UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

A Declarative and Fine-Grained Policy Language for the Web Application Domain

by

Seyed Hossein Ghotbi

Thesis for the degree of Doctor of Philosophy

June 2014

Declaration of Authorship

I, Seyed Hossein Ghotbi, declare that the thesis entitled *A Declarative and Fine-Grained Policy Language for the Web Application Domain* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was carried out wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: [79, 78]

Signed:...

Date:..

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
Electronics and Computer Science

Doctor of Philosophy

A DECLARATIVE AND FINE-GRAINED POLICY LANGUAGE FOR THE WEB APPLICATION DOMAIN

by Seyed Hossein Ghotbi

A Web application that deploys on a set of servers and can be accessed by a large number of users over the Internet requires efficient security mechanisms. The core element in security is access control that enforces desired policies over the shared objects of the system and stops the unauthorised users to operate on these objects. Moreover, the used access control mechanism needs to be managed, through authorisation management elements, during the run-time of the system by the administrators. Therefore, the development of such models and their mechanisms are a main concern for secure systems development. Fine-grained access control and their authorisation management models provide more customisation possibilities and administrative power to the developers; however, in Web applications these models are typically hand-coded without taking advantage of the data model, object types, or contextual information.

This thesis presents the design, implementation and evaluation of Φ, a declarative, fine-grained policy language that enables the developer to define a set of fine-grained access control and authorisation management models for a Web application. For Φ, three types of access control and authorisation management models were designed and implemented. These models, used by Φ, are based on four main access control approaches, namely attribute-, discretionary-, mandatory-, and role-based access control models. For efficiency and flexibility, each access control model can be used with an authorisation management model. Φ compiler, first validates and verifies all these models based on written transformation strategies and verifies them by translating them into logical satisfiability problems to check the models for correctness and completeness, and against independently defined coverage criteria. If the models pass these tests, the generator then compiles them down to the existing tiers of WebDSL, a domain specific Web programming language.

Contents

Declaration of Authorship iii

Acknowledgements xix

Nomenclature xxi

1 Introduction 1
- 1.1 Security and Access Control . . . 3
- 1.2 Access Control and Web Applications . . . 3
- 1.3 Problem Statement . . . 4
- 1.4 Research Contributions . . . 8
- 1.5 Thesis Structure . . . 8

2 Background and Related Work 11
- 2.1 Access Control and Management Models . . . 11
  - 2.1.1 Access Control Models . . . 12
  - 2.1.2 Authorization Management Mechanisms and Architectures . . . 21
- 2.2 Access Control and Authorization Management Development Mechanisms 27
  - 2.2.1 Policy Languages . . . 27
  - 2.2.2 Database Approaches . . . 35
  - 2.2.3 Automated Policy Testing . . . 37
- 2.3 WebDSL . . . 40
  - 2.3.1 WebDSL Language Layers . . . 42
  - 2.3.2 Extending WebDSL . . . 50
- 2.4 Characteristic Requirements . . . 56
- 2.5 Summary and Conclusions . . . 57

3 Syntax and Semantics of Φ Models 59
- 3.1 Introduction . . . 59
  - 3.1.1 Linear Temporal Logic . . . 61
  - 3.1.2 Specification Language . . . 62
  - 3.1.3 Abstract Web Application Semantics . . . 63
  - 3.1.4 Semantics of a Web Application with Sessions . . . 64
- 3.2 ΦDAC Syntax and Semantics . . . 66
  - 3.2.1 ΦDAC Syntax Definition . . . 66
  - 3.2.2 Generic ΦDAC Semantics . . . 70
- 3.3 ΦMAC Syntax and Semantics . . . 71
  - 3.3.1 ΦMAC Syntax Definition . . . 72

3.3.2 Generic ΦMAC Semantics . . . 73
3.4 ΦRBAC Syntax and Semantics . . . 75
3.4.1 Syntax Definition . . . 75
3.4.2 Generic Semantics . . . 77
3.5 ΦDACAM Syntax and Semantics . . . 79
3.5.1 Abstract Syntax . . . 79
3.5.2 Generic Semantics . . . 80
3.5.2.1 ΦDACAM enforcement on ΦDAC . . . 80
3.5.2.2 ΦDACAM enforcement on ΦMAC . . . 82
3.5.2.3 ΦDACAM enforcement on ΦRBAC . . . 83
3.6 ΦMACAM Syntax and Semantics . . . 85
3.6.1 ΦMACAM Syntax . . . 85
3.6.2 Generic ΦMACAM Semantics . . . 86
3.6.2.1 ΦMACAM enforcement with ΦDAC . . . 86
3.6.2.2 ΦMACAM enforcement with ΦMAC . . . 87
3.6.2.3 ΦMACAM enforcement with ΦRBAC . . . 88
3.7 ΦRBACAM Syntax and Semantics . . . 89
3.7.1 ΦRBACAM Syntax . . . 89
3.7.2 Generic Semantics . . . 90
3.7.2.1 ΦRBACAM enforcement with ΦDAC . . . 91
3.7.2.2 ΦRBACAM enforcement with ΦMAC . . . 92
3.7.2.3 ΦRBACAM enforcement with ΦRBAC . . . 93
3.8 Summary and Conclusion . . . 95

4 Compilation of Φ Models 97
4.1 Introduction . . . 97
4.2 Validation and Verification Phase . . . 99
4.2.1 Error Handling . . . 101
4.2.1.1 Access Control Multiplicity Validation . . . 101
4.2.1.2 Object Scope Validation . . . 101
4.2.1.3 Object-Operation Relation Validation . . . 103
4.2.1.4 Policy Term Correctness . . . 103
4.2.1.5 Access Element Correctness . . . 107
4.2.1.6 Each Case Correctness . . . 110
4.2.1.7 Overlapping Cases . . . 114
4.2.1.8 Dead Authorisation Code . . . 116
4.2.2 Warning Handling . . . 119
4.2.2.1 Completeness Verification . . . 119
4.2.2.2 Coverage Verification . . . 121
4.3 Transformation Phase . . . 122
4.3.1 Data Model Elements and Enumeration Types . . . 122
4.3.1.1 Enumeration Types, Hierarchy- and Attribute-based Data Model Elements . . . 124
4.3.1.2 Discretionary- and Attribute-based Models . . . 126
4.3.1.3 Mandatory- and Attribute-based Models . . . 127
4.3.1.4 Role- and Attribute-based Models . . . 129
4.3.2 Sorting the defined Φ Models . . . 131

4.3.3 Transforming Access and Authorisation Management Rights 131
4.3.4 Predicate Handling Mechanisms 135
4.3.5 Access Control Predicates 137
4.3.6 Generating and Weaving Φ Panel 141
4.3.6.1 Access Control and Authorisation Management Activations 141
4.3.6.2 Authorisation Management Assignments 143
4.4 WebDSL Extension 144
4.5 Summary and Conclusion 144

5 Case Studies and Φ Evaluation 147
5.1 Introduction 147
5.2 Academic Research Group 148
5.2.1 Data Model 149
5.2.2 Page Structures 150
5.2.3 Page flows and their related functionalities 151
5.2.4 Defined ΦRBAC with ΦRBACAM authorisation models 153
5.2.5 Defined Φ Model 155
5.2.6 Quantitative Evaluation 159
5.3 Case Study Two: A Social Networking System 160
5.3.1 Data Model 161
5.3.2 Page Structures, flows and their related functionalities 162
5.3.3 Access Control and Authorisation Management 163
5.3.4 Defined Φ Model 165
5.3.5 Quantitative Evaluation 168
5.4 Φ Evaluation 169
5.4.1 Findings 169
5.4.2 Weaknesses 170
5.5 Summary and Conclusions 170

6 Conclusions and Future Work 173
6.1 Conclusion 173
6.1.1 Summary of contributions 173
6.1.2 Answers to Research Questions 175
6.2 Future Work 176

A Syntax Definition 179
A.1 Φ Language 179
A.1.1 Lexical Syntax 179
A.1.2 Objects 180
A.1.3 Policy terms 181
A.1.4 Policy and Coverage Cases 183
A.1.5 Access Control Models 184
A.1.6 Authorisation Management Models 186
A.1.7 Attachment to WebDSL's Syntax 187
A.2 Z3 Formalism 187
A.3 Phi.ini 187

B Validation and Verification in Φ **189**
B.1 Error Handling 189
B.1.1 Multiple Access Control Validation 189
B.1.2 Object Scope Validation 189
B.1.3 Validation of Object-Operation Relation 190
B.1.4 Policy Terms Correctness Validation 190
B.1.5 Access Element Correctness Validation and Verification 192
B.1.6 Validation and Verification for the Correctness of Each Case 193
B.1.7 Overlapping Validation and Verification 198
B.1.8 ΦMAC and ΦMACAM Models 199
B.1.9 ΦRBAC and ΦRBACAM Models 200
B.1.10 Dead Authorisation Code Verification 200
B.2 Warning Handling 200
B.2.1 Completeness Verification 201
B.2.2 Coverage Verification 201

C Defined Data, Access, and Authorisation Management Models **203**
C.1 Academic Research Group 203
C.1.1 User-based data models and their access control requirements 203
C.1.1.1 Person 204
C.1.1.2 Partner 205
C.1.1.3 Visitor 205
C.1.1.4 Intern 206
C.1.1.5 University Member 207
C.1.1.6 Academic 208
C.1.1.7 Postdoc 209
C.1.1.8 Postgraduate 209
C.1.1.9 Project Student 211
C.1.2 Activity-based data models and their access control requirements 212
C.1.2.1 External and Internal Events and Seminars 212
C.1.2.2 Blog, Post, and Comment 213
C.1.2.3 Project 214
C.1.2.4 Publication 215
C.1.2.5 Viva 216
C.1.2.6 Interests 217
C.1.2.7 News 217
C.1.2.8 Address 218
C.1.2.9 Calendar 218
C.2 Social Networking 219
C.2.1 Data model and their access rights 219
C.2.1.1 User Profile 219
C.2.1.2 Network 219

References **223**

List of Figures

1.1 Abstract View of Web Application Interactions 2

2.1 Confidentiality-Based mandatory access control [140] 13
2.2 Integrity-Based Mandatory Access Control [140] 14
2.3 Role-Based Access Control [72] 16
2.4 Flat Role-Based Access Control Model [72] 16
2.5 Hierarchical Role Based Access Control Model [72] 18
2.6 Constrained Role Based Access Control Model with static SOD [72] 19
2.7 Constraint Role Based Access Control Model with dynamic SOD [72] . . . 20
2.8 Symmetric Role Based Access Control Model [72] 20
2.9 Akenti Authorization Model [157] 23
2.10 PRIMA System Overview [118] 24
2.11 POSIX ACL and Grid Access Control List [118] 25
2.12 ShareEnabler Architecture [103] 26
2.13 Authorization Policy Syntax [61] 28
2.14 Instance declarations from policy type definition [61] 28
2.15 Filters on positive authorisation actions [61] 29
2.16 Information filter policy [61] 29
2.17 A Delegation Policy in Ponder [61] 29
2.18 A Refrain Policy in Ponder [61] 30
2.19 Obligation policy Syntax [61] 30
2.20 An obligation policy [61] 30
2.21 XACML abstract syntax [134] 31
2.22 XACML abstract architecture [32] 33
2.23 Related roles, permissions, and context based on the testing requirement [133] 41
2.24 WebDSL language layers 43
2.25 WebDSL to Seam transformation function [91] 44

4.1 Φ's Validation and Verification Phase 100
4.2 Multiple ΦDAC definitions for a Web application led to an error. 101
4.3 The used object, `Students` page does not exist within the Web application.102
4.4 An error based on the relation between an object and its operation. 103
4.5 An error based on used undefined roles as a policy term 104
4.6 A usage of undefined role led to an error. 104
4.7 A duplication in a policy term led to an error. 105
4.8 The defined user identifier is incorrect. 106
4.9 This model led to an error because label `Sec` is duplicated. 107

4.10 Shared label names between *confidentiality* and *integrity* labels are not allowed. . . . 107
4.11 A duplication within *SSOD* relations led to an error. . . . 108
4.12 Inheritance relations are transform to an FOL model by the `ET5Checker` strategy and then automatically checked by the Z3. . . . 109
4.13 An incorrect case within the ΦDAC model based on the user identifier conflict. . . . 110
4.14 An incorrect case based on comparison between user- and data-based attributes. . . . 111
4.15 An incorrect case based on MAC-based properties of ΦMAC model. . . . 112
4.16 An incorrect case based on defined confidentiality-based characteristics. . . . 112
4.17 An incorrect case within the ΦRBAC model based on defined SSOD relation. 113
4.18 Architecture of overlapping and incompleteness checks 115
4.19 Overlapped cases in ΦDAC model created an error. . . . 116
4.20 Nested controlled objects and predicates may create a set of conflicts. . . . 117
4.21 Three main steps that check the *Dead authorisation code* within ΦDAC and ΦDACAM models. . . . 118
4.22 Φ's transformation pipeline 123
4.23 Three sets of enumeration types that hold the *controlled object* and *operation* types, and comparison signs 124
4.24 Generated Data Model elements that will store the *attribute-based* constraints 125
4.25 Generated Data model for the defined ΦDAC model 126
4.26 Generated Data model for the defined ΦDACAM model 127
4.27 Generated Data model for the defined ΦMAC model 127
4.28 Generated Data model for the defined ΦMACAM model 129
4.29 Generated Data model for the defined ΦRBAC model 129
4.30 Generated Data model for the defined ΦRBACDM model 130
4.31 Sorting policy cases based on the *type* of the object and its related operation, and assign a unique identifier to the list 131
4.32 *Global variables* are generated based on list of sorted policy cases. . . . 132
4.33 *Global variables* are generated based on a list of sorted user-based attribute policy cases 133
4.34 *Global variables* are generated based on a list of sorted user-based attributes policy cases 134
4.35 The function `PDACCheck` checks the Φ*DAC-based* policies, by finding the object's rights and calling other functions to check the related *attribute-based* policies. . . . 136
4.36 Simple illustration on weaving a function call (i.e., *PDACCheck*) around *all* the instances of the controlled object and its related operations 137
4.37 Guarding the entity `Student` and its *create* operation by generating the function calls and weaving them around its input and action call 138
4.38 Guarding the entity `User` (part *A*), and property `Staff.salary` and their related *output* elements 138
4.39 Guarding the *edit* input mechanism and its related *action*, by generating and weaving predicate function call around them 139
4.40 Guarding the *delete* operation related to the `Publication` entity 139
4.41 Guarding the the objects `salaries` template call and XML node `address` and their related *read* and *secret* operation 140

4.42 Guarding the the objects `marks` group and block `sideNote` and their related *read* and *secret* operation. 140
4.43 Φ guards the the page `topCustomers` and its navigation link, by the generating and weaving the function call. 140
4.44 Φ Modifies authentication template by adding a navigation 142
4.45 Φ Panel Workflow 142

5.1 PLUS homepage 149
5.2 Different user types in the PLUS data model 150
5.3 Adding an event through a user panel 151
5.4 Academic page 151
5.5 Interests page 152
5.6 Defined roles and their inheritance relation for the PLUS web application 154
5.7 Defined SSOD relations for the defined roles 154
5.8 Social networking homepage 162

C.1 Different user types in the PLUS data model 204
C.2 Event hierarchy in the research group data-model 212

Listings

1.1 Hard Coded Fine-Grained Role-Based Access Control 5
1.2 Hard Coded Fine-Grained Role-Based Access Control 5
2.1 A rule example in XACML 34
2.2 Student entity 45
2.3 Data validation in WebDSL 46
2.4 User interface in WebDSL RBAC 46
2.5 Defining authentication credentials and an access rule for a page 47
2.6 Defining Role entity and extending User entity and securityContext 48
2.7 Point cuts for initial access setting on the pages of the web application . . 49
2.8 Student function in WebDSL 49
2.9 An example of access rules on pages and templates 50
2.10 An example of context-free syntax 51
2.11 Using Concrete Syntax 53
2.12 Type checking 54
2.13 Syntax normalization 54
2.14 Requiring list of Blog 55
2.15 Local-to-global syntactic normalization 55
2.16 Generative Derive Mechanisms 56
3.1 The LTL Specification Language 62
3.2 Semantics of a Web app in the first-order temporal logic 63
3.3 The signature of a Web app with sessions 65
3.4 Semantics of a Web app with sessions in first-order temporal logic 66
3.5 Syntax of ΦDAC 67
3.6 ΦDAC Example 69
3.7 Semantics of ΦDAC in first-order temporal logic 71
3.8 The ΦMAC Abstract Syntax 72
3.9 An example of ΦMAC model 73
3.10 Semantics of ΦMAC in first-order temporal logic 75
3.11 Controlled Objects and related operations 76
3.12 A ΦRBAC Example 77
3.13 Semantics of ΦRBAC in first-order temporal logic 78
3.14 ΦDACAM Abstract Syntax 79
3.15 A ΦDACAM example 80
3.16 Semantics of a Web application with ΦDAC and ΦDACAM 81
3.17 Semantics of a Web application with ΦMAC and ΦDACAM 82
3.18 Semantics of a Web application with ΦRBAC and ΦDACAM 84
3.19 ΦMACAM Abstract Syntax 85

3.20 A ΦMACAM Example 86
3.21 Semantics of a Web application with ΦDAC and ΦMACAM 86
3.22 The semantics of a Web application with ΦMAC and ΦMACAM 88
3.23 The semantics of a Web application with ΦRBAC and ΦMACAM 89
3.24 The ΦRBACAM Abstract Syntax 90
3.25 A ΦRBACAM Example 90
3.26 Semantics of a Web application with ΦDAC and ΦRBACAM 91
3.27 The semantics of a Web application with ΦMAC and ΦRBACAM 93
3.28 The semantics of a Web application with ΦRBAC and ΦRBACAM 94
5.1 Defined access control and authorisation management model for the academic research group 155
5.2 Defined ΦDAC model with discretionary-based authorisation management for the user profiles 166
A.1 The defined lexical syntax 180
A.2 Used controlled objects within Φ language 181
A.3 Policy terms in Φ 182
A.4 Policy cases syntax definition 183
A.5 Syntax definition of coverage cases 184
A.6 The ΦDAC defined syntax 184
A.7 ΦMAC defined Syntax 185
A.8 ΦRBAC defined syntax 186
A.9 Φ-based authorisation management models 186
A.10 Φ models are attached to the WebDSL's definition element 187
A.11 Defined syntax related to Z3 188
A.12 Defined syntax for initialisation file 188
C.1 Person data model 204
C.2 Partner data model 205
C.3 Visitor data model 206
C.4 Intern data model 206
C.5 Data model related to the university members 207
C.6 Academic data model 208
C.7 Defined data model for postdocs 209
C.8 Defined data model for postgraduate students 210
C.9 Defined data model for master degree students 211
C.10 Defined data model for internal and external events and seminars 213
C.11 Defined data models for Blog, Post, Comment, and Tag 214
C.12 Project data model 215
C.13 Defined data model for publication 215
C.14 Defined data model for viva 216
C.15 Defined data model for users' interests 217
C.16 Defined data model for news 217
C.17 Defined data model for address 218
C.18 Calendar data model 218
C.19 Defined entities for the user, status, and wall post 220
C.20 Defined Model for the network, topics, and their related comments 221

List of Tables

1.1 Instances of the student data . . . 4

2.1 An example of an access control matrix [140] . . . 13
2.2 Different Levels of Role Based Access Control [140] . . . 17
2.3 Comparisons between modern Web Frameworks . . . 42

4.1 Errors based on individual *attribute-based* policy terms . . . 106

B.1 Different types of errors related to the shared object scopes within elements of access and authorisation management models . . . 190
B.2 Different types of errors related to the object-operation relation validation within elements of access and authorisation management models . . . 190
B.3 Different types of errors related to the out of context validation within elements of access and authorisation management models . . . 191
B.4 Different types of errors related to the duplication validation within elements of access and authorisation management models . . . 191
B.5 Different types of errors related to the validation of the out of *attribute-based* policy terms within the access and authorisation management models . . . 191
B.6 Different types of errors related to the duplication validation within ΦMAC, ΦMACAM, ΦRBAC, and ΦRBACAM models . . . 192
B.7 Two types of errors related to the validation of shared labels within the defined *confidentiality* and *integrity* labels . . . 192
B.8 Two types of errors related to the validation of the defined objects within the ΦMAC and ΦMACAM models . . . 192
B.9 Two types of errors related to the roles' hierarchy verification within the ΦRBAC and ΦRBACAM models . . . 193
B.10 Four errors related to the user identifier verification within the ΦDAC and ΦDACAM models . . . 193
B.11 Two types of errors related to the user-based attributes verification within the ΦDAC and ΦDACAM models . . . 193
B.12 Four different error messages related to the data-based attributes verification within the ΦDAC and ΦDACAM models . . . 194
B.13 Four different error messages related to the comparisons between user- and data-based attributes within the ΦDAC and ΦDACAM models . . . 194
B.14 Eight different error messages related to the system-based attributes verification within the ΦDAC and ΦDACAM models . . . 194
B.15 Different error messages related to the un allowed operations within ΦMAC and ΦMACAM models . . . 195

B.16 Different error types related to the MAC property verification within ΦMAC and ΦMACAM models 195
B.17 Different error types related to the strict-property validation within ΦMAC and ΦMACAM models 196
B.18 Different error types related to the *read-down* property within ΦMAC and ΦMACAM models 196
B.19 Different error types related to the *write-up* property within ΦMAC and ΦMACAM models 196
B.20 Different error types related to the *read-up* property within ΦMAC and ΦMACAM models 197
B.21 Different error types related to the *write-down* property within ΦMAC and ΦMACAM models 197
B.22 Different error types related to the attribute-based policy terms and their signs within ΦMAC and ΦMACAM models 198
B.23 Different error types related to the relation and policy signs checks within ΦRBAC and ΦRBACAM models 198
B.24 Different error types related to the attribute-based policy terms and their signs the ΦRBAC and ΦRBACAM models 199
B.25 Different error types related to *overlapping* cases within ΦDAC and ΦDACAM models 199
B.26 Different error types related *overlapping* cases within ΦMAC and ΦMACAM models 199
B.27 Different error types related *overlapping* cases within ΦRBAC and ΦRBACAM models 200
B.28 Different types of errors with the Φ's access and authorisation management models 200
B.29 Different categories of warnings related to the completeness verification within Φ models 201
B.30 Six different categories of warning related to the coverage verification within Φ models 201

Acknowledgements

It is my pleasure to thank a number of people who helped me to be where I am today. First, I have to thank Dr. Bernd Fischer, who accepted me to be a PhD student at the University of Southampton, and supported and guided me patiently, throughout my PhD. Second, I am very obliged to School of Electronics and Computer Science, University of Southampton for awarding me the full studentship for conducting my PhD research. Third, I would like to thank a number of lecturers that motivated me to continue my studies, Mr. James King from London Metropolitan University, Professor Mark Harman from UCL and Professor Michael Luck from Kings College London.

I would also like to thank my parents and family who supported me throughout this process. Their support was essential throughout the years in enabling me to never distract from the thing I love the most to do in my life, regardless how hard it is.

Last but not least, I owe a big thank you to the love of my life Dr. Faezeh A. Hassani, who was the massive positive force throughout my PhD. Even though she is into electronics, we did and still have lots of discussions on various things including my area of work. I am sure she can give a talk for 30 minutes on access control.

Nomenclature

ACL	Access Control List
DCC	Dependency Core Calculus
DAC	Discretionary-based Access Control
DSOD	Dynamic Separation of Duties
FOL	First-order Logic
GUI	Graphical User Interface
PAP	Policy Administration Point
PEP	Policy Enforcement Point
PDP	Policy Decision Point
LOC	Lines of Code
LTL	Linear Temporal Logic
MAC	Mandatory-based Access Control
MIM	Model Implementation Mapping
MDSS	Metadata Sharing Service
RBAC	Role-based Access Control
ROA	Role-based Originator Authorisation
SDF	Syntax Definition Formalism
SMT	Satisfiability Modulo Theories
SUT	System Under Test
SOD	Separation of Duties
SSOD	Static Separation of Duties
XACML	XML-based Access Control Modelling Language

To people in my life for their love and support:

My parents: Ezat and Moein
My wife: Faezeh Arab Hassani

Chapter 1

Introduction

The Web is an important aspect of computer science, and Web application development is thus an important factor of software development. Web applications, such as Facebook or Amazon, are deployed on a set of servers, and become accessible through a web browser, wherever there is an Internet connection. The ease of use and accessibility of such applications, via any web browser and operating system, make them popular among users. The available server-oriented deployment mechanisms for web applications such as Jetty [10], Tomcat [1], or a Google engine [7] also make them a suitable development choice for different domains (such as e-business [4] or social networking [6, 18]) for providing and/or sharing their services [12, 19]. As the number of users of a web application grows, its security becomes a major concern [54, 138]. Authentication mechanisms [128, 52, 151] can be used to differentiate inside users from outside users, and help users to operate on their own personal data. This is, however, not enough when the data is shared among inside users, as for example in an organization. In such an environment different users typically have different sets of privileges on different subsets of the shared data. Therefore, there is a need for controlling the access to the shared data, based on specific policies [140, 71, 123]. One way of enforcing a desirable security mechanism for inside users is by using one of the many types of access control [140, 146, 135, 43, 143], such as discretionary (DAC), mandatory (MAC), and role-based access control (RBAC) [71].

Web applications consist of elements that have different granularity levels and that are scattered through the entire application code. Therefore, it is essential to have an efficient access control model and mechanism to deal with the distributed fine-grained access control code that protects the Web applications' elements. In this thesis we investigate modelling and implementing a set of fine-grained access control models and automated deployment mechanisms for the domain of Web applications. Our work provides a novel environment for declaratively defining a combination of DAC, MAC, or RBAC policies with data and system constraints, and to test objectives over a range of objects with different granularity levels. These models can be formally analysed and verified. The proposed approach is implemented on top of WebDSL [162], a domain specific language for Web application development. It uses code generation techniques to generate and weave the access control predicates around the objects within the application code written in WebDSL.

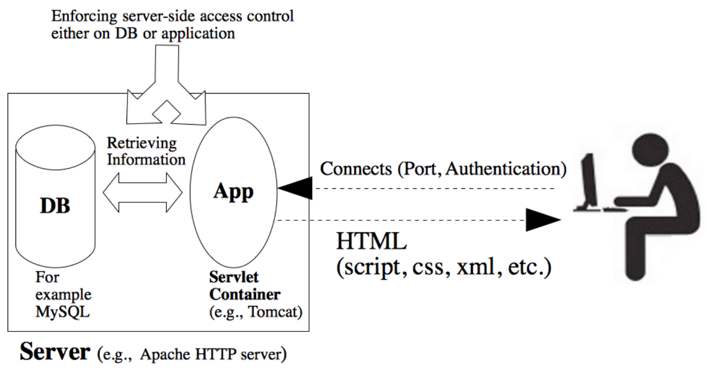

Enforcing server-side access control either on DB or application
Retrieving Information
DB
App
For example MySQL
Servlet Container (e.g., Tomcat)
Connects (Port, Authentication)
HTML (script, css, xml, etc.)
Server (e.g., Apache HTTP server)

Figure 1.1: Abstract View of Web Application Interactions

There are two main approaches that I can choose from when we try to introduce an access control mechanism within the domain of Web applications (see Figure 1.1). First, I can use database-oriented techniques such as query rewriting [155, 136, 88]. Here the Web application remains unchanged and all access control is delegated to the database. When the application then tries, on behalf of a user with certain authentication rights, to retrieve data from the database, the database management system modifies the query to take these rights into account and to ensure the data out of user's rights cannot be retrieved. Second, I can use a code-based approach that integrates the access control into the application code that is deployed on the server. The focus in this research is on the code-based approach, for the following reasons:

Validity and Verifiability: In this research I aim to provide an automated mechanism for validating and verifying the defined access control model with respect to itself and its target application, and then generate and weave the access control code into the target application. In this way I do not need to wait until the deployment of the Web application is finished, and then check for all the access requests based on all the access rights and analyse the retrieved information from the database to ensure the effectiveness of the access control elements.

Clear Relation: Similar to D.Groenewegen and E.Visser [85], I aim to create a very clear connection between the access control and the application's elements. Therefore the access control mechanism and its related policies can be easily analyzed and adjusted with respect to themselves and their target application, before its deployment.

Configuration: In line with the WebDSL original access control approach [85], I want our approach to be easily extensible and flexible to disallow inconsistency between access control policies and mechanisms, and allow integration of access control administrative tasks within the application.

1.1 Security and Access Control

Security is a multifaceted concept, with different types of concerns such as encryption, authentication, or authorisation in different domains such as networking [96] and Web applications [150]. Access control mechanisms [69, 165, 148, 151] are a part of the authorisation concerns, but not a full solution for these. In general, the aim of enforcing security policies in a system is to stop both internal and external users from violating data confidentiality (i.e., data-flow violations) [167] and data integrity (i.e., data modification violations) [47]. To fulfil these goals, both authentication [52, 151] and authorisation [139, 99] mechanisms are needed. Authentication is used by users to identify themselves to the system [52], typically by providing their credentials (e.g., their password or biometric data). After this point, authorisation is used to enforce the access rights specified for the internal users. In this thesis we focus on authorisation and assume that authentication is handled elsewhere.

In line with Sandhu et al. [145], I assume that, in an access control model, three types of basic elements exist, on which the evaluation of an access request is based. These elements types are subjects, objects and operations. A *subject* represents any actor of a system; typically subjects are users, but they could also be processes, agents, etc. An *object* is a system component that needs to be protected by the access control mechanism against the *operations* available on these objects, such as database create, update, delete (CRUD) operations, or file execution. As Samarati et al. [140] point out, there is a distinction between the access control *model* and the access control *mechanism*. An access control model deals with the protection requirements to be enforced, but its mechanism is dealing with how to implement it during software development. In this research, I address fine-grained access control models and their mechanisms.

1.2 Access Control and Web Applications

Web applications typically consist of elements that have different granularity levels and that are scattered throughout the entire application code, which makes the implementation and validating and verifying of the access control model and its mechanism difficult. For example, a page in a Web application can contain smaller elements such as sections or even more fine-grained elements such as a date of a record that is retrieved from the database and displayed in a cell of a table. Currently, developers need to hand-code the fine-grained access control elements around the objects that require access control, using languages such as Scala [94], XACML [119, 87], or Ponder [61]. However, approaches as those listed above have drawbacks in several areas which complicate development of fine-grained access control models and can easily introduce security holes.

Abstraction level: They lack the right *abstraction level* to define flexible access control models that allow the specification of different policies for different individual objects based on the context of the access control terminologies.

Separation of concerns: They lack the *separation of concerns* [63] between access control and application [56]. If I can develop the access control components separately, the developers can check for potential vulnerabilities separately and mechanically, instead of manually analyzing the access control predicates scattered throughout the application code, which is an error-prone and time consuming process.

Validation and Verification phase: They do not provide any validation and verification mechanism for the developer to validate and verify the hand-written code. This can lead to misconfiguration of the access control element of the application. There must be validation and verification mechanism for the developer so that the defined access control model and mechanism, can be checked for errors and warnings, with respect to the model itself and its target language.

Access control for the social Web: Web applications will become more collaborative, and none of the mentioned languages provides an efficient access control model and/or mechanism to define and implement a combination of different access control models for a single Web application. The developer should be able to define a set of access control models for a single Web application such that each access control model enforces a certain authorization requirement for a set of controlled objects.

1.3 Problem Statement

I use the following example, with the assumption that a standard RBAC mechanism is controlling access to individual pages, to illustrate the problem of manually implementing a fine-grained role-based access control approach.

Table 1.1: Instances of the student data

Username	**Homepage**	**Medical History**	**Mark**
John	www.ecs.soton.ac.uk/js10	Diabetes	90
Bryan	www.ecs.soton.ac.uk/br1	None	80
Douglas08	www.ecs.soton.ac.uk/d11	None	70

I consider a simple student table (shown in Table 1.1) that contains each users' username, homepage, medical history, and mark. If the access control policy states that the administrator can edit all the students' information except their mark, and that a teacher can edit each student mark, see (but not edit) the information related to username, and homepage, and cannot see the medical history, the developer needs to define two tables and put them into two different pages, because of the coarse granularity of the existing access control mechanism. However, such an approach leads to duplicated code. Instead the developer can employ a fine-grained approach, by hard-coding the access control checks around fine-grained components (in this case, the table's header, rows, and columns). The following high-level pseudo code illustrates how such an implementation could look.

```
page example1 {
 table {
  tableHeader {"User name" , "Homepage", "Medical History"}
    if (currentUser.role == admin){
     for ( u in User) {
      row {edit(u.username), edit(u.url), edit(u.medHistory), text("***")}
     }
    }
    if(currentUser.role == teacher){
     for (u in User){
      row {text(u.username), text(u.url), text("***"), edit(u.mark)}
     }
    }
  }
}
```

Listing 1.1: Hard Coded Fine-Grained Role-Based Access Control

This very simple example illustrates on a high level the flexibility of hard-coding access control policies. However, if also shows that as the number of roles grows, more and more embedded access control checks need to be inserted to achieve a fine-grained access control approach.

Moreover, let us go beyond the notion of tables, and consider all elements inside the pages. As in the previous example, the access control checks could cover any element such as the header of a page or a sentence inside a paragraph. In this case, as the following high-level pseudo code shows, the embedded fine-grained role-based access control could be used the enforce the desirable policy. It should be clear from the code that the developer needs to do more hard-coding to enforce the access control on more fine-grained elements. This is not a desirable approach as it leads to problems such as poor maintainability and reliability, code duplication, etc.

```
page example2 {
  if (currentUser.role == admin){
    header {This is the header of a page seen by the admin}
  }
  text("This is a sentence that could be seen by everyone.")
  if (currentUser.role == teacher){
    text("This is a sentence that could be seen by the teachers.")
  }
  if (currentUser.role == student){
    text("This is a sentence that could be seen by the students.")
  }
}
```

Listing 1.2: Hard Coded Fine-Grained Role-Based Access Control

The following points reflect the research problems in the area of access control and Web applications, which is addressed in this research.

1. The lack of efficient declarative fine-grained access control approaches [79, 119, 85] in the domain of web application engineering.
2. The granularity of access control code over the web application code, in particular, the enforcement of access control concerns on a fine level of granularity [119, 9].
3. The lack of validation and verification mechanisms for fine-grained access control models and their mechanisms, with respect to both, interval consistency and integration with the target application.
4. An efficient access control model and mechanism for the social and collaborative Web.

Research Questions

From the problems identified above I can now derive four specific research questions that will be answered in the reminder of the thesis.

Research Question 1: *What are the advantages of providing an abstract fine-grained access control modelling mechanism to the developer?*

One efficient way to dealing with distributed fine-grained access control code is to use an approach with a higher level of abstraction, compared to the available mainstream languages that support DAC, MAC, or RBAC [119, 9]. Then, instead of writing scattered access control code, code generation techniques could be used to generate and merge it with the rest of the application. This solution provides three main advantages to the developer:

Productivity: The developer just needs to define the access control on a high level of abstraction and the required access control checks will be generated automatically.

Maintainability: The abstract approach that enables the developer to define the fine-grained role-based access control makes it possible to change the access control in one point, and the required change will be generated and merged with the application code.

Reliability: The validation and verification of the access control component becomes easier, as the developer uses a more abstract and compact notation to define and enforce the security policy.

Research Question 2: *What are the notions of fine-grained access control models independently of their implementation approach?*

In the existing literature, the notion of fine-grained access control refers only to models that can control access to fine-granular objects but where the policies themselves remain coarse-grained, and thus lack flexibility. For example, a number of studies [156, 113] discuss fine-granular access control in the context of databases in terms of the table structure (i.e., columns, rows, etc.), while others [95, 154] discuss it in the context of XML and the hierarchical structure of XML documents. Therefore the challenge here is to define a fine-grained access control based on the access control domain terminalogies and not the context of the programming languages.

Research Question 3: *What elements need to be considered during the validation and verification phase of a fine-grained access control model and its mechanism?*

Access control as a software component needs to be semantically verified [158]. Recent studies [163] have shown that although verifying the access control model is essential, validating and verifying the model on its own is not enough. Any access control is defined to cover a set of objects in an application. Therefore, the validation and verification mechanism should also take into account the target application as well as the access control model.

Research Question 4: *What are the benefits and constraints of the fine-grained access control models and their mechanisms?*

Access control models are the important part of the security component of the system. They are build on the top of other elements such as authentication, and themselves become the foundation of the other part of the security components such as monitoring and provenance. Therefore it is very important to investigate the benefits and weaknesses of our fine-grained access control model.

1.4 Research Contributions

In the answering the research questions above, this thesis makes the following research contributions:

- **Fine-grained Access Control Modelling:** To address the first two research questions, I provide a set of fine-grained access control models for the Web application domain that enforces a separation of concerns between application and access control model at the right abstraction level. Therefore the developer can define the access control model based on the context of the access control models. Moreover, the developer can define a set of fine-grained access control models on a single Web application.
- **Validaiton and Verification Mechanism:** To tackle the third research question, I provide a novel validaiton and verification mechanism to check the correctness, completeness, and sufficiency of the model and the application via using an SMT solver and developer defined criteria. We also introduce the concept of *dead authorisation code* that could occur of a fine-grained access control mechanism, and show how we can check for dead authorisation code.
- **Code Generation Mechanism:** Our fine-grained access control model is implemented as an extension to a domain-specific language, WebDSL [83]. During this process I highlighted a set of benefits and constraints.

1.5 Thesis Structure

The thesis is structured as follows:

Chapter 2 (Background and Related Work): This chapter provides a literature review on access control models, mechanisms, and their granularity. Moreover, it discusses the code generation domain and gives an overview of the underlying language, WebDSL.

Chapter 3 (Syntax and Semantics of Φ models): This chapter first gives an overview look on Φ models and the Web application core semantics. Then, it provides a discussion on the concept behind coverage checks and controlled objects. Furthermore, it discusses the syntax and semantics of access and authorisation management models that can be defined by Φ.

Chapter 4 (Compilation of Φ Models): This chapter discusses in details about the the Φ architecture and its integration to a Web programming language, WebDSL.

Chapter 5 (Case Studies and Evaluation): First this chapter shows two cases studies that are motivated based on the direction of this research. Furthermore, it focuses on the findings and evaluation outcomes of the fine-grained access control model and the set of case studies.

Chapter 6 (Conclusions and Future Work): This chapter first give a summary of this research and an overview of the set of contributions of this research. Then it discusses my future work and publications as an outcome of this research.

Chapter 2

Background and Related Work

This research focuses on design and implementation of a declarative fine-grained policy language in the domain of Web applications. The aim of this policy language is to give the developer the ability to define a set of fine-grained access control and authorisation management models. As an outcome, it provides a declarative environment by extending a domain specific base language, WebDSL.

This chapter presents an overview of the prior work in the following four foundations that shaped the direction of this research:

Access Control and Authorisation Models: In the first part of this chapter I discuss the three main access control models and a set of authorization models that were introduced in the last four decades.

Development Mechanisms: I then discuss the development mechanisms related to these models in terms of language, data base, and the available testing techniques. At the end of this section I discuss the shortcomings of these approaches.

WebDSL: At the end I focus on the WebDSL language itself and its compiler elements and architecture. I also discuss how developers can extend this language by defining new syntax (using SDF) and transformation rules (using Stratego/XT).

2.1 Access Control and Management Models

This part of the thesis discusses the following two foundations of the security domain:

Access Control: Access control models are used to define a set of policies for a software system to enforce a set of rights to fulfil the security concerns.

Authorisation Management: During the system run-time a set of users need to assign the defined rights to the other users in the system. As the system becomes more complex this authorisation management may become a main concern.

In this section, I first introduce three main access control models and then in Section 2.1.2 I discuss different types of authorisation management systems and their related architecture.

2.1.1 Access Control Models

Data protection is essential for all multi-user systems and as such also for Web applications [66, 105, 97]. As Bishop [48] showed before, the data protection should be against violation of data confidentiality (i.e., unauthorized disclosure) and data integrity (i.e., improper modifications), and always provide access to authorized users. The process of protecting the access to the controlled data objects of the system is called *access control* [81, 31, 123]. There is a difference between an access control model and its mechanism. An access control *model* specifies a set of authorization rights, e.g., by means of rules; however, the access control *mechanism* is the actual implementation of the access control model in the system.

Discretionary Access Control

The first class of access control models discussed here is called *discretionary access control* (DAC) models [140]. DAC is based on authorization rights, meaning that both the identity of a requester and the data the user wants to access and operate on, are checked against the defined policy, which states what a requester is allowed or not allowed to do [140].

The *access control matrix* by Lampson et al. [89, 140] is generally considered to be the first access control model; it was introduced and illustrated in the context of operating systems. Access matrixes can be used to implement DAC models. As Table 2.1 shows, the access matrix consists of a two-dimensional table that simply represents the allowed operation relations between the users (i.e., subjects) and the available data (i.e., objects) of the system, such as files and directories [89, 140]. This access relation, which states which subject can operate on which object, is granted and maintained by an administrator [140]. As the access matrix represents the explicit access relation between each individual subject and object, it grows very large very quickly, but remains sparse as most subjects remain unrelated to most objects [140]. Even though solutions like authorization tables, access control lists (ACL) or capability lists are used to store access rights of the non-empty cells only [140], the data structures used in these approaches are still large, as they still store explicit access relations between each of the individual subjects and the objects they want to operate on. This also leads to a complex authorization management [140]. The main drawback of any discretionary access control mechanism such as access matrix is that they are unable to enforce any sort of control of the flow of information from the process that is operating on behalf of the user [140]. This allows "Trojan Horse" processes to leak information [108].

Table 2.1: An example of an access control matrix [140]

	File 1	File 2	File 3	Program 1
Ann	own, read, write	read, write		execute
Bob	read		read, write	
Carl		read		execute, read

Mandatory Access Control

The second class of access control models discussed here is called *mandatory access controls* (MAC) [130]. Among MAC models the most widely used access policy is the multi-level security policy model, which is based on subject and object classifications, as presented in the systems by Samarati et al.[140] and Osborn et al. [130]. In such an access control model, the subject, on the behalf of users, actively requests to access the object [140]. Note that the subject entity in MAC is different from the authorized subject in DAC; in MAC the subject is the process which is trying to access to the object on the behalf of user, not the user herself as in DAC [140]. Therefore MAC can control indirect accesses created by executed processes [140]. There are different types of mandatory access control models such as lattice based approaches [65, 141]. However, I focus on the two most wide-spread types of mandatory policies, confidentiality-based [38] and integrity-based [57] mandatory access control.

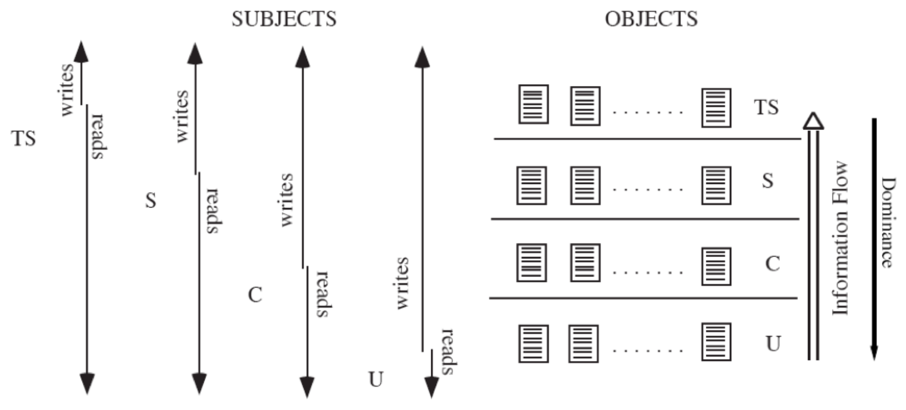


Figure 2.1: Confidentiality-Based mandatory access control [140]

In confidentiality-based mandatory access control data confidentiality is achieved by two principles, which were first proposed by Bell and LaPadula [38]. They are concisely summarized as ”no-read-up” and "no-write-down”. By applying these two principles any information flow from higher security levels to the lower ones will be prevented [140].

Figure 2.1 shows an example. Each subject and each object is defined with a confidentialty level, from *top secret* over *secret* and *classified* down to *unclassified*. As Figure 2.1 shows, ”no-read-up” means that subject of the system can have read access to an object only if the subject has higher (or *dominant*) access class than the object [140]. The "no-write-down" principle, as Figure 2.1 shows, means that the subject can have write access to an object if the object's access level dominates the subject's access level. Therefore, as Figure 2.1 shows, by using a dominance relation from high security levels to lower ones (top secret → secret, etc.) and the "no-read-up" and, "no-write-down" principles I achieve data confidentiality in the system, in the sense that the information can flow only from the lower security levels to the upper and more dominant ones. Note that in the mandatory data confidentiality model, when a subject belongs to a lower security level, it has less reading access rights, but it is not less privileged in general [140].

