
⟨accessETG⟩ ::= ⟨ETClassName⟩ Access | ⟨EClassName⟩ Access
⟨ETClassName⟩ ::= ⟨identifier⟩ in '('⟨EventType⟩ '=' ⟨identifier⟩')'
⟨EClassName⟩ ::= ⟨className⟩ in '('⟨Event⟩ '=' ⟨className⟩')'

Figure B.1: Access control rule definition in BNF

B.2 Other BNF and EBNF Definitions

This section provides agent-, resource-, and event-related definitions. Figures B.2 and B.3 show agent-related definitions in extended BNF and BNF, respectively. These two figures use symbols provided in Tables 3.7 and B.1. The same explanation also holds when resource- and event-related definitions are provided.

The agent-, resource-, and event-related definitions (i.e., the grammar) in BNFs (Figures B.3, B.5, and B.7) are checked with a tool called Gold Parser [24]. As mentioned previously, this tool requires the symbols such as parenthesis, arithmetic equalities or inequalities (e.g., > or =), and commas to be enclosed by single quotation marks. Therefore, these symbols are enclosed by single quotation marks in the figures identified as BNF. In addition, the Gold Parser requires that definitions start with "start symbol"; therefore, when checking the definitions in Figures B.3, B.5, and B.7, the first rule in these figures must be as follows: "Start Symbol" = ⟨modelDef⟩. The last statement also applies for all figures that are identified as BNF definitions.

After providing agent-related definitions in Figures B.2 and B.3, the definitions are

described for both figures together by identifying the corresponding starting line of each definition in these two figures. The definitions and their descriptions for resource- and event-related definitions are very similar to the one provided for agent-related definition.

Description of Agent-related definitions: Figures B.2 and B.3 are described by identifying the corresponding starting line of each definition in these two figures.

- The expression starting with modelDef or ⟨modelDef⟩: This expression defines modleDef as a class or an instance of a class (i.e., Iclass) or an association.
- The expression starting with class or ⟨class⟩: This expression identifies a class with its components: attribute names and types (i.e., attrNameType), operation names and types (if they exist), and operation return values (if they exist).
- The expression starting with AgentDesignation or ⟨AgentDesignation⟩: An AgentDesignation can be one of the following three elements: agent, agent type, or agent group.
- The expression starting with className or ⟨className⟩: The class name is a string.
- The expression starting with attrNameType or ⟨attrNameType⟩: One or more attributes can exist, and each attribute has a name and a type.
- The expression starting with attrName or ⟨attrName⟩: An *attrName* is a string.
- The expression starting with type or ⟨type⟩: A type can be a basic type or a class.
- The expression starting with basicType or ⟨basicType⟩: A basic type can be a number, character, string, or bool.
- The expression starting with opNTR or ⟨opNTR⟩: An operation has a name and can include optional elements of paramNameType and resType.
- The expression starting with Agent or ⟨Agent⟩: An agent is defined as a variable that holds a value of a string type within identifierA.
- The expression starting with AgentType or ⟨AgentType⟩:: An agent type is defined as a variable that holds a value of a string type within identifierAT.
- The expression starting with AgentGroup or or ⟨AgentGroup⟩:: An agent group is defined as a variable that holds a value of a string type within identifierAG.

Agent-related Definitions in Extended BNF

```
modelDef = class | Iclass | association;
class = ([AgentDesignation "="] className "," attrNameType { "," attrNameValue}
        { "," opNTR});
AgentDesignation = Agent | AgentType | AgentGroup;
className = "string";
attrNameType = (attrName ":" type) { (attrName ":" type) };
attrName = "string";
type = basicType | class;
basicType = number | "character" | "string" | bool;
opNTR = opName([paramNameType]) [resType];
Agent = "var" "string" identifierA;
AgentType = "var" "string" identifierAT;
AgentGroup = "var" "string" identifierAG;
opName = "string";
paramNameType = (paramName ":" type) { (paramName ":" type) };
paramName = "string";
resType = type;
identifierA = "Agent";
identifierAT = "AgentType";
identifierAG = "AgentGroup";
Iclass = ("object" [AgentDesignation "="] objectName":"className
          attrNameValue { "," opNTR});
objectName = "string";
attrNameValue = (attrName attrValue) { (attrName attrValue) };
attrValue = relationN valueNum | relationC valueC;
relationN = "<" | ">" | "≥" | "≤" | "=" | "≠";
relationC = "equlas" | notEquals;
valueNum = number;
number = "integer" | "real";
valueC = "character" | "string" | bool;
```

Agent-related Definitions in Extended BNF (contd.)

```
bool =  "True" | "False";
association =  "association"  associationName  Agent  className
               multiplicity  AgentTG  className  multiplicity
               | "association"  associationName  AgentType  className
               multiplicity  AgentGroup  className  multiplicity;
associationName =  "string";
multiplicity =  [lower".."]upper;
AgentTG =  AgentType | AgentGroup;
lower =  "integer";
upper =  "*" | "integer";
```

Figure B.2: Agent-related definitions in Extended BNF

Agent-related Definitions in BNF

```
⟨modelDef⟩ ::=  ⟨class⟩ | ⟨Iclass⟩ | ⟨ association⟩
⟨class⟩ ::=  '('class ⟨AgentDesignation⟩ '=' ⟨className⟩ ',' ⟨attrNameType⟩
             ⟨attrNameValue⟩ ',' ⟨opNTR⟩')' | '('class ⟨className⟩ ','
             ⟨attrNameType⟩ ⟨attrNameValue⟩ ',' ⟨opNTR⟩')'
⟨AgentDesignation⟩ ::=  ⟨Agent⟩ | ⟨AgentType⟩ | ⟨AgentGroup⟩
⟨className⟩ ::=  string
⟨attrNameType⟩ ::=  '('⟨attrName⟩ ':' ⟨type⟩')' | '('⟨attrName⟩ ':' ⟨type⟩')'
                    ⟨attrNameType⟩
⟨attrName⟩ ::=  string
⟨type⟩ ::=  ⟨basicType⟩ | ⟨class⟩
⟨basicType⟩ ::=  ⟨number⟩ | character | string | ⟨bool⟩
⟨opNTR⟩ ::=  ⟨opName⟩'()' | ⟨opName⟩'('⟨paramNameType⟩')'
             | ⟨opName⟩'('⟨paramNameType⟩')' ⟨resType⟩
             | ⟨opName⟩'()' ⟨resType⟩ | ⟨opName⟩'()' ⟨opNTR⟩
             | ⟨opName⟩'('⟨paramNameType⟩')' ⟨opNTR⟩
```

Agent-related Definitions in BNF (contd.)

```
                | ⟨opName⟩'()' ⟨resType⟩ ⟨opNTR⟩
                | ⟨opName⟩'('⟨paramNameType⟩')' ⟨resType⟩ ⟨opNTR⟩ | empty
⟨Agent⟩ ::= var string ⟨identifierA⟩
⟨AgentType⟩ ::= var string ⟨identifierAT⟩
⟨AgentGroup⟩ ::= var string ⟨identifierAG⟩
⟨opName⟩ ::= string
⟨paramNameType⟩ ::= '('⟨paramName⟩ ':' ⟨type⟩')' | '('⟨paramName⟩ ':' ⟨type⟩')'
                    ⟨paramNameType⟩ | empty
⟨paramName⟩ ::= string
⟨resType⟩ ::= ⟨type⟩
⟨identifierA⟩ ::= Agent
⟨identifierAT⟩ ::= AgentType
⟨identifierAG⟩ ::= AgentGroup
⟨Iclass⟩ ::= '('object ⟨AgentDesignation⟩ '=' ⟨objectName⟩':'⟨className⟩
             ⟨attrNameValue⟩ ',' ⟨opNTR⟩')' | '('object ⟨objectName⟩':'⟨className⟩
             ⟨attrNameValue⟩ ',' ⟨opNTR⟩')'
⟨objectName⟩ ::= string
⟨attrNameValue⟩ ::= ',' ⟨attrName⟩ ⟨attrValue⟩ ⟨attrNameValue⟩ | empty
⟨attrValue⟩ ::= ⟨relationN⟩ ⟨valueNum⟩ | ⟨relationC⟩ ⟨valueC⟩
⟨relationN⟩ ::= '<' | '>' | '≥' | '≤' | '=' | '≠'
⟨relationC⟩ ::= equals | notEquals
⟨valueNum⟩ ::= ⟨number⟩
⟨number⟩ ::= integer | real
⟨valueC⟩ ::= character | string | ⟨bool⟩
⟨bool⟩ ::= True | False
⟨association⟩ ::= association ⟨associationName⟩ ⟨Agent⟩ ⟨className⟩
                  ⟨multiplicity⟩ ⟨AgentTG⟩ ⟨className⟩ ⟨multiplicity⟩
                  | association ⟨associationName⟩ ⟨AgentType⟩ ⟨className⟩
                  ⟨multiplicity⟩ ⟨AgentGroup⟩ ⟨className⟩ ⟨multiplicity⟩
```

Agent-related Definitions in BNF (contd.)
⟨associationName⟩ ::= string
⟨multiplicity⟩ ::= ⟨lower⟩'..'⟨upper⟩ | ⟨upper⟩
⟨AgentTG⟩ ::= ⟨AgentType⟩ | ⟨AgentGroup⟩
⟨lower⟩ ::= integer
⟨upper⟩ ::= '*' | integer

Figure B.3: Agent-related definitions in BNF

- The expression starting with opName or ⟨opName⟩: An operation name (*opName*) is a string.
- The expression starting with paramNameType or ⟨opNTR⟩: One or more paramNameType can exist.
- The expression starting with paramName or ⟨paramName⟩: A *paramName* is a string.
- The expression starting with resType or ⟨resType⟩: A result type (*resType*) is a type.
- The expression starting with identifierA: An agent identifier is recognized by the word Agent.
- The expression starting with identifierAT: An agent type identifier is recognized by the word AgentType.
- The expression starting with identifierAG: An agent group is recognized by the word AgentGroup.
- The expression starting with Iclass or ⟨Iclass⟩: An object with its components: attribute names and values, operation names and types, and return values.
- The expression starting with objectName or ⟨objectName⟩: An object name is a string.
- The expression starting with attrNameValue or ⟨attrNameValue⟩: An attribute has a name and a value.
- The expression starting with attrValue or ⟨attrValue⟩: Attribute values are defined as numbers or strings. Relational symbols (*relationN* or *relationC*) are also included for both numbers and strings.

- The expression starting with relationN or ⟨relationN⟩: relationN (relation for numbers) is one of $<$, $>$, $<$, $\geq$, $\leq$, and $=$ symbols for comparing numbers.
- The expression starting with relationC or ⟨relationC⟩: relationC (relation for characters) can be used to compare equalities of characters and strings.
- The expression starting with valueNum or ⟨valueNum⟩: *ValueNum* is a number.
- The expression starting with valueC or ⟨valueC⟩: *valueC* is any of character, string, or bool.
- The expression starting with bool or ⟨bool⟩: bool is either True or False.
- The expression starting with association or ⟨association⟩: This expression identifies the existence of associations between agent and agent type, agent and agent group, and agent type and agent group. Associations have multiplicities on each end.
- The expression starting with associationName or ⟨associationName⟩: An association name is a string.
- The expression starting with multiplicity or ⟨multiplicity⟩: Each association-end can have a single element as a multiplicity (i.e., upper) or can have a range identified with lower and upper limits and two dots in between.
- The expression starting with AgentTG or ⟨AgentTG⟩: AgentTG is defined to be either agent type or agent group.
- The expression starting with lower or ⟨lower⟩: *lower* is an integer.
- The expression starting with lower or ⟨lower⟩: The *upper* range is an integer or many (the symbol star means zero or more).

Similarly, resource-related definitions in both extended BNF and BNF in Figures B.4 and B.5 are provided, respectively.

Resource-related Definitions in Extended BNF

```
modelDef = class | Iclass | association;
class = ([ResourceDesignation "="] className "," attrNameType
        { "," attrNameValue} { "," opNTR})
ResourceDesignation = Resource | ResourceType | ResourceGroup;
className = "string";
attrNameType = (attrName ":" type) { (attrName ":" type) };
attrName = "string";
type = basicType | class;
basicType = number | "character" | "string" | bool;
opNTR = opName([paramNameType]) [resType];
Resource = "var" "string" identifierR;
ResourceType = "var" "string" identifierRT;
ResourceGroup = "var" "string" identifierRG;
opName = "string";
paramNameType = (paramName ":" type) { (paramName ":" type) };
paramName = "string";
resType = type;
identifierR = "Resource";
identifierRT = "ResourceType";
identifierRG = "ResourceGroup";
Iclass = ("object" [ResourceDesignation "="] objectName":"className
         attrNameValue { "," opNTR});
objectName = "string";
attrNameValue = (attrName attrValue) { (attrName attrValue) };
attrValue = relationN valueNum | relationC valueC;
relationN = "<" | ">" | "≥" | "≤" | "=" | "≠";
relationC = "equlas" | notEquals;
valueNum = number;
```

Resource-related Definitions in Extended BNF (contd.)

```
number =  "integer"  |  "real";
valueC =  "character"  |  "string"  |  bool;
bool =  "True"  |  "False";
association =  "association"  associationName  Resource  className
               multiplicity  ResourceTG  className  multiplicity
               |  "association"  associationName  ResourceType  className
               multiplicity  ResourceGroup  className  multiplicity;
associationName =  "string";
multiplicity =  [lower".."]upper;
ResourceTG =  ResourceType  |  ResourceGroup;
lower =  "integer";
upper =  "*"  |  "integer";
```

Figure B.4: Resource-related definitions in Extended BNF

Resource-related Definitions in BNF

```
⟨modelDef⟩ ::=  ⟨class⟩ | ⟨Iclass⟩ | ⟨ association⟩
⟨class⟩ ::=  '('class ⟨ResourceDesignation⟩ '=' ⟨className⟩ ',' ⟨attrNameType⟩
             ⟨attrNameValue⟩ ',' ⟨opNTR⟩')' | '('class ⟨className⟩ ','
             ⟨attrNameType⟩ ⟨attrNameValue⟩ ',' ⟨opNTR⟩')'
⟨ResourceDesignation⟩ ::= ⟨Resource⟩ | ⟨ResourceType⟩ | ⟨ResourceGroup⟩
⟨className⟩ ::=  string
⟨attrNameType⟩ ::=  '('⟨attrName⟩ ':' ⟨type⟩')' | '('⟨attrName⟩ ':' ⟨type⟩')'
                    ⟨attrNameType⟩
⟨attrName⟩ ::=  string
⟨type⟩ ::= ⟨basicType⟩ | ⟨class⟩
⟨basicType⟩ ::= ⟨number⟩ | character | string | ⟨bool⟩
```

Resource-related Definitions in BNF (contd.)

```
⟨opNTR⟩ ::= ⟨opName⟩'()' | ⟨opName⟩'('⟨paramNameType⟩')'
          | ⟨opName⟩'('⟨paramNameType⟩')' ⟨resType⟩
          | ⟨opName⟩'()' ⟨resType⟩ | ⟨opName⟩'()' ⟨opNTR⟩
          | ⟨opName⟩'('⟨paramNameType⟩')' ⟨opNTR⟩
          | ⟨opName⟩'()' ⟨resType⟩ ⟨opNTR⟩
          | ⟨opName⟩'('⟨paramNameType⟩')' ⟨resType⟩ ⟨opNTR⟩ | empty
⟨Resource⟩ ::= var string ⟨identifierR⟩
⟨ResourceType⟩ ::= var string ⟨identifierRT⟩
⟨ResourceGroup⟩ ::= var string ⟨identifierRG⟩
⟨opName⟩ ::= string
⟨paramNameType⟩ ::= '('⟨paramName⟩ ':' ⟨type⟩')' | '('⟨paramName⟩ ':' ⟨type⟩')'
                    ⟨paramNameType⟩ | empty
⟨paramName⟩ ::= string
⟨resType⟩ ::= ⟨type⟩
⟨identifierR⟩ ::= Resource
⟨identifierRT⟩ ::= ResourceType
⟨identifierRG⟩ ::= ResourceGroup
⟨Iclass⟩ ::= '('object ⟨ResourceDesignation⟩ '=' ⟨objectName⟩':'⟨className⟩
             ⟨attrNameValue⟩ ',' ⟨opNTR⟩')' | '('object ⟨objectName⟩':'⟨className⟩
             ⟨attrNameValue⟩ ',' ⟨opNTR⟩')'
⟨objectName⟩ ::= string
⟨attrNameValue⟩ ::= ',' ⟨attrName⟩ ⟨attrValue⟩ ⟨attrNameValue⟩ | empty
⟨attrValue⟩ ::= ⟨relationN⟩ ⟨valueNum⟩ | ⟨relationC⟩ ⟨valueC⟩
⟨relationN⟩ ::= '<' | '>' | '≥' | '≤' | '=' | '≠'
⟨relationC⟩ ::= equals | notEquals
⟨valueNum⟩ ::= ⟨number⟩
⟨number⟩ ::= integer | real
⟨valueC⟩ ::= character | string | ⟨bool⟩
⟨bool⟩ ::= True | False
```

Resource-related Definitions in BNF (contd.)

⟨association⟩ ::= association ⟨associationName⟩ ⟨Resource⟩ ⟨className⟩
⟨multiplicity⟩ ⟨ResourceTG⟩ ⟨className⟩ ⟨multiplicity⟩
| association ⟨associationName⟩ ⟨ResourceType⟩ ⟨className⟩
⟨multiplicity⟩ ⟨ResourceGroup⟩ ⟨className⟩ ⟨multiplicity⟩
⟨associationName⟩ ::= string
⟨multiplicity⟩ ::= ⟨lower⟩'..'⟨upper⟩ | ⟨upper⟩
⟨ResourceTG⟩ ::= ⟨ResourceType⟩ | ⟨ResourceGroup⟩
⟨lower⟩ ::= integer
⟨upper⟩ ::= '*' | integer

Figure B.5: Resource-related definitions in BNF

Figures B.6 and B.7 show event-related definitions in both extended BNF and BNF, respectively.

Event-related Definitions in Extended BNF

```
modelDef =  class  |  Iclass  |  association;
class =  ([EventDesignation  "="]  className ","  attrNameType  { "," attrNameValue}
           { ","   opNTR});
EventDesignation =  Event  |  EventType  |  EventGroup;
className =  "string";
attrNameType =  (attrName ":" type)  { (attrName ":" type) };
attrName =  "string";
type = basicType  |  class;
basicType =  number  |  "character"  |  "string"  |  bool;
opNTR =  opName([paramNameType])  [resType];
Event =  "var"  "string"  identifierE;
EventType =  "var"  "string"  identifierET;
EventGroup =  "var"  "string"  identifierEG;
opName =  "string";
paramNameType =  (paramName ":" type)  { (paramName ":" type) };
paramName =  "string";
resType =  type;
identifierE =  "Event";
identifierET =  "EventType";
identifierEG =  "EventGroup";
Iclass =  ("object"  [EventDesignation "="]  objectName":"className
             attrNameValue  { ","   opNTR});
objectName =  "string";
attrNameValue =  (attrName   attrValue) { (attrName   attrValue) };
attrValue =  relationN  valueNum  |  relationC  valueC;
relationN =  "<"  |  ">"  |  "≥"  |  "≤"  |  "="  |  "≠";
relationC =  "equlas"  |  notEquals;
valueNum =  number;
```

Event-related Definitions in Extended BNF (contd.)

```
number =  "integer"  |  "real";
valueC =  "character"  |  "string"  |  bool;
bool =  "True"  |  "False";
association =  "association"  associationName  Event  className
               multiplicity  EventTG  className  multiplicity
               |  "association"  associationName  EventType  className
               multiplicity  EventGroup  className  multiplicity;
associationName =  "string";
multiplicity =  [lower".."]upper;
EventTG =  EventType  |  EventGroup;
lower =  "integer";
upper =  "*"  |  "integer";
```

Figure B.6: Event definitions in Extended BNF

Event-related Definitions in BNF

```
⟨modelDef⟩ ::=  ⟨class⟩ | ⟨Iclass⟩ | ⟨ association⟩
⟨class⟩ ::=  '('class ⟨EventDesignation⟩ '=' ⟨className⟩ ',' ⟨attrNameType⟩
             ⟨attrNameValue⟩ ',' ⟨opNTR⟩')' | '('class ⟨className⟩ ','
             ⟨attrNameType⟩ ⟨attrNameValue⟩ ',' ⟨opNTR⟩')'
⟨EventDesignation⟩ ::=  ⟨Event⟩ | ⟨EventType⟩ | ⟨EventGroup⟩
⟨className⟩ ::=  string
⟨attrNameType⟩ ::=  '('⟨attrName⟩ ':' ⟨type⟩')' | '('⟨attrName⟩ ':' ⟨type⟩')'
                    ⟨attrNameType⟩
⟨attrName⟩ ::=  string
⟨type⟩ ::=  ⟨basicType⟩ | ⟨class⟩
⟨basicType⟩ ::=  ⟨number⟩ | character | string | ⟨bool⟩
```

Event-related Definitions in BNF (contd.)

```
⟨opNTR⟩ ::= ⟨opName⟩'()' | ⟨opName⟩'('⟨paramNameType⟩')'
          | ⟨opName⟩'('⟨paramNameType⟩')' ⟨resType⟩
          | ⟨opName⟩'()' ⟨resType⟩ | ⟨opName⟩'()' ⟨opNTR⟩
          | ⟨opName⟩'('⟨paramNameType⟩')' ⟨opNTR⟩
          | ⟨opName⟩'()' ⟨resType⟩ ⟨opNTR⟩
          | ⟨opName⟩'('⟨paramNameType⟩')' ⟨resType⟩ ⟨opNTR⟩ | empty
⟨Event⟩ ::= var string ⟨identifierE⟩
⟨EventType⟩ ::= var string ⟨identifierET⟩
⟨EventGroup⟩ ::= var string ⟨identifierEG⟩
⟨opName⟩ ::= string
⟨paramNameType⟩ ::= '('⟨paramName⟩ ':' ⟨type⟩')' | '('⟨paramName⟩ ':' ⟨type⟩')'
                    ⟨paramNameType⟩ | empty
⟨paramName⟩ ::= string
⟨resType⟩ ::= ⟨type⟩
⟨identifierE⟩ ::= Event
⟨identifierET⟩ ::= EventType
⟨identifierEG⟩ ::= EventGroup
⟨Iclass⟩ ::= '('object ⟨EventDesignation⟩ '=' ⟨objectName⟩':'⟨className⟩
             ⟨attrNameValue⟩ ',' ⟨opNTR⟩')' | '('object ⟨objectName⟩':'⟨className⟩
             ⟨attrNameValue⟩ ',' ⟨opNTR⟩')'
⟨objectName⟩ ::= string
⟨attrNameValue⟩ ::= ',' ⟨attrName⟩ ⟨attrValue⟩ ⟨attrNameValue⟩ | empty
⟨attrValue⟩ ::= ⟨relationN⟩ ⟨valueNum⟩ | ⟨relationC⟩ ⟨valueC⟩
⟨relationN⟩ ::= '<' | '>' | '≥' | '≤' | '=' | '≠'
⟨relationC⟩ ::= equals | notEquals
⟨valueNum⟩ ::= ⟨number⟩
⟨number⟩ ::= integer | real
⟨valueC⟩ ::= character | string | ⟨bool⟩
⟨bool⟩ ::= True | False
```

Event-related Definitions in BNF (contd.)

⟨association⟩ ::= association ⟨associationName⟩ ⟨Event⟩ ⟨className⟩
⟨multiplicity⟩ ⟨EventTG⟩ ⟨className⟩ ⟨multiplicity⟩
| association ⟨associationName⟩ ⟨EventType⟩ ⟨className⟩
⟨multiplicity⟩ ⟨EventGroup⟩ ⟨className⟩ ⟨multiplicity⟩
⟨associationName⟩ ::= string
⟨multiplicity⟩ ::= ⟨lower⟩'..'⟨upper⟩ | ⟨upper⟩
⟨EventTG⟩ ::= ⟨EventType⟩ | ⟨EventGroup⟩
⟨lower⟩ ::= integer
⟨upper⟩ ::= '*' | integer

Figure B.7: Event-related definitions in BNF

Appendix C

A Brief Background on Logic

This Appendix is a background on logic. This background is provided because some in the REA community may have a preference for such an overview. Therefore, this appendix describes an overview of *propositional logic*, *predicate logic*, and *linear temporal logic*. This description is mainly based on *Logic in Computer Science: Modelling and Reasoning about Systems* [49] unless another source is cited.

C.1 Propositional Logic

Propositional logic, as the name indicates, is about *propositions* or *declarative sentences*. Declarative sentences can be validated either true or false.

Syntax: The syntax of well-formed formulas in propositional logic, described in Backus Naur Form (BNF), follows:

$$\phi \ ::= \ p \mid (\neg\phi) \mid (\phi \wedge \psi) \mid (\phi \vee \psi) \mid (\phi \rightarrow \psi)$$

where p is any atomic proposition, and ϕ, on the right side of ::=, and ψ represent a formula that has already been constructed according to these rules; $\neg$ (negation), $\wedge$ (and), $\vee$ (or), and $\rightarrow$ (implies) are connectives. An atomic proposition, p, is also a formula.

Semantics: Semantics provide interpretation for well-formed propositional logic syntax. This interpretation is based on the assignment of True (T) or False (F) values to atomic prepositions and formulas. A valuation (v) of a formula (ϕ) is written as $v(\phi)$. This valuation is defined as follows [30]:

$$v(\neg\phi) = \begin{cases} T & \text{if} \quad v(\phi) \ = \ F, \\ F & \text{otherwise.} \end{cases}$$

$$v(\phi \ \wedge \ \psi) = \begin{cases} T & \text{if} \quad v(\phi) \ = \ T \quad \text{and} \quad v(\psi) \ = \ T, \\ F & \text{otherwise.} \end{cases}$$

$$v(\phi \ \vee \ \psi) = \begin{cases} T & \text{if} \quad v(\phi) \ = \ T \quad \text{or} \quad v(\psi) \ = \ T, \\ F & \text{otherwise.} \end{cases}$$

$$v(\phi \ \rightarrow \ \psi) = \begin{cases} F & \text{if} \quad v(\phi) \ = \ T \quad \text{and} \quad v(\psi) \ = \ F, \\ T & \text{otherwise.} \end{cases}$$

Example C.1.1: Does the expression $p \rightarrow p \vee q$ hold?

Because of the semantics of implication, this example holds since in this case whenever $v(p) = T$ then $v(p \vee q) = T$ because of the semantics of $\vee$. Similarly, because of the semantics of implication, if $v(p) = F$, then regardless of true of false evaluation of $(p \vee q)$, the implication holds. Finally, it is not possible to have a case in which $v(p) = T$ and $(p \vee q) = F$. Therefore, $p \rightarrow p \vee q$ holds.

Example C.1.2: Does the expression $p \vee q \ \rightarrow \ p$ hold?

Similar to the previous example, but this implication does not necessarily hold. For instance, if $v(p \vee q) = T$ then according to *or*'s semantics, there can be a case in which $v(q) = T$ and $v(p) = F$; as a result, the valuation of antecedence is true, but the valuation of consequence is false. Therefore, this implication does not hold.

Example C.1.3: Does the expression $(p \rightarrow q) \rightarrow (\neg p \vee q)$ hold?

Because of the implication's semantics, $p \rightarrow q$ is true in all interpretations except when the value of p is true ($v(p) = T$) and the value of q is false ($v(q) = F$). Similarly, because of the semantics of *negation* and *or*, the valuation of $\neg p \vee q$ is also true in all interpretations except in a case in which the value of p is true ($v(p) = T$) and the value of q is false ($v(q) = F$).

$(p \rightarrow q)$ and $(\neg p \vee q)$ are logically equivalent, $(p \rightarrow q) \equiv (\neg p \vee q)$, because their values are the same under all interpretations.

Example C.1.4: Does the expression $(\neg p \vee q) \rightarrow (p \rightarrow q)$ hold? This example is similar to C.1.3.

C.2 Predicate Logic

Two extensions of predicate logic to propositional logic are *quantifiers* and *functions.*

Syntax: *terms* are used in the syntax definition of predicate logic; therefore, terms are first defined. In BNF, terms (t) are defined as follows:

$$t ::= x \ | \ c \ | \ \mathrm{f}(\mathrm{t}_1, \ldots, t_n)$$

where x is a variable and ranges over a set of variables var, c stands for a constant. f is a function over a set of function symbols $\mathcal{F}$ with an arity of $n > 0$.

The syntax of predicate logic, which uses previously defined terms, follows:

$$\phi \ ::= \ P(t_1, t_2, \ldots, t_n) \ | \ (\neg\phi) \ | \ (\phi \wedge \phi) \ | \ (\phi \vee \phi) \ | \ (\phi \rightarrow \phi) \ | \ (\forall x \ \phi) \ | \ (\exists x \ \phi)$$

where $P \in \mathcal{P}$ and is a predicate symbol of arity $n > 0$, and $(t_1, t_2, \ldots, t_n)$ are terms. x is a variable, and the quantifiers $\forall$ and $\exists$ mean "for all" and "there exists or for some," respectively. The occurrence of ϕ on the right side of ::= represents any formula that has already been constructed using the rules shown here.