The integrity element could be added to MAC. Biba [47] proposed a dual policy (also called the Biba Model), which controls the flow of information and prevents indirect information modifications by the subject [140]. As Figure 2.2 shows, the classification of integrity levels (crucial, important, unknown) and dominance flow is similar to the confidentiality model (see Figure 2.1). However, the read and write accesses have the opposite direction, compared to the confidentiality model. For achieving integrity the read access must follow a "no-read-down" rule, meaning a subject can read the object if it has a lower integrity level than the object (see Figure 2.2). Also, the subject can write to the object if it has higher integrity level than the object (see Figure 2.2). Therefore, integrity is achieved if these two principles (”no-read-down” and ”no-write-up”) exist in a mandatory access control system.

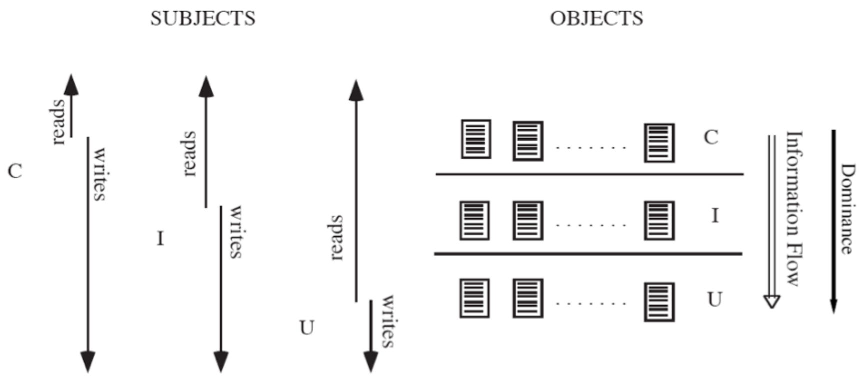


Figure 2.2: Integrity-Based Mandatory Access Control [140]

Finally, in a system that needs to enforce both confidentiality and integrity of the data, two sets of access classes need to exist [140] at the same time, the security access classes and the integrity access classes.

Role-Based Access Control

The third access control model class discussed here is role-based access control (RBAC) which belongs to the ”grouping privileges” access control class [140]. In this class of access control models, privileges are collected based on common aspects, and then authorizations are assigned to these collections [140]. The advantage of using grouping privilege based access control models (such as RBAC), compared to DAC or MAC models, is that this class the access control factors out similarities, and so handles changes in the system better, which leads to easier authorization management [140]. The first work in this class was by Baldwin [36], who introduced the notion of *name protection domain* (NPD), which simplified the security management, by stating the set of all privileges that need to exist to do each task in the system (i.e., the grouping privileges). Then, the groups were assigned to each user based on the required policy. Here I focus on the most recognisable access control model in this class, RBAC; I ignore other models in the grouping privileges class, because our approach is based on RBAC.

RBAC is the most widely researched [15, 85] and used [15] access control since its introduction by Kuhn and Gerraiolo in 1992 [71]. RBAC is used in many domains, such as operating systems [70], Web security and related technologies [132, 73, 40, 147, 169, 152], distributed systems [64, 168], databases [44, 45] and embedded systems [115]. There are also a number of tools and languages that support RBAC [74, 85, 61, 55].

Simply speaking, the RBAC concept uses the notion of "role" as the central authorisation mechanism [140, 71, 15]. Intuitively, a role is an abstract representation of a group of subjects that are allowed to perform the same operations on the same objects. This has also been formalized as a (formal) concept using concept lattices [142]. As Figure 2.3 shows, the objects (i.e., accessible shared data) in the system are assigned to the authorized roles and the subjects (i.e., users) need to identify themselves to acquire these roles to access and operate on the objects. For example, a school system can use teacher and student roles to represent the authorization rights of the teachers and students. The role teacher could be associated with two permitted operations (reading and writing on the mark object) while the subjects associated with the student role can just see their own mark. In this example, if a user, has an authorization right to be a teacher, can mark the student, and if a user has the student authorization right, can see the mark. This highlights the main advantage of using RBAC over DAC and MAC models: because RBAC uses the concept of ”role”, it fits more naturally to the organizational structure [140], which leads to an easier design and development of the access control mechanism. The third advantage is based on the fact that in an organization the roles are changing less frequently than the users, so that using the role as a central authorization element decreases the maintenance cost and administrative tasks [58], as shown in a recent survey [129].

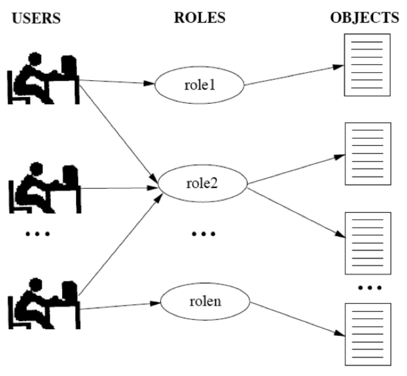


Figure 2.3: Role-Based Access Control [72]

Users (i.e., subjects) and permissions (i.e., the relation between the operations and objects) can be assigned to one or many roles [72, 71].

There have been many early proposals for defining RBAC elements [31, 98, 112, 131, 144], however, RBAC was standardized by the *National Institute of Standards and Technology* (NIST) in 2001 [72], and an update to the standard is soon to be released [15]. The rest of this section related to RBAC is based on the NIST RBAC standard [72], which divides RBAC into four levels that each build on the previous. I will describe each of these levels in one of the following subsections. In each subsection, an example and overview of the implementation requirements for each level are also given.

Flat RBAC

Flat RBAC, as the base level of RBAC, defines the minimal requirements for an RBAC approach as an access control part of the authorization mechanisms of a system [72]. Figure 2.4 shows its elements and Table 2.2 shows its functional capabilities. The central element is the role, and users and permissions (i.e., relations between the data and operations) have one-to-many or many-to-many relations to roles [72]. In addition to this, the user has to be able to review (but not necessary change) the assigned roles and to use a number of permissions from multiple roles simultaneously [72].

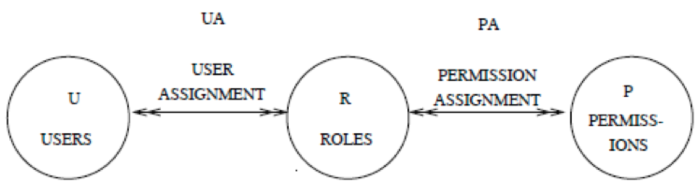


Figure 2.4: Flat Role-Based Access Control Model [72]

A very simple example of the flat RBAC would be a system that has teacher and student as roles, and reading or writing of student marks as permissions, as sketched above.

In the following, I assume that, the application's business logic, authentication, and data model exist. In order to introduce a flat RBAC mechanism there are three abstraction levels that I need to examine:

Developer: The developer needs to define and implement five components. First, the notion of role needs to be defined in the data model. Second, the part of the data model that represents the users of the system must be extended (e.g., by a one-to-many inheritance relation), so that the roles can be assigned to and stored for each user. Third, the users' sessions must be extended, so the users can activate a sub-set of their assigned roles in their sessions.Fourth, access restriction to each object and its related operation need to be defined in terms of required authorization rights. Fifth, the authorization management system should be implemented, so the admininistrator of the system can access it for the access control related activities (e.g., role assignment).

Administration: The administrator should be able to add, delete, and edit the roles. The administrator should also be able to assign any permission (i.e., object and its related operation) to a role and then assign the role to any subject (i.e., user).

User: The user (i.e., subject) should be able to identify herself to the system (through an existing authentication mechanism), select and activate a subset of the roles assigned by the administrator, and acquire the related permissions.

Table 2.2: Different Levels of Role Based Access Control [140]

Level	**RBAC Functional Capabilities**
Flat RBAC	1. Must allow users to acquire permissions through roles. 2. Must support many-to-many user-role assignment. 3. Must support many-to-many permission-role assignment. 4. Must support user-role assignment review. 5. Use permissions of multiple roles simultaneously.
Hierarchical RBAC	Flat RBAC + 1. Must support role hierarchy (partial order). 2. **Level 2a** Requires support for arbitrary hierarchies. 3. **Level 2b** Requires support for limited hierarchies.
Constrained RBAC	Hierarchical RBAC + 1. Must enforce separation of duties (SOD). 2. **Level 3a** Requires support for arbitrary hierarchies. 3. **Level 3b** Requires support for limited hierarchies.
Symmetric RBAC	Constrained RBAC + 1. Must support permission-role review with performance effectively comparable to user-role review. 2. **Level 4a** Requires support for arbitrary hierarchies. 3. **Level 4b** Denotes support for limited hierarchies.

Hierarchical RBAC

The second level of RBAC, hierarchical RBAC, is built on top of flat RBAC [72]. Figure 2.5 shows its elements and Table 2.2 shows its functional capabilities. Hierarchical RBAC adds the notion of *role hierarchy* to the role notion of the a system. The role hierarchy component could be of type arbitrary or limited [72]. By using the limited hierarchy means, the model can assign a set of authorization rights to a part of the roles' hierarchy [140].

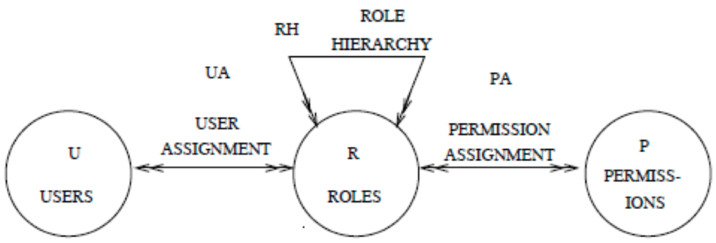


Figure 2.5: Hierarchical Role Based Access Control Model [72]

I expand the example given in the previous subsection to hierarchical RBAC, by adding two elements, a supervisor role and an inheritance relation between the supervisor role and the teacher role. In this case, the supervisor inherits the rights given to the teacher role but can have, in addition, further rights, e.g., the authorization to access for reading each student's medical history who is under supervision.

I can now formulate the requirements of adding a hierarchical RBAC mechanism to the system, in the same terms as before.

Developer: The developer should be able to change the role data model in such a way that the instances of the role in the data model could inherit from an other role.

Administration: The administrator should be able to add, delete, and edit the inheritance relation between the roles.

User: The user should be able to activate a role and the permissions of the inherited roles will be granted to the user automatically.

Constrained RBAC

The third level of RBAC, constrained RBAC, is built on the second RBAC level, hierarchical RBAC [72]. Figure 2.6 shows its elements and Table 2.2 shows its functional capabilities. Constrained RBAC introduces an access control element called *separation of duties* (SOD) [72] on role hierarchies and user-role assignments. SOD is motivated by one of the main design principles of access control, namely that *improper data modification* *is not desirable* and must thus be prevented. If a user of an access-controlled system is able to modify the data improperly, then the administrator has assigned a number of conflicting components (e.g., roles) to the user. The term *conflict components* in an access control model means the components that should not be used together. As an example consider the teacher and student roles in a school. Only the teacher should be able to mark the students. Therefore it is not desirable to assign these two conflicting roles to the same user, so that the user could mark himself. Therefore there is a need to separate these conflicting components by separating their duties. The separation of the duties of conflicting elements can be defined using the SOD component of RBAC [140].

Similary, too many authorization rights might also result in improper data modification. Therefore, overall, SOD can be used to divide the rights among the subjects based on the constraints and/or conflicts that might exist among the roles [72]. The SOD concept itself originated in the data-integrity domain, to stop improper behavior of the subjects [140]; however, this is outside the scope of this research, for more details see [53, 140].

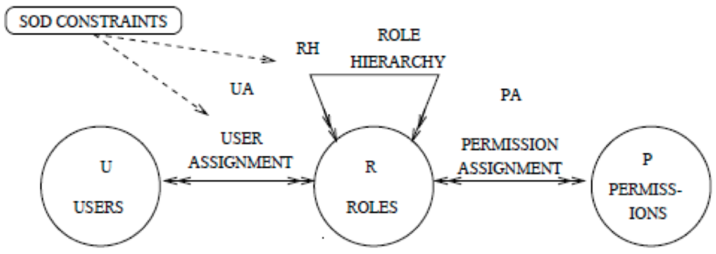


Figure 2.6: Constrained Role Based Access Control Model with static SOD [72]

In constrained RBAC there is also a distinction between static and dynamic SOD:

Static SOD: As shown in Figure 2.6, static SOD affects the allowed role hierarchies and user assignments. Static SOD enforces on the user-role assignments that if two roles always create a security conflict, then these roles cannot be assigned to the same subject at any given time. For example, consider the teacher and student roles. Normally, a student submits her work and a teacher marks it. In this case if a subject (i.e., user) has both authorization rights, can submit and edit own mark, which is prohibited. This can be modelled by a static SOD constraints.

Dynamic SOD: As Figure 2.7 shows, dynamic SOD affects each user session. After user authentication, the system creates a session for the user. Dynamic SOD enforces the separation of authorization rights within each user session. For example, consider the policy that states *a subject cannot be the examiner of a student that, the subject is supervising*. In this case, there must be a dynamic SOD between the roles supervisor and examiner, but only if the student is being supervised needs to be examined. In all other circumstances, there is no conflict between the supervisor and examiner roles. These types of constraints are determined during the user's session, as they cannot be determined with static checks.

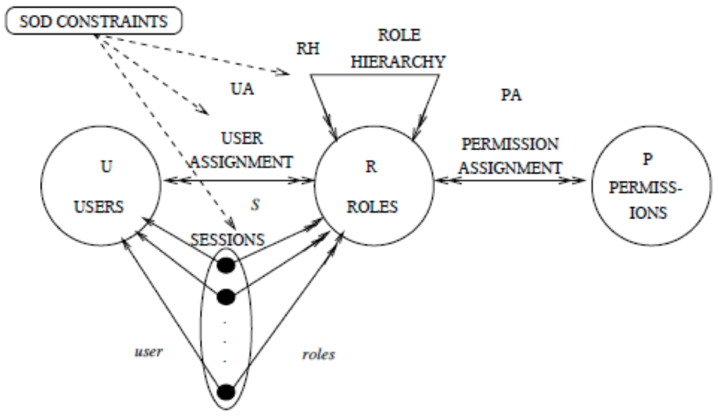


Figure 2.7: Constraint Role Based Access Control Model with dynamic SOD [72]

The difference this *notion of separation of duties* can make in terms of developing and using such an RBAC system, compared to the previous levels, is to introduce static and dynamic checks on the conflict roles. These checks make sure that the conflicting roles do not get assigned to the users. In case of static SOD, the administrator cannot assign conflicted roles to the same subject. In the case of dynamic SOD the administrator can assign both roles to a user but the user cannot activate both conflicted roles in one session.

Symmetric RBAC

The fourth level of RBAC, symmetric RBAC, is built on the third RBAC level, hierarchical RBAC [72]. Figure 2.8 shows its elements and Table 2.2 shows its functional capabilities. In this level the notion of separation of duty constraints extended to constrain permission assignments as well. Therefore, in this level, conflicting permissions cannot be assigned to the same role [72].

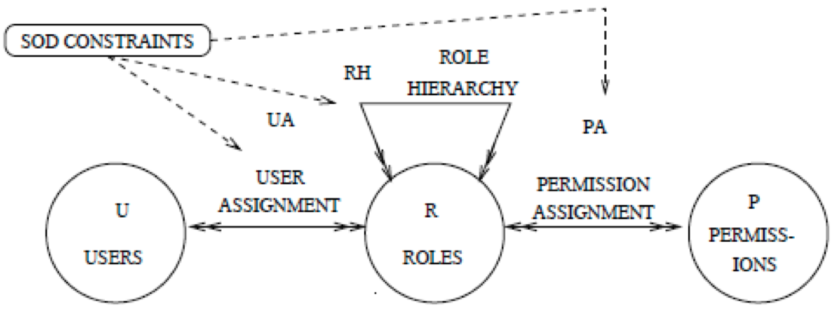


Figure 2.8: Symmetric Role Based Access Control Model [72]

As in constrained RBAC, the SOD at this level is divided into two types, static and dynamic. The element that is added is the permission assignment. The following explains these two aspects of the symmetric RBAC:

Static SOD: As in the previous level, constraints that always create conflicts between permission-role assignments need to be separated such that under no circumstances conflicted permissions would be assigned to the same role. For example, creating and deleting of a bank account should not be possible for any user at the same time. Therefore the creating and deleting permissions cannot be assigned to the same role. Therefore, at this level, the negative permissions are specified, separated and assigned to different roles.

Dynamic SOD: Here the user session is taken into account again, such that if two permissions are conflicting only when they are used together, then these two permissions could be assigned to a role but they cannot be acquired together during any user's session. For an example consider a policy that states *a teacher cannot be the second marker of an exam if the teacher marked the papers for the first time*. Therefore, clearly in this case during the user session, if the teacher attempts to mark the papers then cannot use the same role to give the mark for the second time. The interesting point here is that the conflict is not on two different permissions but on the same permission that relates to the role teacher. Therefore, if a user uses the teacher role for a course to mark the papers, cannot use this role again in the same session to mark the papers for the second time.

In terms of development, the developer can define the static SOD during the development of the system. However, because of the dynamic nature of the dynamic SOD in Symmetric RBAC the developer should provide a feature for the administrator to define and edit the conflicts between the permissions during run-time.

2.1.2 Authorization Management Mechanisms and Architectures

So far I have introduced the three main access control models. After the deployment of such models the administrators need to assign the authorization relations to the users and the objects of the system. The administrative tasks become complex as the system becomes more distributed and more widely shared. Web applications (and their data) are naturally distributed and shared among the Internet users. Therefore, it is necessary to check the authorization management domain.

Authorization management refers to different parts of the system that help the administrator to do the authorization assignments to the users. In this subsection we first find out the administrative authorization elements for the defined access control models.

Then, I inspect a number of research attempts to check the architectural elements of the different access control models.

Administrative Authorization Mechanisms

Based on the description of the different access control models, the following list points out the administrative authorization elements and tasks for each access control model:

DAC: As Section 2.1.1 showed, at the core of the DAC model is a list in which each element is a triple of (u, op, obj) where u represents the user, op refers to a list of allowed operations and obj is the controlled object. Therefore the administrative job is to assign these relations for each user that wants to operate on an object of the system. The mechanism that enables the administrators to do so could be implemented via three input lists (i.e., users, operations, objects) or as an editable table similar to the access control matrix (shown in Figure 2.1).

MAC: During system run-time the administrator of a MAC mechanism needs to assign a set of confidentiality or integrity labels to the users and objects of the system. Then the access control predicates will guard the objects based on these labels and their types (i.e., confidentiality or integrity). These types are defined by the developer, and based on them the access control mechanism enforces the information flow as shown in Figure 2.1 and 2.2. The administrative mechanism can thus consist of three lists of users, labels, and objects, in which the administrators will assign the labels to the users and the objects.

RBAC: The NIST RBAC has four different levels, each with access control elements. I already discussed the administrative jobs in the previous section.

Administrative Authorization Architectures

Authorization assignment is a difficult task in collaborative systems with distributed users [64]. Its complexity might lead the administrators to unwillingly create security breaches within the system by assigning incorrect or inconsistent authorization rights. This part of the thesis focuses on a set of approaches that tackle this problem by providing an authorization management system on the top of the access control mechanisms. I categorize these architectures based on their authorizing access control mechanisms.

Discretionary Management Model

This section focuses on two authorization management architectures, Akenti [157] and PRIMA [118] that are based on DAC. Both were introduced and used [125, 164, 39] over the last decade.

Akenti: Akenti [157] is a certificate-based authorization service that makes access decisions within a system with distributed resources based on a set of user-defined certificates. I categorize authorization management in Akenti as a DAC and group based approach, because in this architecture a resource has a set of stakeholders; and these users define/edit the certificates that lead to access for an authorized user of the system. Therefore overall the Akenti policy engine collects all the certificates for a user and resource and then verifies if the user can access the resource.

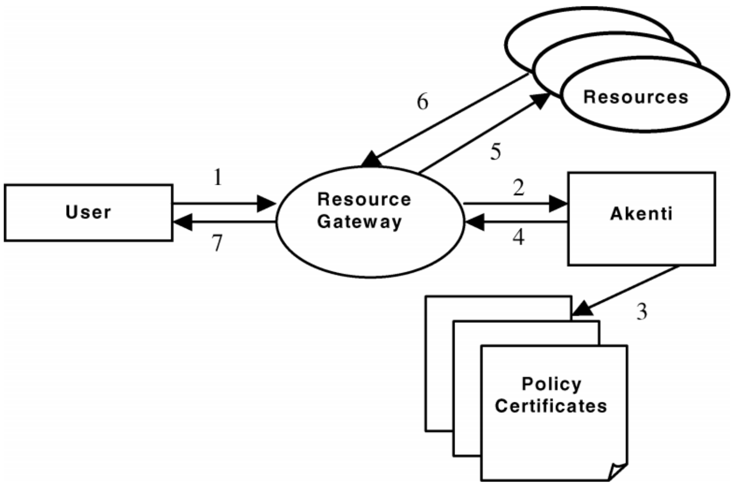


Figure 2.9: Akenti Authorization Model [157]

Authorization Model. As Figure 2.9 shows, the user uses the resource gateway to access to the resources. The stakeholders define the access constraints as a set of signed certificates. Each certificate presents an attribute for each specific right for a resource. Then the resource gateway sends the access request to the Akenti server and the server collects all certificates required for the access, verifies the signatures, and after evaluation sends back the allowed rights for the requested resource to the resource gateway. Furthermore the gateway verifies the user against the allowed rights and, if successful, grants the access to the resource.

Authorization Definition. In Akenti the policies are defined in an XML format. There are three types of signed certificates in Akenti: policy certificates, use-condition certificates, and Akenti attribute certificates. The policy certificate is used for defining the authority sources for a resource. The use-condition certificates are used to define what set of attributes is required for a client to access a resource. Akenti attribute certificate are assigning the required attributes to clients to access a resource. The Akenti attribute certificate assigns attribute to users. Even though there is a DAC based authorization management, for the defined authorization rights the stakeholder can create a Boolean expression (i.e., use-condition certificate) that consists of roles, groups and system context.

PRIMA: I categorize PRIMA [118, 117] as a DAC-based authotization management approach because, similar to Akenti, PRIMA, an authorization management framework, gives the power to the objects' stakeholders to define the authorization privileges. Figure 2.10 shows, the PRIMA system overview that is divided into privilege management and authorization and enforcement layers. As shown, first the resource administrators and stakeholders regulate a policy. Then, in the second step the regulated polices are abstracted from low-level, encoded using XACML [119] and sent back to the administrators [118] or system users. Then, in the third step the administrators or system users can delegate the access privileges to the system users and entities, such as agents. After this, each user of the system can activate a subset of the delegated privileges and request an access (shown as step 4 and 5 in Figure 2.10).

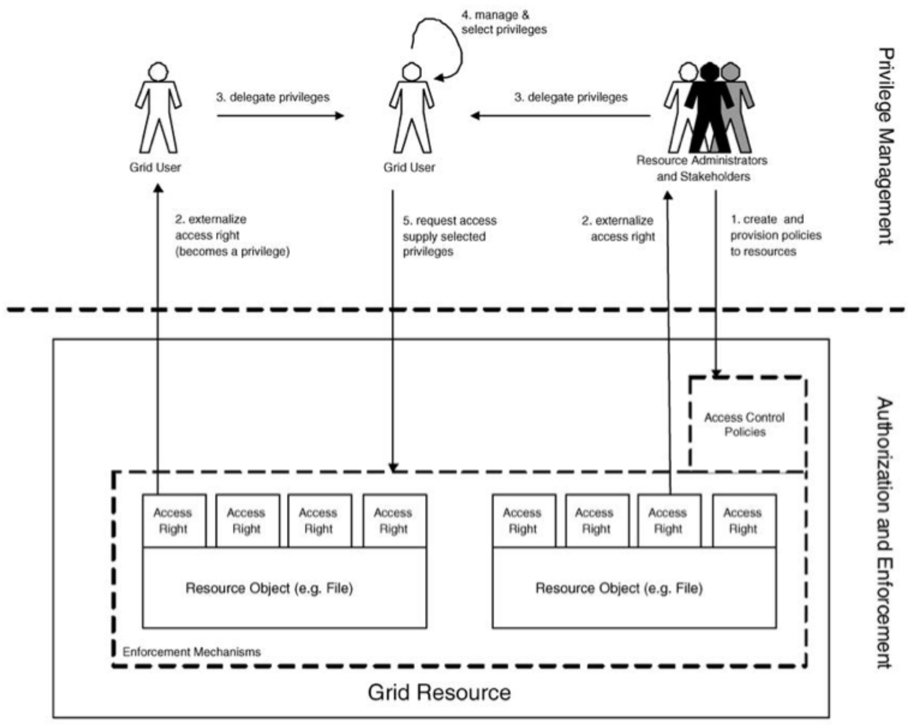

4. manage & select privileges
3. delegate privileges
3. delegate privileges
Grid User
Grid User
Resource Administrators and Stakeholders
Privilege Management
2. externalize access right (becomes a privilege)
5. request access supply selected privileges
2. externalize access right
1. create and provision policies to resources
Access Control Policies
Access Right
Resource Object (e.g. File)
Resource Object (e.g. File)
Enforcement Mechanisms
Grid Resource
Authorization and Enforcement

Figure 2.10: PRIMA System Overview [118]

Policy Enforcement. In PRIMA there are two types of policies [118]: POSIX.1E file system access control lists and Grid access control lists (see Figure 2.11). The file system access control list [153] which is available in Linux operating system allows permissions to each individual PRIMA user. For example in Figure 2.11 the POSIX ACL specifies permissions for the users *sshah* and *kafura* for the file sim.h. On the other hand the Grid ACLs are written in XML format and as Figure 2.11 shows can specifies the relation between the user and the allowed operations. These policies can be defined for a project or a group or a set of files in the system. Both ACL lists are part of the SlashGrid [17] project, and Slash Grid uses the defined ACLs to enforce policies. The difference between these two ACLs is that the Grid ACL is the representation of the file ACL in XML-based format with additional features to support groups and projects related to the PRIMA users.

```
# file: /projects/sim/sim.sh
# owner: mlorch
# group: users
user: : rwx
user: sshah: r-x
user: kafura: rwx
group: : r—
mask: : rwx
others: : ---
```

```
<gacl version="0.0.1">
<entry>
  <person>
   <dn>/CN=Markus Lorch/O=vt/C=US</dn>
  </person>
  <allow>
    <write/><admin/><read/><list/>
  </allow>
</entry>
<entry>
  <person>
   <dn>/CN=Sumit Shah/O=vt/C=US</dn>
  </person>
  <allow>
    <read/><list/>
  </allow>
</entry>
</gacl>
```

Figure 2.11: POSIX ACL and Grid Access Control List [118]

Role-Based Management Model

In this part of the report I examine a role-based authorization management approach in a collaborative environment.

ShareEnabler. This system [102] provides a role-based authorization management for resource sharing for collaborative environments by using an XML-based framework. In ShareEnabler, each participant within these communities is represented as an agent. These agents are in communication with each other through the network. ShareEnabler uses as an independent entity the role notion that is widely used across different communities [102]. ShareEnabler uses the following policy specification elements within role-based originator authorization policy sets (ROA) to define a policy framework for a collaborative environment.

- *Role Policy Set:* This set presents the sharing collaboration domain of each agent. Each set is associated with capability policy set to achieve the role-capability assignment. In this work, role is specified based on subject attributes (e.g., agent_id, etc.)
- *Capability Policy Set:* In this set the actual capabilities for each assigned role is defined. This set includes two elements of *policy* and *rule* to describe actions. It also may contain references to other capability policy sets.
- *Delegation of Delegation Authority Policy Set:* Each originator of a resource issues what other trusted agents are committed to *role assignment.* These role assignment policies are references through another set named, `policySetIdReference`.

- *'Role Assignment Policy Set:* This policy set specifies the assignment relations between roles and participants (i.e., agents). Resource and action are used in the policy set to declare a relation between a role and a resource; also the issuer attribute is used to differentiate between the role assignment issuer and role issuer.

- *Root Meta Policy Set:* This set contains associates of resource originator defined policies for each resource. So this set helps policy enforcement system to locate agent's defined policy set for each resource. For detailed discussion on these policy schemas see [102].

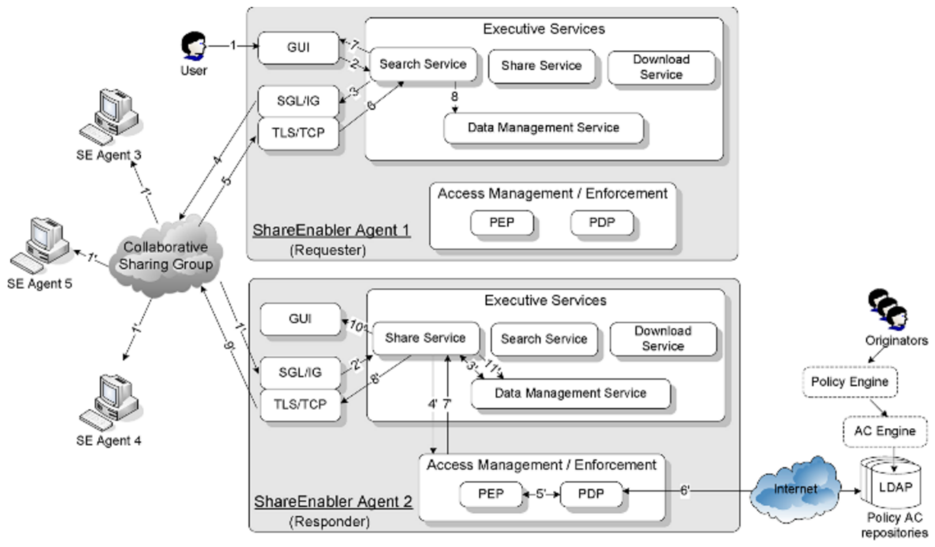


Figure 2.12: ShareEnabler Architecture [103]

ShareEnabler adopts the P2P information structure using SciShare [42]. Each participant is an agent that on behalf of a user uses a query to discover a resource (see Figure 2.12). Here I discuss the architectural components and their interactions through an example that were presented in [102], and illustrated in Figure 2.12. As Figure 2.12 shows, a user interacts with an agent through its GUI. GUI then invokes the search service to create a query and then sends it to its sharing group via SFL/IG port. Then, the second agent gets the 'receive the query' notice; and uses the access management service to get the resources based on the query. The *policy enforcement point* (i.e., PEP) generates an XACML and sends it to the *Policy Decision Point* (i.e., PDP) for the access decision. PDP then evaluate the requested query based on the defined policy for the resource by the originator of the resource. Assume that the query is unauthorised for the given user and queried resource, in this case, the PEP enforces the decision by *removing* the query and sends its response to the *meta data sharing service* (MDSS). At the end the MDSS sends back the *unauthorised* response message to the first agent via TLS/TCP port. Finally the first agent will send the *unauthorised access* to the user.

2.2 Access Control and Authorization Management Development Mechanisms

In this section I investigate two important aspects of developing access control and authorisation management models and mechanisms. First I check how the developer can use the available approaches to develop such systems. Then, I apply the testing techniques on these elements. At the end, this part discusses the shortcomings of the mentioned approaches and set the direction of this research.

2.2.1 Policy Languages

A policy language provides an environment for the developer to enforce a set of access control policies over the defined access controlled objects of the system [68]. There are several research attempts in defining policy languages such as Tower [92, 93], RCL 2000 [30], EPAL [32, 46, 34], ASL [101, 100], and PlexC [76]. However, this part of the report discusses the two most influential policy languages, Ponder [61, 14] and XACML [119].

Ponder

Ponder [61, 14] is an object-oriented, typed policy language that gives the ability to the developer to define a set of declarative rule-based policies at different abstraction levels (i.e., firewall, database, and Java) for the security and management of distributed systems. This part of the report gives an overview on the policy types of the Ponder language for both access and authorisation management models, and additional features of this language.

Ponder Policies. Overall there are two sets of policies for the access and the authorization management. For the access control the developer can define *authorization*, *information filtering*, *delegation*, and *refrain* policies. Moreover for the authorization management models the developer can define *obligation* policies. The following summarises these policy types, defined in Ponder specification [61].

Authorisation: These policies can be used for enforcing access control polices on a set of objects. Figure 2.13 shows the syntax of authorization policy that consists of the following elements:

- *Policy instantiation and reuse:* The keyword `inst` is used when the developer declares a new instance of a policy. Each policy is a type and can be treated as type such that a subjects of its elements can be used in other policy definitions. Ponder also allows developers to reuse policy instances via policy type definition parameterized (see Figure 2.14). The developer can then create

```
inst ( auth+ | auth– ) policyName    “{”
   subject  [<type>]   domain-Scope-Expression ;
   target   [<type>]   domain-Scope-Expression ;
   action              action-list ;
   [ when              constraint-Expression ; ]   ”}”
```

Figure 2.13: Authorization Policy Syntax [61]

multiple instances from the policy type and tailor each of them for a particular environment (see Figure 2.14). Figure 2.14 shows how the policy type definition, `PolicyOpsT` is used in `SwitchPolicyOps` and `routersPolicyOps` policy instances such that `SwitchPolicyOps` is tailored for `NetworkAdmins` to execute their actions on `Nregion/switches` domain; and `routersPolicyOps` is tailored for `QoSAdmins` subjects to execute their actions on `Nregion/routers` domain. As it can be seen, the difference is based on the *subject* and the *target* domain.

```
type auth+ PolicyOpsT (subject s, target <PolicyT> t) {
     action load(), remove(), enable(), disable() ; }

inst auth+ switchPolicyOps=PolicyOpsT(/NetworkAdmins,/Nregion/switches);
inst auth+ routersPolicyOps=PolicyOpsT(/QoSAdmins, /Nregion/routers);
```

Figure 2.14: Instance declarations from policy type definition [61]

- *Positive and negative authorisations:* The `auth+/auth-` keywords can be used for specifying when a subject can/cannot do a set of operations (i.e., actions) on the access control objects (i.e., target). Then the developer defines a *policy name* as a *unique identifier* for a policy.
- *Subject:* An instance of the system (e.g., agent, session, etc.) that acts on the controlled objects on the behalf of the users of the system. It can have an optional type, based on the other defined policies, such as `networkAdmin`. The domain-scope expression is the domain of its usage (e.g., router).
- *Action:* Then the developer can define a set of operations that are used in that domain, such as *remove()* or *load()* in the network domain.
- *When:* Moreover the optional *time constraints* can be used for declaring a set of constraints for the policy rules, such as `time.between("0900","1700")` that states the policy is effective during office hours.

  Note that there is no order between the policy elements.

Information Filtering: As mentioned the positive authorisation policies let the authorised users to do a set of actions on a set of objects. Ponder allows information filtering on the action of policies. For this, the developer can use the following filtering elements within the action of a policy:

- *Condition:* An *optional* if statement can be used to put a set of *attribute-based* restrictions (e.g., time, date, location, etc.). For example the second filter in Figure 2.15 defines a filter that enforces a time limit (i.e., after 7 p.m.).

```
actionName { filter }
filter = [ if condition ]  "{" { ( in parameterName = expression ;  |
                                   out parameterName = expression ; |
                                   result = expression ; ) } "}"
```

Figure 2.15: Filters on positive authorisation actions [61]

- *Input/Output:* The filters transform the input and output of information. For this the developer defines input/output and their related *parameter name* and *evaluation expression*. For example in Figure 2.16 the first filter defines one input for the bandwidth (2 Mb/s) and another input for priority (3 Mb/s); and similarly for the second filter there are two input for the bandwidth and priority.

```
inst auth+ filter1  {
   subject  /Agroup + /Bgroup ;
   target   USAStaff – NYgroup ;
   action   VideoConf(BW, Priority)
            { in BW=2 ; in Priority=3 ; }     // default filter
            if (time.after("1900")) {in BW=3; in Priority = 1; }
}
```

Figure 2.16: Information filter policy [61]

Delegation: Authorizations can be transferred among users. For this the developers can define delegation policies with respect to the defined authorization policies, that are effective during the system run-time. Moreover, because of the sensitivity of this action Ponder allow users to define a set of constraints for each delegation policy rule. For example the defined delegation policy, shown in Figure 2.17, declares a delegation policy `delegSwitchOps` with an association with `switchPolicyOps` such that the subject of the `switchPolicyOps` policy can transfer the defined actions (i,e., `enable()`, `disable()`) from the target domain `/Nregion/switches/typeA` to the `/DomainAdmin` within 24 hours of creation. It is important to note that the actual delegation mechanism is not derived from these policies.

```
inst deleg+ (switchPolicyOps) delegSwitchOps  {
   grantee    /DomainAdmin ;
   target     /Nregion/switches/typeA ;
   action     enable(), disable();
   valid      time.duration(24) ;
}
```

Figure 2.17: A Delegation Policy in Ponder [61]

Refrain Policies: To enforce a *negative* authorisation on the *subjects* performing operations, the developer can define a set of *refrain* policies. The syntax of this policy type is the same as negative authorisation policies. For example, the shown refrain policy in Figure 2.18, states that `test engineers` *must not* use the disclose operation on the `TestResults` and send any information to `analysts` and `develoers` when testing is *not* finished (i.e., in progress).

```
inst refrain testingRes {
   subject s=/test-engineers ;
   target /analysts + /developers ;
   action  discloseTestResults() ;
   when    s.testing_sequence = "in-progress" ;
}
```

Figure 2.18: A Refrain Policy in Ponder [61]

Obligation: For restricting the run-time authorisation management functionalities, the developer can define obligation policies.

```
inst oblig policyName    "{"
   on                    event-specification ;
   subject    [<type>] domain-Scope-Expression ;
   [ target   [<type>] domain-Scope-Expression ; ]
   do                    obligation-action-list ;
   [ catch               exception-specification ; ]
   [ when                constraint-Expression ; ]    "}"
```

Figure 2.19: Obligation policy Syntax [61]

An obligation policy consists of the following elements:

- *Policy instantiation:* Similar to the authorisation policy, the obligation policy instantiating is defined by the keyword `inst`, followed by the policy name.
- *Trigger Event:* This is a definition of an event that invokes the policy. For this the developer should define the event after the keyword `on`. For example in Figure 2.20, the policy is triggered after a user fails to login to the system three consecutive times.

```
inst oblig loginFailure {
   on               3*loginfail(userid) ;
   subject          s = /NRegion/SecAdmin ;
   target <userT>   t = /NRegion/users ^ {userid} ;
   do               t.disable() -> s.log(userid) ;
}
```

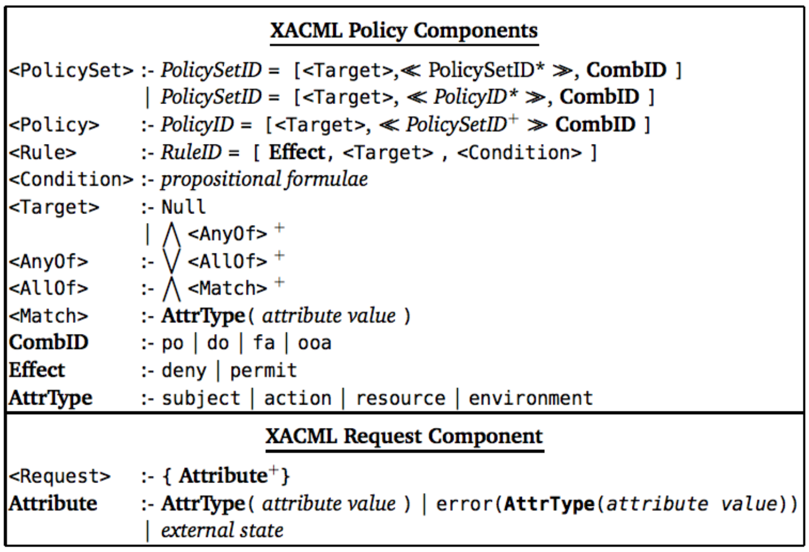
Figure 2.20: An obligation policy [61]

- *subject and target:* It has the same meaning and use, as in authorisation policies.
- *do:* The developer can then define a set of action lists, separated by `->` as the consequence of the triggered event. For example, the obligation policy in Figure 2.20 shows that the access of the user who failed to login is disabled and then logs the `userid` to the `security admin` object.

XACML

One of the most well known policy language is XACML, that is a XML-based language, which got standardised by OASIS[1].

Recently a few number of researchers have been carried out to turn the XACML policies, which have XML-based format, into a formalised representation [121] for further constructive analysis. Since 2003, XACML has three standard versions, in which the last version (version 3.0) was introduced in 2010 [5]. Moreover, a number of tools, such as VDM++ [74], were used to help the developers to define XACML-based policies. This part of the report gives an overview on the syntax and architecture of this language.



XACML Policy Components

```
<PolicySet> :- PolicySetID =  [<Target>,≪ PolicySetID* ≫, CombID ]
            | PolicySetID =  [<Target>, ≪ PolicyID* ≫, CombID ]
<Policy>    :- PolicyID = [<Target>, ≪ PolicySetID+ ≫ CombID ]
<Rule>      :- RuleID = [ Effect, <Target> , <Condition> ]
<Condition> :- propositional formulae
<Target>    :- Null
            | ⋀ <AnyOf> +
<AnyOf>     :- ⋁ <AllOf> +
<AllOf>     :- ⋀ <Match> +
<Match>     :- AttrType( attribute value )
CombID      :- po | do | fa | ooa
Effect      :- deny | permit
AttrType    :- subject | action | resource | environment
```

XACML Request Component

```
<Request>   :- { Attribute+}
Attribute   :- AttrType( attribute value ) | error(AttrType(attribute value))
            | external state
```

Figure 2.21: XACML abstract syntax [134]

XACML is enforcing access on the resources on the Web and shows how the Web applications can access these resources through the provided Web services. As Figure 2.22 shows, the XACML's work flow starts when a request is sent by an application from a resource. The developer defines two things with an XACML, first a set of capabilities in which defines how a Web service should act based on the Web applications requests; and second a set of policy expressions, that define the access rights to enforce authorizations on a set of resources [5].

XACML is based on the following elements:

Policy Expressions: XACML is a policy language, so before the developer defines the behaviour of the policy enforcement and decision points, can define a set of policy expressions. In XACML, a policy is based on the following elements:

[1]OASIS (Organization for the Advancement of Structured Information Standard). Check https://www.oasis-open.org/committees/xacml

- *Target:* This expresses the required capabilities for the requester to access a resource. Each target consists of *subjects*, *action*, *reources*, *environments*, and *conditions*.
- *Rules:* For each target a set of rules is defined. Each rule contains the predicates for the access control decision that either permits or denies a request. The permit and deny are parts of the *effect* concept that is available in each access control decision.
- *Algorithm:* Based on the fact that, policies consist of a set of rules and these rules could permit or deny an access, a set of rules could create a set of conflicts. To define the overall decision when there is a conflict between policy expressions, the developer can define an algorithm, so in case of a conflict the final decision is made based on this algorithm.

Policy Set: XACML allows developers to group a set of policies into a *policySet* and an algorithm for making the final decisions in case of conflicts within the rules of the contained policies. Each policy set can also contain a target that matches with the requester. Note that this matching happens *before* maching the requester to the included policy targets.

Policy Enforcement Point: After a request by an application, the resource access request is created by policy enforcement point (PEP). PEP can be presented by different forms [5] such as a Web server or an agent. Therefore, as E. Rissanen highlighted in [5], all the request responses are converted in a canonical form.

Policy Decision Point: This element makes the following authorisation decisions by matching the values of the request to the values from retrieved policies.

- *Permit/Deny:* The access request is permitted/denied .
- *Not Applicable:* No defined policy can be find to be applied (i.e., matched) to the request.
- *Intermediate:*In the case of PDP cannot evaluate the request based on a missing attribute.

Here I first check its architecture and then the security elements that can be used and defined by XACML.

Architecture Overview. As Figure 2.22 shows, after a subject requested to access a resource, the decision is being made based on the policy enforcement point (PEP), policy decision point (PDP) and other related elements in the system. PEP formulates the subject request and then sends that to the PDP. The formula is based on (S, R, OP, and P) where S is the subject, R is the resource, OP is the operation set that S wants to perform, and P is the purpose of the access. This authorisation decision (i.e., permit,

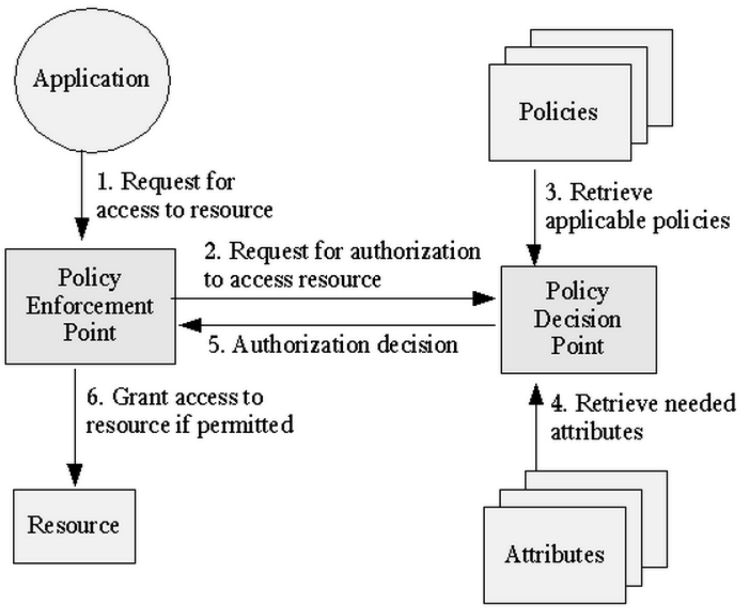

Application
1. Request for access to resource
Policies
3. Retrieve applicable policies
Policy Enforcement Point
2. Request for authorization to access resource
Policy Decision Point
5. Authorization decision
6. Grant access to resource if permitted
4. Retrieve needed attributes
Resource
Attributes

Figure 2.22: XACML abstract architecture [32]

deny, or not applicable) will be send back to the PEP based on the defined applicable policies (written in XACML).