Semantics: The semantics or *interpretation* consists of a pair $\mathcal{I} = (\mathcal{U}_\mathcal{I}, \ \mathcal{A}_\mathcal{I})$. $\mathcal{U}_\mathcal{I}$ is a non-empty set called *domain* or *universe.* $\mathcal{A}_\mathcal{I}$ is defined for every predicate, function, and variable.

If F and G are formulas and $\mathcal{I}$ is an interpretation, then the following holds [106]:

- $\mathcal{I}(x) = x^\mathcal{I}$
- $\mathcal{I}(f(t_1, \ldots, t_k) = f^\mathcal{I}(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_k))$
- $\mathcal{I}(p(t_1, \ldots, t_k)$ = true if $(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_k)) \in p^\mathcal{I}$, false otherwise
- The interpretation of logical connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$ are identical to propositional logic. For instance, to evaluate $\mathcal{I}(F \wedge G)$, one evaluates sub-formulas F and G and then applies the interpretation of these connectives.
- $\mathcal{I}(\forall F)$ = true for every $d \in U : \mathcal{I}_{[x/d]}(F) = true$, false otherwise. (The notation $[x/d]$ means replacing x with d.)
- $\mathcal{I}(\exists F)$ = true if there is a $d \in U : \mathcal{I}_{[x/d]}(F) = true$, false otherwise.

Example C.2.1: Does the expression $\exists x\ p(x)\ \rightarrow\ \forall x\ p(x)$ hold?

This example does not hold because of the meanings of $\forall$ and $\exists$. The semantics provided for $\exists$ defines "$\mathcal{I}(\exists F)$ = true if **there is a** $d \in U : \mathcal{I}_{[x/d]}(F) = true$, false otherwise." (The notation $[x/d]$ means replacing x with d; F is a formula.) Conversely, the meaning provided for $\forall$ defines "$\mathcal{I}(\forall F)$ = true for **every** $d \in U : \mathcal{I}_{[x/d]}(F) = true$." Therefore, if there is a d in which $I(F) = T$, then it does not mean that for all d, $I(F) = T$. As a result, this implication does not hold.

Example C.2.2: Does the expression $\forall x\ p(x)\ \rightarrow\ \exists x\ p(x)$ hold?

This example holds. A similar explanation to example C.2.1 applies. The meaning provided for $\forall$ defines "$\mathcal{I}(\forall F)$ = true for **every** $d \in U : \mathcal{I}_{[x/d]}(F) = true$." (The notation $[x/d]$ means replacing x with d; F is a formula.) The semantics provided for $\exists$ defines "$\mathcal{I}(\exists F)$ = true if **there is a** $d \in U : \mathcal{I}_{[x/d]}(F) = true$, false otherwise." Therefore, if for all d, $I(F) = T$, then there is a d in which $I(F) = T$. As a result, this implication holds.

Example C.2.3: Does the expression $\neg(p(x)\ \wedge\ q(x))\ \rightarrow (\neg p(x)\ \vee\ \neg q(x))$ hold?

If $\neg(p(x)\ \wedge\ q(x))$ holds, then because of the semantics of $\neg$ and $\wedge$, either $p(x)$ or $q(x)$ does not hold, or none of them holds. If $p(x)$ does not hold, then because of the semantics of negation, $\neg p(x)$ holds; then the right-hand side $(\neg p(x)\ \vee\ \neg q(x))$ also holds because of the semantics of $\vee$. If $q(x)$ does not hold, then $\neg q(x)$ holds; as a result, the right-hand side of the implication holds. Similarly, if neither $p(x)$ nor $q(x)$ holds, because of negation, both $\neg p(x)$ and $\neg q(x)$ hold. As a result, the right-hand side holds because of the semantics of $\vee$.

Therefore, $\neg(p(x)\ \wedge\ q(x))\ \rightarrow (\neg p(x)\ \vee\ \neg q(x))$ holds.

Note: $(\neg p(x))\ \vee\ (\neg q(x))\ \rightarrow \neg(p(x)\ \wedge\ q(x))$ is similar.

Example C.2.4: Does the expression $\exists y\ \forall x\ p(x, y)\ \rightarrow\ \forall x\ \exists y\ p(x, y)$ hold?

1) $\exists y\ \forall x\ p(x, y)$

2) $\forall x\ p(x, y)[y/y_0]$ Line 1 to line 2 uses the previously mentioned semantics of $\exists$: $\mathcal{I}(\exists F)$ = true if there is a $d \in U : \mathcal{I}_{[x/d]}(F) = true$, false otherwise. (The notation $[x/d]$ means replacing x with d.)

3) $p(x, y)[y/y_0][x/d]$ Line 2 to line 3 uses the previously mentioned semantics of $\forall$: $\mathcal{I}(\forall F)$ = true for every $d \in U : \mathcal{I}_{[x/d]}(F) = true$, false otherwise.

4) $p(x, y)[x/d][y/y_0]$ Lines 3 and 4 are identical because x, x_0, y, and y_0. are different variables.

5) $\exists y\; p(x,y)[x/d]$ Line 4 to line 5 uses the semantics of $\exists$, shown on line 2.

6) $\forall x\; \exists y\; p(x,y)$ Line 5 to line 6 uses the semantics of $\forall$, shown on line 3.

Therefore, $\exists y\; \forall x\; p(x,y) \;\rightarrow\; \forall x\; \exists y\; p(x,y)$ holds.

Example C.2.5: Does the expression $\forall x\; \exists y\; p(x,y) \;\rightarrow\; \exists y\; \forall x\; p(x,y)$ hold?

This example is similar to C.2.4.

C.3 Linear Temporal Logic

Linear Temporal Logic (LTL) models time as a line or path.

Syntax: The syntax of LTL in BNF follows:

$$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi)$$
$$\mid (\Box\,\phi) \mid (\Diamond\,\phi) \mid (X\,\phi) \mid (\phi \;\mathrm{U}\; \phi) \mid (\phi \;\mathrm{W}\; \phi) \mid \phi \;\mathrm{R}\; \phi$$

p is a propositional atom. The temporal operators $\Box$, $\Diamond$, X, U, R, W mean "always," "eventually," "neXt," "Until," "Release," and "Weak-until," respectively.

Semantics: LTL models time as a sequence of states. This sequence of state is called a *computation path* or just a *path*. The semantics of LTL in which π is a path follows.

- $\pi \models \top$ (i.e., $\top$ is always true)
- $\pi \not\models \bot$ (i.e., $\bot$ is always false)
- $\pi \models p$ iff $p \in L(s_1)$ (i.e., atomic propositions are evaluated along the path)
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$ (similar to propositional logic)
- $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$ (similar to propositional logic)
- $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$ (similar to propositional logic)
- $\pi \models \phi_1 \rightarrow \phi_2$ iff $\pi \models \phi_2$ whenever $\pi \models \phi_1$ (similar to propositional logic)
- $\pi \models \Box\phi$ iff $\forall i \geq 1,\ \pi^i \models \phi$
- $\pi \models \Diamond\phi$ iff $\exists i \geq 1,\ \pi^i \models \phi$
- $\pi \models X\phi$ iff $\pi^2 \models \phi$ (i.e., the next (second) state from the path is ϕ)
- $\pi \models \phi\, U\, \psi$ iff $\exists i \geq 1$ such that $\pi^i \models \psi$ and $\forall j = 1,\ldots,i-1,\ \pi^j \models \phi$ holds
- $\pi \models \phi\, W\, \psi$ similar to the definition of U (above); or $\forall k \geq 1,\ \pi^k \models \phi$ holds
- $\pi \models \phi\, R\, \psi$ iff $\exists i \geq 1$ such that $\pi^i \models \phi$ and $\forall j = 1,\ldots,i,\ \pi^j \models \psi$ holds; or $\forall k \geq 1,\ \pi^k \models \psi$ holds

Example C.3.1: Does the expression $\Diamond\Box p \rightarrow \Box\Diamond p$ hold?.

The following proof [61] shows that if this theorem holds, then the following holds: for arbitrary K and $i \in \mathbb{N}$, $K_i(\Diamond\Box p) = true$ implies that $K_i(\Box\Diamond)p = true$.

(1) $K_i(\Diamond\Box p) = true \rightarrow K_j(\Box p) = true$ for some $j \geq i$;[1]

(2) $\rightarrow K_k(p) = true$ for some $j \geq i$ and every $k \geq j$;[2]

(3) $\rightarrow K_k(p) = true$ for some $k \geq j$ and arbitrary $j \geq i$;[3]

(4) $\rightarrow K_j(\Diamond p) = true$ for every $j \geq i$;[4]

(5) $\rightarrow K_i(\Box\Diamond p) = true$;[5]

Therefore, the expression $\Diamond\Box p \rightarrow \Box\Diamond p$ holds.

Example C.3.2: Does the expression $\Box\Diamond p \rightarrow \Diamond\Box p$ hold?

If this theorem holds, then the following should hold: for arbitrary K and $i \in \mathbb{N}$, $K_i(\Box\Diamond p) = true$ implies that $K_i(\Diamond\Box)p = true$.

(1) $K_i(\Box\Diamond p) = true \rightarrow K_j(\Diamond p) = true$ for every $j \geq i$;

(2) $\rightarrow K_k(p) = true$ for every $j \geq i$ and some $k \geq j$;

(3) $\rightarrow K_i(\Box p) = true$ for some $k \geq j$;

(4) $\rightarrow K_j(\Diamond\Box p) = true$;

Therefore, $\Box\Diamond p \rightarrow \Diamond\Box p$ does not hold as the subscript j refers to a point of of time equal or greater then i.

Example C.3.3: Does the expression $\Box \bigcirc p \rightarrow \bigcirc\Diamond p$ hold?

For arbitrary K and $i \in \mathbb{N}$

(1) $K_i(\Box \bigcirc p) = true \rightarrow K_j(\bigcirc p) = true$ for every $j \geq i$

(2) $\rightarrow K_{j+1}(p)$ for every $j \geq i$

(3) $\rightarrow K_j(p) = true$ for every $j \geq i + 1$

(4) $\rightarrow K_{i+1}(\Box p) = true$

(5) $\rightarrow K_i(\bigcirc\Box p) = true$

Therefore, $K_i(\Box \bigcirc p) \rightarrow K_j(\bigcirc p)$ holds.

---

[1] note: because of $\Diamond$'s semantics

[2] note: from line (1) to (2) is because of $\Box$'s semantics.

[3] note: from line (2) to (3) is for the following two reasons: 1) if holds for *every* $k \geq j$ on line (2) then it should hold for *some* $k \geq j$; 2) since $k \geq j$ then it should hold for any arbitrary j

[4] note: from line (3) to (4) is because of $\Diamond$'s semantics

[5] note: from line (4) to (5) is because of $\Box$'s semantics

Appendix D

An Overview of SPIN and Alloy

D.1 SPIN

A description of the SPIN model checker follows. This description is mainly based on *The SPIN Model Checker: Primer and Reference Manual* [47], *The Model Checker SPIN [46]*, and *Principles of the Spin Model Checker* [11].

SPIN stands for *S*imple *P*ROMELA *IN*terpreter and accepts a specification written in a C-like language called *PRO*cess *ME*ta-*LA*engage (PROMELA); the correctness property can be specified in Linear Temporal Logic (LTL). SPIN, written by Gerard Holzmann, was developed at Bell Laboratories in the 1980s and became available publicly with its first free version in early 1991. This tool represents a widely-used model checker that constantly evolves. SPIN has been used for the verification of operating systems, call processing software at Bell, and key control algorithms for space missions at NASA, to name a few uses.

PROMELA programs consist of a set of processes, which are represented by either *proctype* or *active proctype* keywords; although it is possible to have a program that consists of a single process. The main numeric data types of PROMELA consist of *bit* (with values of 0 and 1), *bool* (a syntactic sugar for *bit* accepting *false* and *true*), *byte*, *short*, *int*, *unsigned*, and *channel* (*chan*). A *channel* data type, defined by the keyword *chan* can include send and receive operations with messages of particular types as defined. There are two types of channels: a) *rendezvous* channels, with a capacity of zero, are used for synchronous communications holding only one message, and b) *buffered* channels, with a capacity of more than zero, are used for asynchronous communications.

No floating, string, and character variables exist in PROMELA. Literal characters can be printed using a *byte* data type and *printf* with an option of %*c*. Messages with string content can be given numeric values and printed. If floating data types are definitely needed, they can be represented as embedded C code. Another possibility is to use minimum, low, maximum, and high and obtain an approximation of a floating data type.

Two other PROMELA data types consist of *message type* (*mtype*) and *process identifier* (*pid*). The *mtype* data type enables the assignment of mnemonic names to values; as a result, these easily remembered names can be represented and printed whenever needed. Up to 255 values are available because *mtype* is internally represented as positive *byte* values. Although similar to *byte*, *pid* represents a separate data type and can have a value up to 255, which is also the maximum number of processes in a SPIN model. The variable *_pid*, which is assigned to a process every time it is initiated, is predefined as a *pid* data type.

PROMELA uses repetition (loops) and selection (if statements) control structures that are both based on *guarded commands*. Dijkstra [26] introduced *guarded commands* as components for both alternative and repetitive constructs. Both constructs consist of a list of boolean expressions acting as guards: whenever a guard is true, its associated statement can be executed. Guards can be non-disjoint; as a result, one of them can be selected non-deterministically. (This selection can even be different from one execution to another.) The guards and their associated statements are enclosed within *do* and *od* for the repetition structure and are encircled within *if* and *fi* for a selection structure, as shown below.

do	**if**
:: guard_1 -> S_1	:: guard_1 -> S_1
:: guard_2 -> S_2	:: guard_2 -> S_2
od	**fi**

Only one guard is evaluated in each iteration or selection, and the order of a guard is irrelevant in this evaluation; in this sense, the guarded commands selection resembles the *case* control structure [105], but guards can be overlapping in guarded commands.

In addition to PROMELA, Ada and Occam, two other programming languages, also support some form of guarded commands.

The SPIN temporal operators and their notations are shown in Table D.1.

SPIN does not have the *weak until* (W or $\mathcal{W}$) operator, but this operator can be defined using its equivalent as below:

Temporal Operator	**General Notation**	**SPIN Notation**
always	$\Box$	[ ]
eventually	$\Diamond$	<>
until	U, $\mathcal{U}$	U
next	X, $\bigcirc$	X

Table D.1: SPIN's temporal operators and their notations

$$p\, W q \equiv p\, U q \vee \Box p$$

SPIN can be used in two different modes: simulation and verification. The correctness properties can be specified as assertions and either simulated with interactive/random modes or verified. In addition, one can use LTL and specify desired properties and verify specifications. SPIN translates an LTL formula into a *never claim*, which is a PROMELA construct, that can also be written directly. The term *never claim* is used because an LTL property specification represents a *claim*; SPIN uses the negation of a *claim* and includes the keyword *never* in its translation from LTL to PROMELA. For instance, a translation of $[\,]mutex$, where [ ] is the SPIN notation for *always* ($\Box$) and *mutex* is defined in code as a variable with a value of 1 or less, follows:

```
never {
T0_init:
    if
    ::  (! ((mutex))) -> goto accept_all
    ::  (1) -> goto T0_init
    fi;
accept_all:
skip
}
```

The claim $[\,]mutex$ holds, and *mutex* has a value of one; therefore, its negation (*!mutex*) does not hold. Initially, the second guard (1) is selected, but if the property does not hold, the first guard will be selected.

SPIN is described as an on-the-fly model checker because the entire state space is not built in one round and searched in another: SPIN searches the target states while building them. Therefore, if a counter-example exists, then the search space is built up to that

point, and there is no need to continue further. The entire state space will be constructed only when there is no error in a model.

Finally, worth mentioning is the computational cost of the SPIN verification. The cost is linear in terms of reachable states in both CPU time and memory space for the verification of safety properties, such as the absence of deadlock and user-defined assertions (in the latter case, properties are not described in LTL). The verification of LTL properties does not change the required space, but the time requirement can be exponential, in the worst case, in terms of temporal operators within an LTL formula.

D.2 Alloy

The description of Alloy used here is primarily based on Jackson's *Software Abstractions: Logic, Language, and Analysis* [50]. Alloy was introduced in 1997. Similar to Jackson's, this overview is divided into three sections: logic, language, and analysis.

Logic: Alloy's logic is called a *relational logic* and combines the quantifiers of predicate logic with the operators of relational calculus. The logical and relational operators play an important role in Alloy and are shown in Table D.2.

<table>
<tr><th>Relational Operator</th><th>Logical Operator</th></tr>
<tr><td>arrow (->)</td><td>and (&&)</td></tr>
<tr><td>dot (.)</td><td>or (||)</td></tr>
<tr><td>box ([ ])</td><td>not (!)</td></tr>
<tr><td>transpose (~)</td><td>implies (=>)</td></tr>
<tr><td>transitive closure (^)</td><td>iff (<=>)</td></tr>
<tr><td>reflexive transitive closure (*)</td><td></td></tr>
<tr><td>domain restriction (<:)</td><td></td></tr>
<tr><td>range restriction (:>)</td><td></td></tr>
<tr><td>override (++)</td><td></td></tr>
</table>

Table D.2: Alloy's relational and logical operators

The meanings of logical operators are similar to those of predicate logic. A brief description of operational operators follows. The *arrow product* (->) of p -> q represents a binary relation if p and q are sets, but describes a multirelation if p and q are relations.

The *dot* (.) or the *join* operator can be used for object navigation. This operator resembles the relational database join, but in Alloy, the matching element is dropped; e.g., the joining of (a,b) and (b,c) results in (a,c). The *box* ([ ]) operator represents a variation of *join* in which arguments are received in reverse order; e.g., $p[\,]q = q.p$. The transpose (∼) of a binary relation represents its mirror image; e.g., the transpose of {(a,b)} is {(b,a)}. A transitive closure (ˆ) of a relation includes the relation and its closure; e.g., the transitive closure of {(a,b), (b,c)} will be {(a,b), (b,c), (a,c)}.

Four other relational operators include reflexive transitive closure (∗), domain restriction (<:), range restriction (:>), and override (++). A reflexive transitive closure is defined as a relation that is reflexive and includes its closure; a relation is reflexive if for every existing element of *a*, it also includes a->a. The domain restriction (<:) of *s<:r*, where *s* is a set and *r* is a relation, includes tuples with starting elements in *s*. For instance, if r={(a,b), (c,d), (e,a)} and *s*={(a)}, then *s<:r* is represented by {(a,b)}. The range restriction (:>) or *s:>r*, where *s* is a set and *r* is a relation, includes tuples with ending elements in *s*. The range restriction of the set s and relation r, just described, is represented as *s:>r* = {(e,a)}. Finally, the override (++) of relations *p* and *q* is shown by *p++q* and defined as their union, in which q can override p; e.g., if *p*= {(a,b), (c,d)} and *q* ={(a,m), (e,f)}, then *p++q* = {(a,m), (c,d), (e,f)}.

Language: The language component of Alloy provides the ability to organize the structure of a model and communicates with the analyzer. Some features, e.g., *scope* and *signature*, of the language are unique to Alloy; others, such as *function*, are common to many modeling and programming languages. The components that are used most often follow.

- *signature declarations*, by the use of the keyword *sig*, define sets; in addition, relations can also be declared as fields in sig declarations, and subsets can also be defined.
- *constraint paragraphs*, by the use of keywords *fact*, *pred* (predicate), and *fun* (function), describe various expressions.
- *assertions*, by the use of the keyword *assert*, express properties that should hold.
- *commands*, by the use of keywords *run* and *check*, declare a particular desired analysis. By running a predicate, the analyzer looks for a model, whereas by checking an assertion, the analyzer looks for a counterexample.

Analysis: Alloy Analyzer is a compiler that translates a problem into a boolean formula; the formula is subsequently handed to a SAT solver. Alloy Analyzer also translates a

SAT solver's solution into Alloy's language. Therefore, the analysis is a form of constraint solving. Alloy's analysis is based on *scope*, which plays a role in the creation of a boolean formula. A model is called an *instance* in Alloy, and the use of *scope* makes instance findings possible. Therefore, the analysis finds an instance within a specified limit or *scope*. Jackson proposes a hypothesis called the "small scope hypothesis" that states, "most bugs have small counterexamples."

The creation of a boolean formula is as follows: for a relation r from a set A to a set B, an adjacency matrix is created. The dimension of this matrix depends on the scope of sets A and B. If both have the default scope of 3, then there will be a 3 × 3 matrix having 9 boolean variables that creates 2^9 values for these variables. Therefore, if a model has only four relations, there will be over a billion cases ($2^9 \times 2^9 \times 2^9 \times 2^9$) of values. Despite the existence of optimization techniques in this translation and consistent progress in SAT-solving techniques, the state space grows exponentially, as described above; therefore, Alloy's analysis is usually small in scope; the default for the scope is 3.

D.2.1 An Example of AC Policies and Properties Specification and Verification

This section describes a small running example to show the benefit of access control formal analysis [56]. This approach is in contrast with a) the description of access control models, rules, policies, and their combinations in Chapter 3, b) the property categorization provided in the first part of this chapter, and c) the case study approach presented in Chapter 5. In particular, the policies of this section are not based on a model; there is no definition of the structure of an access control rule or a policy and their combinations. Access control rules or policies are specified using a programming language.

The following running example describes the formal specification and verification of access control policies for business processes. First, this section specifies access control policies for REA business processes and examines the addition and combination of these policies, more specifically focusing on the principal of separation of duties. In addition, the verification of access control properties is examined in conjunction with a REA business process, and it is shown that this process can reveal errors. As a running example, this chapter uses the process of renting a car and adds access control policies. Subsequently, these policies are integrated into a REA process and are formally analyzed. An incorrect effect is created by adding new access control policies.

Specification of a REA Business Process: A REA business process of rent is specified in Alloy (an overview of Alloy is provided in Appendix D.2.), and a description of these

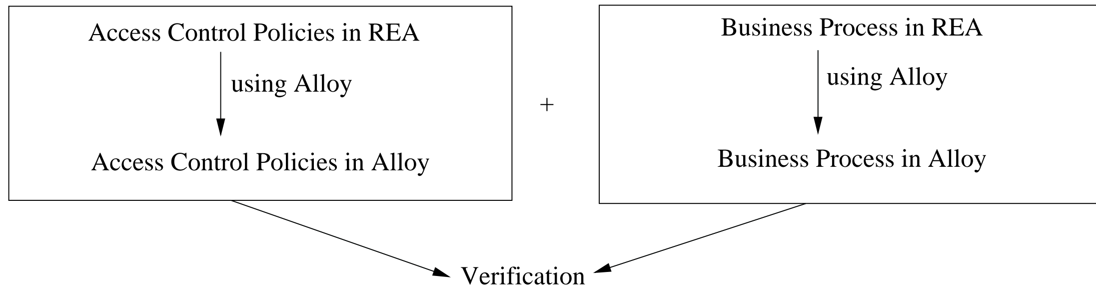


Figure D.1: A verification of policies that are not based on a model.

few lines follows.

Alloy is used for formal specification and verification of access control policies. Alloy is a declarative language with a complete semantic and automatic analysis having the capability of presenting models graphically [50]. Alloy Analyzer is a compiler that translates a problem into a Boolean formula; the formula is subsequently handed to a SAT solver. Alloy Analyzer also translates the SAT solver's solution into Alloy's language [50].

```
module models/REAexchangeBasic
abstract sig Resource, Agent, Event{}
sig CashReceipt extends Event{}
sig RentACar extends Event{}
sig CarRentalEmployee in Agent{}
sig Customer in Agent{}
sig Car extends Resource{}
sig Money extends Resource{}
sig Exchange {inflow: Money -> CashReceipt,
              outflow: Car -> RentACar,
              exchange: (CashReceipt -> RentACar),
              exchangeb: (RentACar ->  CashReceipt),
              provide: (Customer -> CashReceipt),
              provideb: (CarRentalEmployee ->  RentACar),
              receive: (CashReceipt -> CarRentalEmployee),
              receiveb: (RentACar -> Customer) }
```

A *sig* (signature) declares a set; an *extend* keyword introduces subsignatures, and "an abstract signature has no elements except those belonging to its extensions" [50]. Unlike

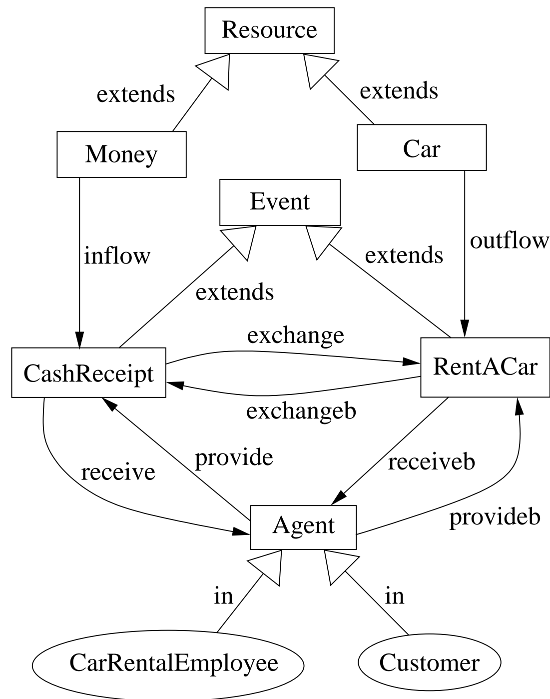


Figure D.2: Visualizing of a REA rent business model using Alloy

subsets introduced by the keyword *extend*, subset signatures declared by the keyword *in* are not necessarily mutually disjoint subsets [50]; e.g., customers and car rental employees form a non-disjoint set because an employee can also rent a car. "exchange and exchangeb" are used to show the bidirectional associations; similarly, "provide and provideb" and "receive and receiveb" are used for the "provide" and "receive," respectively.

Alloy has an option called meta-model that is used to show the process of renting a car as a directed graph. This option is used to create Figure D.2. The car-rental employees and customers are presented inside ellipses using the keyword *in* to represent non-disjoint sets (i.e., car-rental employees can rent cars to themselves and therefore can be customers); on the other hand, cash receipts and rent cars are represented as disjoint sets (i.e., the *extend* keyword) and are shown inside rectangles.

As Fisler et al. [36] describe, "access control is conventionally defined using a matrix. As organizations grow, however, the explicit matrix becomes very cumbersome," and the information in the matrix is captured using rules.

This section introduces three policies as three rules and explains whether the verification of the following three policies holds individually and in composition:

$\mathbf{P}_1$: Gold-club customers get a discount whenever they rent a car.

$\mathbf{P}_2$: People who get a discount (i.e., gold-club customers) cannot be both customers and car rental employees within one exchange.

$\mathbf{P}_3$: Car-rental employees can receive certain discounts whenever they rent a car.

$\mathbf{P}_2$ represents a static separation of duty (SoD) when a discount is involved. A static SoD can use an agent's attribute, and with or without the existence of a discount.

In the next section, these three policies are added to the existing previous model and are analyzed using Alloy to determine their effects. The analysis of access control policies enables determining whether a specification is actually intended.

The typical reasoning analysis includes determining the consistency of these policies and finding out whether there is an instance for a description. In addition, the safety analysis, which is arguably the most important property [59], can be checked (e.g., whether an individual has a permission that he/she should not have).

The next section uses Alloy and provides an example of the verification of access control policies in conjunction with a process of renting a car. More specifically, the principal of separation of duties (e.g., two separate individuals must authorize ordering items and paying for them) is illustrated.

Specification and Verification of Access Control Policies: This section revisits the previously mentioned policies and adds them one by one. The main access control property is to maintain the separation of duties during the car-rental process such that individuals are not able to rent cars to themselves. The first property, "gold-club customers get a discount whenever they rent a car," is indicated as PolicyGCC and defined as an extension of the set Policy.

```
abstract sig Policy{}
sig Policy_GCC extends Policy{apply: GoldClubCustomer}
```

The *predicate* and *run* keywords of Alloy are used to find models of specifications. The idea is to examine whether there is a model for a specific specification.

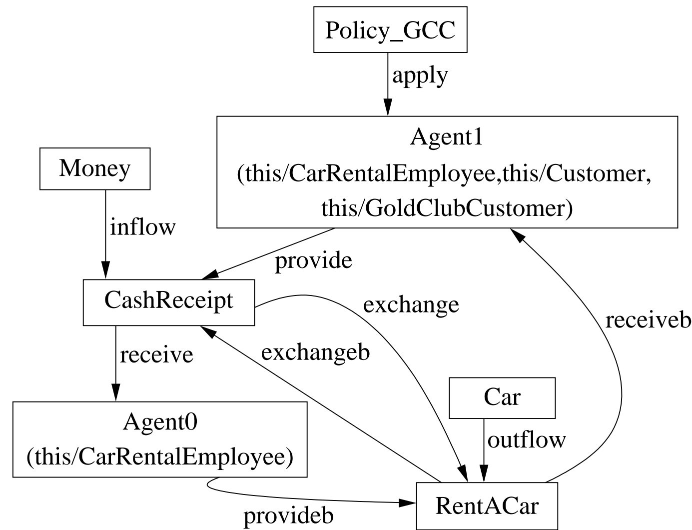


Figure D.3: A counter-example indicating an error

```
pred showPolicy[]{}
run showPolicy for 2 but 1 Policy_GCC, 1 RentACar,
                          1 Exchange
```

The statement, “run showPolicy” is sufficient to create models. Other options such as “for 2 but 1 Policy_GCC, 1 RentACar,” are added to create a small model that has only one policy and is applicable to only one exchange. The numbers “2,” “1” of Policy_GCC, and “1” of RentACar, from showPolicy just mentioned, are the scope of analysis. Scope is used to bound (i.e., to restrict or to limit) the size of an analysis. The default scope is 3, and scope is the number of elements of top-level signatures that will be included in analysis. Figure D.3 represents one possible model. Names, such as Car and Agent0, in Figure D.3 and other upcoming ones, are assigned by Alloy and can be changed by the users of this tool.