Policy Language Model. The policy language model consists of `rule`, `policy` and `policy set` main elements.

Rule. The simplest policy element is rule [32] and to make them exchangeable within the actors of the system, I need to encapsulate them within a policy. The rules get evaluated based on their contents:

- **Target:** This element makes a rule a logical expression that must be evaluated based on the requests (as illustrated in Listing 2.1). If the rule does not have the `target` node, the rule will be considered as a policy.

  In Listing 2.1, `subjects` represents the users, `resources` represents the data, `actions` represents the operations, and `environments` represents the required policy of the system. This rule will be applied to a request if the following predicate will be `true`: `(predicate1 OR predicate2) OR (predicate3) OR (predicate4) OR (predicate5)`. Note that for each group the structure of the nodes will affect the predicates. For example, if `predicate1` and `predicate2` were under one `subject` tag, their related predicates will be `predicate1 AND predicate2`.

  The other attributes that can be used within the rule descriptions are `effect`, `condition`, and `obligations`. The `effect` is used by the rule's write to indicate that the rule should `Permit` or `prohibit` when the rule is evaluated to `true`. The optional `condition` attribute is used for attaching additional

```
<target>
  <subjects>
    <subject>
      <predicate1>
    </subject>
    <subject>
      <predicate2>
    </subject>
  </subjects>
  <resources>
    <resource>
      <predicate3>
    </resource>
  </resources>
  <actions>
    <action>
      <predicate4>
    </action>
  </actions>
  <environments>
    <environment>
            <predicate5>
    </environment>
  </environments>
</target>
```

Listing 2.1: A rule example in XACML

predicates to the rule; and `obligation` attribute is used to state what must be carried out before and after the access grant, such as destroying the provided information after a month.

Policy. The policy administration point (PAP) combines rules within a policy. Each policy has the following elements:

- **Target:** For a policy, the `target` specifies on what set of conditions a set of requests apply.
- **Rule-combining algorithm:** A policy consists of a number of rules and rule-combined algorithms. These rules in the policy state how the evaluation of these contained rules will be combined.
- **Obligation expressions:** When the PDP evaluates a policy, if the policy has obligation expressions, the PDP will transform them into obligations and send those obligations to PEP.

Policy set and its elements. Policies can be gathered into a policy set. Similar to a policy, a policy set has also `target`, `Rule-combining algorithm` and `obligation expressions`.

2.2.2 Database Approaches

Classically access control and fine-granularity of an access control, and their challenges involving extending relational databases to support classical access control models [45], were discussed in terms of the database and extending their related query-based languages [29]. On one hand, this research is based on Web application domain. In this domain, the users of the system *do not* use query-base languages to interact with the database *directly*, and as A. Roichman and E. Gudes pointed out in [137], to the database, every user is a server-side super user, that tries to interact with the database. On the other hand, it is important to have an overview look on the domain of fine-granularity of access control and the database to understand how fine-granularity of access control where enforced in the context of database domain. This part of the report, gives an overall view on the following two main types of query approaches in enforcing fine-grained access control in relational databases.

Query Modification Approaches

Truman Models: R. Agrawal et al. in [29] named this category of query modification Truman models, in which *each* user of the system has a *restricted view* of the full database. In these models a parametrized authorization view is created by the database admin is created for each user of the system that covers all the relations within the database. Therefore, during the run-time of the system, the user's query is modified based on the parametrized views of each relations within the query; and plugged in run-time values (e.g., userID, time, etc.) I present the first presented query modification approach, and a current influential approach of this type.

- *First approach:* The first query modification approach, to enforce fine-granularity of access control, was presented by Michael Stonebraker and Engene Wang [155]. They defined an access control system that is based on a query modification algorithm that gets the query from the user and then modify the query based on the access rights of the user. Therefore the user view is a subset of the overall view of the database that is based on the user's rights. For example, if John has the rights to only retrieve information about his salary, and he sends a query indicating he wants to read *all* the salaries (`Select * from User.salary`), the `query modifier`, based on John's access rights, automatically adds a conjunctive expression (`where User.username == "John"`). Therefore the query response *only* discloses John's salary to John. Note that, this approach were presented four decades ago based on the `INTEGRS` system, and this part, for adequacy, used SQL to give an example.
- *Oracle Virtual Private Database:* This is the only main stream approach in defining fine-grained access control for the relational database within the

row level. In this approach the `admin` writes a set of security policies for *each* table in the database using an extended version of SQL, PL/SQL [20]. These functions are applied for SQL-based data manipulation approaches (i.e., `SELECT`, `INSERT`, `UPDATE`, and `DELETE`). Moreover, it supports application context in which the decision on each user is made by the application based on the user's *attributes* (e.g., userID). These attributes are saved by the application context during the sessions of each user after his/her authentication. Oracle provides customised views to each user based on his/her *access control rights* by attaching a *where* clause predicates to the user's query. For the full details on the concept behind Oracle database architecture, please refer to [20].

- *Weaknesses:* As R. Agrawal pointed out in [29], the issue related to Oracle's VPD and other Truman-based models are based on the fact that, user's queries are modified which leads to inconsistencies based on the retrieved data from database and user's *original* query; that overall leaks to misleading responses. For example, if a user wants to see the average of all the marks in the class, but because of his/her access rights, gets *only* the average marks as a result. Furthermore, the users are *unaware* of parametrized views, and they write their query based on the database relations and *not* views. Therefore there could be a case in which a parametrized view allows a user to retrieve a data, but because user's query is based on the *database relation* this query is modified based on the relation. So the user is unable to see the results even though has the rights to do so. Moreover, as R. Agrawal showed in [29], writing parametrized views are very error-prone and complex.

Non-Truman Models: In these models, the user's query is first validated, if it *passes* the authorization validation, the query is executed normally, and it is rejected if it does not passes the test. This part gives an overview to two influential approachs in this domain introduced by Motro et al.[127] and Rizvi et al. [29].

- *Algebraic manipulation of view definitions:* In this method [127], the administrator is using *conjuncive* relational calculus to define *views* to describe the access permissions. Therefore the database related access control is based on these views and a subset of these views are granted to each user of the system. Then, for each user's query the system checks the definitions of views for the user (i.e., named *mask*); and the query result from the database without the views (i.e., named *answer*). Then the answer is applied to the mask to be provided to the user. Unlike Truman based approaches, to *avoid misleading answers*, this approach provides a statement regarding to the provided data. However this approach is more suitable compare to prior approaches, the main issue is that its not a complete approach. It *only* handles conjunctive query/views and does not support *aggregate* functions. Also it lacks extendibility description to support *disjunction* or *aggregation*.

- *Extending query rewriting:* This approach was presented by Rizvi et al. [29], that tried to cover the limitations on mentioned query rewriting techniques, by using a validity test mechanism for each query. In this model, the access control is defined based on parametrized authorization views, similar to [127]. Here, first the developer defines a *SQL view* definition with *attribute- based* parameters (e.g., userID, time, etc.) are created for each access control policy. Each of these parametrized authorizaiton views validates each user's query. Unlike [127] because of the use of SQL views, the database admin does not need to encode these views with a separate language (relational calculus in case of [127]). Also because of supporting the transparent querying in this model, the user defines a query based on the database relations, and can be answered based on the defined authorization views. In this model, the queries can be validated *conditionally* or *unconditionally*. When the user writes a query `P`, the mechanism of this model writes a query `P'` based on the user rights. If both results are the same answer the query `P` is unconditionally valid through all the database states. For example, if the user writes a query based on retrieving a mark, and his/her authorization views also only allows the user to see his/her mark, then this query *is* unconditional. Moreover, if the query is valid *only* on a state of a DB then the query is *conditionally* valid. For example, if the view states that the user can check the average mark of any class with more than 15 students, however the user defines a query `q` that checks for the average marks of a class with five students, then this query is *conditional* because it depends on the database state. In this mode, the conditional validity of each query is handled by *inference rules* that are implemented as a part of the query optimizer. To check different types of inference rules with their related algorithm refer to [29].

2.2.3 Automated Policy Testing

In the previous section I discussed different approaches to define and implement access control policies. Regardless of the development mechanism it is crucial to test these policies during the development and running of the system. During the development of a system, defined access control models, manually or mechanically, become a set of policies that will guard a set of controlled objects throughout the application; and therefore by testing these elements the efficiency of the defined models can be guaranteed. As a whole, testing the security elements of the system could be based on the following two points of views [166]:

Penetration Testing: In this case the testers need to attack the system based on different techniques and check for the system vulnerabilities, such as session hijacking [60], SQL injection [109, 33], and/or cross-site scripting [86].

Policy Testing: In this category the testers manually or mechanically test the model and/or its derived policies based on different perspectives [120, 126, 122, 124, 166, 133, 107, 106].

In this thesis the focus is on policy testing. Moreover, I are interested in fully automated testing, to design and implement an automated testing mechanism within the language environment. In this case the developers can benefit from this mechanism during the development of their model. The rest this part focuses on a number of automated tests generation techniques that were introduced over the recent years.

Model-based Approaches

Model-based testing is focusing on checking the efficiency of the test objectives, and their automated or semi-automated case generation [149]. In this part of the report I discuss a number of recent approaches in this domain.

Xu et al. [166] focused on detecting errors in the handwritten policies, that are derived from a set of RBAC rules, by using an automated model-based testing technique. In this study an access rule is a 5-tuple of `role`, `object`, `actions` (i.e., operations), `contexts` (i.e., access predicates), and `permissions`. They studied the following four characteristics in an access control policy:

Consistency: A set of rules is consistent where there are no conflicts within any two rules in this set. In this study two rules are conflicting when they have different permission rights (ie., permission or prohibition) and the union of their contexts with conjunction connector is true. For example the following two rules are conflicting: [(`student`, `thesis`,`submission`,`weekDays`,`permission`), (`student`,`thesis`,`submission`,`true`,`prohibition`)].

Non-redundancy: A set of rules are non-redundant when there is no two rules in the set that share the same role, object, action and a context in one rule subsumes the context in the second rule.

Completeness: A set of rules are complete when there is at least one rule for each `role`,`object`,`activity` and `context`. The incompleteness rules are detected based on the following algorithm:

- **Authorization Types:** First they extend the authorization types with the additional undefined (no defined authorization) element that states a set of role, activity, object does not have an authorization definition.
- **Additional Rules:** For each undefined set of role, object, and activity a rule is created with the context true, and authorization type undefined. Also the

type undefined is added to a set of consistent rules where their union of their negation contexts with disjunction connector implies to true. Then if these additional rules do not create inconsistency and redundancy the overall set of defined rules are in fact incomplete.

Test Model Construction. They automatically derive their test cases from access rules based on role-based partitioning. For this, they divide the access control model (a set of access control rules) into a number of subset. Each subset represents a role and its related activities. Then a PrT Net is constructed for each subset. PrT Nets are high-level petri nets, a formal method technique for system verification and modelling [80]. PrT Net can be represented in terms of logical formulas and therefore they are used for checking the correctness of the defined policies. The PrT Nets get constructed based on permission and prohibition authorization rights; and then PrT Nets composition, based on a role and its related activities. For more information on PrT Net construction algorithms see [166].

Test Model Analysis. For this, they create an initial marking for the test models (PrT Nets) based on data (i.e., activities) and configuration (i.e., contexts) tests. Then they analyze the test models based on verifying transition and state reachability, and model simulation. Each transition in constructed PrT Nets (test models) is representing an access control rule under an involved action and condition; and therefore should be accessible from the defined initial state. Also, if a state is known for its reachability but the verification declares it is unreachable or vice versa, then either the test model (PrT Net) or its initial state defined incorrectly. Moreover, they provide a simulation mechanism for analyzing the test models in terms of the behavior of the test models.

Generation of test cases and their executable code. They are using MISTA [11] to generate a set of test cases from the test model based on a set of criteria such as reachability tree coverage, in which, each edge in the graph is covered by a test case. The generated test cases have the tree structure so each test case is a path from the defined initial node to a leaf. The automated test cases need to transform into executable test code for the System Under Test (SUT). First they create a Model-Implementation Mapping (MIM) description based on elements in the test model (represented in PrT Nets) into related constructs in the SUT. Then they create the Model-Implemented Description, which consists of the test model and MIM. Finally they use SUT to automatically generate model-level tests.

Approach Efficiency. They evaluated their approach based on library management system (LMS) and auction management system (AMS) as a case study and used mutation testing to check the fault detection capability. For each case study they used the generated test cases against the hand-written fault injections.

Pretschner et al. [133] introduced a model based technique for testing the access control requirements based on generating test target and generation procedure evaluation. They considered a scenario where the accesses control implementation done manually. So in this case they want to check if the hand-written implementation reflects the defined access control model. This system consists of the following elements:

Access Control Model: They considered the access control model that consists of `roles`, `permissions`, and `context` hierarchies.

Combinatorial Testing: This technique [82] is used (i.e., using paired and related access control elements) to generate test targets with and without access control policy dependency:

- **Without the policy:** In this case the generated test targets are only take the access control information into consideration; and therefore the combinatorial testing will be fully applied on all the `roles`,`permissions`, and `contexts` and their related hierarchies. Then the random instantiation testing construct picks a random element from the gathered set.
- **Policy Dependent:** In this case, the test targets are generated from the access control policies that were implemented based on access control rules. For this, for each access rule they take the triple of `role`, `policy` and `context` name into consideration. Then, they generate the test targets based on all the combinations of all `roles`, `policies`, and `context` nodes that are below the considered nodes. Moreover because an access control rule can be either `permission` or `prohibition` they give the possibility to the tester to choose the nodes that do or do not conform to the node. For example, as Figure 2.23 shows, a rule consists of role `r`, permission `p`, and context `p` (i.e., shown in tick border boxes) in the overall hierarchies. Now the tester wants to define a test in which a set of roles are different from `r`, a set of permission that derive from `p`, and a set of context that does not apply to `c`. Then in this case all these combinations will be used to generate test targets.

  Then, same as first case, they choose random instances for each role, permission and context. Moreover for non-prohibitive instances, combinatorial testing considers respective instances.

Concrete Test Cases. The generated target tests are not executable and therefore they need to transform into executable code by considering the business logic of the target application. For this step, they claim that except the Java code the rest is generated.

2.3 WebDSL

Web application development typically requires a large amount of duplicated and boiler plate code that reduces the development efficiency. Moreover, in the Web application

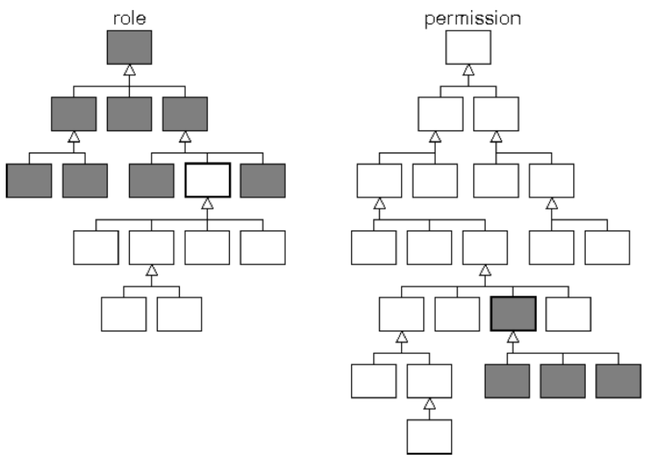


Figure 2.23: Related roles, permissions, and context based on the testing requirement [133]

domain, different technologies are merged together to build an application. For example, a very simple web-based library system could be written using languages such as *HTML*, *XML*, *CSS*, *JavaScript*, and *MySQL*; to be deployed on the server. This web application requires five different languages to be used together to make it working. These languages are changing rapidly and new functionalities often are added to them. Due to these reasons and the fact that software systems are getting more complex, I need a more abstract and generative web application development mechanism. To choose a suitable mechanism, I look into six modern Web frameworks based on the following criteria:

Open Source: It is essential for my research that, the chosen mechanism is extendible. So these frameworks are checked on being an open source project. As Table 2.3 shows, all the chosen frameworks are open source projects.

Supports authentication: As mentioned in Section 1.1, access control elements of an application, are built on the top of the authentication element, and therefore, it is essential for the framework to support authentication. As can be seen in Table 2.3, all the frameworks do support authentication.

Coarse-grained access control: It is important to see if the framework supports any type of coarse-grained access control model (i.e., DAC, MAC, or RBAC). As shown in Table 2.3, WebDSL is the only framework that supports DAC, MAC and RBAC.

Authorisation Management Modelling: Authorization management mechanism refers to a set of functionalities for the admin of the system to manage the rights of the users of the system. Therefore, it is important for the Web application mechanism to support modelling of such mechanisms. As Table 2.3 shows, non of the chosen frameworks do support authorisation management modelling.

Documentation: Each framework provides a set of functionalities for the developer. To extend these functionalities the framework needs to be documented, such that it

Table 2.3: Comparisons between modern Web Frameworks

Charactrestics	**Ruby on Rails** [16]	**WebDSL** [22]	**Play** [13]	**Wakanda** [21]	**Grails** [8]	**Django** [3]
Open Source	Yes	Yes	Yes	Yes	Yes	Yes
Authentication Support	Yes	Yes	Yes	Yes	Yes	Yes
Coarse-Grained Access Control	No	Yes (DAC, MAC, and RBAC)	No	No	Yes (roles)	Yes (roles, groups)
Authorization Management Modelling	No	No	No	No	No	No
Documentation	Yes	Yes	Yes	Yes	Yes	Yes
Internal DSL	Yes	Yes	Yes	No	Yes	Yes

can be used by the developers who extend the framework. As Table 2.3 shows, all the chosen frameworks are fully documented.

Support Internal DSLs: This research wants to provide the developer a declarative and fine-grained policy language by extending a Web development mechanism. Moreover, the developer's defined models, will be validated and verified by the policy language during its compile time. Therefore, it is essential that the framework provides the ability for the developer, to be extended internally, by a DSL. By this way, the developer is able to design and implement the desired syntax definition and code transformation and validation and verification functionalities. As Table 2.3 shows, all the selected frameworks provide internal DSL functionalities for the developers.

Based on the above-mentioned characteristics, WebDSL, Grails, and Django satisfy all of them but WebDSL is the only framework that supports coarse-grained access control for DAC, MAC, and RBAC. Therefore, among these technologies, I choose *WebDSL*, an open-source domain specific language for creating dynamic web application with rich data models [83, 84, 160, 162]. WebDSL also allows the developer to declaratively define different types of access controls (DAC, MAC, RBAC) using *access rules.*

The aim of this section is to give an overview of the WebDSL language with a focus on the language components and its compiler.

2.3.1 WebDSL Language Layers

WebDSL consists of a number of sub-languages [91] that are linked together in a pipeline manner. The pipeline, as shown in Figure 2.24, has a top-down transformation direction which means that the higher language layers are compiled down their subsequent lower layer. For example in Figure 2.24 the specified access control code will generate code in the *Base WebDSL*, which in turn will generate code in *Core WebDSL*. In WebDSL the structured code is considered as a *model* [91], and there are different types of generation mechanisms that are used in the WebDSL pipeline.

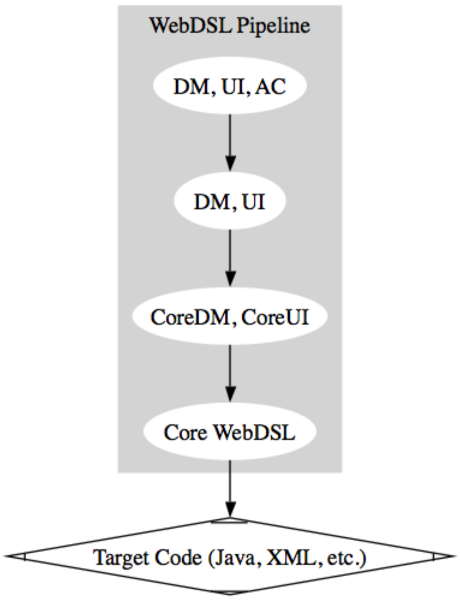


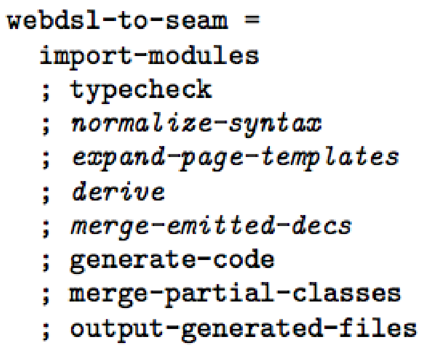
Figure 2.24: WebDSL language layers

According to Hemel et al. [91], the pipeline approach has a number of advantages:

Easy Extension: Extending WebDSL is easy, as the compiler is structured as a pipeline, and the extension component could as well transform to its lower layer without interfering with the other components.

Improved and Simplified Error Handling and Analysis: The modularity transformation mechanism in WebDSL's compiler happens incrementally. For example, as discussed in [91] (see Figure 2.25) different types are linked together and create a transformation pipeline. Therefore in this figure, the type checking strategy can be also added to the WebDSL-to-Seam transformation function. This type checking is context sensitive and can check global to local variables. For instance, it can check the whole application, or a page inside an application, depends on where they are used within the transformation functions. These checks can be extended and used throughout the other part of transformation functions as well. Therefore in WebDSL, it provides an efficient mechanism for implementing which has better error handling and analysing mechanisms, while transforming.

Less Complex Transformations: From the time the developer writes a WebDSL program, a large number of steps are used to transform WebDSL code to the target



```
webdsl-to-seam =
  import-modules
  ; typecheck
  ; normalize-syntax
  ; expand-page-templates
  ; derive
  ; merge-emitted-decs
  ; generate-code
  ; merge-partial-classes
  ; output-generated-files
```

Figure 2.25: WebDSL to Seam transformation function [91]

code. Therefore, I avoid relying on a single complex transformation step, and WebDSL compiler breaks the transformation into simpler steps, which leads to less complex transformation rules.

I now provide an explanation for each part of the language based on the pipeline structure. For each part, except WebDSL core, I will provide an example using WebDSL source code.

Core WebDSL

The overall WebDSL compiler task is to desugar all higher layers to core WebDSL and then to transform the core components to the Java-based web application. The core components of WebDSL are *core data model* and *core user interface*. WebDSL programmer cannot use these components, directly in their programs, but their presence, as an intermediate representation, makes the transformation easier, by acting as a middle ground between WebDSL and the target code [91]. Therefore, the core data model and user interface structures are close to the Java structure, and the data model and user interface written in WebDSL will be desugared into these core modules.

Base WebDSL

The Base WebDSL layer consists of two main components: *data model* and *user interface*. WebDSL programmers can use these components directly to define their web application. In the data model the developer defines the data model structure that is later transformed to SQL tables. The user interface describes the pages and the elements allowed inside the pages. By just using the base WebDSL, the developer can already define a web application and deploy it on the server. This part of the report describes the *data model* and *user interface* components of WebDSL.

As shown in Listing 2.2, the *data-model* in WebDSL specifies the application's entities and their properties. Properties are holding a name and its type. The different types of the properties are:

Value Types: In addition to the basic data types (e.g., float, int, etc.), WebDSL also includes some domain-specific data types such as URL. Value types are indicated by *::* between the property name and its type, for example `username :: String`.

Set Types By using the *Set* constructor, I are indicating that the property holds the set of instances of the base type or entity. For example in Listing 2.2 at line three, I am stating that the courses property of the student entity holds the instances of a set of courses.

Composite Association: By using <> between the property name and its type, the developer defines a composite association type. In this type, the property owns the associated object. For example, the property `marks <> Set<Mark>` inside a student entity indicates that the student owns a set of the entity marks that were given to him/her.

Referential Association: By using of → between the property name and its type, the developer defines a referential association type. In this type, the property owns the associated object. For example, the property `myTutor -> Teacher` inside a student entity indicates that the property myTutor holds an instance of the teacher entity, this means, each student has a tutor of type teacher.

Annotations on Types: There are also a number of annotations that are used for types, in particular *name* for using the property instance as the display name of the entity (see line two of Listing 2.2) and *inverse* to create the inverse relation between two properties of two entities.

Note that the two association types are very similar, but the difference is that in a composite association, if the referrer instance is deleted, all the instances owned by the referrer will be deleted too. This does not apply for the referential association. There is also a derived functionality that can automatically derive CRUD (i.e., create, read, update, delete) pages for an entity.

```
entity Student {
  studentID    :: String  (name)
  courses       -> Set<Course>
  myTutor       -> Teacher
  marks         <> Set<Mark>  (inverse=Mark.student)
}
```

Listing 2.2: Student entity

WebDSL provides a data validation mechanism to enforce the required validation checks on the data model. The developer can use this to define the provided validation checks on the properties of the entities. For example, as Listing 2.3 shows, the *isUnique()* validation check will display an error to the user if the chosen username is already taken.

```
entity User {
  username :: String (id, validate(isUnique(), "Username is taken"))
}
```

Listing 2.3: Data validation in WebDSL

In WebDSL, the user interface belongs to pages and components inside the pages (i.e., form, group, section, table, etc.) Moreover the pages can be defined using the following code: *define page pagename()*. For example the developer can define an empty page name blog by writing `define page blog()`.

As any other language, the user interface presents and collects the data on the web application's pages. The layout of the data presentation on the page could be structured with the available mark up components such as *group*. Manipulating the data is also possible with the *form*, *action* and the *control of the navigation*. These components also can be embedded in other presentation components such as *table*.

Action code in WebDSL described using expression and available statements in WebDSL, such as literals and operators [22]. The following Listing 2.4 shows the *viewUser* page which uses different page elements such as *form group*, and *table*. For structuring the pages in WebDSL, each page can have a number of groups and forms. The table consists of rows and columns, for example in the following code the first row of the table indicate its columns names, and the rest of the rows are outputs based on the each user information. It also shows the use of an imported CSS style file (see line three of Listing 2.4) for the page.

```
define page viewUser() {
  header{"View All Users Information"}
  includeCSS("style2.css")
  form{ group("University Related Information"){
    table{
     row{column{output("User Name")}      column{output("Password")}
         column{output("University Id")} column{output("DSSE ID")}
         column{output("Uni Tel")}        column{output("Uni Email")}
         column{output("Joined date")}
       }
     for (u: User){
      row{ column{output(u.userName)}   column{output(u.password)}
           column{output(u.uniID)}      column{output(u.dsseID)}
           column{output(u.uniTel)}     column{output(u.uniEmail)}
           column{output(u.joinedDate)}
```

```
          }
    }}}}}
```

Listing 2.4: User interface in WebDSL RBAC

Note that the Java and Ajax source code could be used inside a page in WebDSL. However, this aspect of WebDSL is out of the scope of our research. For more information, see [22].

Access Control in WebDSL

This part of the report presents the components that are related directly or indirectly to the access control part of WebDSL. These components are discussed in detail as they are directly related to our research. Also at the end of this part, I discuss WebDSL's approach towards fine-grained access control and explain why I choose a different approach rather than WebDSL.

As shown in the WebDSL pipeline, the access control components are built on the top of *base WebDSL*, therefore the code written on this level will be transformed to the base WebDSL during compile-time.

In order to deploy any access control policy, an *authentication* mechanism needs to exist. WebDSL provides the notion of *principal* which is used to define the *credentials* on which authentication is based. For example, as Listing 2.5 shows (taken from [22]), the credentials that are used for authentication are the *name* and *password* of the user entity. However, this is not enough for developing an authentication mechanism in WebDSL, because WebDSL, so far, does not present a specific notion for authentication but it provides a template, through its website, that could be used as an off-shelf component [22].

Note that the result of the defined credentials is a session entity that holds each user's session after the authentication. The general idea of the access control component in WebDSL is based on the notion of *access rules* for page and template accesses, and on preventing the links to be shown if they refer to inaccessible pages/templates. For example, as the code in Listing 2.5 shows (taken from [22]), the access rule restricts the access if the user wants to access an editUser page which is not hers. Please note that the access rules are defined separately from the page or template definitions, which are later imported into the main web application module.

```
//Defining authentication credentials
principal is User with credentials username, password

//One access rule for a page
access control rules
 rule page editUser(u:User){
      u == principal
  }
  //More access rules on pages and templates
  //could be defined here.
```

Listing 2.5: Defining authentication credentials and an access rule for a page

Different access rules could be used to define the three basic access control policies DAC, MAC, and RBAC. The following discussions shows how RBAC can be defined in WebDSL and if the control is fine-grained enough.

The first elements I need to define are the data models for the RBAC components, such as roles. The data model should contain the following elements:

Role: The entity *role* needs to be defined for holding the role names and the users related to each role, as shown in Listing 2.6.

User Role relation: In the role entity I define that each role is related to a set of users. Now I need to define that each user can have many roles, based on the role's name and the user they are assigned to. For this reason, I extend the entity *User* to obtain this relation, as shown in Listing 2.6.

Active Roles: After the administrator assigned a number of roles to the user, the user needs to activate these roles. Moreover, the system should know which roles are active in each user's session. For this reason, I extend the entity session *securityContext* to hold a set of roles activated by the user, as shown in Listing 2.6.

```
//Entity Role
entity Role {
  name :: String (name)
  users -> Set<User>
}

//Extending User entity
extend entity User {
 roles -> Set<Role> (inverse=Role.users)
}

//Adding activatedRoles to the user's session
extend session securityContext {
  activatedRoles -> Set<Role> }
```

Listing 2.6: Defining Role entity and extending User entity and securityContext

> Note that, the session entity securityContext is automatically *derived* from the *principal* notion that I defined for each user authentication (i.e., username and password).

After defining the RBAC data model, I need to define which access checks are required to access a resource such as a page. Note that as soon as the developer uses the principal and authentication mechanism, the default setting for each page turns to *Deny Access*. For this reason I need to define point cuts to set the initial accessing requirement to true, as shown in Listing 2.7.

```
pointcut openPages() {
page root(*)
// point cuts to the other pages
// could be added here.
}
rule pointcut openPages() {true}
```

Listing 2.7: Point cuts for initial access setting on the pages of the web application

After defining the data model and initial access setting, I need to define a mechanism for checking the resources, based on the user's activated roles. As shown in Listing 2.8, I use a function that iterates over the user's activated roles list and determines if the user has the role(s) required to use that resource. The result of the function is either true (i.e., grant access to the resource) or false (i.e., deny access to the resource). For example, the following code shows if any resource is based on the studentFunc function (such as viewStudent page in Listing 2.9), then the user needs to have a student role to access it.

```
function studentFunc(activatedRolesSet : Set<Role>) : Bool
{ var activatedRoles := activatedRolesSet.list();
  for (activatedRole : Role in activatedRolesList) {
    if (studentViewPage(activatedRole)) {
     return true; //Grant Access
    }
   }
    return false; //Deny Access
  }
  predicate studentViewPage(r :Role){
    r == student_role
  }
```

Listing 2.8: Student function in WebDSL

Next, I need to define the access control access rules for each page or template. As shown in the Listing 2.9, for the page *viewStudent* I are passing the activated roles from the user to the studentFunc (see Listing 2.8). In this case, the user needs to have the student role to access the page. In the following code, line four, I are granting full access to the about page.

```
rule page viewStudent(*){
  studentFunc(securityContext.activatedRoles)
}
rule page about(*){true}
```

Listing 2.9: An example of access rules on pages and templates

After discussing how I can define the RBAC in WebDSL, I now examine the limitations of this approach in terms of access control and fine-granularity. As I have seen above, the access rules are on the pages and templates; recently this has been applied to actions as well. This results in nested rules, which according to the WebDSL creators is not desirable [22]. On WebDSL's website these nested rules are referred to fine-grained access control; however I *do not* consider these nested rule functionalities to be "fine-grained" because they ignore the underlying data model where the fine-grained components, actually are defined (i.e., properties of entities). In our point of view, there is a difference between *putting access on the components of the page (i.e., WebDSL approach)* and *putting the access control control on the page where the fine-grained instances are (our approach).* Please note that, although I do not refer to their approach as fine-grained access control, I will generate these nested functionalities in our future work to put the access rules on the desired fine-grained components.

Finally, in our opinion, the WebDSL approach is not abstract enough, in the sense that the notion of RBAC needs to be defined based on the steps discussed. It is more desirable to bring the abstraction close to the RBAC definitions.

2.3.2 Extending WebDSL

This part gives an overview on two essential elements, syntax definition and code transformation, for extending WebDSL [162]. WebDSL has been built using syntax definition formalism (SDF) [90] for syntax definition purposes, and Stratego [50] for code transformation functionalities. This section explains these two elements.

Syntax Definition

After gathering the domain knowledge, the first step for developing a DSL is to define the syntax and semantic features of the language. There are a wide range of languages to define the syntax of the languages such as DMS [37], Antlr [116], and Syntax Definition Formalism (SDF) [90, 161, 159]. Based on the fact that I are using WebDSL as a host language of our approach, I chose SDF and its related toolset to describe common aspects behind defining syntax for a language.

Syntax Definition Formalism

SDF [90, 161] is declarative, modular, class-free formalism language for defining arbitrary context-free grammars. The aim of the SDF definition is to define a set of strings, abstract syntax trees (AST) and the relations between them [90]. The language developer can construct a set of SDF definitions based on the following elements:

Sorts: Each sort consists of non-terminal and domain names that will be used in other part of the syntax definitions.

Lexical Syntax: The lexical syntax definition describes the low structure of the programming language based on a set of production rules. Each production is a definition of symbol and has the `symbol* -> symbol {attribute*}` structure [90] where the symbol on the right hand side defined based on an arbitrary list of left hand side symbols; and the following list of attributes could be applied to the production to avoid the lexical ambiguities:

- **left, right, non-assoc, assoc:** Productions can be overlapped with themselves. These associative attributes can be used to define different associativity types.
- **prefer:** To define priority between a rule and its other derivatives, the developer can use prefer attribute to say to the parser to choose the branch that derived from prefer attribute compared to the other ambiguous branches with no prefer attribute.
- **reject:** This can be used for defining a difference between two context-free operators.

Context-Free Syntax: Concrete and abstract syntactic structure of a language are described by context-free productions. As the following code shows, Listing 2.10, the left hand side represents the syntax structure and the right hand side represents the rule constructer name for that rule.

```
exports
 sorts ComWord Comment
 lexical syntax
   [\ \t\n] -> LAYOUT
   [\ \n\t\/]+ -> ComWord

 context-free syntax
   "/*" ComWord* "*/" -> Comment
   Comment            -> LAYOUT

 context-free restrictions
   LAYOUT? -/- [\ \t\n]
```

```
LAYOUT? -/- [\/].[\*]
```

Listing 2.10: An example of context-free syntax

SDF2 Toolset: This part of the report discusses the required SDF tools to construct parse table and signature files from the developer's defined SDF definitions.

- **pack-sdf:** The first step the developer needs for a set of defined SDF definitions is to pack all of them into a single SDF definition using pack-sdf tool.
- **sdf2table:** Then the developer needs to create a parse table based on the packed SDF file. This table enables the parser (e.g., SGLR parser) to parse the language and construct an AST based on the defined grammar.
- **sdf2rtg and rtg2sig:** For creating a signature file, the developer needs to use sdf2rtg and rtg2sig tools. The signature file is used during the parsing step to validate the expressions.

Code Transformation

Any programming language requires a code generator to transform the written source code [59]. There are various code generators such as Coccinelle [104], TXL [114], ASF+SDF [159], RASCAL [110], and Stratego/XT [50]. However because of the direction of this research I focus on the Staratego/XT code generator as it is used within WebDSL compiler.

Moreover, I focus on Stratego/XT as the code generator for the WebDSL language.

Stratego/XT

This part of the thesis discusses Stratego/XT as a language and as a code transformer in the WebDSL compiler.

The input of a Stratego transformation, is the *abstract syntax tree* (AST) generated from the parser, and the output is list(s) of AST(s) based on the stage of transformation (i.e., WebDSL, Java, XML, etc.). Moreover, Stratego can be used to apply the transformation to any models with an abstract syntax definition. Stratego code consists of *rewrite rules* and *strategies*. Conceptually, there is no difference between these two and both are used for code transformation based on a set of conditions, but they are syntactically different. Here, I just discuss rewrite rules; for more information on strategies please refer to [50].

The rewrite rules in Stratego have the following structure:

$$R \ : \ Pattern1 \ \rightarrow \ Pattern2 \ where \ Conditions* \tag{2.1}$$

In the above rewrite rule, *R* is a name of the rule, which is used for invoking the rule. *Pattern1* is a match term used against the terms inside the AST. *Conditions** is a sequence of conditions. I can use *with* instead of where, when I want all the conditions to be true before the transformation occurs. *Pattern2* is a built term, that is a term which was transformed based on the matched term (Pattern1) and the condition set, *Conditions**.

Stratego also provides the functionality of using *concrete syntax* for the built term. This means, the patterns can be written using the actual syntax of the respective languages (e.g., Java, WebDSL, etc.). This is helpful because the Stratego programmer does not need to take care of *annotations* and the internal *grammatical structure* of the code [50]. If the programmer just uses the target code, as a built term, I achieve *static transformation*. It is static because the target code does not contain any dynamic component (i.e., variable, function, etc.). However, Stratego also allows us to use *meta variables* inside the concrete syntax fragments [50], as shown in the following Listing 2.11.

```
tableHeader : X -> code-to-emit := webdsl*|[
  var headString : List<String> := [e_headerNames*]
  row{for(hdr:String in headString) {column{output(hdr)}}}
  ]|
 where
   e_headerNames* := <map(getHeaderNames)>X
```

Listing 2.11: Using Concrete Syntax

Meta-variable as shown in the above code (i.e., e_headerNames*) is a component that holds a value that can be passed inside the target code. By using meta-variables the transformation also contains a dynamic component (value of the meta-variable) that can be calculated during the transformation. The concrete syntax code is located between |*[...]*|, and *code-to-emit* is a variable that holds this concrete syntax fragment, and is used somewhere else as a part of a bigger code fragment. Additional transformations can be applied to the generated code, which makes it possible to have the separation of concerns, and divide the code generation steps into number of steps based on the required concerns inside the compiler.

Let us now explain different transformation types that are used in the pipeline and written with Stratego, which can be summarised as *semantic analysis*, *syntactic normalisation*, and *generative derive mechanisms*.

Semantic Analysis

Stratego provides us with *static semantic constraints*. The constraints related to these models are valid syntactically but are not valid based on the overall model, for example *invalid entity call*. Therefore the generator needs another separate type checking stage, from the normal type checking, to check these constraints. The information gathered at this stage is also used for other stages, as a context information, for the rest of the *model-to-model* transformation [91].

Stratego supports *context sensitive local-to-global* transformation for type checking. In this transformation the type information is distributed from the identifier to the required locations inside the AST [50, 91]. These types of transformation could be written in Stratego by using *dynamic rewrite rules*. In this type of rewrite rule, a definition of new rewrite rule is possible during the *run time*.

In Listing 2.12 (taken from [91]), the rewrite rule *typecheck-variable* checks the abstract syntax *Var(x)* as a match term and uses the dynamic rule type during the runtime to evaluate the type of x and attach it to the terms as an annotation. If the type check call fails, then the declaration for x is missing and the rule fails with an error message.

```
typecheck-variable :
  Var(x) -> Var(x){Type(t)}
  where
   if not(t := <TypeOf> x ) then
    typecheck-error (|["Undeclared Variable ",x ,"referenced"])
   end
```

Listing 2.12: Type checking

The Stratego supports the scope mechanism and the dynamic rules could be used for fragment of the code, because they could be defined based on the limited scope.

Syntactic Normalization

Syntactic normalization or *desugaring*, gives the ability to the compiler developer to define a new language construct that compactly represents functionality already provided by the base language. *local-to-local* and *local-to-global* transformation rules are required for these types of new construct.

In *local-to-local transformation*, the match fragment replaces with target code fragment without providing any other code fragments as a result, such as shown in Listing 2.13 (taken from [91]). In this code multiple arguments inside the text list will transform to a single argument by decomposing (See Line 2) the first element and taking the first element of the list repeatedly (See Line 6).

```
NormalizeSyntax :
  |[ text(e1,e2,e*) {} elem*]| -> |[ text(e1) text(e2,e*){} elem*]|
NormalizeSyntax :
  |[ for (x : srt in e1) order by e2){elem*}]| ->
  |[ for (x : srt in e1) where true  order by e2){elem*}]|
normalize-syntax = topdown(repeat(normalizeSyntax))
```

Listing 2.13: Syntax normalization

The *local-to-global* rewrite rule, rewrites the local element but provides a fragment that should be placed somewhere else inside the model (AST code) [91]. As Listing 2.14 (taken from [91]) shows, the list of blogs returned back to the user based on the data in a reverse chronological order.

```
[e.title
 for(e : BlogEntry in b.entries) where
  e.created > date
  order by e.created desc)]
```

Listing 2.14: Requiring list of Blog

The Listing 2.15 (taken from [91]) shows the required transformation for the Listing 2.14. The lift rewrite rule in Listing 2.15 generates a global function (x_func) in which the call to the generated function sits instead of the expression ($arg*$). The free variables passed to the *emit-webdsl-dec* rule which stores the new defined function inside a dynamic rule. These separated fragments of emitted code will be merged together, during the *merge-emitted-decs* generator stage [91].

```
lift:
 |[  [e for (x: srt in e2 where e3 order by e4)] ]| ->
 |[ x_fun(arg*)]|
where x_fun := <newname>
    ; free-vars := <collect-free-vars> (e,e2,e3,e4)
    ; param*     := <map(build-param)> free-vars
    ;arg*            := <map(build-arg)> free-vars
    ;<emit-webdsl-dec> |[
      globals {
          function x_fun(param*) : List <srt> {
            var y : List<srt> := [];
            for(x: srt in e2 where e3 order by e4){
              y.add(e);}
            return y; } }
     ]|
```

Listing 2.15: Local-to-global syntactic normalization

Generative Derive Mechanisms

Generators can be invoked for defining functionalities based on generative abstractions. For example, in a create page based on the property type the input instance is created. For instance, a text box for string properties.

As Listing 2.16 shows (taken from [91]), type of the expression is derived based on each input type. There is also a rule for derive output. For more information on generative abstractions refer to [91].

```
DeriveInput :
|[ input(e) {}]| ->
|[ select (s: srt, Select, e) ]|
where SimpleSort(srt) := <get-type> e
     ; <defined-entity> SimpleSort(srt)

DeriveOutput :
|[ output(e) {} ]| -> |[ navigate(x_view(e)){text(e.name)}]|
where SimpleSort(s) := <get-type> e
; <defined-entity> SimpleSort(s)
; x_view := <view-page-for-entity> s
```

Listing 2.16: Generative Derive Mechanisms

2.4 Characteristic Requirements

In the previous section I showed why WebDSL was chosen among other Web development mechanisms, however this mechanism is not fine-grained in terms of security policies, separation of concerns, and does not support any validation and verification mechanism for their coarse-grained policies.

By considering these shortcomings, this section gathers the overall characteristic requirements of the fine-grained policy language that will be built to satisfy the direction of this research. The characteristics are as following:

Context-free and Declarative Language: By looking at the policy languages defined in the last decade I see that all of them are defined with the context of the host language. For example, as shown in XACML the rules are defined within XML nodes. This adds to the complexity and readability of the language. The policy language needs to define policies declaratively regardless of the language terminology. This leads the non-developers in the security domain to use the language as well as the developers. In this case the security properties defined in plain natural language could be easily translate to the policy language.

Support for different granularity levels: The language should support access control and authorization management mechanisms for the objects of the system in different granularity levels.I want to give the ability to the developer to define models to put authorization on fine-grained to coarse-grained objects. I do not want to restrict the developers to define models for a certain granular level.

Separation of Concerns: It is very important the defined models by the language do not merge with the business logic of the application during the development of the Web application. I would like to provide the environment separated form the business logic.

Validation and verification Mechanism: The compiler of the DSL needs to validate and verify the defined access control and authorization management policies during the compile time. For this, I need to adopt a formal verification technique to translate the defined policies into a logical format and test the properties of the defined model. Moreover,I need to validate the target language as well. These properties are largely missing in the domain of the access control and authorization management modelling.

2.5 Summary and Conclusions

In this chapter, I investigated the research background and related domains to this research. In the first section, I presented three main access control models, discretionary-, mandatory-, and role-based access control and discussed their structures. Then, I showed the authorisation management elements during the run-time of the system. Moreover, I study the access control and authorisation management mechanisms in the area of languages, database approaches and validation and verification techniques.

After that, I investigated different factors of six frameworks. I found that among those frameworks WebDSL was the only framework that supports coarse-grained access control for DAC, MAC, and RBAC. For this reason, WebDSL was chosen to be extended to support declarative- and fine-grained access control policies. It is important to note that, WebDSL, itself, does not support fine-grained access and authorisation management policies during its development phase, and validation and verification mechanism, for its defined coarse-grained policies, during its compilation phase. These shortcomings of WebDSL were highlighted as requirements for my research. Finally, for extending WebDSL, I studied its compiler structure for syntax definition and code transformation. WebDSL uses SDF and Stratego for defining its syntax and transformation functions. These languages will be used in this research to extend the WebDSL to support declarative- and fine-grained policy language.