Figure D.3 shows that one individual both receives cash and rents a car. The goal is to tighten this case so that these two tasks cannot be performed by one individual. Therefore, a new property is added to ensure a separation of duties: employees cannot be counted as gold-club customers. The second property, “people who get a discount (i.e., gold-club customers) cannot be both customers and car rental employees in one exchange.” Gold-

club customers get discounts, and therefore, the following *fact* (an Alloy keyword) is added to ensure that if a gold-club customer provides cash, he/she cannot be the person who is the provider of the rental car. This *fact* is described in Alloy as follows:

```
fact{all e:Exchange, gc:GoldClubCustomer, cr:CashReceipt, rac:RentACar |
         gc = e.provide.cr
            => e.provideb.rac != gc}
```

The above *fact* can be added as an assertion (using the keyword *assert*) and then checked to determine whether a counterexample exists. No counterexample is found when the assertion is checked. An assertion is essentially a constraint that follows from implicit or explicit *fact*s and is added to detect flaws in specifications [50]. Now, the third policy, "Car-rental employees can receive certain discounts when they rent a car," is added to enable car-rental employees to get discounts.

```
sig EmpCus in Customer{}
sig Policy_Emp extends Policy{apply: EmpCus}
assert empSP2{all e:Exchange, ec:EmpCus, cr:CashReceipt, rac:RentACar |
                 ec = e.provide.cr
                        => e.provideb.rac != ec}
```

Checking this assertion indicates the existence of a counterexample: "Counterexample found. empSP2 is invalid." Whenever an assertion is not valid, Alloy creates a graphical counterexample to show the violation(s). Figure D.4 represents this violation and a counterexample for this case.

This counterexample (Figure D.4) shows that a car-rental employee or EmpCus (i.e., a car-rental employee who is also a customer) can receive money (i.e., the edge named "receive" from "CashReceipt" to "Agent1" in Figure D.4); in addition, a car rental employee can provide a car during the rental event (i.e., the edge named "provideb" from "Agent1" to "RentACar in the same figure). Figure D.5 shows the graphical representation of an "Agent" and its non-disjoint subsets before and after allowing discounts for car-rental employees, as shown in Part a and Part b, respectively.

The reason for the existence of a counterexample can be observed if Figure D.5 is compared to the previously mentioned *fact*, shown again below.

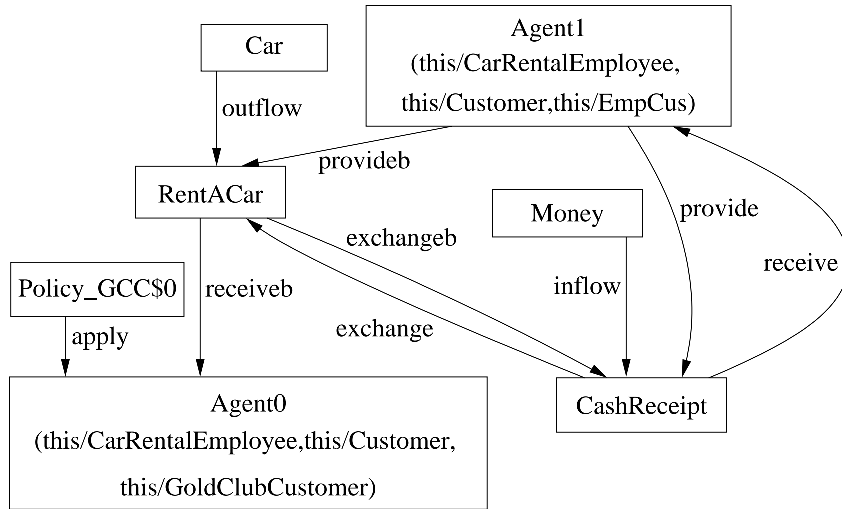


Figure D.4: A counterexample of separation of duties

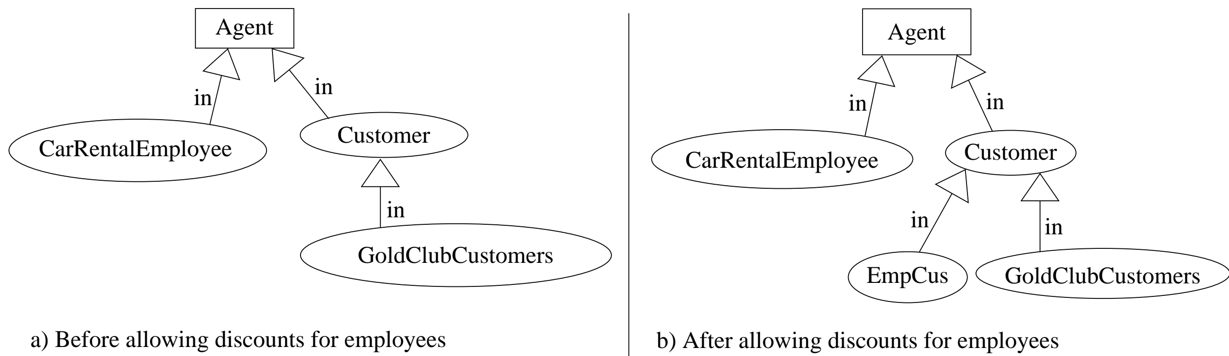


Figure D.5: Employees and Customers

```
fact{all e:Exchange, gc:GoldClubCustomer, cr:CashReceipt, rac:RentACar |
        gc = e.provide.cr
            => e.provideb.rac != gc }
```

Initially, this *fact* was stated for gold-club customers, who were the only people entitled to receive discounts. The *fact* was designed to reduce the possibility of fraud and to enforce separation of duties when only gold-club customers were eligible for discounts. The addition

of the last policy so as to make employees eligible for a discount creates a counterexample because car-rental employees are now customers too.

One possibility for correcting this counterexample is the inclusion of another *fact*, similar to the previously mentioned one. With the inclusion of the following *fact*, checking the previous assertion does not find a counterexample.

```
fact{all e:Exchange, ec:EmpCus, cr:CashReceipt, rac:RentACar |
        ec = e.provide.cr
          => e.provideb.rac != ec }
```

The drawback to this approach is the inclusion of another similar *fact*; on the other hand, that all the possible cases are enumerated and are not implicit can be an advantage. Another possibility for correcting this counterexample is to change the *fact* specified for gold-club customers on Page 175 and the *fact* just described for employees who are customers and consider both groups as customers and have one single *fact*. The following *fact* not only ensure that the provider of a car and the receiver of cash are different whenever a discount policy exists but also is true for all other cases. This strict separation of duties might sometimes be required. The single resulting *fact* follows:

```
fact{all e:Exchange, c:Customer,  cr:CashReceipt, rac:RentACar |
       c = e.provide.cr
         => e.provideb.rac != c }
```

Checking the assertion after this change does not find a counterexample. This assertion is initially checked for the default scope of 3. Then, the scope is gradually increased to 27, and no counterexample is found within this large scope (the analysis runs out of memory for a scope of 28).

This simple example illustrates how combining policies can create an incorrect effect. Iterating over all possible combinations and determining their effects require formal analysis when the number of these polices increases.

Clarification of the approach described and of the access control safety analysis for a general case is necessary. Similar to Jha et al.'s work [52], the presented verification is not the safety analysis mentioned on Page 23 of this thesis; instead, the intention is here to determine whether access control policy invariants are preserved. (Invariants are correctness properties that should hold at certain control points of a program's execution [83].) Jha et al. show that this analysis is decidable.

In summary, a small example that includes access control policies in connection with a REA business process has been specified. In addition, the access control policies have been verified, and the formal verification of policies showed that an incorrect effect occurs with the addition of the last policy; more specifically, the separation of duties has been violated.

Appendix E

The CONTINUE Policies and Properties

The CONTINUE policies and the properties are provided in Sections E.1 and E.2, respectively.

E.1 The CONTINUE Policies

A prose description of twenty-five policies for CONTINUE conference management follows. The first-applicable combining rule within each of the following policies holds. These policies are available in the XACML format from the CONTINUE web site.[1]

In CONTINUE, a *review-set* consists of four resources: *paper-review*, *paper-review-info*, *paper-review-info-reviewer*, and *paper-review-info-submissionStatus*. Similarly, a *review-content-set* consists of four resources: *paper-review-content*, *paper-review-content-rating*, *paper-review-content-commentsAll*, and *paper-review-content-commentsPc*. Furthermore, the first-applicable combining rule within each of the following policies holds.

Convention: The dashes within names and the suffix *rc* (*rc* stands for *resource class*) are omitted; therefore, *paper review* is used instead of *paper-review_rc*.

Policy One: An *administrator* has permission to read and write *conference* resources, and a *pc chair* possesses permission to read these resources. A *pc member* at a meeting

[1] http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/continue/CodeB/

is permitted to read *conference* resources; an unidentified subject has no access to these resources.

Policy Two: An unidentified subject has access to read *conference info* resources. Any other permission to these resources is based on the same access rules applicable to *conference* resources.

Policy Three: A *pc member* has access to read *pc member* resources. An *administrator* possesses permission to write, create, and delete *pc member* resources. A *pc member* whose user-id is equal to the user-id of a *pc member* resources has no permission to perform any action on these resources. Any other individual's access to these resources follows the same rules for accessing *conference* resources.

Policy Four: A *pc chair* possesses permission to read and write *pc member assignment* resources, whereas a *pc member* is allowed to read his/her own assignments (i.e., a *pc member*'s user-id is equal to the user-id of a *pc member assignment* resources). An unidentified subject has no access to these resources. Other types of access to these resources follow the same rules for accessing *pc member* resources.

Policy Five: A *pc chair* has read and write access to *pc member conflict* resources, whereas a *pc member* is capable of reading his/her *conflict* resources. An unidentified subject has no access to these resources. In addition, other types of access to *pc member conflict* resources follow the same rules for accessing *pc member* resources.

Policy Six: Access to *pc member assignment count* resources is according to the rules for accessing *pc member* resources.

Policy Seven: A *pc chair* possesses permission to read and write *pc member info* resources, whereas a *pc member* has access to read and write his/her *pc member info* resources. An unidentified subject has no permission to access these resources. Furthermore, the same permission rules for accessing *pc member* resources hold for *pc member info* resources too.

Policy Eight: A *pc member* has write access to his/her *pc member info password* resources, and an administrator has the same permission whenever *pc member info password* resources are not pending. An unidentified subject does not posses any access to these resources. Additionally, the same permission rules for accessing *pc member info* resources also hold for accessing *pc member info password* resources.

Policy Nine: A *pc member* has access to read *pc member isChairFlag* resources, whereas a *pc member* whose user-id is equal to the user-id of these resources has no access to *pc member isChairFlag* resources. An unidentified subject has no access to these resources. Furthermore, the same permission rules for accessing *pc member info* resources also hold to access *pc member isChairFlag* resources.

Policy Ten: A *pc chair* possesses access to delete *paper* resources. A *pc member* has permission to read a paper if the *paper* is designated for a meeting; in addition, a *pc member* is allowed to create *paper* resources. Any other access to *paper* resources is based on the same rules for accessing *conference* resources.

Policy Eleven: A *pc chair* and a *pc member* are permitted to read *paper submission* resources, whereas a *sub-reviewer* is allowed to read only his/his own *paper submission* resources. In addition, the same permission rules for accessing *paper* resources are also applicable for accessing *paper submission* resources.

A *pc member*, P, designates a *sub-reviewer*, S, to review P's papers. S submits reviews for the assigned papers; after submitting these reviews, S has no future access to these reviews. P can access the reviews by S and modify and submit them. This arrangement makes S capable of using the conference management interface to read submitted papers and to write reviews. Otherwise, P has to make copies of submitted papers for S and retrieve S's reviews without using the conference management interface.

Policy Twelve: Access to *paper submission info* resources follows the same criteria as those for accessing *paper submission* resources.

Policy Thirteen: The same rules for accessing *paper submission* resources are also applicable for accessing *Paper submission file* resources.

Policy Fourteen: A *pc chair* in a meeting has read and write access to *paper decision* resources. Other criteria for accessing *paper decision* resources are based on the same rules as those for accessing *paper* resources.

Note: In the following policies, the words "conflicted" and "unconflicted" indicate that people in a role may face conflicts of interest, such as when reading and writing reviews.

Policy Fifteen: A *pc chair* and an *administrator* are allowed to read and write *paper conflict* resources, whereas a *pc member* who is conflicted is permitted to read *paper conflict* resources. In addition, a *pc member* in a meeting has access to read *paper conflict* resources. An unidentified subject has no access to *paper conflict* resources. Furthermore, other types of access to *paper conflict* resources follow the same rules for accessing *paper* resources.

Policy Sixteen: A *pc chair* and an *administrator* are permitted to read and write *paper assignment* resources. An unidentified subject who is conflicted possesses no access to *paper assignment* resources. A *pc chair* in a meeting is allowed to read a *paper assignment* resource that is related to the meeting. An unidentified subject who is in the meeting is allowed to read *paper assignment* resources. An unidentified subject has no access to *paper assignment* resources. In addition, the same criteria for accessing *paper* resources are applicable for determining access permission for *paper assignment* resources.

Policy Seventeen: An unconflicted *pc chair* has all types of access to *paper review* resources, whereas a *pc chair* in a meeting for particular *paper review* resources is allowed to read only those resources. A *pc chair* is permitted to create and delete *paper review* resources. A conflicted subject has no access to *paper review* resources. An unconflicted *pc member* is permitted to read *paper review* resources. All have all types of access to their own *paper review* resources. All types of access are permitted to discussion phase *paper review* resources. An unidentified subject who is assigned to *paper review* resources and has already done his/her task is allowed to have any type of access to the resources, whereas an unidentified one assigned to particular *paper review* resources has all types of access to them. An unidentified subject is not allowed to have any access to unassigned *paper review* resources. Furthermore, other access rules to *paper* resources are also applicable to *paper review* resources.

Policy Eighteen: A *pc chair* has all types of access to *paper review info* resources; in addition, other types of access to *paper review info* resources are based on the same criteria for accessing *paper review* resources.