Chapter 3

Syntax and Semantics of Φ Models

3.1 Introduction

This research develops Φ, *a declarative and fine-grained policy language for the Web application domain*. By using Φ, the developer can declaratively define a set of fine-grained access control models together with sufficiency checks, and authorisation management models. Φ's declarative approach gives developer the ability to spend less time developing a set of access control models and focus on *what* they want them to do, while the compiler knows *how* to implement the mechanisms. The following principles were considered in the process of the designing the Φ language:

Domain-specificity: The developer should be able to define the access control model based on access control terminologies (e.g., SOD relations, confidentiality labels, etc.), instead of using programming language terminologies (e.g., class, function, method, etc.).

Granularity and Flexibility: A Web application consists of a number of a elements that have different granularity levels. For example, a Web application can have a coarse-grained element such as a page that consists of more fine-grained items such as the `username` that is presented in a cell of a table, or a template call that consists of an XML hierarchy of the instances of the data model. Therefore, an access control *must* provide an environment to support *both* coarse and fine-grained objects and their relations to their components and the other objects. Moreover, the concept of fine-grained access control model does not only mean an access control model that enforces authorisation on individual objects, but also it should be fine-grained by itself. This means, it should be flexible enough so that the developer can use fine-grained policies.

Attribute Constraints: An access control model becomes more efficient if it additionally supports constraints that are based on *user*, *data*, or *system* attributes. For example, an access control model that supports a *time* attribute is more effective than the one that does not, because the developer can enforce a set of policies with additional time range (e.g., office hours) that helps the security of the system.

Authorisation Management: An authorisation management system (AMS) is a mechanism that *displays* the relations between users of the system and the defined and in-use policies; and provides an *editing* operation on these relations during the run-time of the system. Furthermore, these operations can be access controlled as specified by an authorisation management model (AM) that is defined by the developer of the system. For example, an AMS in a system that enforces RBAC displays and provides an editing operation on a set of relations involving *users*, *roles*, and *permissions* (i.e., the relation between objects and operations). However, how *these* operations are access controlled is based on the type of AM (i.e., DAC-, MAC-, or RBAC-Based) that is defined by the developer, and their underlying access control model. In the literature, there are various approaches that discuss access control models (both coarse and fine-grained) [140, 130, 71] regardless of how they are managed by the administrators of the system. It is clear that failing this step would be a main drawback of any access control model, because vague understanding of the AMS can lead to security leaks. Any access control model *must* support authorisation management based on some access control type, so that it is defined how the access control element of the system is managed during the system run-time.

Developer Control: The Φ compiler can verify and validate the models against the language semantics, but only the developer knows the nature and the usage context of Φ models. There must thus be a set of additional checks defined by the developers. I call these checks *coverage checks*. For example, an access control that covers 10 percent of an online forum is more appropriate than the access control that covers 80 percent of a country's nuclear information; because in the context of an online forum 10 percent access control coverage is enough but in case of nuclear information *all* the information needs to be access controlled.

Multiplicity of Approaches: A Web application can be used for covering different aspects of an organisation and its operations which can have different access control needs; therefore it is essential for a policy language to support multiple access control models in one application. The developer can then enforce different types of access control policies for each aspect of the application. For example, in a hospital the routine management operations such as editing personnel information can be controlled by role-based policies, while the critical information about patients who need to be monitored by a nurse can be controlled with discretionary access control policies. These design decisions should be given to the developer of the Web application, to be considered during the access control development of a Web application.

The Φ language is implemented as an extension to the domain-specific language WebDSL. This chapter describes the concepts behind the elements of Φ models as well as their syntax and semantics. More specifically, the current chapter describes Φ models in terms of their meanings through semantical definition, syntax, examples, and their example transitions. Sections 3.1.3 and 3.1.4 give the semantical meaning of a Web application without and with sessions. Then, in Sections 3.2, 3.3, and 3.4, the semantics, syntax, examples of Φ-based access control model are discussed. In Sections 3.5, 3.6, and 3.7, the semantical meaning, syntax definition with example of the Φ-based authorization management models are given and discussed. Finally, in Section 3.8, the summary and conclusion of this chapter is provided.

This thesis uses linear temporal logic (LTL) [51] to formalize Φ models. So this part of the thesis discusses LTL, the specification language that defines the Φ model semantics, and the standard formalisation of a Web application.

3.1.1 Linear Temporal Logic

In computer science, LTL is used to model the behaviour of software systems during their run-time based on the following concepts:

Time: LTL is a version of temporal logic that uses the notion of *time* in a linear order. Compared to classical logic, the temporal characteristic of an LTL formula is useful for expressing the logical state of an event or state that changes over the system run-time.

Possible Worlds: LTL defines the system behaviour as a computational path based on a set of states, such as *S0* → *S1* → *S2* → *S1*. This path shows how the system evolves over time. At each given moment, the system is in a logical state (a world) which holds a set of event or state of the system. The system changes over time based on different logical states (i.e., worlds). The union set of all these logical states creates all *possible worlds* that shows all the logical states that the system can evolve based on, over its run-time. Note that the time is linear, so that each possible world has only one successor.

Temporal Operators: LTL uses temporal operators to show when a set of elements of the system are held through time. The temporal operators used in this thesis are: $\bigcirc$ (the proposition is true in next step), $\Box$ (the proposition is true in the future, i.e., in the current and all the following steps), $\diamond$ (sometime in the future, the proposition is true), *until* (the elements are true until another elements stays true).

3.1.2 Specification Language

This part shows the specification language to formalise the semantics of Web applications and Φ-models, in LTL [51]. Each specification can consist of the following elements as shown in Listing 3.1:

```
spec specificationName =
  sorts s1,...,sN
3
  rigid    op   rop1; ...
  rigid    op   ropN;
  flexible op   fop1; ...
  flexible op   fopN;
  rigid    pred rpd1 ...
  rigid    pred rpdN
  flexible pred fpd1 ...
  flexible pred fpdN
12
  for all n: s1,..., m: sN;
    □ (formula1) ...
    □ (formulaN)

end
```

Listing 3.1: The LTL Specification Language

Sorts: A list of identifiers is declared after the `sorts` keyword constitutes the core elements of each LTL model specification (Listing 3.1 line 2). These elements can be used through out the model to specify a set of relations and system behaviour over discrete time (see Listing 3.1 line 13-15).

Function Symbols: A set of identifiers are used to represent the available functionalities within the system. Each function symbol has a result of type core element. The correctness of each function symbol can be changed over time or stays *true* or *false* through *all* system run-time. For this, the function symbols that do not change over time are defined after the keyword *rigid* and the ones that change are defined after *flexible* keyword (see Listing 3.1 lines 4-7).

Predicate Symbols: An element of type predicate symbol (see Listing 3.1 *rpd1* or *fpd1* lines 8 to 11) holds a set of relations that are constructed based on the sorts elements. Similar to the function symbols, the *stateless* predicate symbols are defined after the **rigid** keyword, and the *state-based* predicate symbols are defined after the **flexible** keyword.

Axioms: The possible interpretations of both rigid and flexible function and predicate symbols are given by a number of axioms, which are universally closed *LTL formula* over the specification symbols. The axioms follow the declaration of the universal variables and their sorts.

Import Mechanism: An LTL specification can have an import mechanism in which it imports the *sorts*, *function and predicate symbols*, and *axioms* from another specification. As a result, the *name* of the imported LTL specification must be included. For example if the LTL specification in Listing 3.1 is importing an LTL specification with the name *importedSpec*, then the first line of the Listing 3.1 would be '`spec specificationName = importedSpec then`'.

3.1.3 Abstract Web Application Semantics

In our semantics, a Web application is modelled as a possibly infinite number of operations, that act on the Web application's objects. Listing 3.2 provides a formalised abstract representation of a Web application *without* the authentication and authorisation capabilities, based on the following elements:

Sorts: The sorts *User*, *Object*, and *Event* represent the users, objects and available operations on these objects, respectively, throughout the Web application run-time. All the data manipulation operations (i.e., create, read, update, delete) are abstracted to the sort *Event*. The specific type of the event and its arguments are formalised by different predicates, as described below. All the operations of type *Event* are applied to the system's objects (i.e., elements of sort *Object*) during the run-time of the system.

```
spec WebApp

  sorts User, Object, Event

  flexible pred:
    createOp: Event × Object
    readOp:   Event × Object
    updateOp: Event × Object × Object
    deleteOp: Event ×  Object

  for all e, e1, e2, e3: Event, o,o': Object;

    ◇ (createOp(e,o) ⇒ □ createOp(e,o))
    ◇ (readOp(e,o)   ⇒ □ readOp(e,o))
    ◇ (updateOp(e,o) ⇒ □ updateOp(e,o))
    ◇ (deleteOp(e,o) ⇒ □ deleteOp(e,o))
    ◇ (deleteOp(e,o) ⇒ (¬ readOp(e1,o) ∧ ¬ updateOp(e2,o,o'))
                           until create(e3,o)))

end
```

Listing 3.2: Semantics of a Web app in the first-order temporal logic

System behaviour: A Web application is possibly an infinite sequence of events (i.e., data manipulation operations) on its objects. For each type of operation (create, read, update, delete), there is a corresponding predicate that holds for each event *iff* the event was of the same type, and has been applied to the object. These operations represent all the create, read, update, and delete events on the system's objects. The intuition is that Xop(e,o) is true when *e* is an X-type event that has been occurred sometime in the past, to the object *o*. Hence Xop(e,o) must remain true once it has become true. The world in which it becomes true for the first time can be considered to be the time step at which the event (operationally) happened. As Listing 3.2 shows (i.e., lines 9-14), the system behaviour/operation executions are as follows:

Create: In case of create execution of e on the object o (i.e., line 11), createOp(e,s), remains true after the operation is executed some time in the future during the system run-time.

Read: the object is read, when the read execution of e on the object o (i.e., line 12) is executed.

Update: In some future moment of the Web application run-time, after the execution of update operation, updateOp(e,o, o') occurs, and the object o is updated by the object o'.

Delete: In some future moment, after the execution of delete operation e, on the object o, read operation e1 or update operation e2 can be applied to the object o until a second create operation e3 is applied to the object o.

3.1.4 Semantics of a Web Application with Sessions

After introducing the abstract semantics of a Web application in first-order temporal logic, I now add the concept of *users' sessions* within a Web application. This concept is essential in defining the semantics of the users' authentication and authorisation, and their interactions with the Web application. Accordingly, this part discusses the signature of a Web application with session (*WebAppWithSessionSign* in Listing 3.3) and its semantics (*WebAppWithSession* in Listing 3.4). Here, the semantical representation of a Web application with sessions is separated into a signature (i.e., *WebAppWithSessionSig*), and a list of axioms that specify the operational executions during system run-time (i.e., *WebAppWithSession*). This is because of the fact that, the signature is extended by the later semantical models; however the interpretation of CRUD operations (i.e., *cexec*, *rexec*, *uexec*, *dexec*) evolves in the later semantics based on the new features of these models. Therefore, the *WebAppWithSession* only represents the Web applications with sessions without any authorisation rights, and is thus not included in the later semantics.

Signature

As Listing 3.3 shows, the signature of a Web application with sessions is an extension of *WebApp* (Listing 3.2), so it has all the core and behavioural characteristics of a *WebApp* with the additional concept of user sessions (Sort *Session* in Listing 3.3 line 3); and a set of flexible predicates that represent the execution of the different data manipulation operations within each session. The signature is constructed based on the following concepts:

Session Model: As shown in Listing 3.3, the core elements of the *WebAppWithSessionSig* are the sorts *User*, *Object*, *Event* (from *WebApp*), and *Session* that represents all the sessions in the system which are created for authenticated users. Each session of type *Session* represents a unique session of a single user of the system who is authenticated to the system. Moreover, a Web application with session must support an authentication mechanism for the system's users. Therefore, as Listing 3.3 shows, this semantic has a rigid function *user* (line 5), that shows all the users of the system who are interacting with the system within their sessions of system run-time. It also contains the rigid predicate *happensIn* (line 6) that captures which (abstract) events happen in which sessions.

Operational Execution: For formulating the users' interactions with the system, there is a list of "execution" predicates (line 9-12) *cexec* (for creating an object), *rexec* (for reading an object), *uexec* (for updating an object with a new object), and *dexec* (for deleting an object). These predicates are true, if and only if the user was allowed to execute the operation captured in the event on the object; when then also interpret this to mean that, the execution has been successful.

```
spec WebAppWithSessionSig = WebApp then

3   sorts Session

    rigid Op user: Session -> User;
    rigid Pred happensIn: Session × Event;

8   flexible pred:
      cexec: User × Event × Object
      rexec: User × Event × Object
      uexec: User × Event × Object × Object
      dexec: User × Event × Object
```

Listing 3.3: The signature of a Web app with sessions

Axioms

Listing 3.4 shows the semantics of a Web application where users interact with the system through their sessions. The description of operational semantics are as follows:

Create: The formula in line 5 states that the user u has been able to execute a create operation e, on object o, in a session s, if and only if a create operation e on object o has occurred in this session, and the session s is in fact related to the user u.

Read: Also, the formula in line 7 states, if and only if, a read operation occurs in the user u's session, the user is able to execute a read operation e, on the object o.

Update: The formula in line 9 states the semantics of the update operation in a Web application session. It states, the user u has been able to execute the update operation, by updating the object o with o', if and only if, the session *s* relates to the user *u*, and update operation e, on object o has occurred.

Delete: Moreover, for the delete operation, as the formula in line 11 Listing webapp3 states, the user *u* has been able to execute the delete operation e, on object o, in session s, if and only if, a delete operation e on object o has occurred in the session, and the session s is related to the user *u*.

```
spec WebAppWithSession = WebAppWithSessionSig then

   for all u: User, e: Event,  o,o': Object;

5     □ (cexec(u,e,o)  ⇔ ∃ s: Session.(user(s) = u ∧
           happensIn(s,e) ∧ createOp(e,o)))

7     □ (rexec(u,e,o)  ⇔ ∃ s: Session.(user(s) = u ∧
           happensIn(s,e) ∧ readOp(e,o)))

      □ (uexec(u,e,o,o')  ⇔ ∃ s: Session.(user(s) = u ∧
           happensIn(s,e) ∧ updateOp(e,o,o')))

11    □ (dexec(u,e,o)  ⇔ ∃ s: Session.(user(s) = u ∧
           happensIn(s,e) ∧ deleteOp(e,o)))

end
```

Listing 3.4: Semantics of a Web app with sessions in first-order temporal logic

3.2 ΦDAC Syntax and Semantics

This part of the thesis describes the syntax and semantics of ΦDAC models. Here the syntax is represented in an abstract version, while the full concrete syntax definition is presented in Appendix A. The semantics of ΦDAC is defined in LTL as an extension of a Web application with sessions.

3.2.1 ΦDAC Syntax Definition

Listing 3.5 represents the syntax definition of ΦDAC, which is based on *controlled objects*, *policies*, *policy cases*, and *coverage criteria*, and its related *authorisation* management. This part of the thesis discusses these elements, and in the following sections the concept of coverage and authorisation management will be discussed.

Controlled Objects

Currently, WebDSL already supports access control on the pages and templates of the Web application. However, I do not want to force the developer to use the Φ models and existing access control models to declare two different types of access control models at the same time, so these coarse-grained elements are also supported by all Φ models, including ΦDAC (see Listing 3.5). In addition, Φ models support access control for a number of fine-grained elements, as shown below:

```
Statements

  "PhiDAC"  "{" ControlledObjs  Policies  PCases  Coverage? AMS "}"

  ControlledObjs ::= "objects" "(" Obj* ")"
  Policies ::= "policies" "(" PPred* ")"
  PCases   ::= "{" PCase* "}"
  PCase    ::= "(" PSign* ")" "->" "(" ObjOper* ")"

  Coverage ::= ...
  AMS      ::= ...

Expressions

  Obj     ::= PageObj | TempObj | BlockObj | GroupObj | EntObj | XMLObj | PropObj
  ObjOper  ::= "c" | "r" | "u" | "d" | "i" (constant)
  PolSign  ::= "+" | "-" | "?" (constant)


  PageObj  ::= "P"    "(" constant* ")"
  TempObj  ::= "T"    "(" constant* ")"
  BlockObj ::= "B"    "(" constant* ")"
  GroupObj ::= "G"    "(" constant* ")"
  EntObj   ::= "E"    "(" constant* ")"
  XMLObj   ::= "X"    "(" constant* ")"
  PropObj  ::= "Pr"   "(" PElems*   ")"
  PElems*  ::= constant "." constant | constant "." "[" constant* "]"

  PPred    ::= UserTest
             | AttTest

  AttTest  ::= UserAtt  RelOp (constant | DataAtt)
             | DataAtt  RelOp (constant | UserAtt)
             | SysAtt   RelOp constant

  UserTest ::= "Self" "==" UserId
  UserId   ::= String
  UserAtt  ::= "Self" "." constant*
  DataAtt  ::= "This" "." constant*

  RelOp    ::= ">" | "<" | "<=" | "=>" | "!="

  CovValue ::= C  (constant) | V  (variable)

Constants

  constant ::= String | Int
```

Listing 3.5: Syntax of ΦDAC

Group: A WebDSL developer can define text boxes by using a group code block, and Φ developer can use *G(GroupNames)* to declare that a set of group names needs to be covered by the access control model.

Style Blocks: WebDSL developers can use an external CSS style file with the Web applications and then use *block(blockName){...}* to style a block of WebDSL code. Φ also supports blocks as controlled objects and the developer uses *B(BlockNames)* to define a set of blocks that needs to be access controlled.

XML Hierarchies: In WebDSL, an XML hierarchy can be used within the template code. Here, the developer uses *X(NodeNames)* to declare XML node names that need to be controlled.

Data Model: Any Web application that is more than just a set of linked static pages needs a supporting data manipulation mechanism, e.g. a relational data base. The data structure is defined in a data model, which is then translated into a database type, such as tables in a relational database. The main benefits of using the data model as a part of controlled objects is, that I can define access control on the data model elements *without* considering where or by whom they are used within the application code. It is important to note, that the access control models consequently support relations, such as inheritance, between data-model components. For example, the type of the `speaker` property can be the `Person` entity. If this entity is access controlled, then the model automatically adds all the access control predicates from the `Person` entity to `speaker` s predicate. Therefore, the developer uses `E(EntityNames)` and `Pr(PropertyNames)` to define a set of entities and/or properties as controlled objects.

Policies

Here for the efficiency and readability of the access control model, I factorize the policy terms, and they are defined after the *policies* keyword. The *User Identifier* can be written by using the `Self` keyword. For example, the first policy term in Listing 3.6 for the user that is/is not "John Smith". Note that the positive or negative authorisations on these policy terms are based on the policy signs. Also, more than the original DAC, *any* property that is defined for the entity that represents the user of the system, can be used as an atomic value or a range of values to identify a set of users within a system. This helps the developers to create a group of users based on common criteria that reduces the development time and gives more flexibility to the developer. For example, in Listing 3.6, the third policy term states: *a set of users in the system that were/were not born between first of Jan 2000 and first of Jan 2010.*

Attributes. Similar to all the Φ models, ΦDAC supports *User*, *Data*, and System attributes. This part of the thesis explains these attributes. In any system there is a unique identifier (e.g., `userID`) that is used to identify each individual user. In the original DAC proposal [89], only the users' unique identifiers are associated to the controlled objects (i.e., files in the system). However, in this model *any* property that is defined for the user of the system can be used to identify a set of users. This helps the developers to create a group of users based on common criteria that reduces the development time and gives more flexibility to the developer. For example in Listing 3.6, the third policy term states: a set of users in the system that were or were not (depends on the policy signs) born on or after the first of January 2000. Also the developer can define a very specific policy term such as Self.username == "John Smith".

The developer can also use the content of the data instances as a policy term after the keyword *This*. For example, the policy term This.username == "John Smith" refers to the instances of the property username where its content is "John Smith". Note the difference between user and data contexts. In the user context the value is checked against or in favour (depends on the policy sign) of the current user. However the value in data context represents an instance of the data that is retrieved from the database and is presented within the Web application. The developer can also specify the system attribute-based policy term based on the time and/or date after the keyword *Sys*. For example, Sys.time > 9:00 means: when the time on Web application's deployed server is/is not after 9:00 a.m.

```
PhiDAC{
        objects{G(studentMarks,classes),X(address,telephoneNumber),
                  Pr(Person.password),E(marks),B(editUser,addUser)}

        policies{Self.username == "John Smith", This.username == "John Smith",
                  Self.dob> 1/1/2000, Sys.time (>= 9:00, <= 17:00)}

8       cases{ (+,?,-,+) -> (r,c,s,[r,u],i),
                (+,?,+,+) -> (r,r,e,i,u)}

11        coverage {
          objects{P(root)}
          policies{Self.username == "John Smith"}
          cases { (+) -> ([r,100],[i]),
                  (-) -> ([r,(>10,<50)])
          }
        }
}
```

Listing 3.6: ΦDAC Example

Policy Cases

The aim of policy cases is to define a set of predicates based on a set of logical states and their relation to the controlled objects and their associated operations (see Listing 3.6 Lines 8 and 9). I call them policy cases because these block of code gives the meaning to the policy terms and related set of object operations to the objects. In which the Φ can formulate a set of predicates for an object and its related operations as I discussed earlier.

Create: The keyword `c` is used to denote that the authorised users are allowed to create an instance of the controlled objects or a set of objects that are embedded within the controlled objects. For example, if the controlled object is an entity, this case controls the create operations of this entity throughout the application.

Read: The keyword `r` refers to read operations of the controlled object itself or its embedded objects (i.e., properties as sub-elements).

Update: The keyword `u` refers to update operations of the controlled object.

Delete: The keyword `d` refers to the delete operations related to the controlled object.

Secret: The keyword `s` is used for hiding the content of the object itself or its embedded objects. For example, if the controlled object is `User.username` then its instance will be hidden to the user.

Ignore: The keyword `i` states that the defined policy states of this case do not affect the predicates of the controlled object and its embedded objects.

3.2.2 Generic ΦDAC Semantics

Listing 3.7 defines the semantics of a Web application with ΦDAC as access control element. *WebAppWithPhiDAC* is an extension of *WebAppWithSessionSig* (Listing 3.3), so it has all the core and operation execution characteristics of a Web application with sessions, but with the additional ΦDAC authorisation elements. Note that the constructors of sort *Attribute* change for the different Φ models and will be included via refinement. The semantics of such Web application is constructed based on the following elements:

Attribute-based Model: ΦDAC is a discretionary- and attribute-based access control model. For this, the semantics of ΦDAC has an additional *Attribute* element that abstractly represents all the *user*, *data*, and *system* attribute-based constraints. These attributes are also part of the user authorisation during the run-time of the system, and therefore they need to be evaluated during the run-time based on a flexible predicate *evalAtt*.

Authorisation Rights: As Listing 3.7 shows (see line 8), the authorisation of the objects' data manipulation operations (i.e., *authRights*) in a Web application during the users' sessions is based on the sorts *Session*, *Attribute*, and *Event*. Authorisation rights are specified as a time-based predicate, based on the fact that the changing values of the attributes can make the authorisations both true and false at different times. For example, if an authorisation check A is based on office hours, then A is true *only* during this period of time.

System executions: In this model, I added ΦDAC as an authorisation enforcement. As Listing 3.7 (see lines 11-19) shows, there are two additional predicates evalAtt and authRights compared to the Web application with session (see Listing 3.4) that guards the executions of the data manipulation operations. The system behaviour/operation executions in Listing 3.7 are as following:

Create: As Listing 3.7 line 12 shows, a create operation e for an object o, has been executed if and only if during the user u's session, u has the required rights (i.e., discretionary- and attribute-based) to do so for the create operation, based on the user u's related attributes, and the create operation.

Read: The formula in line 14 shows the semantics of the read operations in a ΦDAC model. A read operation e for an object o, has been executed if and only if, during the users u's session, u has the required discretionary- and attribute-based rights to execute the read operation during the Web application run-time.

Update: Also as the formula in line 16 shows, the update operation e for an object o has been executed, if and only if, during the users u's session, u has the required rights (i.e., discretionary- and attribute-based) to execute the update operation; sometime during the Web application's run-time.

Delete: Listing 3.7, line 18 shows the semantics of the delete operation execution semantics. In this case, the delete operation e for an object o has been executed, if and only if, during the user u's session, u has the discretionary- and attribute-based rights, to execute the operation.

```
spec PhiDAC  = WebAppWithSessionSig then

  sorts Attribute

  flexible pred:

   evalAtt:Attribute;

   authRights:User × Attribute × Event;

  for all u: Users, e: Event, a: Attribute, o,o': Object;

   □ (cexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
      evalAtt(a) ∧ authRights (u,a,e) ∧ happensIn(s,e) ∧ createOp(e,o)))

   □ (rexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
      evalAtt(a) ∧ authRights (u,a,e) ∧ happensIn(s,e) ∧ readOp(e,o)))

   □ (uexec(u,e,o,o') ⇔ ∃ s : Session.(user(s) = u ∧
      evalAtt(a) ∧ authRights (u,a,e) ∧ happensIn(s,e) ∧ updateOp(e,o,o')))

   □ (dexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
      evalAtt(a) ∧ authRights (u,a,e) ∧ happensIn(s,e) ∧ deleteOp(e,o)))
end
```

Listing 3.7: Semantics of ΦDAC in first-order temporal logic

3.3 ΦMAC Syntax and Semantics

ΦMAC is the second type of access control model that can be defined by Φ language. Moreover ΦMAC, is a declarative and fine-grained mandatory- and attribute-based access control model. As mentioned previously in Section 2.1.1, MAC's authorisation rights for *confidentiality* and *integrity* of the data, are based on labels [130]. In this model, the labels are assigned to *both* objects (i.e., classification) and users (i.e., security clearance), and the users with the correct set of labels can use *create* and/or *read* operations on a set of objects (see 2.1.1 for details). The next following parts discuss the ΦMAC in terms of its *generic semantics*, *syntax definition*, and *example* with its *transition* to Φ*MAC's semantics*.

3.3.1 ΦMAC Syntax Definition

As Listing 3.8 shows, the ΦMAC's syntax are constructed based on the following elements:

Confidentiality or Integrity Labels: The developer can use ΦMAC model to enforce *confidentiality* and *integrity* models. Therefore the developer first can define a set of confidentiality and/or integrity labels after the *secLabels* and *intLabels* keywords. As Listing 3.8 (line 19) shows, these labels has a *one directional* flow from a dominant label (i.e., left hand side) to (i.e., `->`) a less dominant label (i.e., right hand side). For example in listing 3.9, there are four confidentiality labels (see line 3) in which the most dominant label is `TopSecret` and the label with no dominancy on any other label is `Normal`. Note that MAC and therefore ΦMAC only support *one-directional* information flow, therefore in ΦMAC, the developer cannot define multiple inheritance relation between two sets of labels.

```
Statements
  "PhiMAC" "{" CLabels ILabels ControlledObjs Policies PCases Coverage? AMS "}"
4
  CLabels          ::= "comLabels" "{" LabelElems* "}"
  ILabels          ::= "intLabels" "{" LabelElems* "}"

  ControlledObjs   ::= "objects"   "("    ObjElem* ")"
9
  Policies         ::= "policies"  "("    PPred*   ")"
  PCases           ::= "{" PCase* "}"
  PCase            ::= "(" PSign* ")" "->" "(" ObjOper* ")"
13
  Coverage         ::= ...
  AMS              ::= ...

Expressions
18
  LabelElems       ::= { constant "->"}*
  ObjElem*         ::= Obj "." "label"  (Constant)
                     | Obj "." "labels" (Constant, Constant)
  PPred            ::= AttTest | constant*
```

Listing 3.8: The ΦMAC Abstract Syntax

Controlled Objects: These are the objects that need to be controlled by ΦMAC model, that are defined between the brackets of the *objects* keyword. The type of available objects used in ΦMAC is identical to the ΦDAC model, however in ΦMAC, the confidentiality and integrity classification labels must be assigned to the objects and users of a system. So in this model the developer assigns confidentiality and/or integrity labels to the objects (see Listing 3.8 lines 18 and 19). For example, in Listing 3.9 line 6, the developer assigned the classification label `secret` (for confidentiality) to the controlled object `student marks`.

Policy terms and cases: In ΦMAC model the developer can use the user, data, and system attributes along with the confidentiality and/or integrity labels (see Listing 3.9 lines 2 and 4). The aim of using this attributes is to give the developer the flexibility to use positive and/or negative authorisation enforcement. For example in Listing 3.9 the defined policy terms `Self.username == "John Smith"` is creating an exceptional case for the user with the `username` John Smith. Policy cases in ΦMAC is constructed similar to ΦDAC model with a few differences. In terms of confidentiality or integrity classification levels, the policy sign + represents the user who has the privilege directly or indirectly (based in the defined hierarchies) to the level; and that represents the user who does not have the authorisation rights, directly or indirectly (i.e., through labels' hierarchies), to the level. Also the operations on the object can be either *read* or *write* based on the MAC definitions (see Section 2.1.1 for more details). As discussed in the literature, generally speaking ΦMAC is suitable for inflexible restrictive environments.

```
PhiMAC{

  conLabels{  TopSecret          -> Secret          -> InternallySecret -> Normal}
4
  intLabels{  TopConfidential -> Confidential -> Common              -> ExtCommon  }

  objects{  G(studentMarks).label(secret),  X(telNum).label(secret),
            Person.password.label(TopSecret),  P(marks).label(TopSecret),
            B(students).label(secret)}
10
  policies{Self.username == "John Smith", This.username == "John Smith",
          Secret,   ExtCommon,  Sys.time (>= 9:00, <= 17:00)}

  cases{  (+,-,-,?,+)   -> (,r,s,[r,c],i),
          (-,-,+,-,-)   -> ([r,c],r,s,i,c)}
15
  coverage{
    objects{P(root)}
    policies{Secret, Confidential}
    cases {  (+,?) -> ([r,100],i),
             (?,+) -> (i,  [r,(>10,<50)])
    }
  }
}
```

Listing 3.9: An example of ΦMAC model

3.3.2 Generic ΦMAC Semantics

This part discusses the generic ΦMAC semantics that is defined using LTL [51]. As Listing 3.10 shows the ΦMAC formal specification extends *WebAppWithSessionSign* formal specification (see Listing 3.3) and it is based on the following elements:

Mandatory- and Attribute-based Model: As mentioned before, ΦMAC is mandatory- and attribute-based access control model. So, the semantics of ΦMAC has *Attribute* and *Label* as two additional elements. Similar to the ΦDAC specification, the *Attribute* abstractly represents all the *user*, *data*, and *system* attribute-based constraints. Therefore these constraints need to be evaluated during run-time, based on the flexible predicate *evalAtt*. Moreover, the *Label* sorts elements abstractly represents both *confidentiality* and *integrity* labels for the *users* (i.e., User) and the *objects* (i.e., Object) of the system. There are two label related activities during the run time of the system. First, the user with the administration responsibility *assigns* a label to a user; and the target user *activates* the label after the authentication and before any allowed data manipulation. The *assignment* activity relates to the *authorisation management* mechanisms and their semantics (see Sections 3.5, 3.6, and 3.7). However, the *label activation* directly relates to the ΦMAC model and it is represented by the *labelActivation* flexible predicate (see Listing 3.10 (line 7). After label *activation* by the users of the system, the *Label* needs to be evaluated at run-time, by the *evalLabel* flexible predicate (Listing 3.10 line 8).

Authorisation Rights: In a Web application the authorisation of objects' data manipulation operations under ΦMAC access control model, during the users' sessions is based on *User*, *Attribute*, *Label* (assigned to users), and *Event*. Similar to ΦDAC predicates, authorisation rights in ΦMAC are specified as time-based predicates and they are periodically true.

System Executions: In this model the authorisation enforcement is ΦMAC-based. Therefore, to guard the executions of the data manipulation operations, as Listing 3.10 shows (lines 9-15), there are *three* additional predicate elements (i.e., *evalAtt*, *evalLabel*, and *authRights*) compared to the Web application with session (Listing 3.3). Also compare to the ΦDAC formal specification, ΦMAC specification adds the abstract notion of *Label* in its *authRights* predicate with additional *evalLabel* predicate. Also note that ΦMAC only allows *create* and *read* operations for protecting the data's *confidentiality* and *integrity*. Therefore as Listing 3.10 shows (lines 10-15) ΦMAC formal specification covers *cexec* (i.e., create execution) and *rexec* (i.e., read execution). In this model, the authorisation enforcement is ΦMAC-based. Therefore, to guard the executions of the data manipulation operations, as Listing 3.10 shows (lines 9-15), there are three additional predicate elements (i.e., evalAtt, evalLabel, and authRights) compared to the Web application with session (Listing 3.3). Also compared to the ΦDAC formal specification, ΦMAC specification adds the abstract notion of Label in its authRights predicate with additional evalLabel predicate. Also note that ΦMAC only allows create and read operations for protecting the data's confidentiality and integrity. Therefore, as Listing 3.10 shows (lines 10-15), ΦMAC formal specification covers cexec (i.e., create execution) and rexec (i.e., read execution). The descriptions of the create and read operation in ΦMAC model are as follows:

Create: As Listing 3.10 shows (cf. line 12), a create operation e for an object o has been executed if and only if, the user u has the required mandatory- and attribute-based rights, and the session s is related to the user u, to execute the create operation.

Read: In Listing 3.10 line 14, a read operation e for an object o, has been executed if and only if, the user (i.e., u) is in the session s and has the required mandatory- and attribute-based rights for the read operation; based on the user's related attributes and evaluation of the user's assigned label (i.e., security clearance).

```
spec PhiMAC = WebAppWithSessionSign then

  sorts Attribute, Label
4
  flexible preds
    evalAtt: Attribute;
    labelActivation: User × Label;
    evalLabel: Label × User × Object × Label;
    authRights: User × Attribute × Label × Event;
9
  for all j: Users, e: Event, a: Attributes, l,l': label, o: Object;

   □ (cexec(u,e,l,a,o) ⇔ ∃ s : Session.(user(s) = u ∧
      labelActivation(u,l) ∧ evalAtt(a) ∧ evalLabel(l, u, o, l') ∧
      authRights (u, a, l, e) ∧ happensIn(s,e) ∧ createOp(e,o)))

   □ (rexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
      labelActivation(u,l) ∧ evalAtt(a) ∧ evalLabel(l, u, o, l') ∧
      authRights (u, a, l, e) ∧ happensIn(s,e) ∧ readOp(e,o)))
15

end
```

Listing 3.10: Semantics of ΦMAC in first-order temporal logic

3.4 ΦRBAC Syntax and Semantics

ΦRBAC is an approach for declaratively defining and implementing a flexible, expressive, and high-level *role-* and *attribute-based* model. It enables the developer to define an access control model on fine- grained objects of the system. This part discusses the *generic semantics*, *syntax definition* with an example and its *translation* to the defined semantics.

3.4.1 Syntax Definition

This part describes the ΦRBAC abstract syntax (see Listing 3.11) and provides an example (Listing 3.12) for each part of the model. As Listing 3.11 shows, the defined ΦRBAC syntax is constructed based on the following elements:

Roles: The developer first defines roles and their cardinalities (cf. line 6), which specify the maximum number of *subjects* that may acquire the respective roles at any given time. For example in Listing 3.12 (line 4) the role *admin* has the cardinality of one.

```
Statements
2
  "PhiRBAC" "{" Roles Hierarchies SSODRels DSODRels ContObjects
                Policies PolicyCases Coverage? AMS "}"
5
  Roles       ::= "roles"     "(" roleElem* ")"
  Hierarchies ::= "hierarchy" "(" hElem     ")"
  SSODRels    ::= "ssod"      "(" sodElem   ")"
  DSODRels    ::= "dosd"      "(" sodElem   ")"
  ContObjs    ::= "objects"   "(" obj*      ")"
  Policies    ::= "(" PElem* ")"
  Coverage    ::= ...
  AMS         ::= ...
14

Expressions

  roleElem   ::= constant "(" constant ")"
  helem      ::= constant "->" constant |
                 constant "->" "(" constant* ")" |
                 "(" constant* ")" -> constant

  sodElem   ::= constant "<->" constant |
                 constant "<->" "(" constant* ")" |
                 "(" constant* ")" "<->" constant
```

Listing 3.11: Controlled Objects and related operations

Optional Relations: The developer can also define an optional role *hierarchy* after the `hierarchy` keyword (line 5). In the example (see Listing 3.12 line 5), the `advisor` role is defined as a specialisation of `teacher`. In addition the developer can define optional *static and dynamic separation of duty* constraints after the `ssod` and `dsod` keywords (lines 6 and 7). For example in Listing 3.12 (line 7), any user cannot activate the role *manager* and *admin* at the same time.

Controlled Objects: ΦRBAC supports all the mentioned (see 3.2.1) coarse- and fine-grained objects. They are defined after the `objects` keyword (line 9).

Policy Terms and their cases: Similar to ΦDAC and ΦMAC, I use a matrix-structure to specify the actual access control policy. The labels of the matrix rows and columns are given as the set of controlled objects and the different policy terms, while the entries of the matrix are given on a line-by-line basis as policy cases. These show the relation between the policy combinations and allowed operations on the respective objects. A developer can select an arbitrary number of desired policy terms following the *policies* keyword (line 10). These policy terms are either *roles*, or *user*, *data*, and *system* attributes for enforcing negative or positive authorisations; which is dependent on the logical status (i.e., +, -, ?). For example as shown in Listing 3.12, the first policy case (line 11) creates a predicate that states: the user with the activate role *teacher*, and the roles *student* and *admin* are not active.

```
PhiRBAC{

   roles{ teacher(10), admin(1), manager(1), advisor(10),student(*)}

   hierarchy{(advisor) -> (teacher)}
   ssod{(teacher,admin,advisor,manager) <-> (student)}
   dsod{(and(advisor,teacher),admin)    <-> (manager)}

  objects{G(studentMarks),X(address),Person.password,P(marks)}
  policies{teacher,student,admin,Self.username == "John Smith"}
  cases{ (+,-,-,+) -> ([r,u],r,s,[r,u]),
         (-,-,+,?) -> ([r,u],r,s,i)
      }

  coverage {
   objects{P(root),student.marks}
   policies{admin,teacher}
   cases { (+,?) -> ([r,100],[i]),
           (-,+) -> ([i],[u,(>80,<=100)])
         }
  }
}
```

Listing 3.12: A ΦRBAC Example

3.4.2 Generic Semantics

Listing 3.13 shows the semantics of the Web application with ΦRBAC as an access control element. This specification extends the *WebAppWithSessionSign* formal specification (Listing 3.3) and it is based on the following elements:

Role- and Attribute-based model: As mentioned ΦRBAC is a role- and attribute-based access control model, so this formal semantics has two additional *Role* and *Attributes* sorts elements. The *Role* represents all the defined roles within the ΦRBAC model and their optional relations (e.g., SSOD). Similar to ΦDAC and ΦMAC models, the *Attribute* element abstractly represents all the *user*, *data* and *system* attribute-based constraints. Also similarly, the flexible predicate *evalAtt* evaluates the attributes.

Role Activation: Similar to the ΦMAC model, in ΦRBAC the users with the administrative rights can assign the authorisation rights, and the target users can activate them (i.e., in this case roles). The assignment semantics related to the authorisation management system will be discussed later. The *role activation* relates to the ΦRBAC model itself, and as it can be seen in Listing 3.13 (see line 6), which is represented as a flexible predicate *roleActivation* that is based on the relations between the users (i.e., User) and the defined roles (i.e., Role) within the system.

Authorisation Rights: As Listing 3.13 shows (line 8), the authorisation rights within the Web application with ΦRBAC model is based on the users of the system (i.e., User), the required roles (i.e., Role), the related user, data, and/or system attributes (i.e., Attribute), and the data manipulation operation on the objects (i.e., Event).

```
spec WebAppWithPhiRBAC = WebAppWithSessionSig then

  sorts Attribute, Role
3
  flexible preds
5
    roleActivation : User × Role;
    evalAtt   : Attribute;
    authRights : User × Role × Attribute × Event;
9
  for all j: Users, r: role, a: Attributes, e: Event, o,o': Objects;

    □ (cexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧ roleActivation (u,r)
                           ∧ evalAtt(a) ∧ happensIn(s,e) ∧ createOp(e,o)))

    □ (rexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧ roleActivation (u,r)
                           ∧ evalAtt(a) ∧ happensIn(s,e) ∧ readOp(e,o)))

    □ (uexec(u, e, r, a, o, o') ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r) ∧ evalAtt(a) ∧ happensIn(s,e) ∧ updateOp(e,o,o')))

    □ (dexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧ roleActivation (u,r)
                           ∧ evalAtt(a) ∧ happensIn(s,e) ∧ deleteOp(e,o)))
18
end
```

Listing 3.13: Semantics of ΦRBAC in first-order temporal logic

System Execution: As Listing 3.13 shows (lines 10-17), the crud execution operation can be enforced by the ΦRBAC model. In comparison with the Web application with session and no access control, there are *two* additional predicate elements (i.e., *evalAtt* and *authRights*). Also compared to the ΦDAC and ΦMAC, the authorisation of the ΦRBAC specification supports *role-based* authorisation rights. Moreover, the ΦRBAC model is more flexible than ΦMAC as it can guard all the available execution operations and not just the *create* and *read* execution operations.

As Listing 3.13 shows (lines 10-17), the crud execution operation can be enforced by the ΦRBAC model. In comparison with the Web application with session and no access control, there are two additional predicate elements (i.e., evalAtt and authRights). Also compared to the ΦDAC and ΦMAC, the authorisation of the ΦRBAC specification supports role-based authorisation rights. Moreover, the ΦRBAC model is more flexible than ΦMAC as it can guard all the available execution operations and not just the create and read execution operations. These semantics of the create, read, update, and delete operation executions are as following:

Create: As Listing 3.13 line 11 shows, a create operation e for an object o, has been executed if and only if, during the user's related session (i.e., *s*), the user (i.e., u) has the rights to do so for the create operation, based on the user's activated roles and related attributes.

Read: Also the formula in Listing 3.13 line 13 shows, a read operation e for an object o, has beed executed, if and only if, during the user *u's* session s, the user has the required roles (i.e., by activating them) and attributes to execute the read operation.

Update: Moreover, an update operation e for an object o (i.e., formula in Line 15), has been executed, if and only if, during the related session s to the user u, the user has the role- and attribute-based rights to execute the update operation.

Delete: Furthermore, a delete operation e for an object o (see Listing 3.13 Line 17) has been executed, if and only if, during the user *u's* session s, the user has the required roles and attributes to execute the operation.

3.5 ΦDACAM Syntax and Semantics

ΦDACAM is a discretionary- and attribute-based authorisation management model that enforces a set of *discretionary-* and *attribute-based* authorisation management policies on a subset of the controlled objects defined in the Φ's access control models (i.e., ΦDAC, ΦMAC, and ΦRBAC). In this part I present its abstract syntax (see Appendix A for the syntax definition) and generic semantics based on the context of the access control.

3.5.1 Abstract Syntax

Listing 3.14 illustrates, that the ΦDACAM model is defined based on the following elements:

Objects: ΦDACAM supports all the mentioned coarse- and fine-grained objects (see 3.2.1) that are supported by the rest of the Φ-based access control models.

Policies and their cases: ΦDACAM supports all discretionary- and attribute-based policies that can be used within the ΦDAC access control model.

```
Statements
    "PhiDACAM"  "{"  ControlledObjs  Policies  PCases  Coverage?  "}"
```

Listing 3.14: ΦDACAM Abstract Syntax

Even though, syntactically, the core (i.e., objects, policies and their cases) of the ΦDAC and ΦDACAM are the same, they carry different meanings. The ΦDAC is an access control model that defines the behaviour of the *authorisation enforcement* mechanism in guarding a set of objects. However, the ΦDACAM is an authorisation management model that defines the behaviour of the *authorisation management* mechanism that enables a set of users to act as the administrators of the system and *assign* the authorisation rights for the defined controlled objects.

```
PhiDACAM{ objects{G(studentMarks), X(address), P(marks)}

   policies{Self.username == "Mary Green",
           This.username == "Mary Green", Self.dob > 1/1/2010}
  cases{(+,−,−) −> (r,r,r),
        (−,−,+) −> (s,s,r)}
  coverage{objects{P(root)}
            policies{Self.username == "Mary Green"}
            cases{ (+) −> ([r,100],i),
                   (−) −> ([r,(>10,<50)])}
           }
}
```

Listing 3.15: A ΦDACAM example

3.5.2 Generic Semantics

This part of the thesis discusses the *generic semantics* of the ΦDACAM when it is used as an authorisation management element with ΦDAC, ΦMAC, or ΦRBAC models. Therefore there is a semantical representation for each relation between ΦDACAM and an access control model. The next three parts of this thesis discuss these semantics and then for each semantic, the example translation will be given.