Policy Nineteen: A *pc member* is permitted to write, create, and delete *paper review content* resources if a *pc member*'s user-id is equal to the user-id of the *paper review content* resources, whereas a *sub-reviewer* is allowed to create *paper review content* resources only if the *sub-reviewer* user-id is equal to the user-id of the *paper review content* resources. Furthermore, other types of access to *paper review content* resources follow the same criteria for accessing *paper review* resources.

Policy Twenty: A *pc member* has permission to write *paper review info submission status* resources if the *pc member*'s user-id equals the user-id of these resources and the content of these resources is already in place. Other types of access to *paper review info submission status* resources are based on the same rules for accessing *paper review info* resources.

Policy Twenty-one: All types of access to *paper review content rating* resources are based on the same rules as those for accessing *paper review content* resources.

Policy Twenty-two: All types of access to *paper review content comments all* resources are based on the same rules as those for accessing *paper review content* resources.

Policy Twenty-three: All types of access to *paper review content comments pc* resources are based on the same rules as those for accessing *paper review content* resources.

Note: CONTINUE currently does not permit *comments* by *pc members* who have not written reviews for a paper, and therefore, have not read the paper in as much detail as the reviewers of that paper have but intend to provide *comments*, which are distinct from reviews, for authors.

Policy Twenty-four: All types of access to *paper review info reviewer* resources are based on the same rules as those for accessing *paper review info* resources.

Policy Twenty-five: A *pc chair* has read and write access to *is meeting flag* resources, whereas a *pc member* possesses only read access. In addition, other types of access criteria for *is Meeting flag* resources follow the same rules for accessing *conference* resources.

E.2 The CONTINUE Properties

The CONTINUE properties are provided next. Note that because of the use of model checking and temporal logic and implications (Figure 4.5), *always*, *eventually*, and *next* exist in the description of these properties.

Property 1 (Pr_1): For any state, if there is a request, then it will eventually allow only a deny or permit response (i.e., no NA response is possible). This property should hold.

Property 2 (Pr_2): It is always the case that a program committee (PC) member who owns reviews can eventually edit his/her reviews. This property should hold.

Property 3 (Pr_3): For any state, if an individual is neither a PC chair nor an administrator, then he or she cannot eventually set the meeting flag. In verification, this property should hold.

Property 4 (Pr_4): It is always the case that if an individual role is not described (i.e., no roles exist for a subject), then no permit exists for the individual. In verification, this property should fail.

Property 5 (Pr_5): For any state, if an individual's role is not described and a resource is not conference-information, then eventually no permits exist for the individual. This property should hold.

Property 6 (Pr_6): It is always the case that if a person is neither a PC chair nor an administrator, then the person should eventually never be allowed to read the *paper review* resources for which he/she has a conflict of interest. This property should hold.

Property 7 (Pr_7): For any state, if an individual is neither a PC chair nor an administrator, and he/she is conflicted, then the individual should never be eventually permitted to read either any part of the *review-content-set* resources that are not written by the individual, or read the reviewer-info resource. This property should hold.[2]

[2] Pr_7 and Pr_6 are similar in the sense that the latter refers to a subset of the former. (Pr_6 refers

Property 8 (Pr_8): For any state, if a PC chair calls for a meeting, then the chair can eventually read anything related to the subject of the meeting. This property should fail.[3]

Property 9 (Pr_9): For any state, if a PC chair calls for a meeting, then the chair can eventually read any part of the reviews to be discussed at the meeting. This property should hold.[4]

Property 10 (Pr_{10}): It is always the case that a non-conflicted PC member at discussion phase can eventually read all parts of the reviews. This property should hold.

Property 11 (Pr_{11}): For any state, a non-conflicted PC member who has submitted a review of a paper can eventually read all parts of others' reviews of that paper. This property should hold.[5]

Property 12 (Pr_{12}): It is always the case that when the phase is not discussion, a PC member who has been assigned to submit a review of a paper but who has not done so cannot eventually read any part of *review-content-set* of others' reviews of that paper. This property should hold.

---

to *paper review* resources, and Pr_7 includes all review resources.) A discrepancy between Pr_7's prose description and the specification of this property in Scheme exists. The specification in Scheme considers only review-content-set and paper-review-info-reviewer resources, not all resources; nevertheless, either case is acceptable.

[3]This property possibly exists to check whether a paper's certain information, such as a paper's author(s), can be revealed in a meeting. Generally, the specific information, which can be known, must be clearly defined.

[4]Pr_9, a specific case of Pr_8, allows access to all eight review resources (described previously).

[5]For both Pr_{11} and Pr_{12}, the provided Scheme code specifies non-conflicted PC members.

Appendix F

Other Combining Access Control Algorithms

This appendix presents algorithmic forms and state machines for the following combining algorithms: the strong-consensus policy-combining algorithm, strong-majority policy-combining algorithm, and super-majority-permit policy-combining algorithm. This appendix also shows algorithmic forms and state machines for the weak-consensus rule-combing algorithm and weak-majority rule-combining algorithm.

Strong-consensus [66]: "All sub-policies must agree: Permit a request if all sub-policies permit a request. Deny a request if all sub-policies deny a request. Yield conflict otherwise." This algorithm differs from the weak-consensus policy-combining algorithm as one or more policies may have a *not applicable* result. In this situation, the result provided by the strong-consensus algorithm is *conflict*, whereas the weak-consensus algorithm may provide a *permit* or *deny* result because one or more *not applicable* results have no effect in the final decision.

Figure F.1 shows the strong-consensus policy-combining algorithm according to the approach shown in Chapter 3. Figure F.2 shows the corresponding state machine for this algorithm according to the approach presented in this thesis.

```
initial state = state_00;
set PermitRes to false;
set DenyRes to false;
set ConflictRes to false;
set NAres to false;
for i = 1 to n do
    // n = the number of policies in a policy set
    if premise-rule_i = false then
        set NAres to true;
        move to state_io;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            set PermitRes to true;
            move to state permitRes;
        else if Event_i Access(deny) = true for every element of EventResult then
            set DenyRes to true;
            move to state denyRes;
        end
    end
end
if i = n and NAres = true then
    set ConflictRes to true;
    move to state conflict;
else if i = n and PermitRes = true and DenyRes = false then
    move to state permit;
else if i = n and DenyRes = true and PermitRes = false then
    move to state deny;
else if i = n and PermitRes = true and DenyRes = true then
    set ConflictRes to true;
    move to state conflict;
end
```

Figure F.1: The algorithmic form for strong-consensus policy-combining algorithm

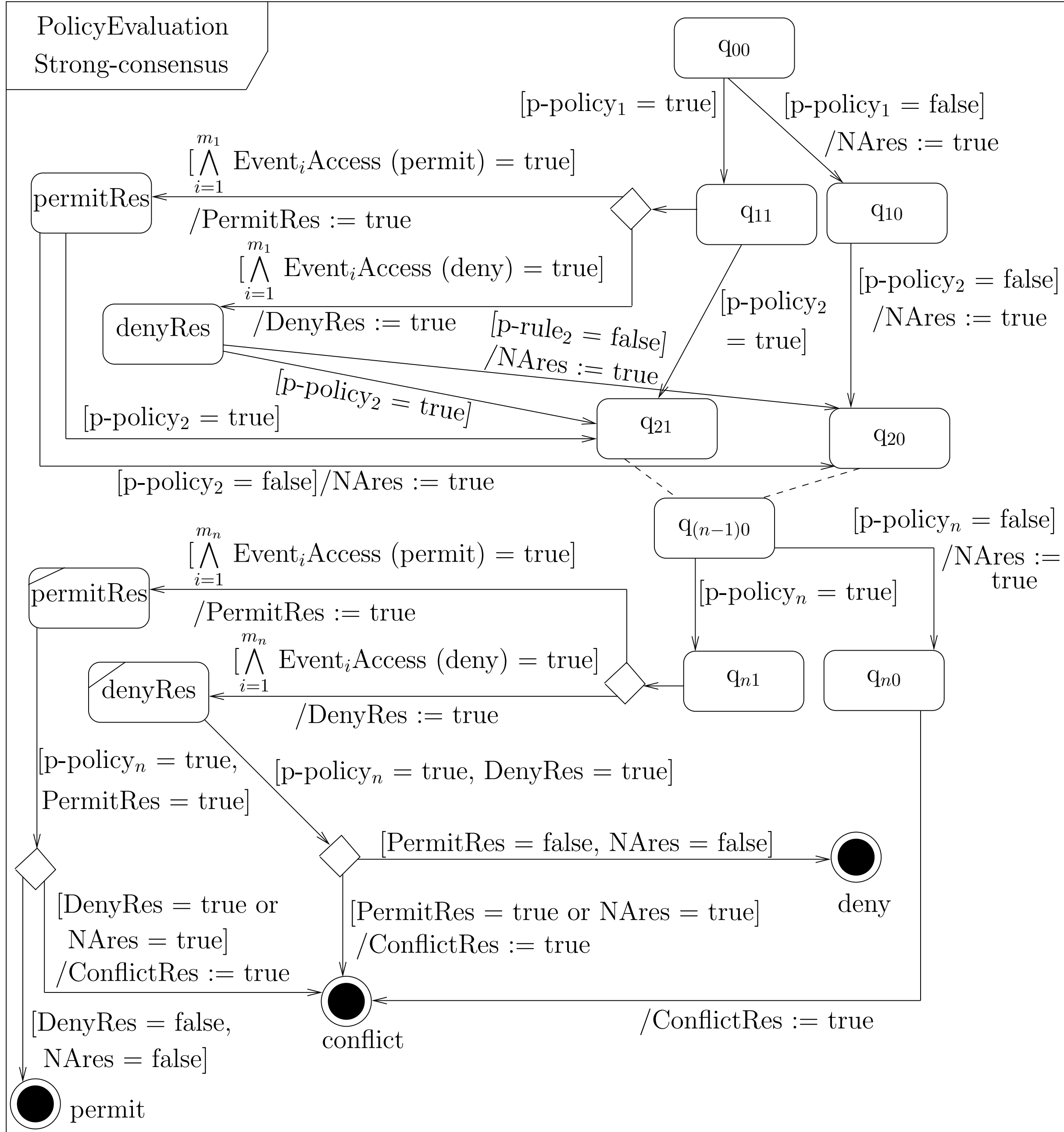


Figure F.2: A UML state machine for strong-consensus policy-combining algorithm

Strong-majority [66]: "Permit a request if over half of all sub-policies permit it, and deny the request if over half deny it."

Figure F.3 shows the algorithmic form for the strong-majority policy-combining algorithm.

```
initial state = state_00;
set NumPermit to zero;
set NumDeny to zero;
for i = 1 to n do
|   // n = the number of policies in a policy set
|   if premise-rule_i = false then
|   |   move to state_io ;
|   else
|   |   move to state_i1;
|   |   if Event_i Access(permit) = true for every element of EventResult then
|   |   |   add one to NumPermit;
|   |   |   move to state permitRes;
|   |   else if Event_i Access(deny) = true for every element of EventResult then
|   |   |   add one to NumDeny;
|   |   |   move to state denyRes;
|   |   end
|   end
end
if i = n and NumPermit > (n/2) then
|   move to state permit;
else if i = n and NumDeny > (n/2) then
|   move to state deny;
end
```

Figure F.3: The algorithmic form for strong-majority policy-combining algorithm

Figure F.4 shows the state machine corresponding to the algorithmic format.

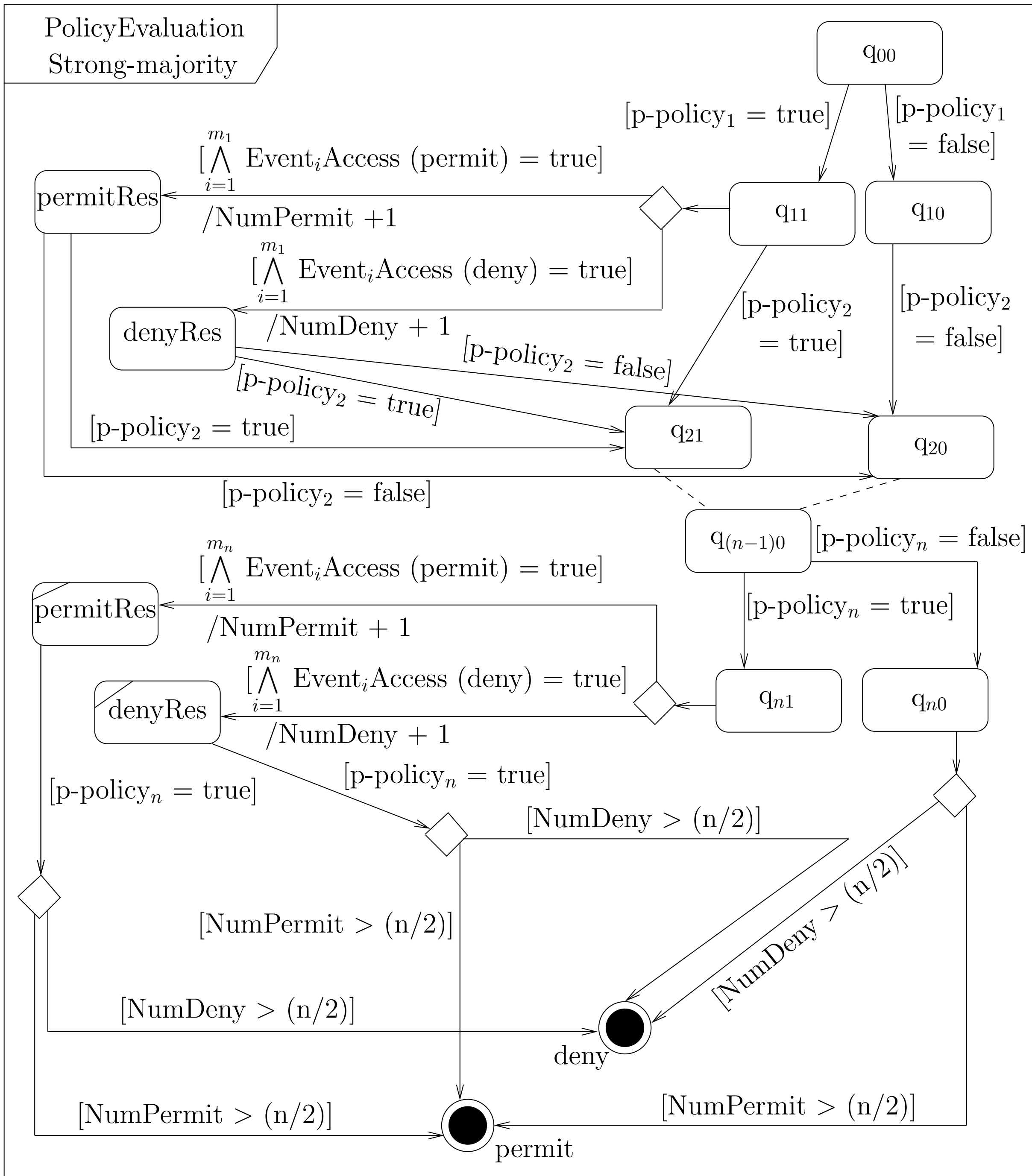


Figure F.4: A UML state machine for strong-majority policy-combining algorithm

Super-majority-permit [66]: "Permit a request if over 2/3 of all policies permit it, and deny the request otherwise."