3.5.2.1 ΦDACAM enforcement on ΦDAC

Listing 3.16 shows the semantics of the Web application where it has ΦDAC and ΦDACAM as an access control and authorisation management elements. This specification extends the *PhiDAC* (Listing 3.7), so it has all the characteristics of a Web application with session (Listing 3.4) and ΦDAC (Listing 3.7) with additional elements.

The ΦDAC formal specification, when it is used with ΦDAC, is based on the following elements:

Discretionary- and Attribute-based model: As mentioned before, ΦDACAM is a discretionary- and attribute-based model. For this ΦDACAM uses the abstract notion of *Attribute* and *evalAtt*, from ΦDAC formal specification, for the user, data, and system attributes and their evaluation during the run-time.

Authorisation Management Rights: The ΦDACAM formal specification uses the flexible predicate *assignAuthRight*, defined in the ΦDAC formal specification (Listing 3.7), to define the managerial authorisation rights element. This flexible predicate is based on the users of the system (i.e., User), data, and system attributes (i.e., Attributes), and the operational events on the objects of the system (i.e., Event).

System Executions: Within the ΦDACAMWithPhiDAC specification the users with the authorisation rights can assign the authorisation rights to a set of users to execute the create, read, update, and update operations. The semantics of the systems executions are as follows:

```
spec PhiDACAMWithPhiDAC  = PhiDAC then

   flexible pred :

    assignAuthRights: User × Attribute × Event;
6
   for all u, u': Users, e, e': Event, a,a': Attribute, o: Object;

    □ (assigncrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
       evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
       ∧ authRights (u',a',e')))

    □ (assignrrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
       evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
       ∧ authRights (u',a',e')))

    □ (assignurightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
       evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
       ∧ authRights (u',a',e')))

    □ (assigndrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
       evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
       ∧ authRights (u',a',e')))

end
```

Listing 3.16: Semantics of a Web application with ΦDAC and ΦDACAM

Create: In Listing 3.16 line 9 shows, the user u can assign the create authorisation right to the user u', when there is a session s for the user u and the authorisation management right related to the object o is true for the user u and attribute a. The user *u*, can assign the create operation *e* to the user *u'* based on a set of discretionary- and attribute-based constraints.

Read: The formula in Listing 3.16 line 12 shows the semantics of read assign authorisation rights for user *u'* by the user *u*. This occurs sometime during the Web application run-time, in a session *s* that related to the user *u*, u has the required discretionary- and attribute-based authorisation management rights, related to the object o. Then, the user *u* assigns the read operation *e* to the user *u'*, based on the discretionary- and attribute-based constraints.

Update: The formula in line 16 shows the assignment rights semantics of the update operation e, for the user *u'* by the user *u*. The assignment of the update operation occurs, in a session *s* that is related to the user *u*, where u has the discretionary- and attribute-based authorisation management rights, related to the object o. This way the user *u* assigns the update operation *e* to the user *u'* based on the discretionary- and attribute-based rights.

Delete: The formula in line 20 presents the semantics of the assignment rights of the delete operation emphe from user u to the user *u'*. This assignment occurs, in a session *s* related to the user *u*, where u has the required rights (i.e., discretionary- and attribute-based constraints), related to the object *o* and the delete operation *e*. Therefore, based on discretionary- and attribute-based rights, the user *u* can assign the delete operation *e* to the user *u'*.

3.5.2.2 ΦDACAM enforcement on ΦMAC

Listing 3.17 shows the formal semantics of a Web application when it has ΦMAC and ΦDACAM as an access control and authorisation management elements. This application extends the *PhiMAC*, therefore it has all the characteristics of the Web application with session (Listing 3.4), and ΦMAC (Listing 3.10), with its additional elements. The formal specification of ΦDACAM when it is used by ΦMAC is as follows:

Discretionary- and Attribute-based: This specification uses the abstract notion of *Attribute* and *evalAtt* from the ΦMAC formal specification as ΦDACAM is also an attribute-based model.

```
spec PhiDACAMWithPhiMAC  = PhiMAC then

  flexible pred:

   assignAuthRights: User × Attribute × Event;

  for all u, u': Users, e, e': Event, a, a': Attribute, o: Object, l: Label;

10     □ (assigncrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
       evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
       ∧ authRights (u', a', l, e')))

13     □ (assignrrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
       evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
       ∧ authRights (u', a', l, e')))

16     □ (assignurightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
       evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
       ∧ authRights (u', a', l, e')))

19     □ (assigndrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
       evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
       ∧ authRights (u', a', l, e')))

end
```

Listing 3.17: Semantics of a Web application with ΦMAC and ΦDACAM

Authorisation Management Rights: Similar to the previous formal specification *ΦDACAMWithMAC* specification uses the flexible predicate *assignAuthRights* that gives the administrative power to a set of users (i.e., User), based on a set of attributes and authorisation assignment event.

System Executions: Within this specification the users with the assigned authorisation rights are able to create, read, update, and delete operation executions to a set of users. As Listing 3.17 shows, the semantics of the system executions are as follows:

Create: Listing 3.17 line 10 shows, the user u can assign the create authorisation right to the user u', when there is a session s for the user u and the authorisation management right related to the object o is true for the user u and attribute a. The user *u*, can assign the create operation *e* to the user *u'* based on a set of mandatory- and attribute-based constraints.

Read: The formula in line 13 shows the assignment rights semantics of the read operation e, for the user *u'* by the user *u*. The assignment of the update operation occurs, in a session *s* that is related to the user *u*, where u has the discretionary- and attribute-based authorisation management rights, related to the object o. This way the user *u* assigns the update operation *e* to the user *u'* based on mandatory- and attribute-based rights.

Update: The formula in line 16 shows, the user u can assign the update authorisation right to the user u', when there is a session s for the user u and the authorisation management rights (discretionary- and attribute-based rights) related to the object o is true for the user u and attribute a. This way, the user *u* can assign the update operation *e*, to the user *u'*, based on a set of mandatory- and attribute-based constraints.

Delete: The formula in line 19 presents the semantics of the assignment rights of the delete operation emphe from user u to the user *u'*. This assignment occurs, in a session *s* related to the user *u*, where u has the required rights (i.e., discretionary- and attribute-based constraints), related to the object *o* and the delete operation *e*. Then, the user *u* can assign the delete operation *e* to the user *u'*, based on mandatory- and attribute-based rights.

3.5.2.3 ΦDACAM enforcement on ΦRBAC

Listing 3.18 shows the formal semantics of a Web application when it has ΦDACAM and ΦRBAC as an authorisation management and access control elements. This application extends the *PhiRBAC*, therefore it has all the characteristics of the Web application with session (Listing 3.4), and ΦRBAC (Listing 3.10), with its additional elements.

The formal specification of ΦDACAM, when it is used by ΦRBAC, is as follows:

Discretionary- and Attribute-based: This specification uses the abstract notion of *Attribute* and *evalAtt* from the ΦRBAC semantics, as the ΦDACAM is also an attribute-based model.

Authorisation Management Rights: Similar to the previous formal specification ΦDACAMWithRBAC specification uses the flexible predicate *assignAuthRights* that make a set of users of the system to act as an administrator of the system and assign the operational executions to a set of users (i.e., User), based on a set of attributes and authorisation management event.

System Executions: When ΦDACAM is used with the ΦRBAC access control model, the administrators can assign the create, read, update, and delete operation executions to a set of users (i.e., u'). The semantics of assignments of the system execution of a ΦDACAM model for ΦRBAC are as follows (see Listing 3.18):

```
spec PhiDACAMWithPhiRBAC  = PhiRBAC then

  flexible pred:

   assignAuthRights: User × Attribute × Event;

  for all u, u': Users, e, e': Event, a, a': Attribute, o: Object, r: Role;

9     □ (assigncrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
        evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
        ∧ authRights (u', a', r, e')))

14    □ (assignrrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
        evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
        ∧ authRights (u', a', r, e')))

17    □ (assignurightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
        evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
        ∧ authRights (u', a', r, e')))

20    □ (assigndrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
        evalAtt(a) ∧ assignAuthRights (u,a,e) ∧ happensIn(s,e)
        ∧ authRights (u', a', r, e')))

end
```

Listing 3.18: Semantics of a Web application with ΦRBAC and ΦDACAM

Create: As the formula in line 9 shows, the user u can assign the create authorisation right to the user u', when there is a session s for the user u and the authorisation management rights (discretionary- and attribute-based rights) related to the object o is true for the user u and attribute a. This way, the user *u* can assign the update operation *e*, to the user *u'*, with a set of role- and attribute-based constraints.

Read: The formula in line 14 shows, the user u can assign the read authorisation right to the user u', when there is a session s for the user u and the authorisation management rights (discretionary- and attribute-based rights) related to the object o is true for the user u and attribute a. This way, the user *u* can assign the update operation *e*, to the user *u'*, based on a set of role- and attribute-based constraints.

Update: Line 17 shows the formula related to the assignment of the rights related to the update operation. Here, the user u can assign the create authorisation rights (i.e., role- and attribute-based) to the user u', when there is a session s for the user u and the authorisation management right related to the object o is true for the user u and attribute a.

Delete: The formula in line 20 presents the semantics of the assignment rights of the delete operation emphe from user u to the user *u'*. This assignment occurs, in a session *s* related to the user *u*, where u has the required rights (i.e., discretionary- and attribute-based constraints), related to the object *o* and the delete operation *e*. Then, based on discretionary- and attribute-based rights, the user *u* can assign the delete operation *e* to the user *u'*.

3.6 ΦMACAM Syntax and Semantics

ΦMACAM is a mandatory- and attribute-based authorisation management model that enforces a set of *mandatory-* and *attribute-based* authorisation management policies on the controlled objects defined within the Φ's access control models (i.e., ΦDAC, ΦMAC, and ΦRBAC). This part looks at syntax and generic semantics of ΦMACAM based on its access control context.

3.6.1 ΦMACAM Syntax

Similar to the ΦMAC, as Listing 3.19 and 3.20 show, the ΦMACAM syntax is defined based on the following elements:

Labels: ΦMACAM supports *confidentiality* and *integrity* labels. For example, in Listing 3.20, the confidentiality labels `TopSecret` and `Secret` are used to define the MAC-based confidentiality model based on the flow of information within these two models.

Objects: ΦMACAM supports all the mentioned coarse- and fine-grained objects (see Listing 3.19) that are supported by the rest of the Φ models prior to this.

```
Statements
 "PhiMACAM"  "{" CLabels ILabel ControlledObjs  Policies PCases Coverage? "}"
```

Listing 3.19: ΦMACAM Abstract Syntax

Policies and their cases: Identical to the ΦMAC, ΦMACAM supports *mandatory-* and *attribute-based* policy terms. Also for each policy case it allows `+`, `-`, and `?` signs for policy related logical representations, and `c`, `r`, `i`, and `s` for the object related operations.

Coverage: The coverage can be defined based on a set controlled objects, mandatory- and attribute-based polices, and a set of coverage cases. These cases are used during the validation and verification phase to check the sufficiency of the ΦMACAM with its underlying Φ-based access control model.

The syntactic core of the ΦMAC and ΦMACAM are identical however similar to the relation between ΦDAC and ΦDACAM, the ΦMACAM is used to enforce authorisation management policies and the ΦMAC is enforcing access control policies on the objects.

```
PhiMACAM{

 confLabels{TopSecret -> Secret}

 objects{G(studentMarks), X(address), P(marks)}

 policies{Self.username == "Mary Green",
          This.username == "Mary Green", Self.dob > 1/1/2010}
 cases{(+,-,-) -> (r,r,r),
       (-,-,+) -> (s,s,r)}
 coverage{objects{P(root)}
             policies{Self.username == "Mary Green"}
             cases{ (+) -> ([r,100],i),
                      (-) -> ([r,(>10,<50)])}
          }
}
```

Listing 3.20: A ΦMACAM Example

3.6.2 Generic ΦMACAM Semantics

This part discusses the generic semantics of ΦMACAM when it is used as an authorisation management element for the ΦDAC, ΦMAC, and ΦRBAC models. The next three parts discuss the semantical representation of the ΦMACAM based on the context of its access control model.

3.6.2.1 ΦMACAM enforcement with ΦDAC

Listing 3.21 shows the semantics of a Web application where it has ΦDAC and ΦMACAM as an access control and authorisation management elements. This specification extends the ΦDAC's semantics, so it has the following characterisctis:

```
spec PhiMACAMWithPDAC = PhiDAC then

  sorts Attribute, Label

  flexible preds
    evalAtt: Attribute;
    labelActivation: User × Label;
    evalLabel: Label × User × Object × Label;
    authRights: User × Attribute × Label × Event;

  for all u, u': Users, e, e': Event, a, a': Attributes, l: label, o: Object;

    □ (assigncrightexec(u,e,l,a,o) ⇔ ∃ s : Session.(user(s) = u ∧
       labelActivation(u,l) ∧ evalAtt(a) ∧ evalLabel(l, u, o, l') ∧
       assignedAuthRights (u, a, l, e) ∧ happensIn(s,e) ∧ authRights (u', a', e')
    ))

    □ (assignrrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
       labelActivation(u,l) ∧ evalAtt(a) ∧ evalLabel(l, u, o, l') ∧
       assignedAuthRights (u, a, l, e) ∧ happensIn(s,e) ∧ authRights (u', a', e')
    ))
```

Listing 3.21: Semantics of a Web application with ΦDAC and ΦMACAM

Discretionary- and attribute-based: This property extends the ΦDAC semantics, therefore it has all the characteristics of the Web application with session and ΦDAC for the discretionary- and mandatory-based access control constraints, with additional authorisation management based on the ΦMACAM's semantics.

Authorisation Management Rights: The formal specification of ΦMACAM uses the flexible predicate *assignAuthRight* to define the mandatory- and attribute-based authorisation management right elements. The flexible predicate is based on the defined labels (i.e., Label), user, data, and system attributes (i.e., Attributes), and the operational events on the objects of the system (i.e., Event).

System Executions: Within the ΦMACAMWithPDAC specification, the users with the assign authorisation rights can only create and read assignments for the users of the system with regards to their rights. The system execution of this system are as follows (i.e., lines 9-15 in Listing 3.21):

Create: As the formula in line 12 shows, the user u can create an assignment to the user u', when there is a session s, in which the labels and the attributes are held for the user u. Therefore, the user *u* can assign the mandatory- and attribute-based rights over the management of the discretionary- and attribute-based rights. This is because the ΦMACAM is defined for authorisation management of the ΦDAC model.

Delete: The line 15 shows, the user u can assign the delete authorisation right to the user u', when there is a session s for the user u and the authorisation management right related to the object o is true for the user u and attribute a. Then the user *u*, can assign the create operation *e* to the user *u'* based on a set of discretionary- and attribute-based constraints.

3.6.2.2 ΦMACAM enforcement with ΦMAC

Listing 3.22 shows the formal semantics of a Web application when it has ΦMACAM as an authorisation management and ΦMAC as an access control elements. Therefore, the semantics of this application extends the application with ΦMAC (see Listing 3.4); and so it has all the characteristics of the Web application with session (Listing 3.10), and ΦMAC (Listing 3.13) with additional authorisation management elements related to the ΦMACAM as the following:

Mandatory- and Attribute-based: This formal specification uses the abstract notion of *Label*, *Attribute*, *labelActivation*, *evalLabel* and *authRights* and *evalAtt* for mandatory- and attribute-based properties of the ΦMACAM models.

System Executions: Within the ΦMACAMWithMAC specification the users with the administrative authorisation rights can create and read the authorisation

```
spec PhiMACAMWithPMAC = PhiMAC then

 for all u, u': Users, e, e': Event, a, a': Attributes, l,l': label, o: Object;

5   □ (assigncrightexec(u,e,l,a,o) ⇔ ∃ s : Session.(user(s) = u ∧
      labelActivation(u,l) ∧ evalAtt(a) ∧ evalLabel(l, u, o, l') ∧
      assignedAuthRights (u, a, l, e) ∧ happensIn(s,e) ∧ authRights (u', a', l',
    e')))

9   □ (assignrrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
      labelActivation(u,l) ∧ evalAtt(a) ∧ evalLabel(l, u, o, l') ∧
      assignedAuthRights (u, a, l, e) ∧ happensIn(s,e) ∧ authRights (u', a', l',
    e')))
```

Listing 3.22: The semantics of a Web application with ΦMAC and ΦMACAM

assignments related to the underlying ΦMAC model. The system execution for a ΦMACAM for a ΦMAC model is as follows:

Create: Listing 3.22 shows (cf. line 5), the user u can create a new assignment for an object o to the user u' when the user's authorisation management rights are evaluated based on user's activated label and the related user, data, and system attributes. This way the user *u* can define the mandatory and attribute-based requirements for assigning the authorisation rights over the mandatory- and attribute-based policies of the ΦMAC model.

Delete: As the formula in Listing 3.22 shows (cf. line 9), the user *u* in the related session *s* can assign the delete operation e, to the user *u'*, if and only if, the required mandatory- and attribute-based constraints are held for the user u.

3.6.2.3 ΦMACAM enforcement with ΦRBAC

Listing 3.23 shows the formal semantics of a Web application when it has ΦMACAM as an authorisation management and ΦRBAC as an access control elements. Therefore, it has the semantics of a Web application with ΦRBAC (see Listing 3.13), with additional authorisation management elements as following:

Mandatory- and Attribute-based: Identical to the the two prior semantics, its formal specification of a mandatory- and attribute-based models.

System Executions: Within the ΦMACAMWithRBAC specification, the users with the administrative authorisation rights can create and/or read ΦRBAC-based authorisations to the users of the system. As Listing 3.23 shows, the system execution semantics are as follows:

Create: As the formula in line 10 states, the user u can create a new role- and attribute-based assignment for an object o to the user u' when the user's

```
spec PhiMACAMWithPRBAC = PhiRBAC then

 for all u, u': Users, e, e': Event, a, a': Attributes, l: Label, r: Role, o: Object;

10    □ (assigncrightexec(u,e,l,a,o) ⇔ ∃ s : Session.(user(s) = u ∧
        labelActivation(u,l) ∧ evalAtt(a) ∧ evalLabel(l, u, o, l') ∧
        assignedAuthRights (u, a, l, e) ∧ happensIn(s,e) ∧ authRights (u', a', r,
   e')))

14    □ (assignrrightexec(u,e,o) ⇔ ∃ s : Session.(user(s) = u ∧
        labelActivation(u,l) ∧ evalAtt(a) ∧ evalLabel(l, u, o, l') ∧
        assignedAuthRights (u, a, l, e) ∧ happensIn(s,e) ∧ authRights (u', a', r,
   e')))
```

Listing 3.23: The semantics of a Web application with ΦRBAC and ΦMACAM

authorisation management rights are evaluated based on user's activated label and related attributes.

Delete: The formula in line 14 states the semantics of the assignment of the delete operations when the ΦMACAM is the authorisation management system for the ΦRBAC access control model. Therefore, here, the user *e* can assign the delete operation to the user *e'*, if and only if, the mandatory- and attribute-based constraints are held for the user u and the session s is related to the user u.

3.7 ΦRBACAM Syntax and Semantics

ΦRBAC is a role- and attribute-based authorisation management model that enforces a set of *role-* and *attribute-based* authorisation management policies on the controlled objects defined using the Φ's access control models (i.e., ΦDAC, ΦMAC, and ΦRBAC). This part discusses its abstract syntax, an example, and generic semantics based on the access control context it is associated with.

3.7.1 ΦRBACAM Syntax

Similar to the ΦRBAC, as Listing 3.24 and 3.25 show, the ΦRBACAM syntax is defined based on the following elements:

RBAC Elements: As Listing 3.24 shows, ΦRBACAM syntax supports roles, hierarchy, and SSOD and DSOD relations.

Objects: Similar to the previous authorisation management models, ΦRBACAM supports all the mentioned coarse- and fine-grained objects (see Listing 3.24).

```
Statements

 "PhiRBACAM" "{" Roles Hierarchies SSODRels DSODRels ControlledObjs
                 Policies PCases Coverage? "}"
```

Listing 3.24: The ΦRBACAM Abstract Syntax

Policies and their cases: Identical to the ΦRBAC, ΦRBACAM supports *mandatory-* and *attribute-based* policy terms. It for each policy case it allows `+`, `-`, and `?` signs for policy related logical representations, and `c`, `r`, `u`, `d`, `i`, and `s` for the object related operations.

Coverage: Similar to the prior Φ models, ΦRBACAM could also contain the coverage cases. The coverage cases are checked during the validation and verification phase, to check the sufficiency of the ΦRBACAM with its underlying Φ-based access control model.

```
PhiRBACAM

   roles{ teacher(10), admin(1), manager(1), advisor(10),student(*)}

   hierarchy{(advisor) -> (teacher)}
   ssod{(teacher,admin,advisor,manager) <-> (student)}
   dsod{(and(advisor,teacher),admin)    <-> (manager)}

  objects{G(studentMarks),X(address),Person.password,P(marks)}
  policies{teacher,student,admin,Self.username == "John Smith"}
  cases{ (+,-,-,+) -> ([r,u],r,s,[r,u]),
         (-,-,+,?) -> ([r,u],r,s,i)
      }

  coverage {
   objects{P(root),student.marks}
   policies{admin,teacher}
   cases { (+,?) -> ([r,100],[i]),
            (-,+) -> ([i],[u,(>80,<=100)])
         }
  }
```

Listing 3.25: A ΦRBACAM Example

The syntactic core of the ΦRBACAM is identical to the ΦRBAC model, however the ΦRBACAM is used to enforce role- and attribute-based authorisation management policies, but ΦRBAC is used to enforce role- and attribute-based access control policies on the controlled objects of the system.

3.7.2 Generic Semantics

This part discusses the generic semantics of ΦRBACAM when it is used as the authorisation management element for the Φ-based access control models (i.e., ΦDAC, ΦMAC, and ΦRBAC). The next three parts discuss the semantical representation of the ΦRBACAM based on the context of its access control model.

3.7.2.1 ΦRBACAM enforcement with ΦDAC

Listing 3.26 shows the semantics of a Web Application where it has ΦDAC and ΦRBACAM as an access control and authorisation management elements. This specification extends the ΦDAC's semantics, so it has the following characteristics:

```
spec PhiRBACAMWithPDAC = PhiDAC then
 sorts Attribute , Role

  flexible preds

    roleActivation : User × Role;
    evalAtt  : Attribute;
    authRights : User × Role × Attribute × Event;

  for all u, u': Users, r: role, a, a': Attributes, e: Event, o: Objects;

      □ (assigncrightexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r)                 ∧ evalAtt(a) ∧ happensIn(s,e) ∧
    authRights(u', a', e',o) ))

      □ (assignrrightexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r)                 ∧ evalAtt(a) ∧ happensIn(s,e) ∧
    authRights(u', a', e',o)))

      □ (assignurightexec(u, e, r, a, o, o') ⇔ ∃ s : Session.(user(s) = u ∧
     roleActivation (u,r) ∧ evalAtt(a) ∧ happensIn(s,e) ∧ authRights(u', a', e',o
    )))

      □ (assigndrightexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r)                 ∧ evalAtt(a) ∧ happensIn(s,e) ∧
    authRights(u', a', e',o)))

end
```

Listing 3.26: Semantics of a Web application with ΦDAC and ΦRBACAM

Access Control Rights: This property extends the ΦDAC semantics, therefore it has all the characteristics of the Web application with session and ΦDAC for the discretionary- and mandatory-, with additional authorisation management behaviour based on the ΦRBACAM's semantics.

Authorisation Management Rights: The formal specification of ΦRBACAM uses the flexible predicate *assignAuthRight* to define the role- and attribute-based authorisation management right elements. The flexible predicate is based on the defined roles (i.e., Label), their relations (i,e., hierarchy, SSOD, and DSOD), user, data, and system attributes (i.e., Attributes), and the operational events on the objects of the system (i.e., Event).

System Executions: Within the ΦRBACAMWithPDAC specification, the users with the assign authorisation rights can create, read, update, and delete access related assignments for the users of the system with regards to their rights. The semantics of the system execution of the ΦRBACAM for a ΦDAC model is as follows (see Listing 3.26):

Create: Line 12 shows the formula related to the assignment of the rights related to the create operation. Here, the user u can assign the create authorisation rights (i.e., discretionary- and attribute-based) to the user u', when there is a session s for the user u and the authorisation management right related to the object o are held (i.e., role- and attribute-based) for the user u and attribute a.

Read: Line 16 states the semantics of the assignment rights of the read operation. Here, the user *u* in the related session *s* can assign the discretionary- and attribute-based constraints to the user *u'*, if and only if, the role- and attribute-based constraints are held for the user u.

Update: As shown in the formula in line 20, the user u can update an assignment to the user u', when there is a session s, in which the required role was assigned and activated, and the attributes (i.e., user, data, or system) are held for the user u.

Delete: The line 24 states the semantics of the assignment of the rights to the delete operation. So here, the user *u* can assign the discretionary- and attribute-based rights to the user *u'*, if and only if, the role- and attribute-based authorisation rights are held in the session *s* related to the user *u*.

3.7.2.2 ΦRBACAM enforcement with ΦMAC

Listing 3.27 shows the formal semantics of a Web application when it has ΦRBACAM as an authorisation management and ΦMAC as an access control elements. Therefore, the semantics of this application extends the application with ΦMAC (see Listing 3.4); and so it has all the characteristics of the Web application with session (Listing 3.10), and ΦMAC (Listing 3.13) with additional authorisation management elements related to the ΦRBACAM as the following:

Role- and Attribute-based: This formal specification uses the abstract notion of *Role*, *Attribute*, *roleActivation*, *evalrole* and *authRights* and *evalAtt* for role- and attribute-based properties of the ΦRBACAM models.

System Executions: Within the ΦRBACAMWithMAC specification the users with the administrative authorisation rights can create, read, update, and delete the authorisation assignments related to the underlying ΦMAC model. As Listing 3.27 shows, the semantics of the system executions of a ΦRBACAM over a ΦMAC model are as follows:

Create: As the formula in line 12 shows, the user u can create a new assignment for an object o to the user u' when the user's authorisation management rights are evaluated based on user's activated role and related user, data, and system attributes.

```
spec PhiRBACAMWithPMAC = PhiMAC then
 sorts Attribute , Role
3
  flexible preds
5
    roleActivation : User × Role;
    evalAtt   : Attribute;
    authRights : User × Role × Attribute × Event;
9
  for all u, u': Users, l: label,  r: role, a, a': Attributes, e: Event, o: Objects;

12     □ (assigncrightexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r)              ∧ evalAtt(a) ∧ happensIn(s,e) ∧
    authRights(u', a', l, e',o) ))

16     □ (assignrrightexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r)              ∧ evalAtt(a) ∧ happensIn(s,e) ∧
    authRights(u', a', l, e',o)))

20     □ (assignurightexec(u, e, r, a, o, o') ⇔ ∃ s : Session.(user(s) = u ∧
     roleActivation (u,r) ∧ evalAtt(a) ∧ happensIn(s,e) ∧
     authRights(u', a', l, e',o)))

24     □ (assigndrightexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r) ∧ evalAtt(a) ∧
     happensIn(s,e) ∧ authRights(u', a', l, e',o)))

end
```

Listing 3.27: The semantics of a Web application with ΦMAC and ΦRBACAM

Read: The line 16 states the semantics of the assignment of the rights to the read operation. Here, the user u can assign the mandatory- and attribute-based rights to the user *u'*, if and only if, the role- and attribute-based authorisation rights are held in the session s related to the user u.

Update: Furthermore, the formula in line 20 states the semantics of the assignment rights of the update rights. The user u can assign the mandatory- and attribute-based rights to the user *u'*, if and only if, the session s is related to the user u and role- and attribute-based constraints are held for the user u.

Delete: The formula in line 24 states the semantics of the delete assignment rights. Here, the user u can assign the delete authorisation rights to the user *u'*, if and only if, the role- and attribute-based authorisation management constraints are held for the user u and the session s is related to the user u.

3.7.2.3 ΦRBACAM enforcement with ΦRBAC

Listing 3.28 shows the formal semantics of a Web application when it has ΦRBACAM as an authorisation management and ΦRBAC as an access control elements. Therefore, the semantics of this application extends the application with ΦRBAC (see Listing 3.13); and so it has all the characteristics of the Web application with session (Listing 3.4), with additional authorisation management elements related to the ΦRBACAM as the following:

Role- and Attribute-based: This formal specification uses the abstract notion of *Role*, *Attribute*, *roleActivation*, *evalrole* and *authRights* and *evalAtt* for role- and attribute-based properties of the ΦRBACAM models.

```
spec PhiRBACAMWithPRBAC = PhiRBAC then

  for all u, u': Users, r, r': role, a, a': Attributes, e: Event, o: Objects;

12     □ (assigncrightexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r)              ∧ evalAtt(a) ∧ happensIn(s,e) ∧
    authRights(u', a', r', e',o) ))

16     □ (assignrrightexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r)              ∧ evalAtt(a) ∧ happensIn(s,e) ∧
    authRights(u', a', r', e',o)))

20     □ (assignurightexec(u, e, r, a, o, o') ⇔ ∃ s : Session.(user(s) = u ∧
     roleActivation (u,r) ∧ evalAtt(a) ∧ happensIn(s,e) ∧ authRights(u', a', r',
    e',o)))

24     □ (assigndrightexec(u, e, r, a, o) ⇔ ∃ s : Session.(user(s) = u ∧
    roleActivation (u,r)              ∧ evalAtt(a) ∧ happensIn(s,e) ∧
    authRights(u', a', r', e',o)))

end
```

Listing 3.28: The semantics of a Web application with ΦRBAC and ΦRBACAM

System Executions: Within the ΦRBACAMWithRBAC specification the users with the administrative authorisation rights can create, read, update, and delete the authorisation assignments related to the underlying Φ'RBAC model. The semantical formulas of the system execution of the authorisation management rights of the ΦRBACAM over ΦRBAC are as follows:

Create: As the formula shows (see line 12 in Listing 3.28), the user u can create a new assignment for an object o to the user u' when the user's authorisation management rights are evaluated based on user's activated role and related user, data, and system attributes.

Read: The formula in line 16 shows the semantics of the authorisation management assignments of the read operation e, from user *u* to *u'*. Here, the user *u* can assign the read operation *e* to a user *u'*, when the role- and attribute-based constraints are held, and the session *s* is related to the user *u*.

Update: In line 20, the formula states the semantics of the authorisation management assignment of the update operation *e*, from user *u* to a user *u'*. Here, the user *u* and the related session *s* can assign the update operation rights to the user *u'*, if and only if, the user *u* has the role- and attribute-based rights.

Delete: The formula in line 24 represents the semantics of the authorisation management rights of the delete operation *e* from user *e* to a user *u'*. Here, the user *e* can assign the role- and attribute-based rights to the user *u'*, if and only if, the session s is related to the user u, and the role- and attribute-based rights are held for the user u.

3.8 Summary and Conclusion

This part of the thesis describes what has been achieved and gives a comparison between defining the semantics of security properties and other different logical formalisms. This chapter, first highlighted the key points which were considered in designing of the language. Then, it used an abstract version of the syntax, and linear temporal logic for discussing the syntax and semantics of Φ-based access control models (i.e., ΦDAC, ΦMAC, and ΦRBAC), and Φ-based authorisation management models (i.e., ΦDACAM, ΦMACAM, and ΦRBACAM) with respect to their underlying Φ-based access control model; within their environment, a Web application with authentication mechanism. The achievement of this chapter lays on the fact that both the syntax and semantics of a declaration and fine-grained policy languages, were defined and explained. This lead to the simple and clear semantical description of Φ-based models. LTL is used to define the Web applications with authentication mechanism. Then on the top of the Φ-based access control, the semantics of Φ-based authorisation management models were defined. These semantical description explains discretionary-, mandatory-, and role-based access and authorisation management models with respect to data manipulation executions and their elements. Therefore, these defined semantics of the Φ language can be used as the basis of the future research in the domain of software security.

As Martín Abadi pointed out in [26], logic is helpful in designing and understanding the security properties of an access control and its application. Different types of logical formalisms can be used to define the security properties and their applications [24, 23, 25]. In this research, I used LTL to define a simple straight forward semantical definition of security properties of Φ-based models, compared to second-order modal Propositional Logic [77, 111], Dependency Core Calculus [41, 27] (CDD), and a logical program, Binder [67]. Propositional Logic can be used to define security predicates [26, 77, 111], by defining a set of axioms based on Hilbert system [28] as an example [26]. Compared to LTL, using second-order propositional temporal has one main drawback: timelessness. Therefore, using LTL is a better choice, as I can model Φ's attribute properties, or system behaviours. For example, a system when a property becomes true and stays true throughout the system run-time (e.g., create operation in a ΦDAC model). CDD is used to define a logical system, by a set of typing rules, distinct type variables and their types [27]. Moreover, CCD can be used as a type system and logic to define a set of logical typing rules that describe the security properties of an access control model [41, 27]. It is used for language-based authorisation [75] , and to define information flow control dependency [26]. In comparison with LTL, the main drawback of CDD in defining security policies is that, the policies are defined as black or white [26], and therefore, for defining a consistent security policy, LTL is a better candidate; as I can define the notion of *'sometimes true/false'* for the system behaviour. Binder [2] is an alternative approach for defining security properties. Binder uses a set of prolog-style logical rules, to describe the predicates related to access control policies and their context. Even though it has some good features such as, each formula is relative to its context [2, 26], it does not support *import* capabilities. In this chapter, I used import capabilities of LTL to define models based on a set of other models' specifications (e.g., Defining WebAppWithSession based on WebApp's specification). So compared to Binder, LTL is a better candidate to clearly define security policies.

Chapter 4

Compilation of Φ Models

4.1 Introduction

The practical outcome of this research is the policy language Φ. As mentioned in Chapter 3, with Φ the developer can declaratively define three types of access and authorisation management models for a Web application. Φ is implemented as a DSL and added to the compiler of the language WebDSL. WebDSL is used to generate Web applications, therefore during its compile time, Φ generates WebDSL code (e.g., predicates, links to authorisation management page, etc.) and then WebDSL compiler generates these generated code with the other written Web application code to its target language (e.g., XML, HTML, CSS, JavaScript, etc.). The Φ *architecture* consists of a number of constructed steps that represent the Φ's compiler pipeline, which is responsible for *validating*, *verifying*, and then *transforming* the Φ *m*odels into their mechanisms. The *mechanisms* are the generated access controls and their authorisation management elements that guard the controlled objects within the target Web application; and provide an *administrative* tool to the authorised users. This chapter discusses the Φ architecture in the following three parts:

Validation and Verification Phase: In this research the validation and verification definition based is on the Bohem's definition [49]. Validation is to check the correctness of the semantical relation between the defined model and its specifications; and validation is to check if the defined software code is executable or not. Validating and verifying the access and authorisation management models on their own is essential but not enough. Any generated element is defined to cover a set of objects and their content within the Web application code. As a result, the validating and verifying phase have to take the target application, with the generated access control and authorisation management elements into consideration. The main aim of this phase to discover all the possible *semantical* errors and warnings that could occur during the defining the Φ models. Moreover this phase is constructed based on two principles. First, it should provide a *cheap* mechanism to validate and verify the models. For this, this phase's steps constructed based on their *importance* and *cheapness*. To find different errors and warnings, the Φ compiler needs to validate and verify the semantics of Φ models based on different steps. However, the number of sub-checks in each validation and verification step is different. Here, the cheapness means: The less number of sub-checks and time is needed to validate and verify the defined Φ-based models. Therefore, the errors are given to the developer as soon as possible. Second, for the overall efficiency of the compiler, this phase *must* check for all the errors and warnings *before* the transformation phase, even though in some steps it deals with the location of the generated code. For this, Φ takes the advantage of its object-centric approach and provides a set of solutions to make it possible, for Φ compiler, to go through this phase before the transformation phase during the compilation of the Φ models.

Transformation Phase: After the validating and verifying the defined model within themselves and their target application, Φ transforms the declarative and fine-grained models into different types of elements (e.g., data models, predicates, etc.), and then weaves them through the Web application code. This phase enables the developers to define a set of access control and their authorisation management *models only* by using the Φ syntax, and any elements related to their *mechanisms* are handled *entirely* by the Φ generator. For example, if the developer wants to use the original WebDSL's coarse-grained access control implementation, needs to define a set of data models such as `Role` or extend the entity that represent the user to hold the activated roles. Moreover, the user needs to implement all the pages and their navigations related to the access control models (e.g., `roleActivation` page). Furthermore, in case of RBAC- or MAC-based models, the user needs to change the workflow of the Web application by modifying the authentication template, so the users will redirect to their role or label activation page after their authentication.

Φ Compiler: This compiler[1] is implemented, by using the syntax definition formalism [90], and Stratego/XT [50]. Then it is added to the WebDSL compiler as an add-on DSL that is in attached to the list of DSLs (e.g., Ajax, HQL, etc.) that create the WebDSL language as a whole. This approach leads to separation of concerns between the Φ language with the other aspects of the WebDSL language.

This chapter discusses the validating and verifying of Φ models in Section 4.2, their transformations in Section 4.3, and how the Φ compiler got implemented by using the syntax definition formalism (SDF) and Stratego/XT; and attached to the WebDSL in Section 4.4. At last in Section 4.5 provides the summary and conclusion of this chapter.

[1]Available at philang.org/repos/compiler

4.2 Validation and Verification Phase

A number of studies [126, 120, 122] highlighted the fact that developing an access control mechanism is error-prone and therefore need to be validated and verified. Unlike the prior approaches, Φ's architecture reflects the fact that *correctness* and *completeness* of the Φ models on their own is not enough and the target application *must* be considered as well based on the defined models. Even partially doing so, can result in the application code that is compilable (i.e., due to the correctness of the generator and the generated code), but has a number of security holes, that are *not fixable* after the application deployment. This leads to high testing and maintenance costs after the deployment of the application. It is ideal to give a *full guarantee* to the developer for the defined access control model and its target Web application before the deployment phase. Φ's validating and verifying phase is constructed based on the following principles:

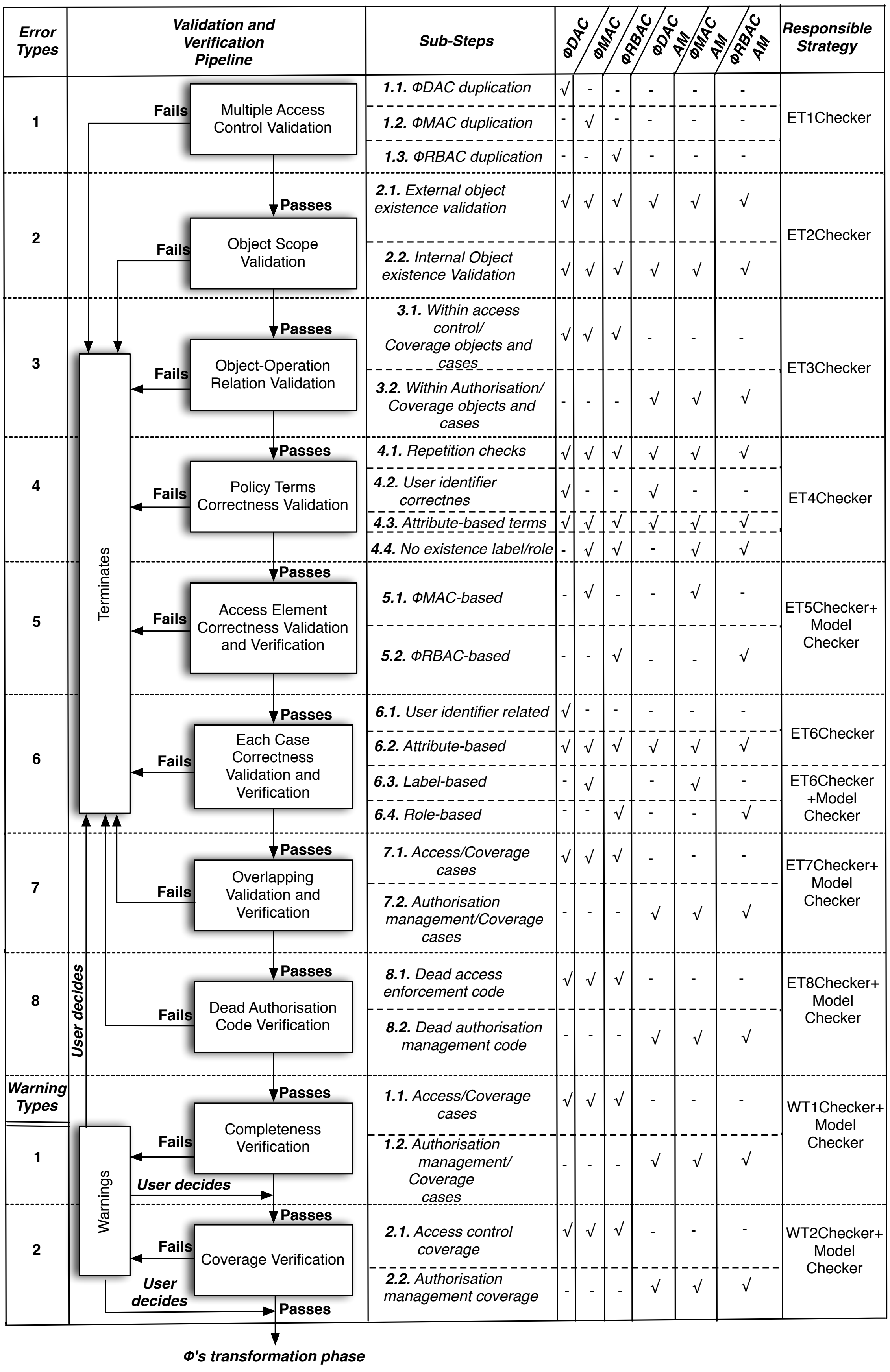
Discover All Semantical Issues: Within this step, Φ is able to deal with any possible error that can occur based on the defined Φ-based models. So the first aim of Φ compiler is to report *any* error or warning to the developer as soon as possible.

Sorted Steps: The Φ's verification and validation phase consists of number of steps that are sorted based on their importance and cheapness. This phase consists of *ten* consecutive white-box validating and/or verifying steps (See Figure 4.1). The first eight cover the eight types errors that each gives the error message and automatically triggers the compilation termination. The last two steps are create a set of warning messages, that give the developer the chance to *manually* trigger the termination or continuation of the compilation.

Φ Pipeline: Within the Φ compiler the validation and verification phase occurs before the transformation phase, even though in some steps the validation and verification needs to deal with the behaviour of the generated code fragments within one another (See Section 4.2.1.8). Therefore overall this architecture leads to *cheap* error handling and transformation phases, within the Φ compiler.

Use of Model Checker: A model checker checks for satisfiability of logical formulas based on a set of theories [35]. So, the semantics of a written program can be checked by a model checker, when the program is transformed into a logical formula and then model checker checks if the model is satisfiable. Here, I use Z3 [62] as a model checker that checks the satisfiability of first order logic formulas. Z3 is used as a part of the Φ compiler element, because it is open source and research-based.

The following two parts discuss these steps in terms of error and warning handling.



Error Types	Validation and Verification Pipeline	Sub-Steps	ΦDAC	ΦMAC	ΦRBAC	ΦDAC AM	ΦMAC AM	ΦRBAC AM	Responsible Strategy
1	Multiple Access Control Validation	***1.1.*** *ΦDAC duplication*	√	-	-	-	-	-	ET1Checker
		1.2. *ΦMAC duplication*	-	√	-	-	-	-	
		1.3. *ΦRBAC duplication*	-	-	√	-	-	-	
2	Object Scope Validation	***2.1.*** *External object existence validation*	√	√	√	√	√	√	ET2Checker
		2.2. *Internal Object existence Validation*	√	√	√	√	√	√	
3	Object-Operation Relation Validation	***3.1.*** *Within access control/ Coverage objects and cases*	√	√	√	-	-	-	ET3Checker
		3.2. *Within Authorisation/ Coverage objects and cases*	-	-	-	√	√	√	
4	Policy Terms Correctness Validation	***4.1.*** *Repetition checks*	√	√	√	√	√	√	ET4Checker
		4.2. *User identifier correctnes*	√	-	-	√	-	-	
		4.3. *Attribute-based terms*	√	√	√	√	√	√	
		4.4. *No existence label/role*	-	√	√	-	√	√	
5	Access Element Correctness Validation and Verification	***5.1.*** *ΦMAC-based*	-	√	-	-	√	-	ET5Checker+ Model Checker
		5.2. *ΦRBAC-based*	-	-	√	-	-	√	
6	Each Case Correctness Validation and Verification	***6.1.*** *User identifier related*	√	-	-	-	-	-	ET6Checker
		6.2. *Attribute-based*	√	√	√	√	√	√	
		6.3. *Label-based*	-	√		-	√	-	ET6Checker +Model Checker
		6.4. *Role-based*	-	-	√	-	-	√	
7	Overlapping Validation and Verification	***7.1.*** *Access/Coverage cases*	√	√	√	-	-	-	ET7Checker+ Model Checker
		7.2. *Authorisation management/Coverage cases*	-	-	-	√	√	√	
8	Dead Authorisation Code Verification	***8.1.*** *Dead access enforcement code*	√	√	√	-	-	-	ET8Checker+ Model Checker
		8.2. *Dead authorisation management code*	-	-	-	√	√	√	
Warning Types									
1	Completeness Verification	***1.1.*** *Access/Coverage cases*	√	√	√	-	-	-	WT1Checker+ Model Checker
		1.2. *Authorisation management/ Coverage cases*	-	-	-	√	√	√	
2	Coverage Verification	***2.1.*** *Access control coverage*	√	√	√	-	-	-	WT2Checker+ Model Checker
		2.2. *Authorisation management coverage*	-	-	-	√	√	√	

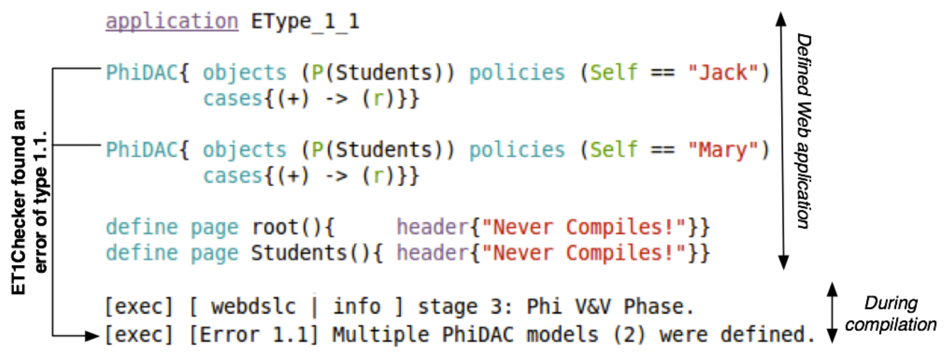
Figure 4.1: Φ's Validation and Verification Phase

4.2.1 Error Handling

This part of the thesis discusses the eight consecutive error handling steps, their sub-steps, their involved strategy within the Φ compiler, and how they handle these errors.

4.2.1.1 Access Control Multiplicity Validation

As mentioned before Φ enables the developers to define ΦDAC, ΦMAC, and/or ΦRBAC in a single Web application. So the first validation check is based on the number of defined access control models. In case of any type of access control multiplication, an error message will be given, and the Φ compilation is terminated. For this, the uppermost strategy `ET1Checker` and its relates strategies, handle these type of errors by checking the application's abstract syntax tree (AST) and giving the corresponding error if necessary (shown in Figure 4.2). Note that the developer is not also allowed to define different types of authorisation management model for a single access control model; however this error (i.e., authorisation management multiplicity) is handled by the parser, because it is against the Φ's syntax definition (see Appendix A). For example as shown in Figure 4.2, the developer wrongfully defined two ΦDAC models and therefore Φ gives the error type one to the developer and terminates the compilation.



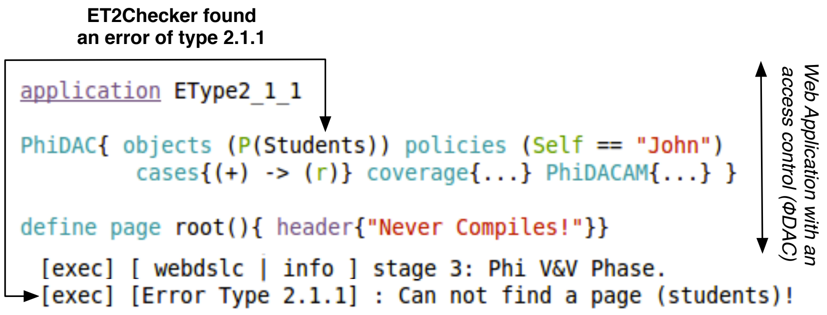
```
application EType_1_1

PhiDAC{ objects (P(Students)) policies (Self == "Jack")
        cases{(+) -> (r)}}

PhiDAC{ objects (P(Students)) policies (Self == "Mary")
        cases{(+) -> (r)}}

define page root(){     header{"Never Compiles!"}}
define page Students(){ header{"Never Compiles!"}}

[exec] [ webdslc | info ] stage 3: Phi V&V Phase.
[exec] [Error 1.1] Multiple PhiDAC models (2) were defined.
```

Figure 4.2: Multiple ΦDAC definitions for a Web application led to an error.

4.2.1.2 Object Scope Validation

This part of the thesis explains the term object scope validation (OSV) and its usage within the Φ compiler. As mentioned before Φ is mainly a data centric approach that enforces different types of authorisations on the set of objects. So within the Φ models, the used objects need to be checked with respect to the defined application and other parts of the model (e.g., coverage within an access control model). The following, are two main types of OSV-based errors that are handled through uppermost strategy `ET2Checker` within the Φ compiler (See Figure 4.1).

1. ***Out of Scope***: Within this step, the used objects within all the *access control* models (i.e., ΦDAC, ΦMAC, and ΦRBAC), are checked against the defined elements of the Web application. So the *controlled objects* within an access control model are checked against the defined objects (e.g., entities, properties, etc.) within the Web application. For example, as Figure 4.3 shows, the defined ΦDAC uses an object (i.e., page `Students`) that does not exist within the Web application code, and as a result, the strategy `ET2Checker` gives an error and terminates the compilation.

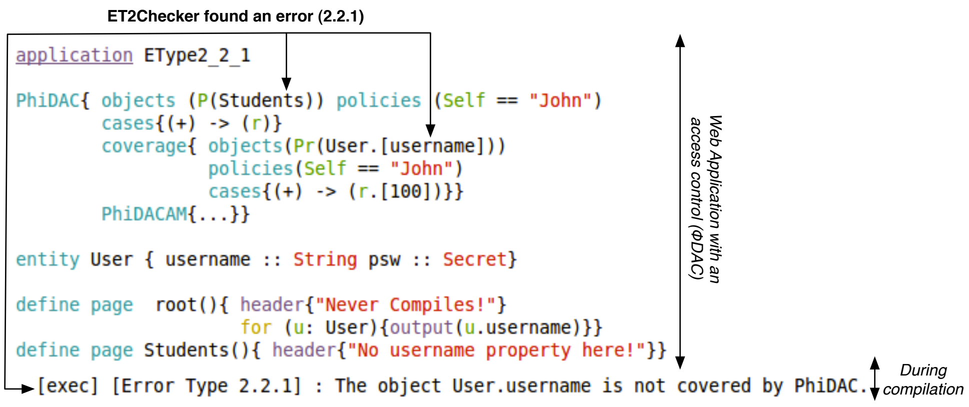

ET2Checker found an error of type 2.1.1
Web Application with an access control (ΦDAC)

```
application EType2_1_1

PhiDAC{ objects (P(Students)) policies (Self == "John")
        cases{(+) -> (r)} coverage{...} PhiDACAM{...} }

define page root(){ header{"Never Compiles!"}}
 [exec] [ webdslc | info ] stage 3: Phi V&V Phase.
 [exec] [Error Type 2.1.1] : Can not find a page (students)!
```

Figure 4.3: The used object, `Students` page does not exist within the Web application.

Note that Φ only validates the controlled objects within the access control models because the other objects used in the access control's coverage, authorisation management and its coverage are the *subset* of the controlled objects within the access control models. Therefore their objects need to be validated *only* against the controlled objects of the access control model, that they are involved with (next step).

2. ***Shared Scopes:*** In this step, for each access control model *three* consecutive steps are validating the objects used within *coverage*, *authorisation management model* and its *coverage* against the access control's controlled objects. Therefore if there is a set of objects that are not *directly* or *indirectly* part of the access control model, the strategy `ET2Checker` terminates the compilation and give an error to the developer. For example in Figure 4.2.1.2 the developer used the property `User.username` as a coverage object, that does not part of the ΦDAC's controlled objects or exists within the content of the page `Students`.


ET2Checker found an error (2.2.1)
Web Application with an access control (ΦDAC)
During compilation

```
application EType2_2_1

PhiDAC{ objects (P(Students)) policies (Self == "John")
        cases{(+) -> (r)}
        coverage{ objects(Pr(User.[username]))
                  policies(Self == "John")
                  cases{(+) -> (r.[100])}}
        PhiDACAM{...}}

entity User { username :: String psw :: Secret}

define page  root(){ header{"Never Compiles!"}
                     for (u: User){output(u.username)}}
define page Students(){ header{"No username property here!"}}
 [exec] [Error Type 2.2.1] : The object User.username is not covered by PhiDAC.
```

4.2.1.3 Object-Operation Relation Validation

As mentioned before, Φ supports access control for *pages*, *templates*, *XML nodes*, *Groups*, *Blocks*, and *data model* (i.e., entities and their properties) elements. Moreover, the developer defines an *operational relation* between these objects with a set of operations, namely, *create*, *read*, *update*, *delete*, *secret*, and *ignore*. However, the data manipulation operations (i.e., *create*, *update*, and *delete*) are solely related to the storable data (i.e., data model elements). Therefore, the Φ compiler checks if any data manipulating operation is used for a non-storable objects. If so, the Φ terminates the compilation and gives an error to the developer as soon as it find this type of error.

These types of errors are handled by the `ET3Checker` strategy and as Figure 4.4 shows, it terminates the compilation when it finds a non-data model object (i.e., page *Students*), that is used with a data manipulative operation (i.e., *delete*). As Figure 4.1 shows, there are two main types of these errors based on the location of the error within the Φ-based defined models. The first, relates to the access control model (i.e., *controlled objects* and *coverage objects* and their *cases*); and second it relates to the authorisation management part of the model (i.e., *management objects*, *management coverage objects* and their *cases*). Please refer to Appendix B to see all the examples related to the sub-types of these errors.

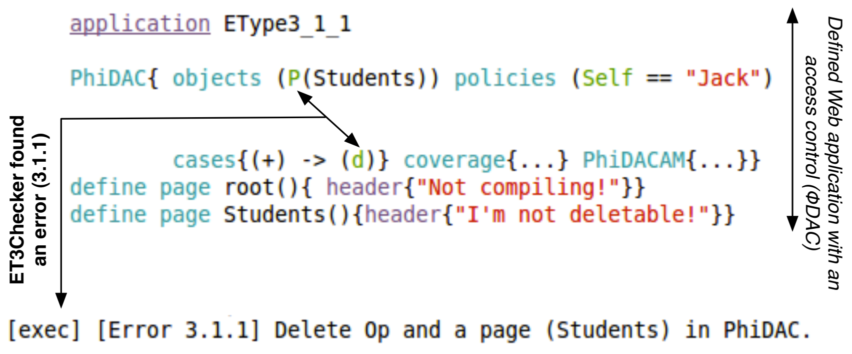

```
application EType3_1_1

PhiDAC{ objects (P(Students)) policies (Self == "Jack")

        cases{(+) -> (d)} coverage{...} PhiDACAM{...}}
define page root(){ header{"Not compiling!"}}
define page Students(){header{"I'm not deletable!"}}

[exec] [Error 3.1.1] Delete Op and a page (Students) in PhiDAC.
```
ET3Checker found an error (3.1.1)
Defined Web application with an access control (ΦDAC)

Figure 4.4: An error based on the relation between an object and its operation.

4.2.1.4 Policy Term Correctness

This step checks the *correctness* of each defined policy term with respect to itself and the other policy terms defined in Φ models. For this as Figure 4.1 shows, Φ compiler uses the `ET4Checker` to handle the following errors:

1. ***Out of Context Validation:*** First, Φ checks if there is any policy term that does not match the context of the defined model. For this, Φ checks these errors based on the following consecutive steps:

- *Access and Authorisation Management Models:* In this step, each defined policy term within the Φ access and control models is validated against the context of the model. For the ΦDAC and ΦDACAM there must not be any defined *label* or *role*; if so, the error of type *4.1.1* (in case of ΦDAC) or *4.1.4* (in case of ΦDACAM) will be given to the developer and the compilation gets terminated. Moreover, for the ΦMAC and ΦMACAM there *must not* be any label, as a policy term, which is not defined as a *confidentiality* or *integrity* labels (i.e., errors of type *4.1.2* and *4.1.5*). Furthermore, for the ΦRBAC and ΦRBACAM models, there *must not* be a role, as a policy term, outside the defined roles, in this case as the Figure 4.5 shows, an error is given to the developer, as the role `writer` is not initiated as a role. Note that these errors also cover the cases in which the developer use a role for ΦMAC and ΦMACAM, or a label for the ΦRBAC or ΦRBACAM models; as within the defined Φ's syntax definition (see Appendix A), labels and roles are both defined as *identifiers*. Therefore, for example, if the developer uses a label as a role within the policies of an ΦRBAC model, Φ cover this error and treats it as an undefined role.

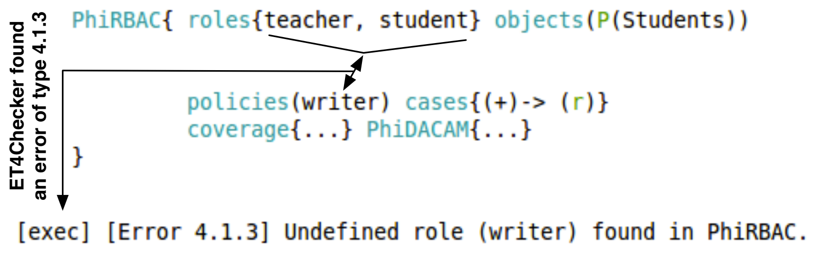

```
PhiRBAC{ roles{teacher, student} objects(P(Students))

         policies(writer) cases{(+)-> (r)}
         coverage{...} PhiDACAM{...}
}
[exec] [Error 4.1.3] Undefined role (writer) found in PhiRBAC.
```

Figure 4.5: An error based on used undefined roles as a policy term

- *Coverage-based Policies:* At this point, Φ (through uppermost `ET4Checker` strategy) validates if the coverage policies are matching the defined policies within the access control and authorisation management model. For this as the Figure 4.6 shows, the `ET4Checker` creates an error and terminates the compilation, based on the fact that the policy term, `admin` which does not match the defined roles within the ΦDACAM model.

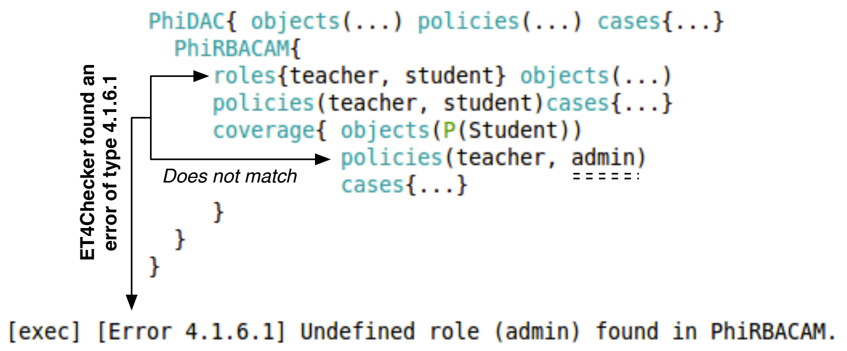
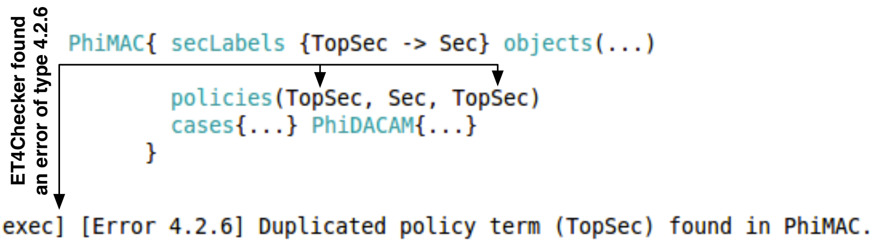

```
PhiDAC{ objects(...) policies(...) cases{...}
  PhiRBACAM{
    roles{teacher, student} objects(...)
    policies(teacher, student)cases{...}
    coverage{ objects(P(Student))
              policies(teacher, admin)
              cases{...}
    }
  }
}
[exec] [Error 4.1.6.1] Undefined role (admin) found in PhiRBACAM.
```

Figure 4.6: A usage of undefined role led to an error.

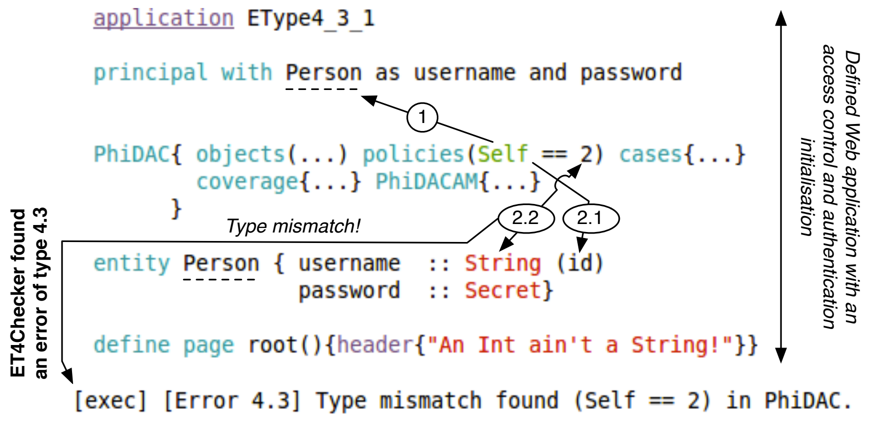
2.*Repetition Validation*: Defined policy terms should be treated as a set, therefore within each policy term there *must not* be any policy term that is repeated. Through all Φ models I have *seven* types of policy terms (i.e., *user identifier*, *user attribute*, *data attribute*, *user-data attributes*, *system attributes*, *labels* and *roles*) and so the types of errors given by Φ in this part is sorted based on the types of the policy terms. For example as Figure 4.7 shows, within the defined ΦMAC model two policy terms are the same (i.e., `TopSec`) and therefore Φ gives an error and terminates the compilation.



```
PhiMAC{ secLabels {TopSec -> Sec} objects(...)
        policies(TopSec, Sec, TopSec)
        cases{...} PhiDACAM{...}
      }
[exec] [Error 4.2.6] Duplicated policy term (TopSec) found in PhiMAC.
```

Figure 4.7: A duplication in a policy term led to an error.

User Identifier-based Validation: As mentioned before ΦDAC and ΦDACAM are discretionary-based access and authorisation management models, and therefore they can use the *user identifier* (i.e., `Self`) as a policy term. Moreover as discussed in Chapter 2, within the WebDSL language the developer uses the *principal definition* with *two* properties (see Figure 4.8) of an entity to declare the *authentication credentials*. Therefore, if within a Φ model the type of the used comparison value does not match the defined user identifier's *type*, then Φ, through `ET4Checker` strategy, will give an error to the developer and will terminate the compilation. So, as Figure 4.8 shows, Φ first checks the *principal definition* to get the *entity* that represents the users of the system (e.g., `Person`). Then, it go through the entity's properties to get the *property*, which is used as the entity's *identifier*. Finally it checks the *type* of the value used within the policy term (e.g., *Int* in Figure 4.8), with the *type* of the property that represents the users' identifiers (e.g., *String* in Figure 4.8). In case of type mismatch, as Figure 4.8 shows, Φ will give an error and terminates the compilation.



```
application EType4_3_1

principal with Person as username and password

PhiDAC{ objects(...) policies(Self == 2) cases{...}
        coverage{...} PhiDACAM{...}
      }

entity Person { username  :: String (id)
                password  :: Secret}

define page root(){header{"An Int ain't a String!"}}
[exec] [Error 4.3] Type mismatch found (Self == 2) in PhiDAC.
```

1
2.2
2.1
Type mismatch!
ET4Checker found an error of type 4.3
Defined Web application with an access control and authentication initialisation

Figure 4.8: The defined user identifier is incorrect.

Attribute-based Validation: At this point the Φ checks the correctness of attribute-based policy terms (*user*, *data*, and *system* attributes). As mentioned in the previous Chapter these policy terms are used with a comparison with an expression (e.g., `Self.username == "John"`), or compared with another attribute-based policy term (e.g., `Self.password == This.User.password`). However, these comparisons can create an error, based on the type mismatch of left and right hand side types. Table 4.1 shows all the possible types of errors that occur based on attribute-based policy terms. As shown in Table 4.1 the first type of error could occur based on using incompatible type such as `Int` with the user attribute term (i.e., `Self.age`) which does not have the type `String`. The second type of errors is based on a comparison between incompatible *basic types* (i.e., `String`, `Int`, `Float`, etc.) with *user* and *data* attributes. The third type of errors is based on incompatible comparison between attribute-based policy terms. For example as shown in Figure 4.1 the user can not compare the `age` property of the `Student` entity when it is of type `Int`, with the `address` property of the `User` entity which is of type `Text`. All these errors are checked by the Φ compiler and related error will be given to the developer.

Table 4.1: Errors based on individual *attribute-based* policy terms

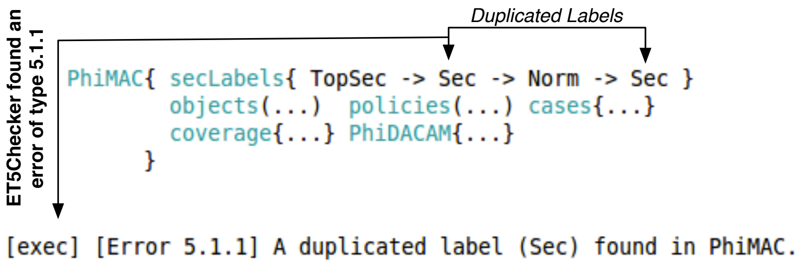
Policy Type	**Error *iff***
UserAtt CompSign Exp **E.g.,** Self.age == "John"	*age.Type != Int*
DataAtt CompSign Exp **E.g.,** This.User.address >= 5.2	*User.address.Type != Float*
DataAtt CompSign UserAtt **E.g.,** This.Student.age == Self.address	*Student.age.Type != User.address.Type*

4.2.1.5 Access Element Correctness

This is the first sub-step in model validation of Φ*MAC*, Φ*MACAM* and validation and verification of Φ*RBAC* and Φ*RBACAM* models. In these models as discussed in the previous chapter, the developer declaratively defines the MAC (i.e., *confidentiality* and/or *integrity labels* and their *flows*) and RBAC (i.e., *roles*, and their *optional relations*) basic elements. Therefore, before Φ processes into *more* expensive validation and verification steps, Φ checks the *correctness* of the following defined access control elements by using the uppermost strategy `ET5Checker` and its sub strategies.

Mac-based Models: At this point Φ checks the Φ's MAC-based models (i.e., ΦMAC and ΦMACAM) based on the following consecutive steps:

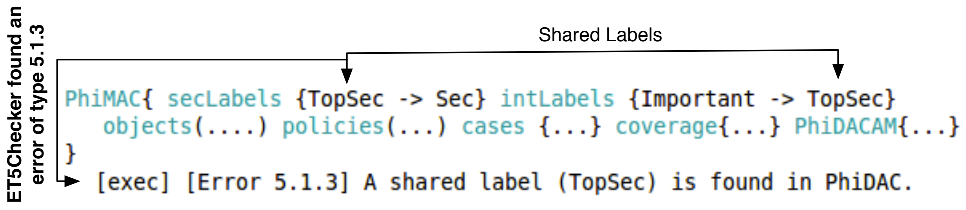
- *1.Duplication Validation:* Here Φ checks if the used label names within the *confidentiality* and *integrity* are *duplicated*. This is an error because as mentioned in the previous Chapter, a set of labels represents different *classifications* and *clearance levels*, are thus need to be *unique*. For example as Figure 4.9 shows, the `Sec` label got repeated and cause an error within the defined ΦMAC model.



```
PhiMAC{ secLabels{ TopSec -> Sec -> Norm -> Sec }
        objects(...)  policies(...) cases{...}
        coverage{...} PhiDACAM{...}
      }
[exec] [Error 5.1.1] A duplicated label (Sec) found in PhiMAC.
```

Figure 4.9: This model led to an error because label `Sec` is duplicated.

- *2.Shared Labels Validation:* For avoiding the confusion for both the developer and the user of the system, the entity that will be generated for the defined ΦMAC and ΦMACAM models (i.e., `Label`), uses the *labelName* property as its identifier (i.e., `label`). Therefore, within the defined MAC-based models, the developers *can not* use the same label name for both confidentiality and integrity labels, and in such a case it leads to an error as shown in the Figure 4.10, and compilation termination.

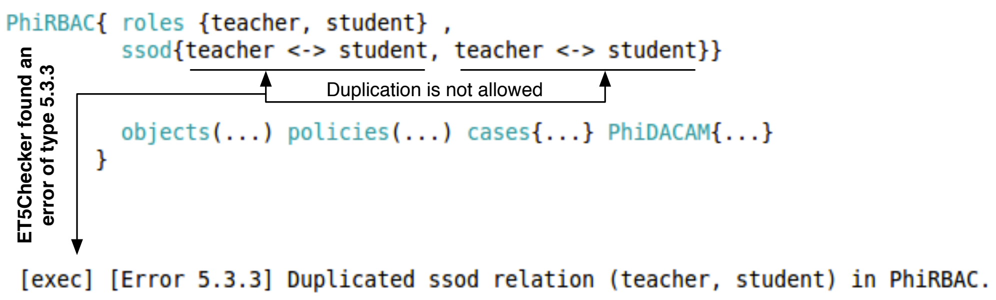


```
PhiMAC{ secLabels {TopSec -> Sec} intLabels {Important -> TopSec}
   objects(....) policies(...) cases {...} coverage{...} PhiDACAM{...}
}
  [exec] [Error 5.1.3] A shared label (TopSec) is found in PhiDAC.
```

Figure 4.10: Shared label names between *confidentiality* and *integrity* labels are not allowed.

Validation and Verification of RBAC-based Models: This part *validates* and *verifies* the defined basic RBAC elements based on the `ET5Checker` strategy and a model checker, Z3 [62]. For this, Φ checks each ΦRBAC and ΦRBACAM model based on the following consecutive steps:

1. *Repetition Validation:* At first, the `ET5Checker` validates the *uniqueness* of defined *roles* and their optional relations (i.e., *hierarchy*, *SSOD*, and *DSOD*). For this the strategy uses the available list-based functionalities within the Stratego/XT library to check each set with respect to itself. For example, as shown in Figure 4.11 the defined ΦRBAC model has a duplication within the SSOD relations. For this `ET5Checker` gives an error and terminates the compilation.



```
PhiRBAC{ roles {teacher, student} ,
         ssod{teacher <-> student, teacher <-> student}}

         objects(...) policies(...) cases{...} PhiDACAM{...}
       }

[exec] [Error 5.3.3] Duplicated ssod relation (teacher, student) in PhiRBAC.
```

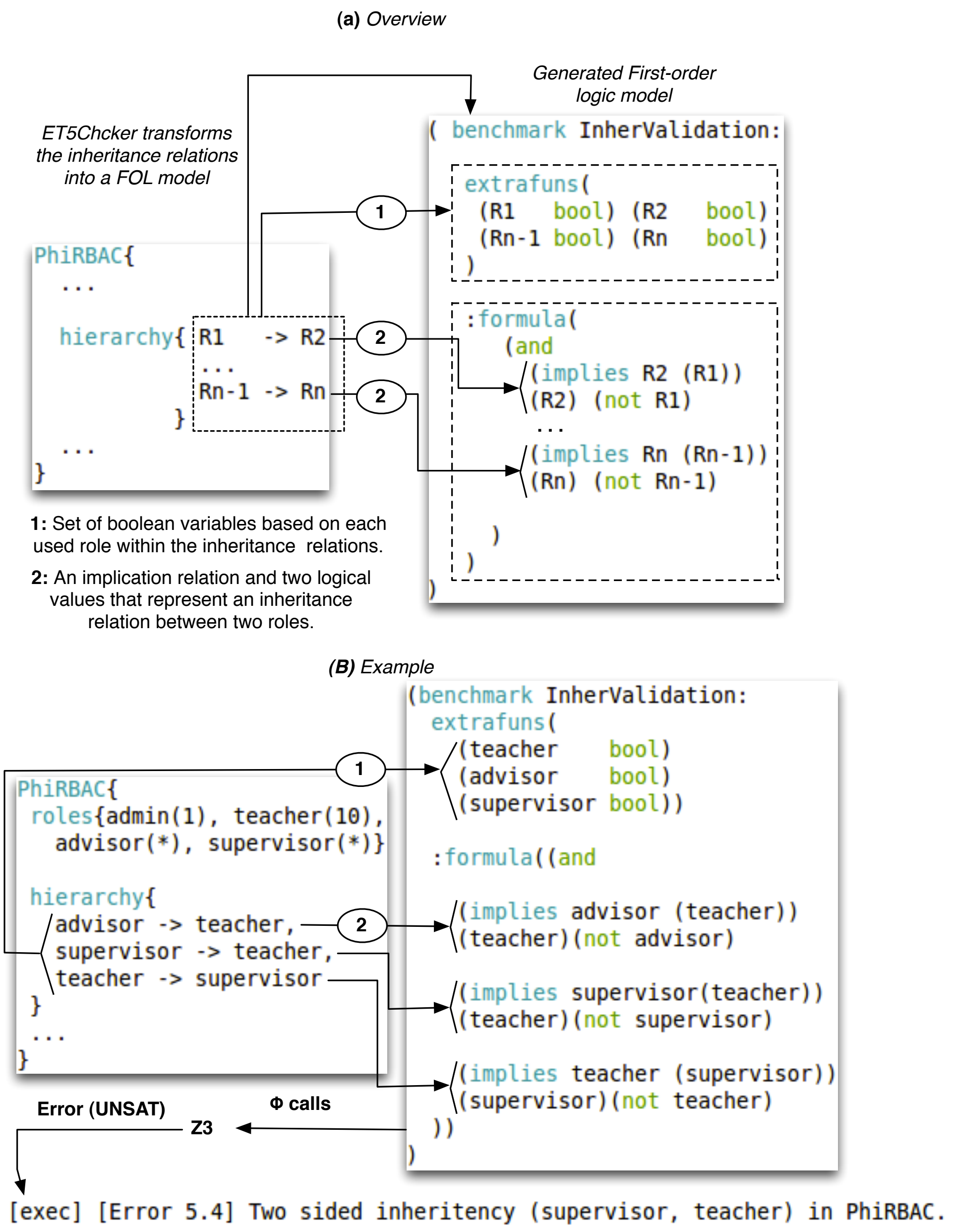
Duplication is not allowed
ET5Checker found an error of type 5.3.3

Figure 4.11: A duplication within *SSOD* relations led to an error.

2. *Hierarchy Verification:* Here Φ checks the *correctness* of the defined inheritance relations, regardless of any *static or dynamic separation of duty relations.* Roles hierarchy and its related checks can grow exponentially based on the hierarchy structure of roles. Checking these errors by hand-written code can be limited and error prone. However, here, I can simply represent the hierarchy in a first order logical formula (i.e., through Φ's transformation rules), and then check its correctness using Z3 and give its corresponding error to the developer. The only semantical error that could occur while defining the role's hierarchies is when within the inheritance definition, *two* roles are inherited from one another. For example, if role A inherits from role B and role B inherits from role A, then if fact these two roles are semantically identical, and therefore it is unnecessary to define two separated roles, or the inheritance relation must be removed by the developer. For this as Figure 4.12 shows, the `ET5Checker` within the Φ compiler transforms the defined inheritance relations into a *first order logic* (FOL) model and then checks its satisfiability through the model checker. As shown in Figure 4.12, the generated FOL model, by Φ, constructed based on the following elements:

   - *Boolean Variables:* For each used role within the inheritance relations role a boolean variable is generated (see Figure 4.12).

- *A Formula:* All the roles hierarchy transforms into a formula such that for each inheritance relation between two roles (e.g., `Supervisor -> Teacher`) an implication relation is defined (e.g., `(implies Teacher (supervisor))`) and the parent role initiated to `True` and the child role is initiated to `False`. Therefore as Figure 4.12 (i.e., Section b) shows in case of `UNSAT` the role's hierarchies contain duplications and therefore an error will be given to the developer and Φ will terminate the compilation.


(a) Overview
Generated First-order logic model
ET5Chcker transforms the inheritance relations into a FOL model
1
2
2
1: Set of boolean variables based on each used role within the inheritance relations.
2: An implication relation and two logical values that represent an inheritance relation between two roles.
(B) Example
1
2
Error (UNSAT)
Φ calls
Z3

```
PhiRBAC{
  ...

  hierarchy{ R1   -> R2
             ...
             Rn-1 -> Rn
           }
  ...
}
```

```
( benchmark InherValidation:
  extrafuns(
    (R1   bool) (R2   bool)
    (Rn-1 bool) (Rn   bool)
  )
  :formula(
     (and
      (implies R2 (R1))
      (R2) (not R1)
      ...
      (implies Rn (Rn-1))
      (Rn) (not Rn-1)

     )
  )
)
```

```
PhiRBAC{
 roles{admin(1), teacher(10),
   advisor(*), supervisor(*)}

 hierarchy{
   advisor -> teacher,
   supervisor -> teacher,
   teacher -> supervisor
 }
 ...
}
```

```
(benchmark InherValidation:
  extrafuns(
   (teacher    bool)
   (advisor    bool)
   (supervisor bool))

  :formula((and

   (implies advisor (teacher))
   (teacher)(not advisor)

   (implies supervisor(teacher))
   (teacher)(not supervisor)

   (implies teacher (supervisor))
   (supervisor)(not teacher)
  ))
)
```

```
[exec] [Error 5.4] Two sided inheritency (supervisor, teacher) in PhiRBAC.
```

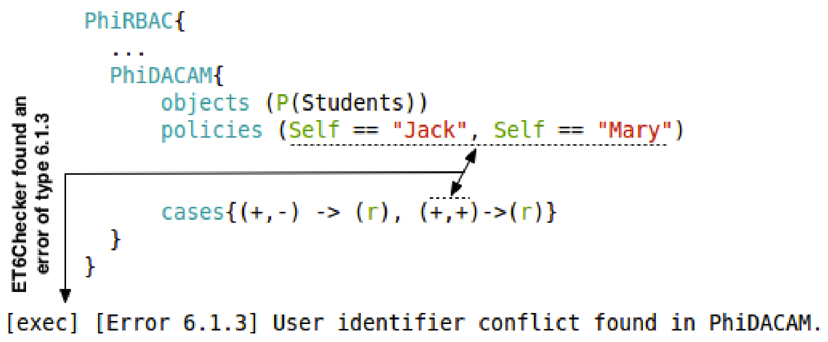
Figure 4.12: Inheritance relations are transform to an FOL model by the `ET5Checker` strategy and then automatically checked by the Z3.

4.2.1.6 Each Case Correctness

In later steps Φ *conjunctively* connects each *policy case* to create an *access* or *authorisation management* predicate; and *coverage case* to create a coverage criteria for the access and authorisation management models. Also as discussed in the last chapter the policy signs give a logical meaning to their related policy terms (i.e., *true*/*activated*, *false*/*not activated*, or *don't care*). Moreover, Φ already checked the correctness of each policy term (see Section 4.2.1.4). Therefore at this step, it is important to check the *correctness* of each defined policy and coverage case. Φ checks (i.e., by using the uppermost strategy `ET6Checker`) the correctness of each defined *policy* and *coverage* cases within *access control*, *authorisation management* models. The following three parts discuss this validation and verification step, based on the *authorisation type* of the Φ-based models.

ΦDAC and ΦDACAM Models. Based on the *cheapness* factor, Φ first verifies each *policy* and *coverage* case in ΦDAC and ΦDACAM models, based on the following consecutive steps:

User Identifier Validation: The `ET6Checker` checks within each case if any two user identifiers create a semantical *conflict*. For example, the second case defined within the Φ*DACAM* model in Figure 4.13 states that the user who is "John" *and* "Mary" at the same time can read the page `Students`. This is an error because the instances of the user identifiers are *unique* throughout the running phase of the application, and therefore a user can not have more than one identifier. So as Figure 4.13 shows, Φ (through `ET6Checker` strategy) gives this error and terminates the compilation. For more similar types of error, please refer to Appendix B.



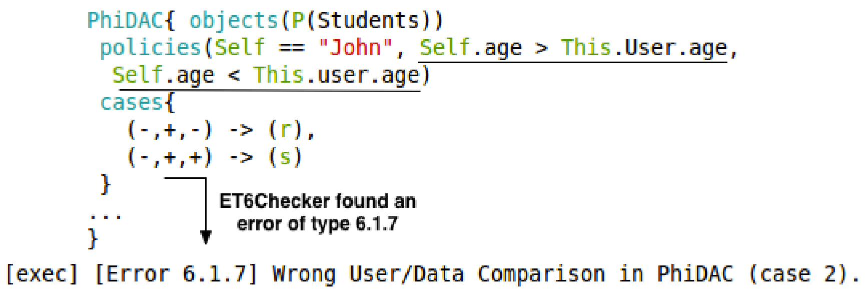
```
PhiRBAC{
  ...
  PhiDACAM{
      objects (P(Students))
      policies (Self == "Jack", Self == "Mary")

      cases{(+,-) -> (r), (+,+)->(r)}
  }
}
[exec] [Error 6.1.3] User identifier conflict found in PhiDACAM.
```

Figure 4.13: An incorrect case within the ΦDAC model based on the user identifier conflict.

Attribute-based Verification: Then, Φ (through `ET6Checker`) verifies each *policy* and *coverage* cases based on the defined *attribute-based* policy terms and their related sings. the following discusses this steps based on the attributes' types:

- *User-based Attributes:* In this part `ET6Checker` checks the correctness of each user-based attributes and their related policy signs for each case. Here, similar to the user identifier validation, `ET6Checker` validates if any two user-based attributes create an error. For example, if in a case there are two + signs for `Self.age > 18` and `Self.age < 18` policy terms, then the `ET6Checker` strategy notifies the developer about the error and then terminates the compilation. See Appendix B for more examples.

- *Data-based Attributes:* Similar to the previous step `ET6Checker` validates the data-based attributes that create a conflict within each case. For example, if in a case there are two $+$ signs for `This.User.name == "Jack"` and `This.User.name == "Mary"` policy terms, the strategy creates an error and terminates the compilation (See Appendix B for examples).
- *User- and data attribute-based comparisons:* Similar to the last to steps, for each policy and coverage case, the `ET6Checker` strategy verifies the policy terms which are the comparison between the user- and attribute-based policy terms, and their related signs within each case. For example, as shown in Figure 4.14, Φ gives an error based on the fact that the second policy case will result in *wrong* access control predicate; because the instance of the current user's age (i.e., `Self.age`) *can not be* greater *and* smaller than the instance of the `User.age` *at the same time.*



```
PhiDAC{ objects(P(Students))
 policies(Self == "John", Self.age > This.User.age,
  Self.age < This.user.age)
 cases{
   (-,+,-) -> (r),
   (-,+,+) -> (s)
 }
 ...
}
[exec] [Error 6.1.7] Wrong User/Data Comparison in PhiDAC (case 2).
```

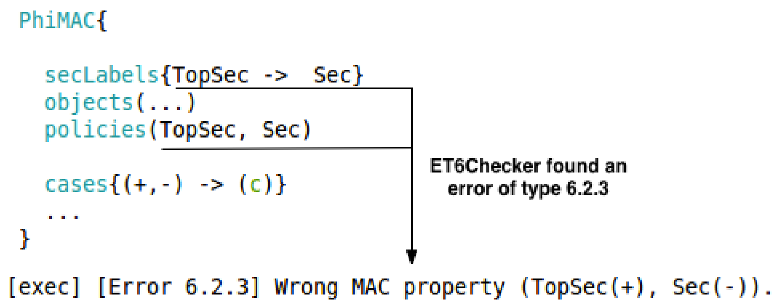
Figure 4.14: An incorrect case based on comparison between user- and data-based attributes.

- *System-based Attributes:* Similar to the last three the `ET6Checker` checks the correctness of each case based on the defined system attribute-based policy term. For this the strategy `ET6Checker` get the defined policy set for each model and check each defined policy case based on the defined system-based (i.e., *date* and *time*) attributes. For example if a case has two $+$ signs for the `Sys.date(>1/1/2010, <1/1/2013)` and `Sys.date(>1/1/2014, <1/1/2015)` policy terms, then this error will be reported to the developer and the compilation terminates by Φ. For examples please refer to Appendix B.

ΦMAC and ΦMACAM models. Here Φ validates and verifies, through `ET6Checker`, the correctness of each defined *policy* and *coverage* case based on the following consecutive sub-steps:

Allowed Operation: ΦMAC and ΦMACAM are MAC-based models, and therefore they only allow *create* and *read* operations. So if the developer uses the *update* or *delete* operation within any case, Φ gives an error and terminate the compilation. Please refer to Appendix B for examples.

MAC Property Verification: Then Φ verifies the information flow of both *confidentiality* and *integrity* dominance levels based on *policy signs* within each policy and coverage case. The error is based on the fact that if an *upper* level have + sign, all its lower levels need to have + sign as well. Therefore, Φ verifies each case based on defined signs and verifies against the informaiton flow, and in case of an incorrect specifiacation, the error is reported to the developer, as shown in Figure 4.15.

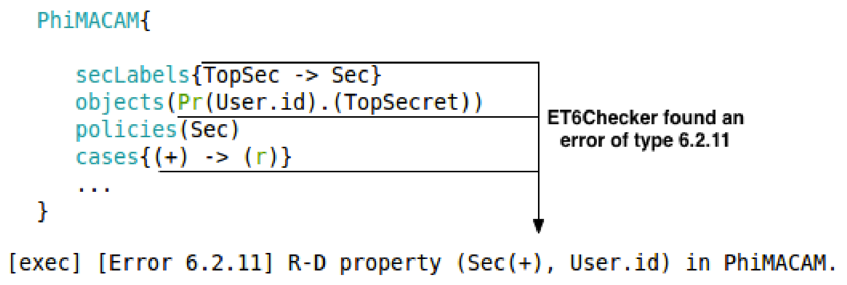


```
PhiMAC{

  secLabels{TopSec ->  Sec}
  objects(...)
  policies(TopSec, Sec)

  cases{(+,-) -> (c)}
  ...
}
[exec] [Error 6.2.3] Wrong MAC property (TopSec(+), Sec(-)).
```

Figure 4.15: An incorrect case based on MAC-based properties of ΦMAC model.

Strict-property Validation: After checking the policy signs related to the information flow, Φ verifies the strict-property validation. This is based on the fact that if the object's *classification label is equal* to the *security clearance* of a case, then the developer can use *both* create (i.e., `c`) and read (i.e., `r`) operation for that particular object. Please refer to Appendix B for examples.

Confidentiality Verification: Similar to the prior step, here Φ, through `ET6Checker` upper-most strategy verifies the *confidentiality-related* properties. For this, Φ verifies the *read-down* and *write-up* concepts (see Chapter 3 for semantics) by verifying the subject's confidentiality-based clearance level against the object's confidentiality-based security classification. For example, as shown in Figure 4.16 the defined ΦMACAM model creates an error because the subject's clearance level is *lower* than object's security classification and the related operation is *read* (i.e., `r`).

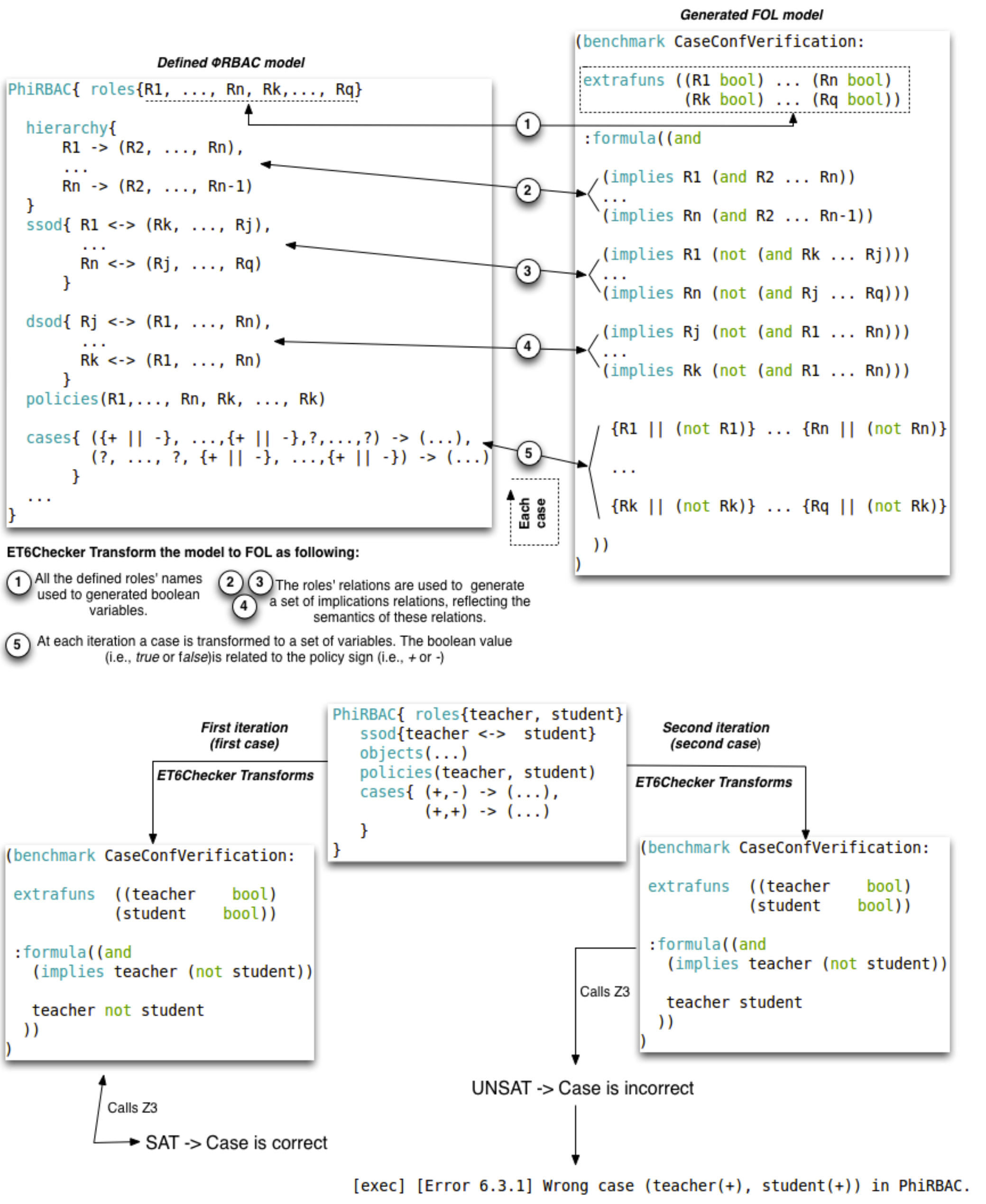


```
PhiMACAM{

  secLabels{TopSec -> Sec}
  objects(Pr(User.id).(TopSecret))
  policies(Sec)
  cases{(+) -> (r)}
  ...
}
[exec] [Error 6.2.11] R-D property (Sec(+), User.id) in PhiMACAM.
```

Figure 4.16: An incorrect case based on defined confidentiality-based characteristics.