Figure F.5 shows the algorithmic form for the super-majority-permit policy-combining algorithm.

```
initial state = state_00;
set NumPermit to zero;
for i = 1 to n do
    // n = the number of policies in a policy set
    if premise-rule_i = false then
        move to state_io ;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            add one to NumPermit;
            move to state permitRes;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state denyRes;
        end
    end
end
if i = n and NumPermit > (2/3n) then
    move to state permit;
else
    move to state deny;
end
```

Figure F.5: The algorithmic form for super-majority-permit policy-combining algorithm

Figure F.6 shows the state machine corresponding to the algorithmic format.

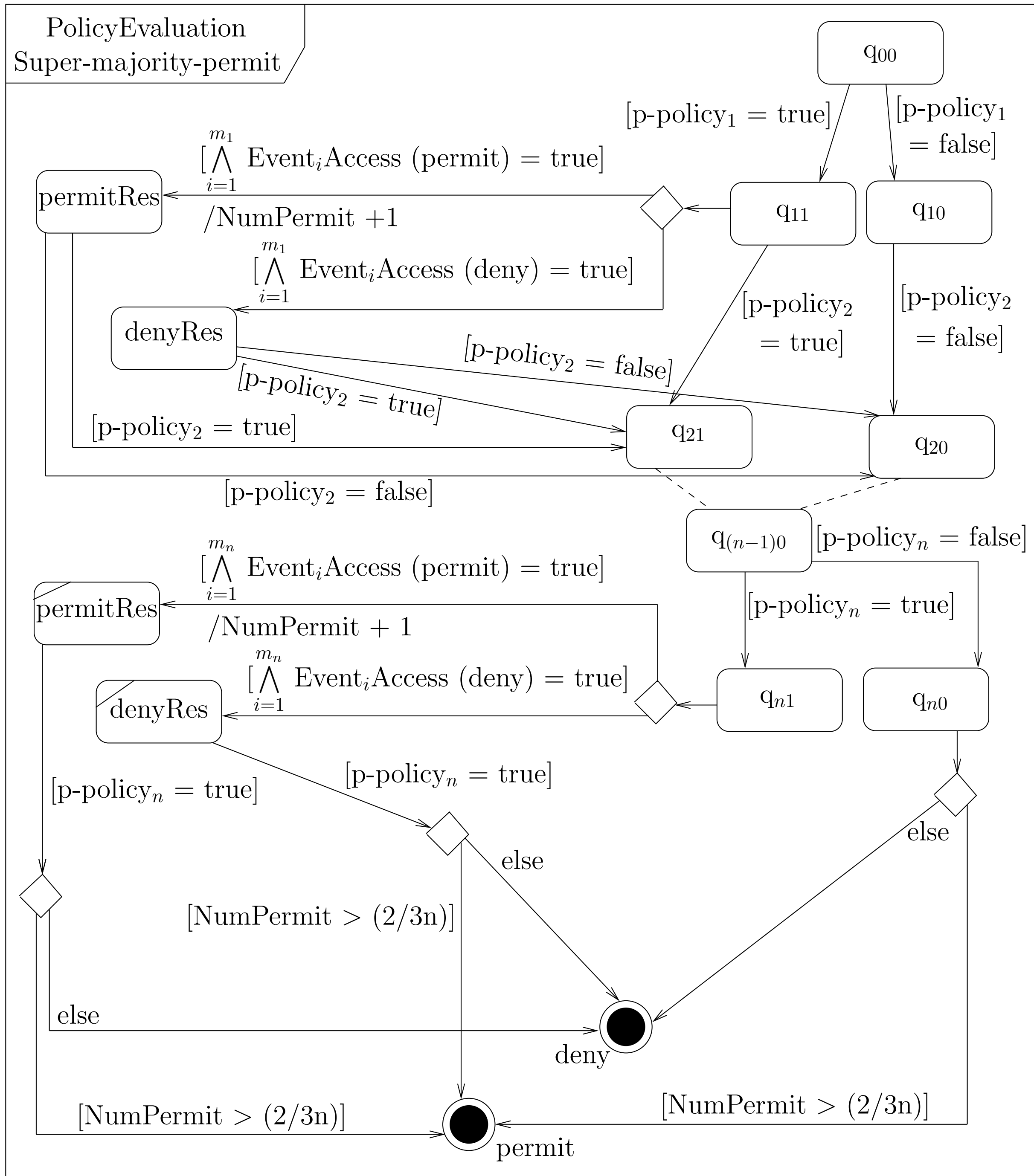


Figure F.6: A UML state machine for super-majority-permit policy-combining algorithm

Figure F.7 shows the weak-consensus rule-combining algorithm that is similar to the weak-consensus policy combining-algorithm, shown in Chapter 3.

```
initial state = state_00;
set PermitRes to false;
set DenyRes to false;
set ConflictRes to false;
for i = 1 to n do
    // n = the number of rules in a policy
    if premise-rule_i = false then
        move to state_io ;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            set PermitRes to true;
            move to state permitRes;
        else if Event_i Access(deny) = true for every element of EventResult then
            set DenyRes to true;
            move to state denyRes;
        end
    end
end
if i = n and PermitRes = true and DenyRes = false then
    move to state permit;
else if i = n and DenyRes = true and PermitRes = false then
    move to state deny;
else if i = n and PermitRes = true and DenyRes = true then
    set ConflictRes to true;
    move to state conflict;
end
```

Figure F.7: The defined AC rules and states for weak-consensus rule-combining algorithm

Figure F.8 represents the state machine corresponding to the algorithmic description.

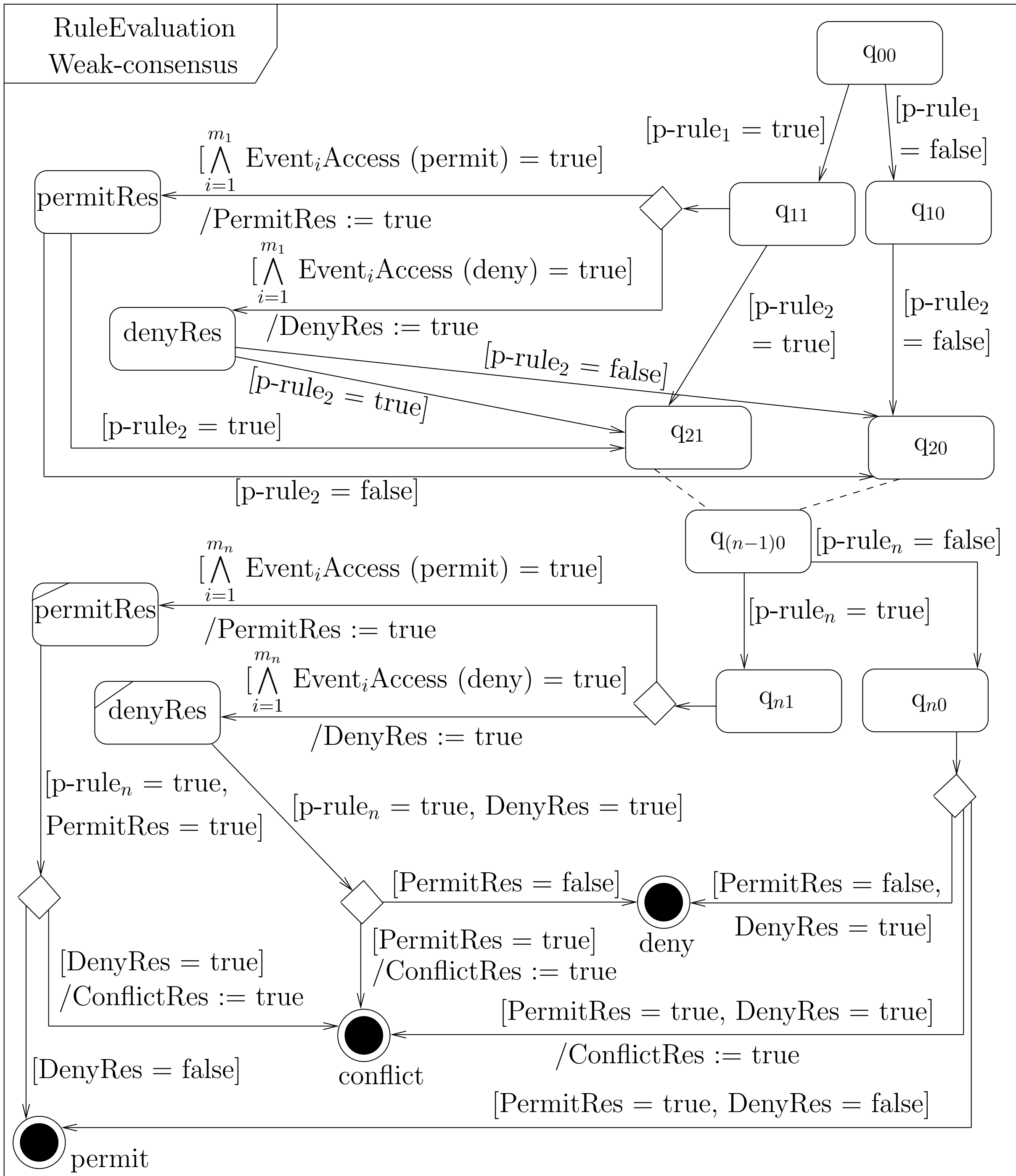


Figure F.8: A UML state machine that uses AC rule definitions for weak-consensus rule-combining algorithm

Figure F.9 shows the algorithmic form for the weak-majority rule-combining algorithm that is similar to the weak-majority policy-combining algorithm, shown in Chapter 3.

```
initial state = state_00;
set NumPermit to zero;
set NumDeny to zero;
for i = 1 to n do
    // n = the number of rules in a policy
    if premise-rule_i = false then
        move to state_io ;
    else
        move to state_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            add one to NumPermit;
            move to state permitRes;
        else if Event_i Access(deny) = true for every element of EventResult then
            add one to NumDeny;
            move to state denyRes;
        end
    end
end
if i = n and NumPermit > NumDeny then
    move to state permit;
else if i = n and NumDeny > NumPermit then
    move to state deny;
end
```

Figure F.9: The defined AC rules and states for weak-majority rule-combining algorithm

Figure F.10 represents the state machine corresponding to the algorithmic description.

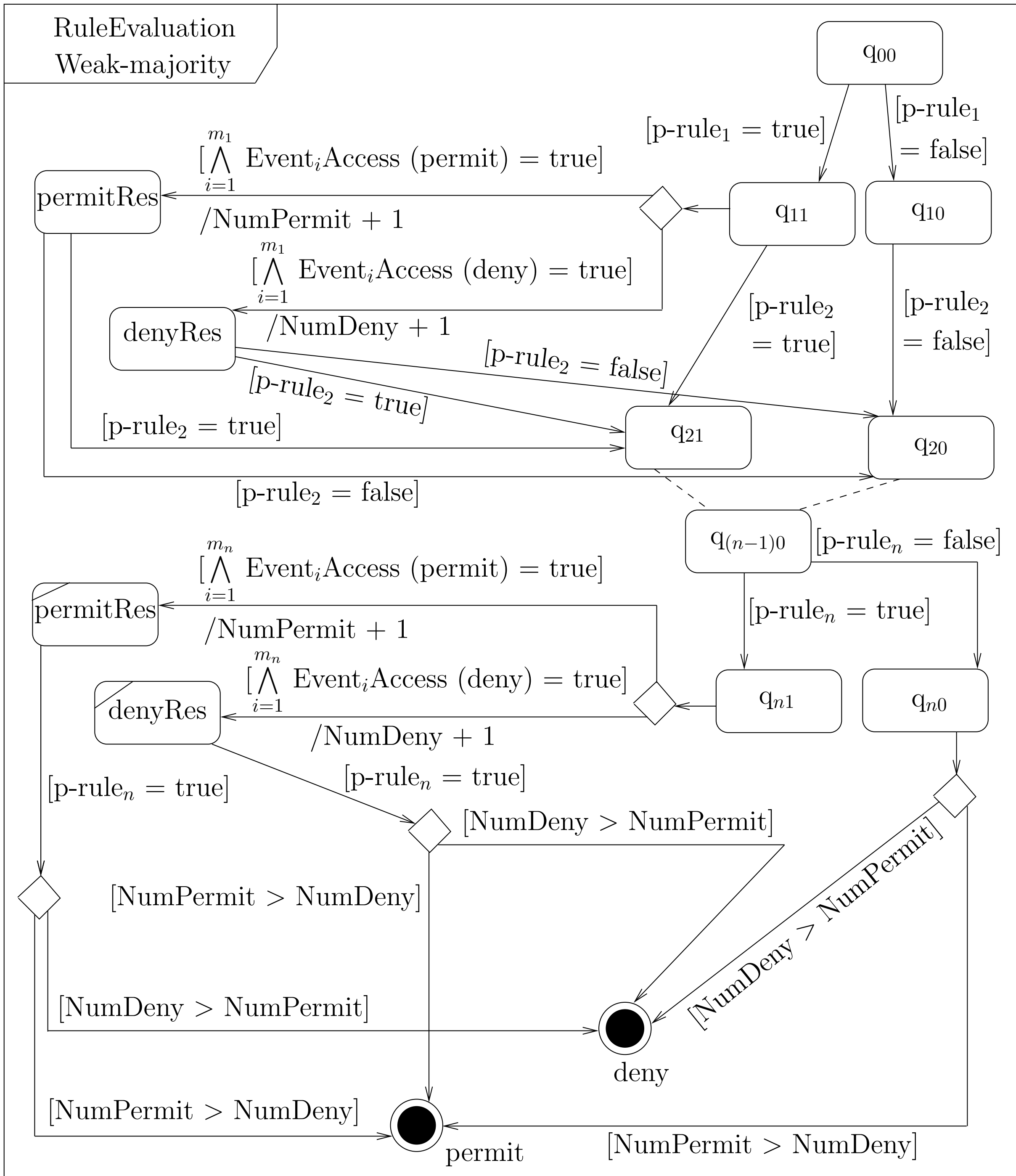


Figure F.10: A UML state machine that uses the AC rule definitions for weak-majority rule-combining algorithm

References

[1] Mohammad Al-Kahtani and Ravi Sandhu. A model for attribute-based user-role assignment. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*, pages 353–364, Las Vegas, USA, December 2002.