Integrity Verification: Then Φ (through `ET6Checker` strategy) verifies the *integrity-related* properties (i.e., *Read-up* and *write-down* properties). Similar to the the previous step, Φ verifies these properties with resecpt to the subject's integrity-based clearance level against the object's integrity-bassed security classification. Please refer to Appendix B for its categories.

Attribute-based Verification: Identical to ΦDAC and ΦDACAM models, the attribute-based policies and their related signs within ΦMAC and ΦMACAM models are verifies. See Appendix B for its categories.



Defined ΦRBAC model
Generated FOL model
1
2
3
4
5
Each case
ET6Checker Transform the model to FOL as following:
1 All the defined roles' names used to generated boolean variables.
2 3 4 The roles' relations are used to generate a set of implications relations, reflecting the semantics of these relations.
5 At each iteration a case is transformed to a set of variables. The boolean value (i.e., *true* or *false*)is related to the policy sign (i.e., + or -)
First iteration (first case)
ET6Checker Transforms
Second iteration (second case)
ET6Checker Transforms
Calls Z3
SAT -> Case is correct
Calls Z3
UNSAT -> Case is incorrect

```
PhiRBAC{ roles{R1, ..., Rn, Rk,..., Rq}

  hierarchy{
     R1 -> (R2, ..., Rn),
     ...
     Rn -> (R2, ..., Rn-1)
  }
  ssod{ R1 <-> (Rk, ..., Rj),
        ...
        Rn <-> (Rj, ..., Rq)
     }

  dsod{ Rj <-> (R1, ..., Rn),
        ...
        Rk <-> (R1, ..., Rn)
     }
  policies(R1,..., Rn, Rk, ..., Rk)

  cases{ ({+ || -}, ...,{+ || -},?,...,?) -> (...),
         (?, ..., ?, {+ || -}, ...,{+ || -}) -> (...)
      }
  ...
}
```

```
(benchmark CaseConfVerification:

extrafuns ((R1 bool) ... (Rn bool)
           (Rk bool) ... (Rq bool))

:formula((and

 (implies R1 (and R2 ... Rn))
 ...
 (implies Rn (and R2 ... Rn-1))

 (implies R1 (not (and Rk ... Rj)))
 ...
 (implies Rn (not (and Rj ... Rq)))

 (implies Rj (not (and R1 ... Rn)))
 ...
 (implies Rk (not (and R1 ... Rn)))

   {R1 || (not R1)} ... {Rn || (not Rn)}

   ...

   {Rk || (not Rk)} ... {Rq || (not Rk)}

 ))
)
```

```
PhiRBAC{ roles{teacher, student}
   ssod{teacher <->  student}
   objects(...)
   policies(teacher, student)
   cases{ (+,-) -> (...),
          (+,+) -> (...)
   }
}
```

```
(benchmark CaseConfVerification:

 extrafuns  ((teacher    bool)
            (student    bool))

 :formula((and
   (implies teacher (not student))

   teacher not student
  ))
)
```

```
(benchmark CaseConfVerification:

 extrafuns  ((teacher    bool)
            (student    bool))

 :formula((and
   (implies teacher (not student))

   teacher student
  ))
)
```

```
[exec] [Error 6.3.1] Wrong case (teacher(+), student(+)) in PhiRBAC.
```

Figure 4.17: An incorrect case within the ΦRBAC model based on defined SSOD relation.

ΦRBAC and ΦRBACAM Models. At last Φ verifies the *correctness* of each case within the ΦRBAC and ΦRBACAM models, based on the following two consecutive steps:

Attribute-based Verification: First, identical to the previous models, the attribute-based policies and their related signs are verified within the *policy* and *coverage* cases. Please refer to Appendix B for examples.

Conflicting Role Verification: Second, Φ transforms each policy and coverage case *with* the defined *roles* and their related relations (i.e., *hierarchy*, *SSOD*, *DSOD*) to an FOL formula and then check for its *satisfiability* through the model checker Z3 [62]. For example as shown in Figure 4.17 the second policy case in the ΦRBAC model creates an error based on the fact that both `Teacher` and `Student` roles can not be activated at the same time, due to the *SSOD* relation between them.

4.2.1.7 Overlapping Cases

As mentioned in the last Chapter, each case links the *access*, *authorisation management*, or *coverage requirement* to a set of operations. Therefore, to avoid any conflicts or repetition within access and authorisation management predicates, it is essential for Φ to verify if each case is in fact *unique*. Therefore any two cases in any Φ model *must not* be equal *syntactically* or *semantically*. Φ checks the *overlapping* between two defined cases of each model, by using the `ET7Checker` strategy and a model checker (i.e., Z3 [62]). As figure 4.18 shows, Φ checks for these cases by transforming the Φ model into *FOL* and then check for its satisfiability through Z3. The reason Φ uses Z3 here, is due to the potential of high number of cases and inner connectivity of some of Φ-based models (e.g., basic RBAC elements and policy cases in ΦRBAC) that leads to large number of lines of codes to check for overlapped cases. Therefore, here to give the full guarantee about the overlapped cases, the representation of these cases are transferred to FOL and then check by Z3, based on the following steps:

Pairing: First, Φ creates a set that consists of all the paired cases, and then for each pair does the following steps:

1. *FOL formula generation:* As Figure 4.18 shows, Φ generates a FOL that consists of:

   (a) *Boolean Variables:* A number of *boolean* variables are created based on the number of policy terms.

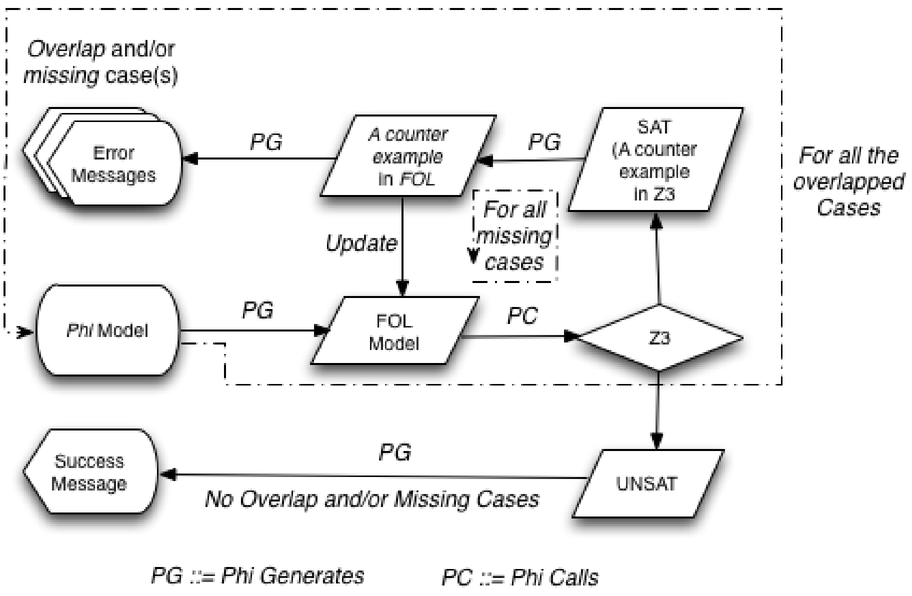
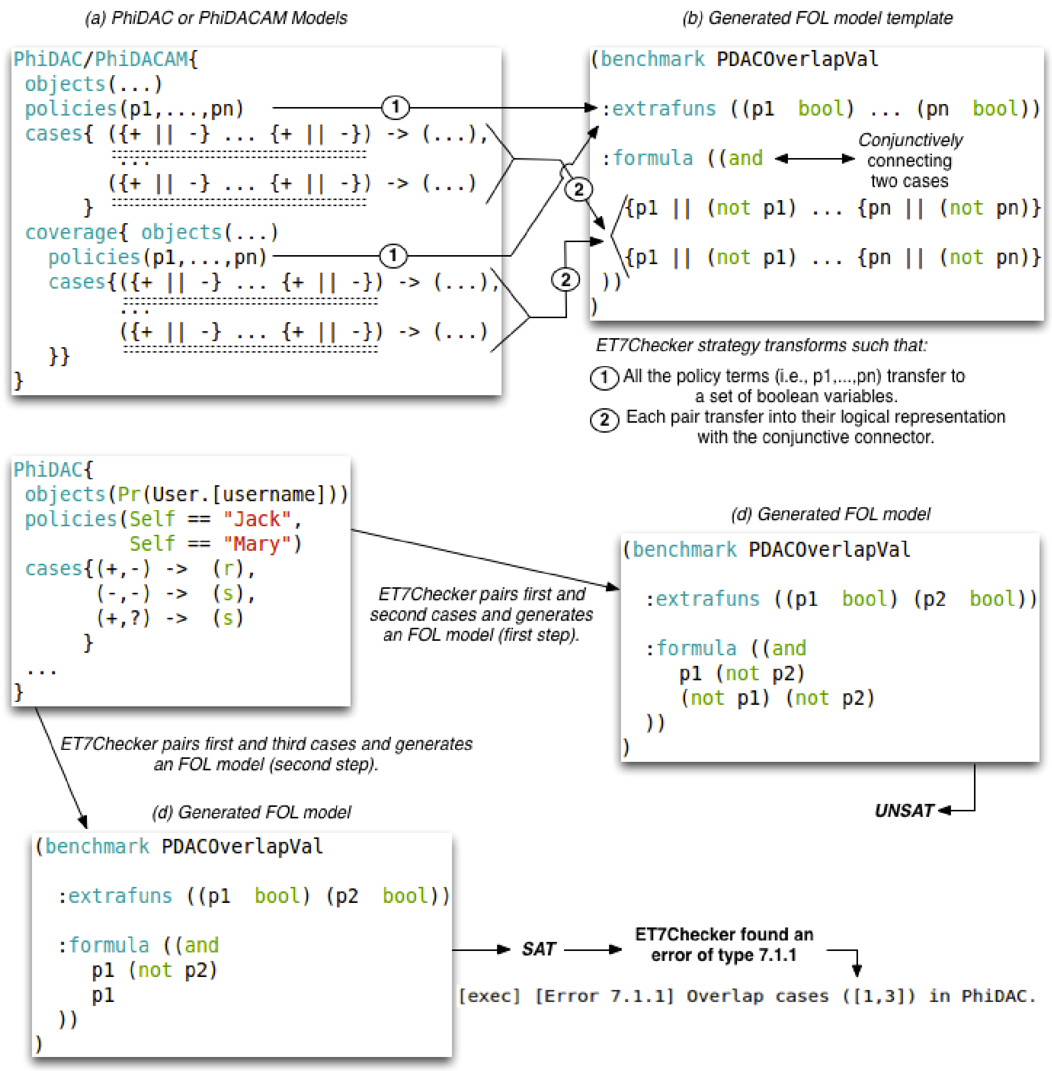


Figure 4.18: Architecture of overlapping and incompleteness checks

The strategy `ET7Cheker` checks for overlapping cases with in Φ models based on the following steps:

(b) *Body:* The body of the formula is created based on the *logical state* (i.e., policy signs) of each policy term. So wherever the state is *true* (i.e., +) the related boolean variable initiated to *true* value and where the state is *false* (i.e., -), the variable will be *false*. Moreover, all the terms of each case, as well as the cases themselves, are connected with a *conjunction*. Note that for the Φ's MAC-based (i.e., ΦMAC and ΦMACAM) and RBAC-based (i.e., ΦRBAC and ΦRBACAM) models a set of *implication* relations are generated to support the *information flow* in the MAC-based, and roles' optional relations (i.e., *hierarchy*, *SSOD*, and *DSOD*) in RBAC-based models. For example if there is an SSOD relation between the role `teacher` and `student` the generated implication relation is `(implies teacher (not student))`. Please refer to Appendix B for examples.

2. *Satisfiability Check:* Here the generated FOL formula is checked by Z3. For this reason, Φ calls Z3 and checks its output. In case of satisfiability, Z3 generates a model, which means two cases are overlapped and therefore they are *syntactically* or *semantically* equal. Then, Φ gets the Z3's output model, and creates an error message for the developer based on the defined ΦDAC or ΦDACAM model. In case of *unsatisfiability*, Z3's output is an output message of *UNSAT*. So, Φ gets the message and knows the paired cases do not overlap, and therefore it does not create an error message and moves on to the rest of the set.


(a) PhiDAC or PhiDACAM Models
(b) Generated FOL model template
Conjunctively connecting two cases
ET7Checker strategy transforms such that:
1 All the policy terms (i.e., p1,...,pn) transfer to a set of boolean variables.
2 Each pair transfer into their logical representation with the conjunctive connector.
ET7Checker pairs first and second cases and generates an FOL model (first step).
(d) Generated FOL model
ET7Checker pairs first and third cases and generates an FOL model (second step).
(d) Generated FOL model
UNSAT
SAT
ET7Checker found an error of type 7.1.1

```
PhiDAC/PhiDACAM{
 objects(...)
 policies(p1,...,pn)
 cases{ ({+ || -} ... {+ || -}) -> (...),
        ...
        ({+ || -} ... {+ || -}) -> (...)
      }
 coverage{ objects(...)
   policies(p1,...,pn)
   cases{({+ || -} ... {+ || -}) -> (...),
          ...
          ({+ || -} ... {+ || -}) -> (...)
   }}
}
```

```
(benchmark PDACOverlapVal
 :extrafuns ((p1  bool) ... (pn  bool))
 :formula ((and
   {p1 || (not p1) ... {pn || (not pn)}
   {p1 || (not p1) ... {pn || (not pn)}
 ))
)
```

```
PhiDAC{
 objects(Pr(User.[username]))
 policies(Self == "Jack",
          Self == "Mary")
 cases{(+,-) ->  (r),
       (-,-) ->  (s),
       (+,?) ->  (s)
      }
 ...
}
```

```
(benchmark PDACOverlapVal
  :extrafuns ((p1  bool) (p2  bool))
  :formula ((and
     p1 (not p2)
     (not p1) (not p2)
  ))
)
```

```
(benchmark PDACOverlapVal
  :extrafuns ((p1  bool) (p2  bool))
  :formula ((and
     p1 (not p2)
     p1
  ))
)
```

```
[exec] [Error 7.1.1] Overlap cases ([1,3]) in PhiDAC.
```

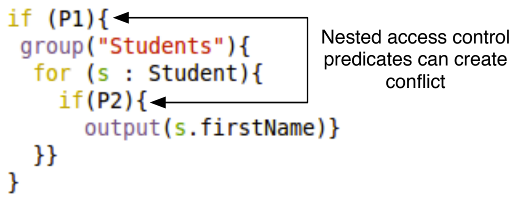
Figure 4.19: Overlapped cases in ΦDAC model created an error.

Rest of the Overlapped Cases: As Figure 4.18 shows, the above mentioned steps are repeated, until the end of the set of paired cases, and all the other overlapping cases are reported to the developer.

4.2.1.8 Dead Authorisation Code

This part of the thesis, first discusses the concept behind *dead authorisation code*, and then explains how Φ validates and verifies the defined Φ-based models for the dead authorisation code.

In Φ automation mechanism, the access control predicates are woven into the application code around the controlled objects. These controlled objects and consequently their predicates may be nested within each other and so create a set of conflicts. For example, in Listing 3, I have two different controlled objects, in which the instances of all students' `firstName` are embedded within the sub-element of the group `Students`. I have `P1` that protects `Students` and `P2` that protects the instances of students' `firstName`. Moreover, `P1` indirectly protects `P2` as `P2` is nested within `P1`. Let us assume that `P1` and `P2` can conflict. For instance, `P1` is true for the users with the activated role `teacher` and `P2` is true for the users with the activated role `admin` but in the access control model there is also an `SSOD` relation between role `teacher` and `admin`. It is clear that users with the activated role `admin` can *never* access the instances of students' `firstName`, even though they have a right to do so. This research calls these unreachable areas *dead authorisation code*. It is more efficient to check for these errors *before* the transformation phase. The following parts discusses how Φ uses validation and verification techniques to find these areas for the Φ access and authorisation based models before the transformation phase.



```
if (P1){
 group("Students"){
  for (s : Student){
    if(P2){
      output(s.firstName)}
  }}
}
```

Figure 4.20: Nested controlled objects and predicates may create a set of conflicts.

Φ validate and verifies each defined model, against the dead authorisation code, based on the following three steps:

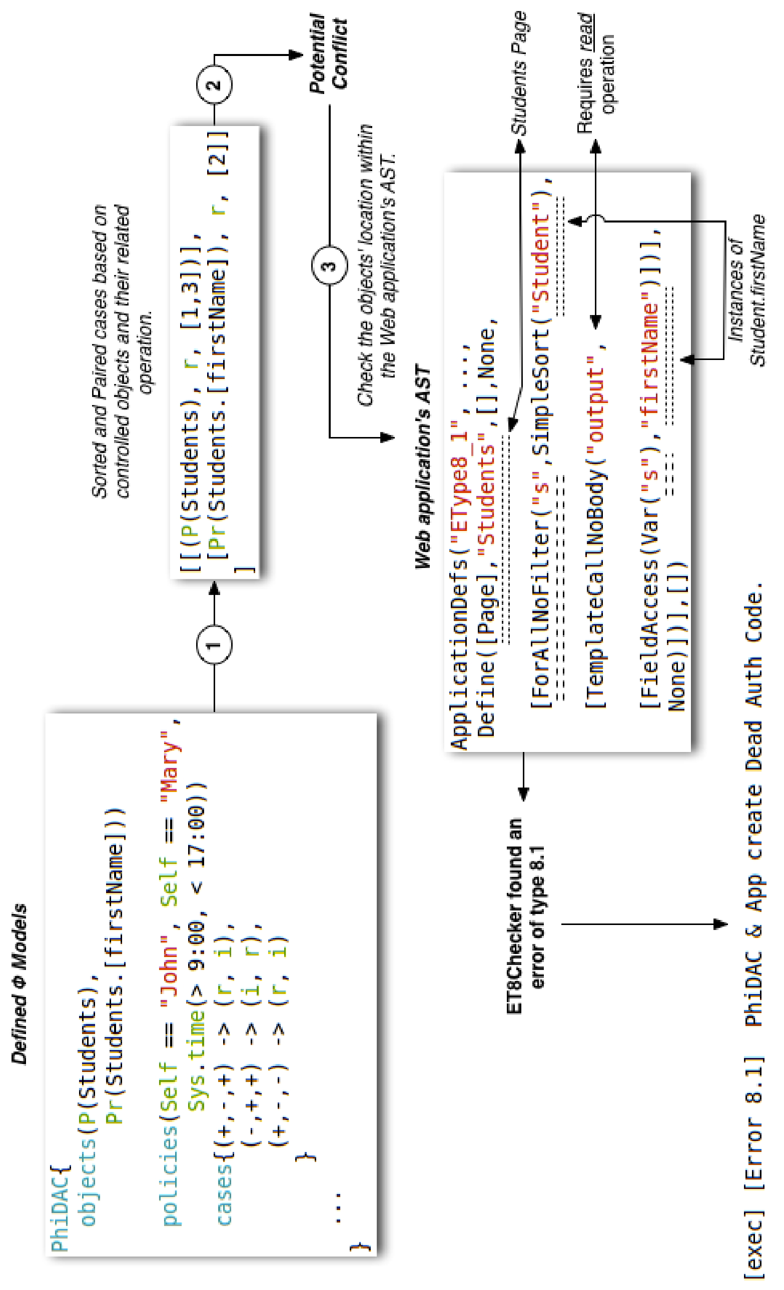
1. **Sorting and Pairing:** First, Φ sorts all policy cases based on the controlled objects and their related operations. For example as shown in Figure 4.21, after the first step, for the `Students` page and its `read` operation the case *one* and *three*; and for the object `Students.firstName` and `read` operation the second case are gathered. Then these two objects are paired.

2. **Potential Conflicts:** After pairing all the used controlled objects and their predicates, Φ checks for potential conflicts, in each pair of *(Obj1, Oper*, {PM ... PN)}* and *(Obj2, Oper*, {PJ... PK})* such that:

$$\{PM...PN)\} \cap \{PJ...PK)\} \equiv \varnothing \vee \neg(\{PM...PN)\} \cup \{PJ...PK)\}) \Rightarrow pc : PotentialConflicts \tag{4.1}$$

   Therefore Φ checks for finding if the two set of related predicates to *Obj1* and *Obj2* are in fact sharing the same set of of cases, if not then the predicates associated to these objects may create a conflict. For example, as shown in Figure 4.21 the paired objects's predicates create a potential conflict as their related predicates are different. In Φ terminology these set of predicates named *potential conflicts*. This is because, Φ, handles this step (i.e., Web application validation) *before* the transformation phase. So, Φ at each iteration, checks for the dead authorisation code based on the *location* of two objects, in which their predicates are conflicting. Therefore because not all the time a controlled object is embedded within the other object, these set of conflicts named potential conflicts.

3. **Traversing Application's AST:** Φ checks for the dead authorisation code, based on the list that contains a set of object pairs in which their predicates are conflicting, throughout the application code, based on the following sub-steps:


Defined Φ Models
Sorted and Paired cases based on controlled objects and their related operation.
1
2
Potential Conflict
3
Check the objects' location within the Web application's AST.
Web application's AST
Students Page
Requires read operation
Instances of Student.firstName
ET8Checker found an error of type 8.1

```
PhiDAC{
  objects(P(Students),
          Pr(Students.[firstName]))

  policies(Self == "John", Self == "Mary",
           Sys.time(> 9:00, < 17:00))
  cases{(+,-,+) -> (r, i),
        (-,+,+) -> (i, r),
        (+,-,-) -> (r, i)
       }
  ...
}
```

```
[[(P(Students), r, [1,3])],
 [Pr(Students.[firstName]), r, [2]]
]
```

```
ApplicationDefs("EType8_1", ...,
Define([Page],"Students",[],None,
   [ForAllNoFilter("s",SimpleSort("Student"),
   [TemplateCallNoBody("output",
   [FieldAccess(Var("s"),"firstName")])]),
   None)]),[])
```

```
[exec] [Error 8.1]  PhiDAC & App create Dead Auth Code.
```

Figure 4.21: Three main steps that check the *Dead authorisation code* within ΦDAC and ΦDACAM models.

(a) *Factorisation and ordering:* To have a cheap algorithm by finding all the *dead authorisation code* without unnecessary repetition, Φ factorizes and orders the potential conflicts list based on coarse-granularity of the objects. For example if the potential conflict list is `{((PageOne, r, Preds1), (User,u,Preds2)), ((PageOne,r,Preds1), (Student, u, Preds2))}`, the factorized and ordered set is `{((PageOne, r, Preds1), ((User,u,Preds2)), (Student, u, Preds2))}`. Note that the order is from coarse-grained to fine-grained (i.e., Page, template, block, group, xml, entity, property).

(b) *Finding dead authorisation code:* Then Φ uses the list created by the last sub-step, and traverse through out the application's AST, and check if one of the two objects with potential conflict, is in fact embedded in the body of the second object. If so and error will be generated by Φ, based on the defined model and the target application and will be given to the developer. For example as Figure 4.21 shows, the object `Students.firstName` property is in fact embedded within the body of the page `Students` and there as it shows, Φ gives an error and terminating the Φ compilation by finding this error.

4.2.2 Warning Handling

After validating and verifying the Φ models against all the essential *errors*, as Figure 4.1 shows, Φ will then check for *incompleteness* and *insufficiency* constraints, that creates and displays a set of warnings to the developer, based on the defined policy and/or coverage cases. Figure 4.1 shows that in a case of a warning in each step, Φ either *terminates* or *continues* the compilation. These termination or continuation decisions are based on the initiated defined options by the developer within the `Phi.ini` file (see Appendix A for details). The following two parts discuss these two error handling mechanisms.

4.2.2.1 Completeness Verification

At this point, the compiler knows the defined Φ models are *correct* in terms of themselves and their target Web application. However Φ needs to check the *completeness* of the defined models. Completeness of each model is determined based on the defined *policy cases*. Considering the number Âăof cases and their relation to the defined policies for access or authorisation management structure, a large number of standalone lines of code needs to be written to check for the completeness of these defined cases. Moreover, both the structure of security models and the policy cases can be transferred to the defined first order logic and be checked by a model checker, Z3. For these reasons, Φ compiler uses Z3 to provide a full guarantee on the completeness of the defined cases.ÂăOverall as Figure 4.18 illustrates the workflow, Φ, through `WT1Checker` uppermost strategy, checks for missing cases by transforming each set of the defined cases (i.e., policy), into a first order logic formula; and verifies for its satisfiability through an SMT solver, Z3. Therefore, for each set of policy cases, within all Φ-based models, Φ performs the following steps to check *all* the missing cases:

1. *Create a Set:* Φ first creates a set, in which each element represents a case based on the relation between the policy signs and policy terms.

2. *FOL formula generation:* Then, Φ generates a FOL formula where its *boolean variables* are representing all the defined policy terms (i.e., PT1, ..., PTN). Then it generates the *negation* of all the cases, where for each case it generates as follows:

   (a) *True/False assignments:* The value for each defined variable is *true* where the logical state for the related policy term is *false* (i.e., `-`), and initiate the value of the variable to *false* where the state of the policy term is *true* (i.e., `+`).

   (b) *Logical Connections:* The elements of each negated case is connected with a *disjunction*. While all the negated cases are connected with a *conjunction*.

   (c) *Implications:* A set of implication relations are generated by Φ for the defined basic RBAC and MAC structure within the ΦRBAC, ΦRBACAM, ΦMAC, and ΦRBACAM. For the inherency, SSOD or DSOD relations, within the ΦRBAC and ΦRBACAM models, the `WT1Checker` strategy, generates a set of implications as following:

      - *Inherency:* For each inheritance relation between the defined roles, Φ generates an implication relation such as `A -> B` where role *B* is inherits the role A authorisation rights.
      - *SSOD/DSOD relations:* For each SSOD or DSOD relation between the defined roles, Φ generates an implication relation such that `A -> !B` where there is an SSOD or DSOD relation between these two roles.

   Moreover, for the confidentiality and integrity information flow within the defined ΦMAC and ΦMACAM models, similar to the inherency relations within the ΦRBAC and ΦRBACAM models; Φ generates an implication relations. For this, for each `upperLabel -> lowerLabel` relation, Φ generates an implication relation such that if upperLabel is true then lowerLabel can be true too.

3. *Satisfiability Check:* Here Φ calls Z3 to check the satisfiability of the generated FOL formula. If the generated formula is *unsatisfiable* (i.e., `UNSAT`), then there are no missing cases. However, if Z3 creates a counter example in case of *satisfiability*, then the counter example is the missing case, and an error message is generated for the developer based on the defined Φ-based models.

4. *Rest of Missing Cases:* Furthermore, for finding the next missing case, if any, Φ generates the negation of the missing case and updates the FOL formula and checks its satisfiability with Z3. The same routine is repeated by Φ (through `WT1Checker` strategy) until there is no more missing cases to report to the developer.

4.2.2.2 Coverage Verification

The aim of this step is to check the required access control and authorisation coverage criteria, based on the defined coverage cases in the defined Φ models. So that it can provide feedback to the developer about the shortcomings of the defined models. In terms of *fine-grained access control models* (i.e., ΦDAC, ΦMAC, and ΦRBAC), a coverage percentage shows, in what percentage the object is protected *directly* or *indirectly* by the derived access control predicates that are defined in the coverage cases of the access control models. In terms of *fine-grained authorisation management models* (i.e., ΦDACAM, ΦMACAM, ΦRBACAM), a coverage percentage shows, in what percentage the access control elements of the controlled objects are managed directly or indirectly through the defined authorisation models. For each controlled object used in the coverage or AMS cases, the coverage percentage is calculated.

As mentioned before Φ gives ability to the developer to define a set of cross checks test cases for the Φ's access control and Φ's authorisation management systems. This part of the report discusses the coverage checks in two categories of access control and authorisation management models:

Coverage checks for Φ's access control models

This section discuss the concept of access control coverage in Φ. As mentioned before in this research, the access control coverage, refers to how many percentage an object is covered through defined access control model policies *directly* or *indirectly* throughout the application code. I find the coverage percentage of each mentioned object in coverage cases based on the following steps:

Object occurrences: First, the Φ compiler checks the application's AST and counts the occurrences of the controlled object.

Related predicates: Second, the related strategies in Φ language, check the related access control predicates in which they are *true*, and make a list of all the objects related to these predicates (i.e., policy cases).

Coverage calculation: Third, I count all the occurrences of the controlled object that are directly or indirectly (i.e., by nesting) related to the list of controlled objects in the last step. At the end, Φ counts the coverage percentage based on dividing the number of occurrences that I get from this step with its total number of occurrences (i.e., first step) and then multiplies the number by 100. Φ compiler creates a coverage error in case the number does not match the number defined in the coverage cases.

Coverage checks for Φ's authorisation management models

This part of the thesis discuss the meaning behind coverage checks for authorisation management models, and how they are calculated in Φ. The coverage percentage in Φ's authorisation management systems, shows how many access controlled objects are *directly* or indirectly are covered through authorisation management mechanisms. Φ checks for the coverage percentage in authorisation models as following:

1. *Related predicates:* Φ first, check the related authorisation management predicates, to each coverage case in the model.
2. *Coverage calculation:* Φ then checks the authorisation coverage percentage for each used controlled object and its related operation, based on dividing number of cases that did not cover the object and the operation over the number of policy cases that did cover the object and its operation. Then the number is multiplied by 100 to get the authorisation management coverage.

As it can be seen from the mentioned steps their difference in coverage concept between access control and authorisation management models is that for the management models that the object occurrence is not calculated, and only the related policy predicates within the management models are important; as they do control 100 percent of the access control objects.

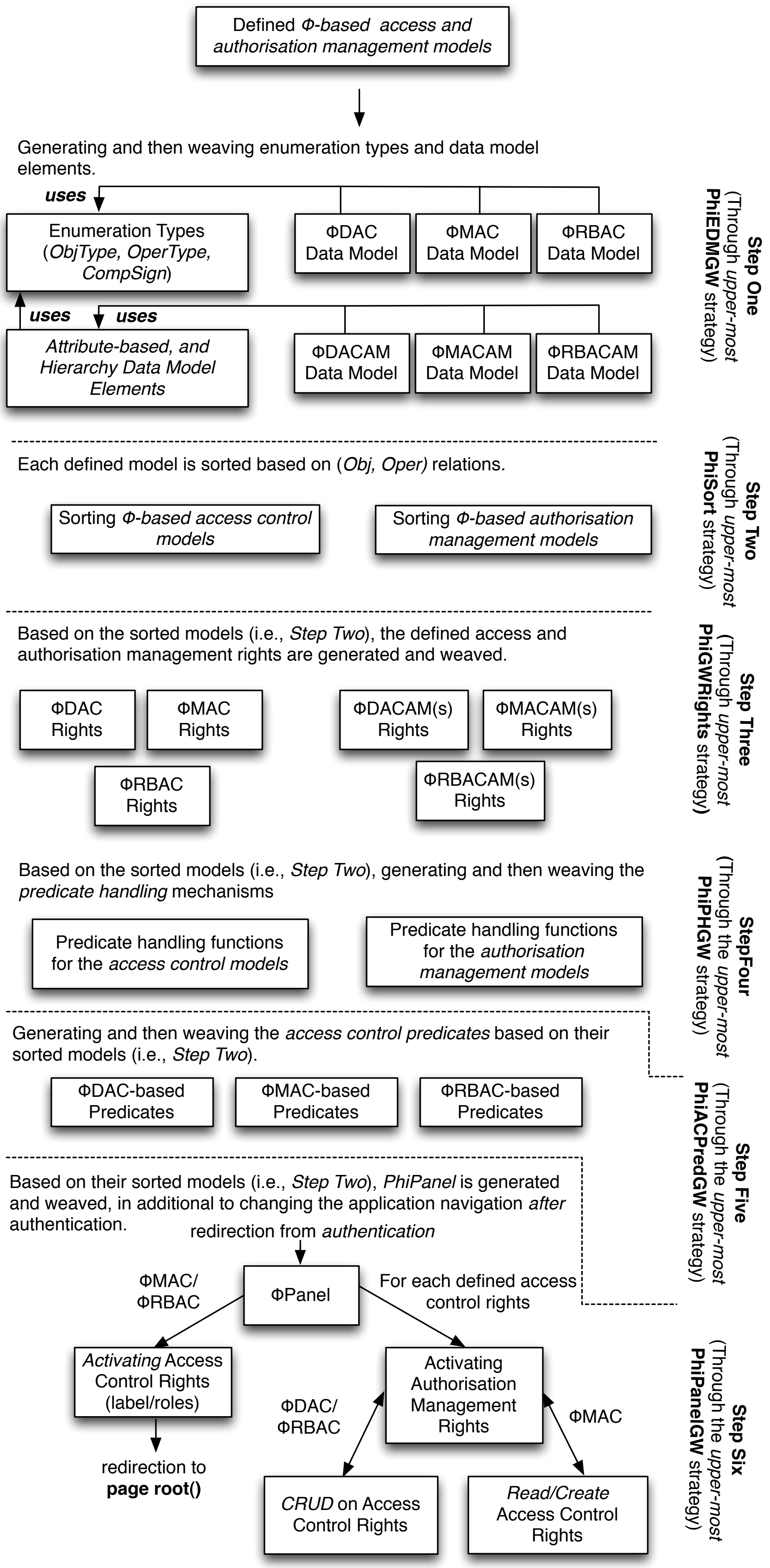
4.3 Transformation Phase

The Φ compiler executes the transformation phase[2], after successfully *validating* and *verifying* the defined Φ-based models with respect to their *semantics*. This phase *generates* and then *weaves* the access and authorisation management *mechanisms* based on the defined models into the WebDSL code. These mechanisms are responsible for *guarding* the controlled objects in terms of their operational usage (i.e., in case of access control) and managements of the user's rights (i.e., in case of authorisation management) during the run-time of the system. This part of the thesis, discusses each step of the architecture of the Φ's transformation phase (shown in Figure 4.22), with regarding to themselves and the Φ-based models.

4.3.1 Data Model Elements and Enumeration Types

During the first transformation step (see Figure 4.22), Φ generates and then weaves the Φ-related *data model* elements with

[2]`http://philang.org/repos/compiler/Phi/transformation.str`



Defined Φ-based access and authorisation management models
Generating and then weaving enumeration types and data model elements.
uses
Enumeration Types (ObjType, OperType, CompSign)
ΦDAC Data Model
ΦMAC Data Model
ΦRBAC Data Model
uses
uses
Attribute-based, and Hierarchy Data Model Elements
ΦDACAM Data Model
ΦMACAM Data Model
ΦRBACAM Data Model
Step One (Through upper-most PhiEDMGW strategy)
Each defined model is sorted based on (Obj, Oper) relations.
Sorting Φ-based access control models
Sorting Φ-based authorisation management models
Step Two (Through upper-most PhiSort strategy)
Based on the sorted models (i.e., Step Two), the defined access and authorisation management rights are generated and weaved.
ΦDAC Rights
ΦMAC Rights
ΦRBAC Rights
ΦDACAM(s) Rights
ΦMACAM(s) Rights
ΦRBACAM(s) Rights
Step Three (Through upper-most PhiGWRights strategy)
Based on the sorted models (i.e., Step Two), generating and then weaving the predicate handling mechanisms
Predicate handling functions for the access control models
Predicate handling functions for the authorisation management models
StepFour (Through the upper-most PhiPHGW strategy)
Generating and then weaving the access control predicates based on their sorted models (i.e., Step Two).
ΦDAC-based Predicates
ΦMAC-based Predicates
ΦRBAC-based Predicates
Step Five (Through the upper-most PhiACPredGW strategy)
Based on their sorted models (i.e., Step Two), PhiPanel is generated and weaved, in additional to changing the application navigation after authentication.
redirection from authentication
ΦPanel
ΦMAC/ ΦRBAC
For each defined access control rights
Activating Access Control Rights (label/roles)
redirection to page root()
Activating Authorisation Management Rights
ΦDAC/ ΦRBAC
ΦMAC
CRUD on Access Control Rights
Read/Create Access Control Rights
Step Six (Through the upper-most PhiPanelGW strategy)

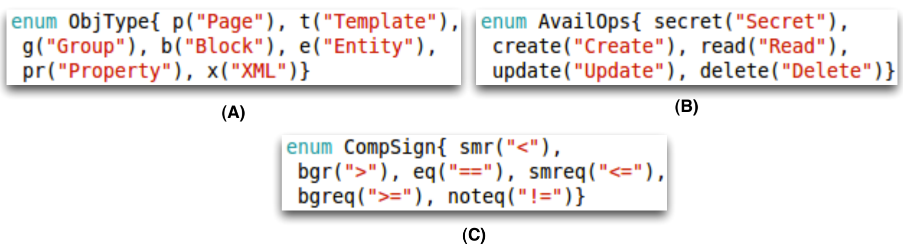
Figure 4.22: Φ's transformation pipeline

respect to the defined access and authorisation management models with three enumeration types. These elements are later used, by Φ during the compile time, to *store* the initial *authorisation rights* based on the defined access and authorisation management models. The main reason to store these authorisation rights is based on the fact that, in this manner, Φ enables administrators of the system to change the defined access control authorisations through their authorisation management mechanisms. The following part discusses the first set of elements which are generated and weaved *regardless* of the type of the defined models. Then the transformation steps for the generated and weaved data models are discussed based on the type of access and authorisation management models.

4.3.1.1 Enumeration Types, Hierarchy- and Attribute-based Data Model Elements

As mentioned before, *all* Φ-based models are *attribute-based*. Moreover, all can use the same type of objects, and use the subset of object related operations. Therefore, during this step, Φ generates and weaves (through uppermost strategy `PhiEDMGW`) the following common elements, without considering the type of the defined models:

Enumeration Types: As shown in Figure 4.23, three sets of enumeration types are generated to cover all the possible objects' *types* (see A), *operations* (see B), *comparison signs* (see C). These enumeration types, are used as a property type, within generated entities which hold the authorisation rights.



```
enum ObjType{ p("Page"), t("Template"),
 g("Group"), b("Block"), e("Entity"),
 pr("Property"), x("XML")}
```

```
enum AvailOps{ secret("Secret"),
 create("Create"), read("Read"),
 update("Update"), delete("Delete")}
```

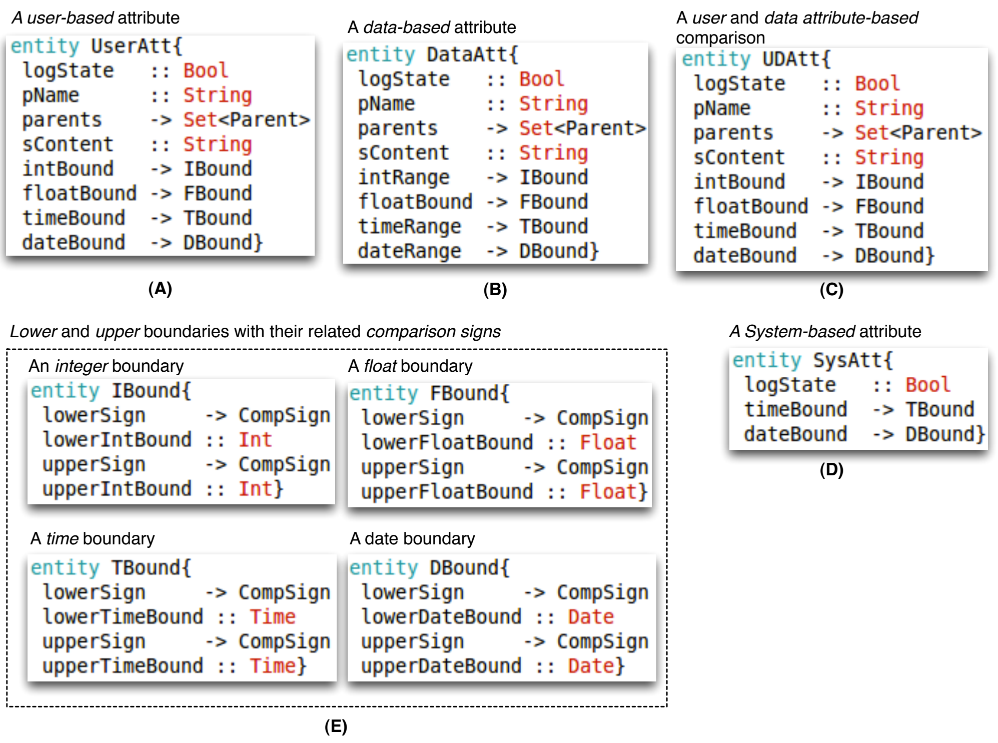
```
enum CompSign{ smr("<"),
 bgr(">"), eq("=="), smreq("<="),
 bgreq(">="), noteq("!=")}
```

Figure 4.23: Three sets of enumeration types that hold the *controlled object* and *operation* types, and comparison signs

Attribute-based Data Models: As Figure 4.24 shows (see part *a* and *b*), the `UserAtt` and `DataAtt` entities are *statically* generated to store the user- and data-based attributes. Both, have the `logState` property for storing the logical state of the user or data constraints, `pName` property for storing the name of the related property, and `parent` for storing the hierarchy of the property. Moreover, for storing the required value, they have `sContent` to store the *String* value, `intBound` of type `IBound` entity (see part *j*), `floatBound` of type `FBound` entity (see part *i*), `timeBound` of type `TBound` entity (see part *g*), and `dateBound` of type `DBound` entity (see part *h*), for storing the *integer*, *float*, *time* or *date* lower and upper boundaries. Note that each instance of the `UserAtt` and `DataAtt` only holds one type of value. For example, if the policy term and its related sign is `Self.username == "John"` and `+`, then the `intBound`, `floatBound`, `timeBound`, and `dateBound` properties are assigned to `null`, where the `sContent` property has a value `"John"`. Moreover, as shown in Figure 4.24 the entity `UDAtt` is generated to store the policy terms related to the comparisons between the user- and data-based attributes and their related policy signs. As shown in the Figure 4.24 the properties are identical to the `UserAtt` and `DataAtt` entities with the same discussed purposes.

Figure 4.24 (see part *D*) shows, the entity `SysAtt` is generated to store the system-based constraints. For this reason, it contains the property `logState` that stores the logical state of the constraints, and `timeBound` of type entity `TBound` (i.e., part *e* in Figure 4.24), that stores the *lower* and *upper* time boundaries. Also it has the property `dataBound` which is of type entity `DBound` (i.e., part *h* in Figure 4.24) that stores the *lower* and *upper* date boundaries. Note that, the time and date boundaries does not necessary need to have values. For example, if the *system-based* constrains is just based on office hours, the *dateBound* entity is assigned to `null`.

Hierarchy-based data model: At the end of this step the entity `ObjParent` is generated and weaved to cover the inheritance relations within the*entities* and their *properties*. For this, this generated entity has a property `Parent`, of type `String` that holds the name of a *data model element*.