[2] Scott Ambler. *The Elements of UML 2.0 Style*. Cambridge University Press, 2005.

[3] Anne Anderson. A comparison of two privacy policy languages: EPAL and XACML. In *Proceedings of the 3rd ACM Workshop On Secure Web Services*, pages 53–60, Alexandria, VA, USA, November 2006.

[4] Jim Arlow and Ila Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*. Addison-Wesley, 2004.

[5] Alessandro Artale, Enrico Franconi, Nicola Guarino, and Luca Pazzi. Part-whole relations in object-centered systems: An overview. *Data & Knowledge Engineering*, 20(3):347–383, November 1996.

[6] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, second edition, 2007.

[7] Franz Baader and Werner Nutt. *Basic Description Logics*, chapter two in The Description Logic Handbook: Theory, Implementation, and Applications, pages 47–104. Cambridge University Press, 2007.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[9] David Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. Automated analysis of security-design models. *Information and Software Technology*, 51(5):815–831, 2009.

[10] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[11] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

[12] Messaoud Benantar. *Access Control Systems: Security, Identity, Management, and Trust Models*. Springer, 2006.

[13] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification: Model-checking Techniques and Tools*. Springer, 2001.

[14] Elisa Bertino, Piero Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. In *Proceedings of the ACM Workshop on Role-Based Access Control*, pages 21–30, 2000.

[15] Matthew Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[16] Michael Blaha and James Rumbaugh. *Object-oriented Modeling and Design with UML*. Pearson Prentice Hall, second edition, 2005.

[17] Hiawatha Bray. Payroll website still not secured. The Boston Globe, March 1, 2005.

[18] The Business Rules Group (BRG). *Defining Business Rules: What are They Really? Revision 1.3*. http://www.BusinessRulesGroup.org, July 2000.

[19] Achim Brucker and Burkhart Wolff. HOL-OCL: A formal proof environment for UML/OCL. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 97–100, Budapest, Hungary, March 2008.

[20] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. Past, present, and future trends in software patterns. *IEEE Software*, 24(4):31–37, 2007.

[21] Carla Carnaghan. Business process modeling approaches in the context of process level audit risk assessment: An analysis and comparison. *International Journal of Accounting Information Systems*, 7(2):170–204, 2006.

[22] Ramaswamy Chandramouli. Application of XML tools for enterprise-wide RBAC implementation tasks. In *Proceedings of the Fifth ACM Workshop on Role-based Access Control*, pages 11–18, Berlin, Germany, July 2000.

[23] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[24] Devin Cook and Multiple Contributors. Gold parseing system, available at http://goldparser.org/index.htm.

[25] Remco Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, 2008.

[26] Edsger Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[27] Matthew Dwyer, George Avrunin, and James Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice(FMSP)*, pages 7–15, Florida, USA, March 1998.

[28] Matthew Dwyer, George Avrunin, and James Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE)*, pages 411–420, Los Angeles, CA, USA, May 1999.

[29] E. Allen Emerson. The beginning of model checking: A personal perspective. In *Proceedings of the 25 Years of Model Checking*, pages 27–45, 2008.

[30] Herbert Enderton. *A Mathematical Introduction to Logic*. Academic Press, second edition, 2001.

[31] Hans-Erik Eriksson and Magnus Penker. *Business Modeling with UML: Business Patterns at Work*. Wiley, 2000.

[32] David Ferraiolo and D. Richard Kuhn. Role-based access control. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, October 1992.

[33] David Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, second edition, 2007.

[34] David Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, August 2001.

[35] Timothy Finin, Anupam Joshi, Lalana Kagal, Jianwei Niu, Ravi Sandhu, William Winsborough, and Bhavani Thuraisingham. ROWLBAC: Representing role based access control in OWL. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 73–82, CO, USA, 2008.

[36] Kathi Fisler, Shriram Krishnamurthi, Leo Meyerovich, and Michael Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 196–205, Saint Louis, US, May 2005.

[37] Jesús Arias Fisteus, Luis Sánchez Fernández, and Carlos Delgado Kloos. Applying model checking to BPEL4WS business collaborations. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 826–830, New Mexico, USA, March 2005.

[38] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

[39] Mark Fox. The TOVE project towards a common-sense model of the enterprise. In *Proceedings of the 5th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Lecture Notes in Computer Science, Springer*, volume 604, pages 25–34, June 1992.

[40] Mark Fox and Michael Gruninger. Enterprise modeling. *AI Magazine*, 19(3):109–121, 1998.

[41] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, 1979.

[42] Guido Geerts and William McCarthy. An ontological analysis of the economic primitives of the extended-REA enterprise information architecture. *The International Journal of Accounting Information Systems*, 3(1):1–16, March 2002.

[43] Guido Geerts and William McCarthy. Policy-level specifications in REA enterprise information systems. *Journal of Information Systems*, 20(2):37–63, Fall 2006.

[44] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007.

[45] Michael Harrison, Walter Ruzzo, and Jeffrey Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.

[46] Gerard Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23(5):279–295, May 1997.

[47] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[48] Pavel Hruby and with contributions by Jesper Kiehn and Christian Vibe Scheller. *Model-Driven Design Using Business Patterns*. Springer, 2006.

[49] Michael Huth and Mark Ryan. *Logic in Computer Science: Modeling and Reasoning about Systems*. Cambridge University Press, second edition, 2004.

[50] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[51] Wil Janssen, Radu Mateescu, Sjouke Mauw, and Jan Springintveld. Verifying business processes using SPIN. In *Proceedings of the 4th International SPIN Workshop*, pages 21–36, Paris, France, November 1998.

[52] Somesh Jha, Ninghui Li, Mahesh Tripunitara, Qihua Wang, and William Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(4):242–255, 2008.

[53] Jan Jürjens, Jörg Schreck, and Yijun Yu. Automated analysis of permission-based security using UMLsec. In *Proceedings of the 11th International Conference Fundamental Approaches to Software Engineering (FASE)*, pages 292–295, Budapest, Hungary, 2008.

[54] Lalana Kagal, Timothy Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 63–74, Lake Como, Italy, June 2003.

[55] Vahid Karimi. Formal analysis of access control policies for pattern-based business processes. In *Proceedings of the IEEE World Congress on Privacy, Security, Trust, and the Management of e-Business, Doctoral Symposium*, pages 239–242, Saint John, New Brunswick, Canada, August 2009.

[56] Vahid Karimi and Donald Cowan. Verification of access control policies for REA business processes. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 422–427, Seattle, Washington, USA, July 2009.

[57] Vahid Karimi and Donald Cowan. Access control models for business processes. In *Proceedings of the Internation Conference on Security and Cryptography (SE-CRYPT)*, pages 489–498, Athens, Greece, July 2010.

[58] Axel Kern and Claudia Walhorn. Rule support for role-based access control. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 130–138, Stockholm, Sweden, June 2005.

[59] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing Web access control policies. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 677–686, Banff, Alberta, Canada, 2007.

[60] Shriram Krishnamurthi. The CONTINUE server (or, how I administered PADL 2002 and 2003). In *Proceedings of the 5th International Symposium Practical Aspects of Declarative Languages (PADL)*, pages 2–16, New Orleans, LA, USA, 2003.

[61] Fred Kröger and Stephan Merz. *Temporal Logic and State System*. Springer, 2008.

[62] SAnToS laboratory. Spec patterns, available at http://patterns.projects.cis.ksu.edu/.

[63] James Lampe. Discussion of an ontological analysis of the economic primitives of the extended-REA enterprise information. *International Journal of Accounting Information Systems*, 3(1):17–34, 2002.

[64] Butler Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971.

[65] Butler Lampson. Computer security in the real world. *IEEE Computer*, 37(6):37–46, 2004.

[66] Ninghui Li, Qihua Wang, Wahbeh Qardaji, Elisa Bertino, Prathima Rao, Jorge Lobo, and Dan Lin. Access control policy combining: theory meets practice. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 135–144, Stresa, Italy, June 2009.

[67] James Martin and James Odell. *Object-Oriented Methods: a Foundation, UML Edition*. Prentice Hall, second edition, 1998.

[68] William McCarthy. The REA accounting model: A generalized framework for accounting systems in a shared data environment. *The Accounting Review*, 57(3):54–78, July 1982.

[69] William McCarthy. Ontologically-driven standards development for business process systems. Ontolog invited Speaker Presentation. http://ontolog.cim3.net/cgi-bin/wiki.pl?ConferenceCall_2008_06_05, 2008.

[70] Deborah McGuinness. Ontologies come of age: Bringing the world wide web to its full potential. In *Proceedings of the Spinning the Semantic Web*, pages 171–194, 2003.

[71] Renate Motschnig-Pitrik and Jens Kaasbøll. Part-whole relationship categories and their application in object-oriented analysis. *IEEE Transactions of Knowledge and Data Engineering*, 11(5):779–797, September/October 1999.

[72] Renate Motschnig-Pitrik and Veda Storey. Modelling of set membership: The notion and the issues. *Data & Knowledge Engineering*, 16(2):147–185, August 1995.

[73] Hiroaki Nakamura and Ralph Johnson. Adaptive framework for the REA accounting model. In *OOPSLA Workshop on Business Object Design and Implementation IV*, 1998.

[74] Qun Ni, Alberto Trombetta, Elisa Bertino, and Jorge Lobo. Privacy-aware role based access control. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 41–50, Sophia Antipolis, France, June 2007.

[75] James Odell. *Advanced Object-Oriented Analysis and Design Using UML*. Cambridge University Press, 1998.

[76] IDEF Family of Methods: A Structured Approach to Enterprise Modeling and Analysis. http://www.idef.com/.

[77] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language Primer, Version 2.0.* http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf, May 2007.

[78] Organization for the Advancement of Structured Information Standards (OASIS). *eXtensible Access Control Markup Language (XACML), Version 3.0, Committee Specification 01*, August 2010.

[79] Organization for the Advancement of Structured Information Standards (OASIS), Tim Moses (editor). *eXtensible Access Control Markup Language (XACML), Version 2.0*, February 2005.

[80] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, May 2000.

[81] Chun Ouyang, Eric Verbeek, Wil van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur ter Hofstede. WofBPEL: A tool for automated analysis of BPEL processes. In *Proceedings of the Third International Conference on Service-Oriented Computing (ICSOC)*, pages 484–489, Amsterdam, The Netherlands, December 2005.

[82] Jaehong Park and Ravi Sandhu. Towards usage control models: Beyond traditional access control. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies(SACMAT)*, pages 57–64, California, USA, June 2002.

[83] Doron Peled. *Software Reliability Methods*. Springer, 2001.

[84] Indrakshi Ray, Na Li, Robert France, and Dae-Kyoo Kim. Using UML to visualize role-based access control constraints. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 115–124, New York, USA, June 2004.

[85] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley Professional, second edition, 2005.

[86] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 43–50, San Diego, California, USA, June 2004.

[87] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[88] Ravi Sandhu and Qamar Munawer. How to do discretionary access control using roles. In *Proceddings of the third ACM Workshop on Role-Based Access Control*, pages 47–54, Virginia, USA, 1998.

[89] Ravi Sandhu and Pierangela Samarati. Authentication, access control, and audit. *ACM Computing Surveys*, 28(1):241–243, 1996.

[90] Andreas Schaad, Jonathan Moffett, and Jeremy Jacob. The role-based access control system of a European bank: a case study and discussion. In *Proceedings of the 6th Symposium on Access Control Models and Technologies (SACMAT)*, pages 3–9, Virginia, USA, May 2001.

[91] R. Scowen. Generic base standards. In *Proceedings of the Software Engineering Standards Symposium*, pages 25–34, 1993.

[92] Graeme Shanks, Elizabeth Tansley, Jasmina Nuredini, and Daniel Tobin. Representing part-whole relations in conceptual modeling: an empirical evaluation. *MIS Quarterly*, 32(3):553–573, September 2008.

[93] Graeme Shanks, Elizabeth Tansley, and Ron Weber. Representing composites in conceptual modeling. *Communications of the ACM*, 47(7):77–80, July 2004.

[94] Richard Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW)*, pages 183–194, Massachusetts, USA, June 1997.

[95] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, first edition, 1997.

[96] Karsten Sohr, Michael Drouineaud, Gail-Joon Ahn, and Martin Gogolla. Analyzing and managing role-based access control policies. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):924–939, July 2008.

[97] John Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole, 2000.

[98] William Stallings and Lawrie Brown with contributions by Mick Bauer and Michael Howard. *Computer Security: Principles and Practice*. Pearson Prentice Hall, 2008.

[99] The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). *International Standard, ISO/IEC 14977. Information technology–Syntactic metalanguage–Extended BNF*, first edition, December 1966.

[100] The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). *International Standard, ISO/IEC 15944-4:2007(E). Information Technology–Business Operational View–Part 4: Business Transaction Scenarios–Accounting and Economy Ontology*, first edition, November 2007.

[101] The Object Management Group (OMG). *Semantics of Business Vocabulary and Business Rules (SBVR), Version 1.0*, January 2008.

[102] Roshan Thomas and Ravi Sandhu. Towards a task-based paradigm for flexible and adaptable access control in distributed applications. In *Proceedings of the Workshop on New Security Paradigms*, pages 138–142, Rhode Island, USA, 1993.

[103] Roshan Thomas and Ravi Sandhu. Task-based Authorization Controls (TBAC): A family of models for active and enterprise-oriented authorization management. In *Proceedings of the Eleventh International Conference on Database Security (DBSec)*, pages 166–181, Lake Tahoe, California, 1997.

[104] Gianluca Tonti, Jeffrey Bradshaw, Renia Jeffers, Rebecca Montanari, Niranjan Suri, and Andrzej Uszok. Semantic web languages for policy representation and reasoning: A comparison of KAoS, Rei, and Ponder. In *Proceedings of the International Semantic Web Conference*, pages 419–437, October 2003.

[105] Jerrold Wagener. Guarded command. In *Encyclopedia of Computer Science*, pages 761–762. John Wiley, 2003.

[106] Wikibook, Logic for Computer Scientists/Predicate Logic/Semantics, available at http://en.wikibooks.org/wiki/Logic_for_Computer_Scientists/Predicate_Logic/Semantics.

[107] Morton Winston, Roger Chaffin, and Douglas Herrmann. A taxonomy of part-whole relations. *Cognitive Science*, 11(4), 1987.

[108] Jim Woodcock and Martin Loomes. *Software Engineering Mathematics*. Addison-Wesley, 1988.

[109] Tom Zeller. Not yet in business school, and already flunking ethics. The New York Times, March 14, 2005.