A user-based attribute
A data-based attribute
A *user* and *data attribute-based* comparison
(A)
(B)
(C)
Lower and *upper* boundaries with their related *comparison signs*
An *integer* boundary
A *float* boundary
A *time* boundary
A date boundary
(E)
A System-based attribute
(D)

```
entity UserAtt{
 logState    :: Bool
 pName       :: String
 parents     -> Set<Parent>
 sContent    :: String
 intBound    -> IBound
 floatBound  -> FBound
 timeBound   -> TBound
 dateBound   -> DBound}
```

```
entity DataAtt{
 logState    :: Bool
 pName       :: String
 parents     -> Set<Parent>
 sContent    :: String
 intRange    -> IBound
 floatBound  -> FBound
 timeRange   -> TBound
 dateRange   -> DBound}
```

```
entity UDAtt{
 logState    :: Bool
 pName       :: String
 parents     -> Set<Parent>
 sContent    :: String
 intBound    -> IBound
 floatBound  -> FBound
 timeBound   -> TBound
 dateBound   -> DBound}
```

```
entity IBound{
 lowerSign      -> CompSign
 lowerIntBound  :: Int
 upperSign      -> CompSign
 upperIntBound  :: Int}
```

```
entity FBound{
 lowerSign        -> CompSign
 lowerFloatBound  :: Float
 upperSign        -> CompSign
 upperFloatBound  :: Float}
```

```
entity TBound{
 lowerSign       -> CompSign
 lowerTimeBound  :: Time
 upperSign       -> CompSign
 upperTimeBound  :: Time}
```

```
entity DBound{
 lowerSign       -> CompSign
 lowerDateBound  :: Date
 upperSign       -> CompSign
 upperDateBound  :: Date}
```

```
entity SysAtt{
 logState   :: Bool
 timeBound  -> TBound
 dateBound  -> DBound}
```

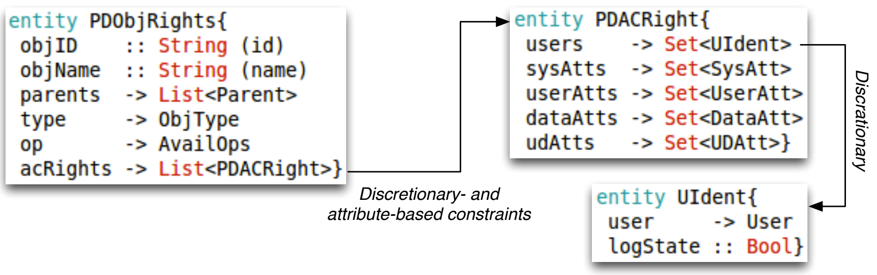
Figure 4.24: Generated Data Model elements that will store the *attribute-based* constraints

4.3.1.2 Discretionary- and Attribute-based Models

As discussed before, ΦDAC and ΦDACAM models are *discretionary-* and *attribute-based* models. Therefore the data model that needs to be generated and weaved for these models needs to be stored discretionary- and attribute-based constraints. The following discusses the generated data model for these models.

ΦDAC: As shown in Figure 4.25, for the ΦDAC model, the following entities are generated to store the *discretionary-* and *attribute-based* constraints:

- *Object Rights:* As Figure 4.25 shows (see part *a*), an entity `PDObjRights` is generated that consists of the object id (i.e., `objID`), the name of the object (i.e., `objName`), its parents entities (i.e., in case of *property*), the related operation (i.e., `op`), the type of the object (i.e., `type`), and its access rights (i.e., `acRights`). Note that for retrieving the access control rights, *only* the object's id and the instances of its access rights are required. However, the rest of the properties are used within the *authorisation management mechanism* for displaying the related access control rights to the authorised users *within* the authorisation management mechanism.



```
entity PDObjRights{
 objID    :: String (id)
 objName  :: String (name)
 parents  -> List<Parent>
 type     -> ObjType
 op       -> AvailOps
 acRights -> List<PDACRight>}
```

```
entity PDACRight{
 users     -> Set<UIdent>
 sysAtts   -> Set<SysAtt>
 userAtts  -> Set<UserAtt>
 dataAtts  -> Set<DataAtt>
 udAtts    -> Set<UDAtt>}
```

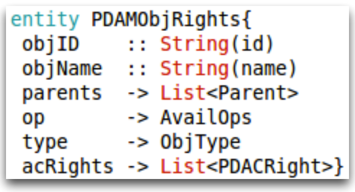
```
entity UIdent{
 user     -> User
 logState :: Bool}
```

Figure 4.25: Generated Data model for the defined ΦDAC model

- *Access Right:* This entity is *statically* generated, which holds an instance of the ΦDAC-based access control rights of an object (See Figure 4.25). So, for storing the *discretionary-based* constraints, this entity has a property `users` to store a set of user identifier-based constraints. Moreover for storing the *attribute-based* constraints, the `AccessRight` entity contains the properties `sysAtts`, `userAtts`, and `dataAtts` for storing a set of *system-*, *user-*, and *data-based* constraints.
- *User Identifier:* Unlike all the other entities, the entity `UIdent` is not entirely generated statically. Φ (through ΦDACGen strategy) gets the defined entity, within the *principal* definition (explained in Chapter 2), that represents the users of the system, and makes the first property `user` based on the type of that entity. For example, if in an application the *User* entity used within the *principal* definition, property `user` is then, of type `User`. Moreover, the

entity `UIdent` has the property `logState` that holds the logical state of the user identifier (see Figure 4.25). For example, if a the sign related to the policy term is `-`, then the `logState` property has the `false` value.

ΦDACAM: The entity `PDAMObjRights` (shown in Figure 4.26) and `PDACRight` (if no ΦDAC model was defined) are generated to store the ΦDACAM constraints. Based on the fact that ΦDACAM has the same *authorisation type* as ΦDAC model, all the properties of the `PDAMObjRights` are identical to the `PDObjRights` entity. However because these authorisations are based on authorisation management and *not* access control, their instances are stored separately.



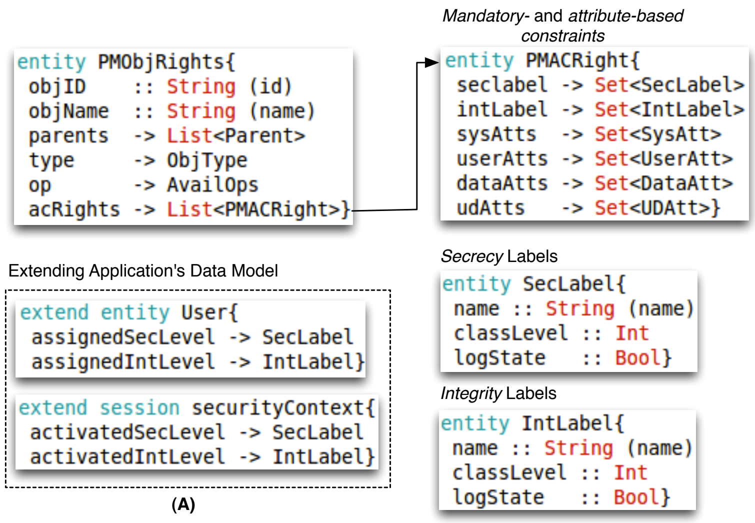
```
entity PDAMObjRights{
 objID    :: String(id)
 objName  :: String(name)
 parents  -> List<Parent>
 op       -> AvailOps
 type     -> ObjType
 acRights -> List<PDACRight>}
```

Figure 4.26: Generated Data model for the defined ΦDACAM model

4.3.1.3 Mandatory- and Attribute-based Models

As discussed before, ΦMAC and ΦMACAM models are *mandatory-* and *attribute-based* models. Therefore the data model elements, that are needed to be generated and weaved for these models need to be able to store, the mandatory- and attribute-based constraints. The following elements are generated for storing the authorisation rights of these models:

ΦMAC: As Figure 4.27 shows, for the defined ΦMAC model, the following entities are generated to store the access control charactrestics of this model:


Mandatory- and *attribute-based constraints*
Extending Application's Data Model
Secrecy Labels
Integrity Labels
(A)

```
entity PMObjRights{
 objID    :: String (id)
 objName  :: String (name)
 parents  -> List<Parent>
 type     -> ObjType
 op       -> AvailOps
 acRights -> List<PMACRight>}
```

```
entity PMACRight{
 seclabel  -> Set<SecLabel>
 intLabel  -> Set<IntLabel>
 sysAtts   -> Set<SysAtt>
 userAtts  -> Set<UserAtt>
 dataAtts  -> Set<DataAtt>
 udAtts    -> Set<UDAtt>}
```

```
extend entity User{
 assignedSecLevel -> SecLabel
 assignedIntLevel -> IntLabel}
```

```
extend session securityContext{
 activatedSecLevel -> SecLabel
 activatedIntLevel -> IntLabel}
```

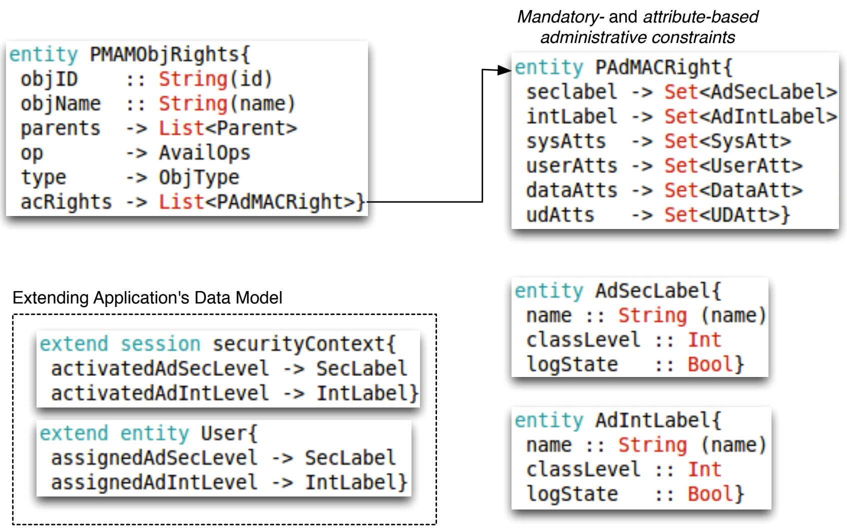
```
entity SecLabel{
 name :: String (name)
 classLevel :: Int
 logState   :: Bool}
```

```
entity IntLabel{
 name :: String (name)
 classLevel :: Int
 logState   :: Bool}
```

Figure 4.27: Generated Data model for the defined ΦMAC model

- *PMObjRights:* This entity contains all the ΦMAC-based constraints for the defined controlled objects and their related authorisation rights. Similar to the `PDObjRights` entity it consists of properties that hold the general information about the controlled objects and their related operations, such as *object's name* (i.e., `ObjName`), and related operation (i.e., `op`). Moreover, it contains the property `acType` which holds the list of *mandatory-* and *attribute-based* constraints (i.e., through `PMACRight`) for each controlled object and its related operation.
- *PMACRight:* This entity holds an instance of the ΦMAC-based constraints. As shown in Figure 4.27, the first two properties (i.e., `confLabel` and `intLables`) hold a list of allowed confidentiality based labels. Moreover, identical to ΦDAC and ΦDACAM models, it consists of the property `sysAtts`, `userAtts`, `dataAtts`, and `udAtts` for storing the *attribute-based* policy terms and their related signs.
- *ConfLabel and IntLabel:* As shown in Figure 4.27, these two entities hold an instance of the *confidentiality* and *integrity* labels. The `classLevel` property holds an integer number that later will be used for *evaluating* the ΦMAC-based constraints. Therefore, the dominance level between two same type labels are determined by the instances of this property.
- *Extensions on Web Application's Data Models:* The instances of the *User* entity is extended so that the labels can be assigned to the users of the system during the run-time of the system. Moreover, the user can activate the assigned labels or their related *less* dominant labels after authentication. Therefore, for this, the Web applications' *session* entity is extended (see part a), to hold the activated labels of the users, during their sessions.

ΦMACAM: The following entities are generated for each defined ΦMACAM (Figure 4.28) for storing *mandatory-* and *attribute-based* constraints. Therefore, as the *authorisation type* of ΦMACAM and ΦMAC are the same, only the name of entities and in case of user and session entities, their properties' names are different. However, they will be used to store the ΦMACAM-based authorisation rights the same way as ΦMAC (discussed previously).


Mandatory- and attribute-based administrative constraints
Extending Application's Data Model

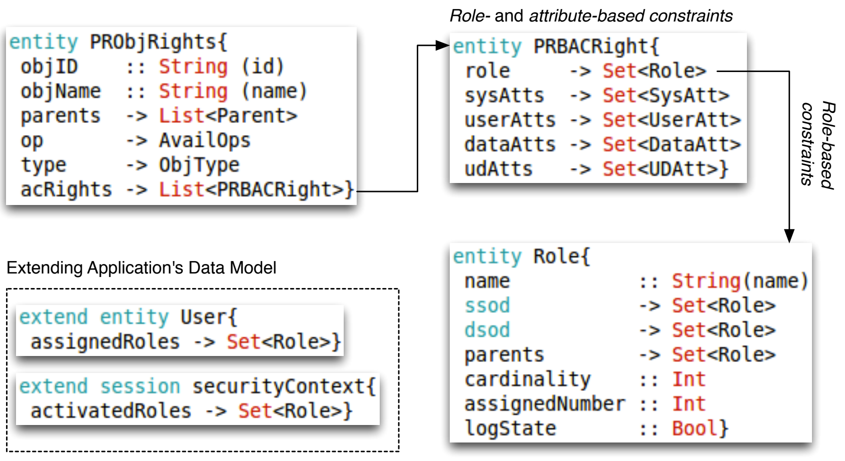
```
entity PMAMObjRights{
 objID    :: String(id)
 objName  :: String(name)
 parents  -> List<Parent>
 op       -> AvailOps
 type     -> ObjType
 acRights -> List<PAdMACRight>}

entity PAdMACRight{
 seclabel  -> Set<AdSecLabel>
 intLabel  -> Set<AdIntLabel>
 sysAtts   -> Set<SysAtt>
 userAtts  -> Set<UserAtt>
 dataAtts  -> Set<DataAtt>
 udAtts    -> Set<UDAtt>}

extend session securityContext{
 activatedAdSecLevel  -> SecLabel
 activatedAdIntLevel  -> IntLabel}

extend entity User{
 assignedAdSecLevel  -> SecLabel
 assignedAdIntLevel  -> IntLabel}

entity AdSecLabel{
 name :: String (name)
 classLevel :: Int
 logState   :: Bool}

entity AdIntLabel{
 name :: String (name)
 classLevel :: Int
 logState   :: Bool}
```

Figure 4.28: Generated Data model for the defined ΦMACAM model

4.3.1.4 Role- and Attribute-based Models

As discussed before, the ΦRBAC and ΦRBACAM models are *role-* and *attribute-based* models. Therefore the data model that needs to be generated and weaved for these models need to stored *role-* and *attribute-based* constraints. The following two parts discusses the generated data model elements for the ΦRBAC and ΦRBACAM models.

ΦRBAC: For the defined ΦRBAC model, the following entities (See Figure 4.29) are generated:


Role- and attribute-based constraints
Role-based constraints
Extending Application's Data Model

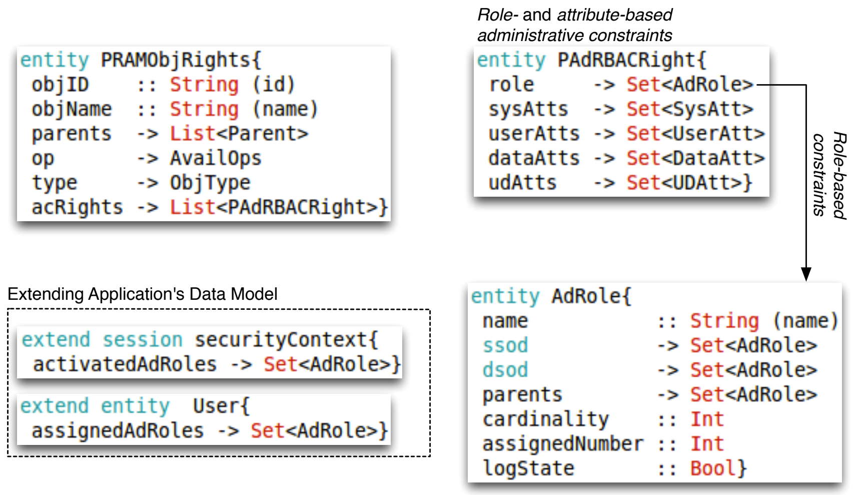
```
entity PRObjRights{
 objID    :: String (id)
 objName  :: String (name)
 parents  -> List<Parent>
 op       -> AvailOps
 type     -> ObjType
 acRights -> List<PRBACRight>}

entity PRBACRight{
 role      -> Set<Role>
 sysAtts   -> Set<SysAtt>
 userAtts  -> Set<UserAtt>
 dataAtts  -> Set<DataAtt>
 udAtts    -> Set<UDAtt>}

extend entity User{
 assignedRoles -> Set<Role>}

extend session securityContext{
 activatedRoles -> Set<Role>}

entity Role{
 name              :: String(name)
 ssod              -> Set<Role>
 dsod              -> Set<Role>
 parents           -> Set<Role>
 cardinality       :: Int
 assignedNumber    :: Int
 logState          :: Bool}
```

Figure 4.29: Generated Data model for the defined ΦRBAC model

- *PRObjRights:* This entity contains all the ΦRBAC-based constraints for each defined controlled objects. Similar to `PDObjRights` and `PMObjRights` entities in ΦDAC and ΦMAC models, it contains a set of properties that hold the general information about the controlled objects, such as name (i.e., `objName`), as well as a list of related *role-* and *attribute-based* constraints, through `acRights` property and `PRBACRight` entity.

- *PRBACRight:* This entity is generated to hold instances of ΦRBAC-based authorisation rights. It holds the set of required *roles* through `role` property, and the *attribute-based* constraints (i.e., *user-*, *data-* and *system-based attributes*) with the rest of its properties.
- *Role:* The entity `Role` (shown in Figure 4.29) is generated to hold the information of the defined roles and their relations (i.e., *hierarchy*, *SSOD*, and *DSOD*). Moreover it consists of the property `cardinality` and `assignedNumber` for storing the cardinality of each role and the number of times a role is assigned to the users of the application. The `logState` property holds the logical representation of the related policy sign for each role-based policy term.
- *Extension on the User and Session Data Model:* In ΦRBAC, similar to the ΦMAC model, the entity that represents the users of the system is extended (see Figure 4.29), for *role assignments and activation*. This is used for storing the the users of the system can *activate* their roles after their authentication. Moreover, the *session* entity is extended to store the users' activated roles during the run-time of the system (see Figure 4.29).

ΦRBACAM: Figure 4.30 shows the generated entities for each defined ΦRBACAM model to store the *role-* and *attribute-based* constraints of such models. The ΦRBACAM's *authorisation type* is the same as ΦRBAC model. Therefore, in principal it contains the same set of entities and properties which are *generated* for the ΦRBAC model (see Figure 4.29). However, these generated elements make it possible to separate between the ΦRBAC and ΦRBACAM defined authorisation rights. Please refer to the previous part to see the discussion on the generated entities and properties for holding the *role-* and *attribute-based* constraints.


Role- and attribute-based administrative constraints
Role-based constraints
Extending Application's Data Model

```
entity PRAMObjRights{
 objID    :: String (id)
 objName  :: String (name)
 parents  -> List<Parent>
 op       -> AvailOps
 type     -> ObjType
 acRights -> List<PAdRBACRight>}
```

```
entity PAdRBACRight{
 role      -> Set<AdRole>
 sysAtts   -> Set<SysAtt>
 userAtts  -> Set<UserAtt>
 dataAtts  -> Set<DataAtt>
 udAtts    -> Set<UDAtt>}
```

```
extend session securityContext{
 activatedAdRoles -> Set<AdRole>}

extend entity  User{
 assignedAdRoles -> Set<AdRole>}
```

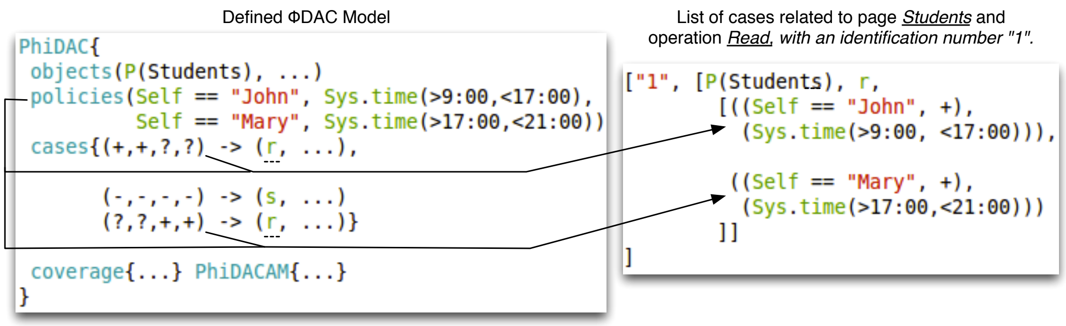
```
entity AdRole{
 name           :: String (name)
 ssod           -> Set<AdRole>
 dsod           -> Set<AdRole>
 parents        -> Set<AdRole>
 cardinality    :: Int
 assignedNumber :: Int
 logState       :: Bool}
```

Figure 4.30: Generated Data model for the defined ΦRBACDM model

4.3.2 Sorting the defined Φ Models

After generating and weaving the data model and Φ-based data model variants, for the correctness of the approach and avoiding the repetition within the generated architecture, Φ needs to sort the defined Φ-based models; before generating the rest of the architecture.

Syntactically, Φ-based models designed in such a way that the list of the objects and policies are factorized from the list of the policy cases. Even though, each defined policy in access or authorisation management models is unique, but policies are interconnected in terms of the object and operation they are protecting. For example, as shown in Figure 4.31, for the controlled object page `Students`, and `read` operation, the first and third case are factorized and created a list of two related cases. Moreover, a *unique identifier* is given to this list. This identifier later is used for *retrieving* the access control rights of an object and its related operation.


Defined ΦDAC Model
List of cases related to page *Students* and operation *Read, with an identification number "1".*

```
PhiDAC{
 objects(P(Students), ...)
 policies(Self == "John", Sys.time(>9:00,<17:00),
          Self == "Mary", Sys.time(>17:00,<21:00))
 cases{(+,+,?,?) -> (r, ...),

       (-,-,-,-) -> (s, ...)
       (?,?,+,+) -> (r, ...)}

 coverage{...} PhiDACAM{...}
}
```

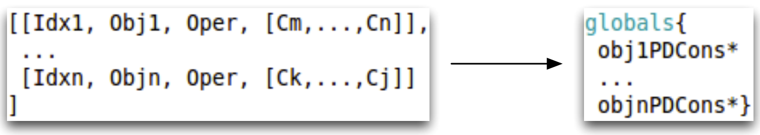
```
["1", [P(Students), r,
      [((Self == "John", +),
        (Sys.time(>9:00, <17:00))),

       ((Self == "Mary", +),
        (Sys.time(>17:00,<21:00)))
      ]]
]
```

Figure 4.31: Sorting policy cases based on the *type* of the object and its related operation, and assign a unique identifier to the list

Therefore, in this step, all the defined policy cases within the Φ-based access control models, are sorted based (through uppermost `PhiSort` strategy) on each defined *controlled object* and data manipulative (i.e., *create*, *read*, *update*, and *delete*) or *secret* operations.

4.3.3 Transforming Access and Authorisation Management Rights

After generating the data model elements and enumeration types, Φ, through uppermost `PhiGWRights` strategy, generates a set of *global variables*, to store the initial access and authorisation management constraints, based on the defined Φ-based models. As Figure 4.32 shows, Φ gets the first step results: a set of *objects*, their *operation*, related *policy cases* and their identifiers; and transforms them into a set of global variables. Note that, after the Φ compilation, WebDSL compiler generates the database, based on the defined and generated (i.e., in case of Φ) data models, and populates it with the defined global variables. Therefore this architecture provides a *dynamic* approach for authorisation *enforcement* and *management*.



```
[[Idx1, Obj1, Oper, [Cm,...,Cn]],
 ...
 [Idxn, Objn, Oper, [Ck,...,Cj]]
]
```

```
globals{
 obj1PDCons*
 ...
 objnPDCons*}
```

Figure 4.32: *Global variables* are generated based on list of sorted policy cases.

Each element within the sorted policy case is transformed to a set of global variables, as following:

Authorisation Rights: Each case related to each *object*, *operation* relation, is transferred to an instances of an entity (i.e., based on their type) as following:

- *User Identifier:* As the following equation shows, each *user identifier* and its related *logical sign*, is transformed to an `UIdent` entity.

$$\texttt{UItoGV(Self} == \mathit{Var}, \mathit{LS}) => \begin{cases} \texttt{var } \mathit{usr1} : \texttt{User} := \texttt{User}\{\ \texttt{user} := V \\ \quad \texttt{logState} := \texttt{true}\}\ \text{ if } \mathit{LS} := + \\ \texttt{var } \mathit{usr1} : \texttt{User} := \texttt{User}\{\ \texttt{user} := V \\ \quad \texttt{logState} := \texttt{false}\}\ \text{if } \mathit{LS} := - \end{cases} \tag{4.2}$$

- *User Attribute:* Each user attribute is transformed to an instance of `UserAtt` entity, based on the *policy term* and its related *policy sign*, as shown in Figure 4.33.
- *Data Attribute:* Similar to the user attributes (Figure 4.33 and 4.34); the instances of the data-based attributes are stored within the `DataAtt` entity.
- *System-based Attributes:* System-based attributes are based on *time* and *date* constrains. The last two points already pointed out how Φ generates the *user* and *data attributes*, and *user/data* attribute-based comparisons. As mentioned, the system-based attributes are stored within the `SysAtt` entities. Identical to part *A.1* and *B.1* the each *time* and *date* constraint is transfered to a *global variable* of type `TBound` (i.e., in case of *time constraints*), or `DBound` (i.e., in case of *data constraints*). Then the instance with its related *logical sign* will be a part of the `SysAtt` entity.

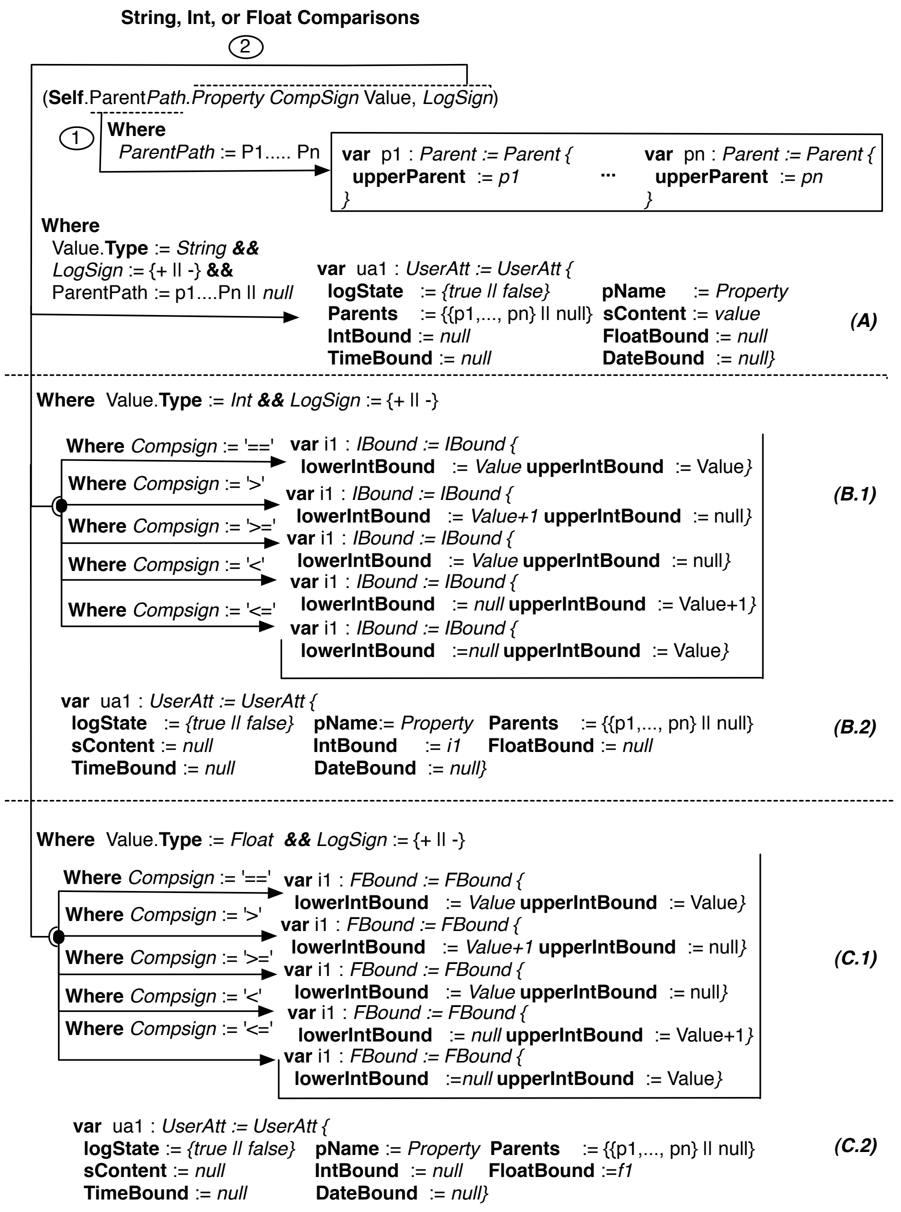

String, Int, or Float Comparisons
2
(Self.ParentPath.Property CompSign Value, LogSign)
1
Where ParentPath := P1..... Pn
var p1 : Parent := Parent { upperParent := p1 }
...
var pn : Parent := Parent { upperParent := pn }
Where Value.Type := String && LogSign := {+ || -} && ParentPath := p1....Pn || null
var ua1 : UserAtt := UserAtt { logState := {true || false} pName := Property Parents := {{p1,..., pn} || null} sContent := value IntBound := null FloatBound := null TimeBound := null DateBound := null}
(A)
Where Value.Type := Int && LogSign := {+ || -}
Where Compsign := '=='
Where Compsign := '>'
Where Compsign := '>='
Where Compsign := '<'
Where Compsign := '<='
var i1 : IBound := IBound { lowerIntBound := Value upperIntBound := Value}
var i1 : IBound := IBound { lowerIntBound := Value+1 upperIntBound := null}
var i1 : IBound := IBound { lowerIntBound := Value upperIntBound := null}
var i1 : IBound := IBound { lowerIntBound := null upperIntBound := Value+1}
var i1 : IBound := IBound { lowerIntBound :=null upperIntBound := Value}
(B.1)
var ua1 : UserAtt := UserAtt { logState := {true || false} pName:= Property Parents := {{p1,..., pn} || null} sContent := null IntBound := i1 FloatBound := null TimeBound := null DateBound := null}
(B.2)
Where Value.Type := Float && LogSign := {+ || -}
Where Compsign := '=='
Where Compsign := '>'
Where Compsign := '>='
Where Compsign := '<'
Where Compsign := '<='
var i1 : FBound := FBound { lowerIntBound := Value upperIntBound := Value}
var i1 : FBound := FBound { lowerIntBound := Value+1 upperIntBound := null}
var i1 : FBound := FBound { lowerIntBound := Value upperIntBound := null}
var i1 : FBound := FBound { lowerIntBound := null upperIntBound := Value+1}
var i1 : FBound := FBound { lowerIntBound :=null upperIntBound := Value}
(C.1)
var ua1 : UserAtt := UserAtt { logState := {true || false} pName := Property Parents := {{p1,..., pn} || null} sContent := null IntBound := null FloatBound :=f1 TimeBound := null DateBound := null}
(C.2)

Figure 4.33: *Global variables* are generated based on a list of sorted user-based attribute policy cases

- *Labels:* All the sorted label-based policy terms related to labels are transferred by Φ, to a set of global variables of type `confLabel` and `IntLabel` (in case of ΦMAC), and `AdConfLabel` and `AdIntLable` (in case of ΦMACAM) entities.
- *Role and attribute-based:* All the sorted role-based policy terms, are transferred by Φ, to a set of global variables of type `Role` (in case of ΦRBAC) and `AdRole` (in case of ΦRBACAM) entities.

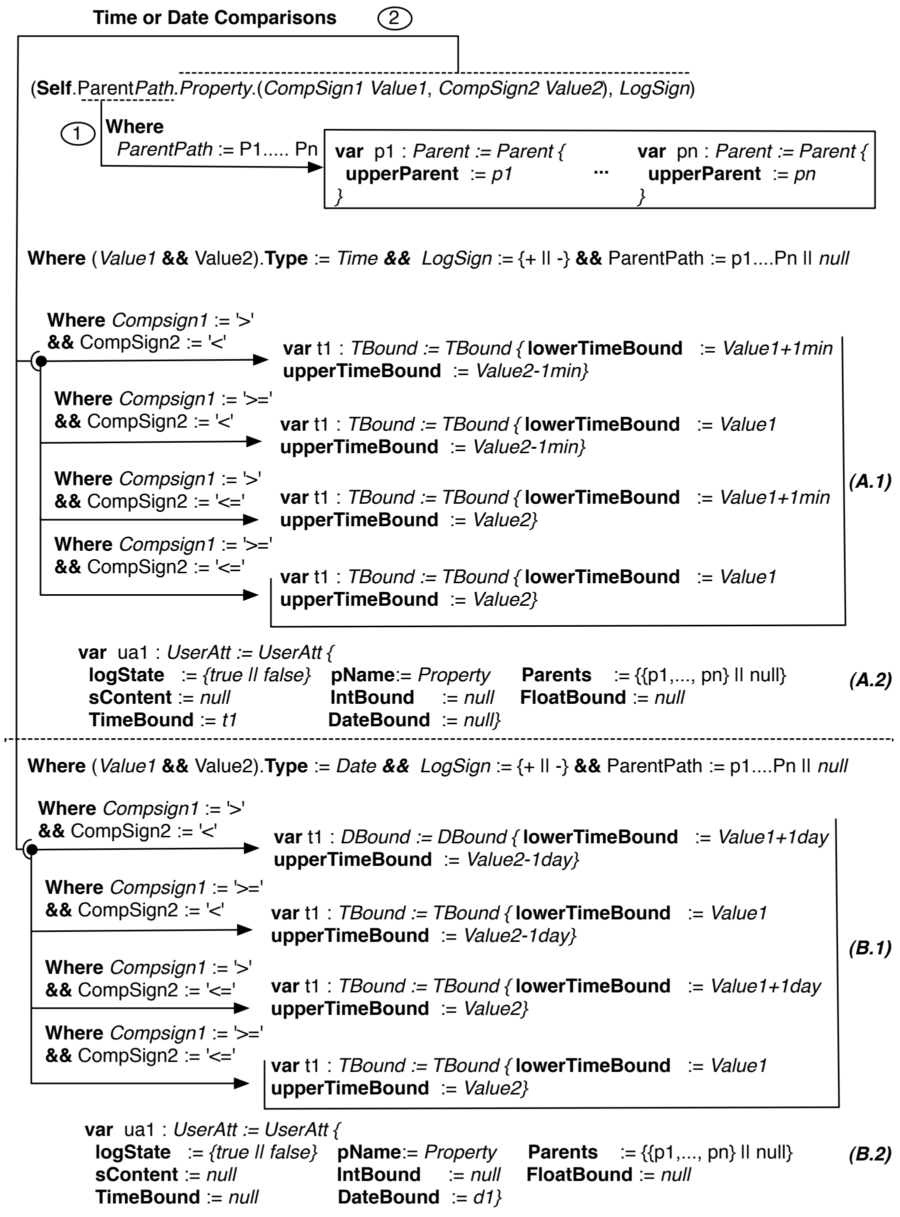

Time or Date Comparisons
2
(Self.ParentPath.Property.(CompSign1 Value1, CompSign2 Value2), LogSign)
1
Where
ParentPath := P1..... Pn
var p1 : Parent := Parent{ upperParent := p1 }
...
var pn : Parent := Parent{ upperParent := pn }
Where (Value1 && Value2).Type := Time && LogSign := {+ || -} && ParentPath := p1....Pn || null
Where Compsign1 := '>' && CompSign2 := '<'
var t1 : TBound := TBound{ lowerTimeBound := Value1+1min upperTimeBound := Value2-1min}
Where Compsign1 := '>=' && CompSign2 := '<'
var t1 : TBound := TBound{ lowerTimeBound := Value1 upperTimeBound := Value2-1min}
Where Compsign1 := '>' && CompSign2 := '<='
var t1 : TBound := TBound{ lowerTimeBound := Value1+1min upperTimeBound := Value2}
Where Compsign1 := '>=' && CompSign2 := '<='
var t1 : TBound := TBound{ lowerTimeBound := Value1 upperTimeBound := Value2}
(A.1)
var ua1 : UserAtt := UserAtt{ logState := {true || false} pName:= Property Parents := {{p1,..., pn} || null} sContent := null IntBound := null FloatBound := null TimeBound := t1 DateBound := null}
(A.2)
Where (Value1 && Value2).Type := Date && LogSign := {+ || -} && ParentPath := p1....Pn || null
Where Compsign1 := '>' && CompSign2 := '<'
var t1 : DBound := DBound{ lowerTimeBound := Value1+1day upperTimeBound := Value2-1day}
Where Compsign1 := '>=' && CompSign2 := '<'
var t1 : TBound := TBound{ lowerTimeBound := Value1 upperTimeBound := Value2-1day}
Where Compsign1 := '>' && CompSign2 := '<='
var t1 : TBound := TBound{ lowerTimeBound := Value1+1day upperTimeBound := Value2}
Where Compsign1 := '>=' && CompSign2 := '<='
var t1 : TBound := TBound{ lowerTimeBound := Value1 upperTimeBound := Value2}
(B.1)
var ua1 : UserAtt := UserAtt{ logState := {true || false} pName:= Property Parents := {{p1,..., pn} || null} sContent := null IntBound := null FloatBound := null TimeBound := null DateBound := d1}
(B.2)

Figure 4.34: *Global variables* are generated based on a list of sorted user-based attributes policy cases

Gathering Access Rights: After generating all the instances of a case, all instances need to be gathered to represent a case in the Φ models. For this Φ gathers each case in the ΦDAC and Φ DACAM model into `PDACRigt`, ΦMAC model into `PMACRight`, ΦMACAM into `PAdMACRight`, ΦRBAC into `PRBACRight`, and ΦRBACAM into `PAdRBACRight` entities.

Object Rights: At last, all the access and authorisation management rights related to the triple of (*object identifier*, *object* and its *operation*), with the information about the triples are transferred to a *global variable* of type `PDObjRights` (i.e., ΦDAC), `PDAMObjRights` (i.e., ΦDACAM), `PMObjRights` (i.e., ΦMAC), `PMAMObjRights` (i.e., ΦMACAM), `PRObjRights` (i.e., ΦRBAC), and `PRAMObjRights` (i.e., ΦRBACAM).

4.3.4 Predicate Handling Mechanisms

So far, Φ sorted, generated, weaved the instances of the access and authorisation management policies. At this point Φ through the uppermost strategy `PhiPHGW` generates a set of functions for checking the authorisation rights of each controlled object in both access and authorisation management models. Note that the calls for checking the predicates of the controlled objects are generated and weaved within the next step. The common points within all these functions are they all get the `objID` and the return value of type *Bool*. In this way, each function can find the object's authorisation rights based on its *id*, and return *True* (i.e., the access can be granted) or *False* (i.e., the access is not granted).

The following points discusses generated functions and its elements for each Φ-based models:

PDACCheck and PDACAMCheck: The *PDACCheck* and `PDACAMCheck` functions, are generated and weaved by Φ, based on the defined ΦDAC or ΦDACAM models. Both these functions contain five function calls for checking the discretionary- and/or attribute-based constraints. They are as following:

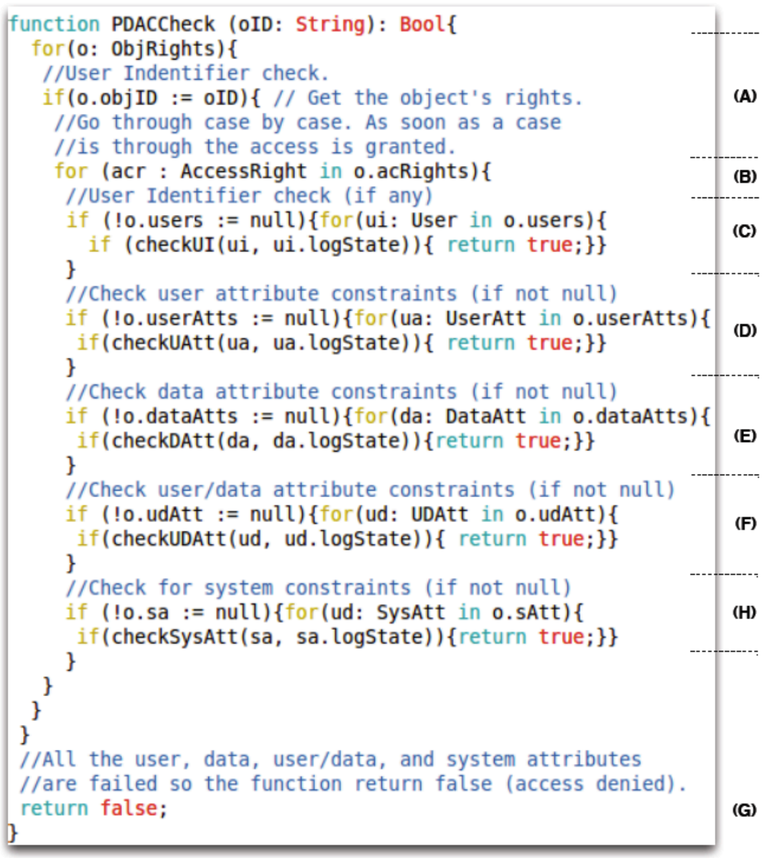
CheckUAtt: This function checks the *user attribute-based* policies by getting their instances and their logical state which were stored within the global variables of type `UserAtt` entity.

CheckDAtt: This function checks for the *data attribute-based* policies by getting their instances and their logical state, which were stored within the global variables of type `DataAtt` entity.

CheckUDAtt: This function checks for the *user and data attribute-based* comparisons by getting their instances and their logical state, which were stored within the global variables of type `UDAtt` entity.

CheckSAtt: This function checks for the *user and data attribute-based* comparisons by getting their instances and their logical state, which were stored within the global variables of tyoe `UDAtt` entity.

PDACChecks: This function gets the object identifier and check for the access control rights of the given controlled object (see Figure 4.35). Note that the calls for checking the predicates of the controlled objects are generated and weaved within the next step. First this function for the *user identifier* and then goes through the instances of the `ObjectRight`, find the related instance (based on the object identifier), and then check the following:



```
function PDACCheck (oID: String): Bool{
  for(o: ObjRights){
   //User Indentifier check.
   if(o.objID := oID){ // Get the object's rights.
    //Go through case by case. As soon as a case
    //is through the access is granted.
    for (acr : AccessRight in o.acRights){
     //User Identifier check (if any)
     if (!o.users := null){for(ui: User in o.users){
       if (checkUI(ui, ui.logState)){ return true;}}
     }
     //Check user attribute constraints (if not null)
     if (!o.userAtts := null){for(ua: UserAtt in o.userAtts){
      if(checkUAtt(ua, ua.logState)){ return true;}}
     }
     //Check data attribute constraints (if not null)
     if (!o.dataAtts := null){for(da: DataAtt in o.dataAtts){
      if(checkDAtt(da, da.logState)){return true;}}
     }
     //Check user/data attribute constraints (if not null)
     if (!o.udAtt := null){for(ud: UDAtt in o.udAtt){
      if(checkUDAtt(ud, ud.logState)){ return true;}}
     }
     //Check for system constraints (if not null)
     if (!o.sa := null){for(ud: SysAtt in o.sAtt){
      if(checkSysAtt(sa, sa.logState)){return true;}}
     }
    }
   }
  }
 //All the user, data, user/data, and system attributes
 //are failed so the function return false (access denied).
 return false;
}
```

Figure 4.35: The function `PDACCheck` checks the Φ*DAC-based* policies, by finding the object's rights and calling other functions to check the related *attribute-based* policies.

- *User Identifier:* For this, this function calls the `checkUI` function and passes an instances of the a property *users* and its logical sign, if its instances are not *null* (see part *C* in Figure 4.35).
- *User Attribute:* Then as Figure 4.35 shows (see part *D*), if the property `userAtts` is not *null*, then the function passes an instances of this property and its logical sign to the `checkUAAtt` function.
- *Data Attribute:* As Figure 4.35 shows (part *E*) this function calls the function `checkDAtt`, an passes an instances of it with its logical sign, to check the correctness of the data attributes.
- *User Data Attribute-based comparisons:* Then, as Figure 4.35 shows (see part *F*), this function first check if the property `UDAtt` is not *null*. Then it calls the function `checkUDAtt` and sends an instances of this with its logical sign to see its correctness, if it is correct it returns `true`.
- *System-based Attributes:* As Figure 4.35 shows (see part *H*), the function first checks if the property `systemAtt` is not *null*. If not, it calls the function `checkSysAtt`, and sends an instance of this property with its related logical sign. If the `checkSysAtt` returns *true*, then this function returns *true*.

PMACCheck and PMACAMCheck: The *PMACCheck* and `PMACAMCheck` functions, are generated and weaved by Φ, based on the defined ΦMAC or ΦMACAM models. Both these functions contain four function calls for attribute-based constraints and one function call to check its mandatory-based requirements. The attribute-based function call are identical to the previous point, however the function call `checkLabel`, checks the current user's activated confidentiality and/or integrity labels in regard to the access right for the controlled object. So, both for both `PMACCheck` and `PMACAMCheck` functions, if all the function calls are true then in this case they return *True* (i.e., grand access), and if not they return *False* (i.e., no access must be granted).

PRBACCheck and PRBACAMCheck: These two functions, and their generated function calls and their related functions are generated for the ΦRBAC and ΦRBACAM models. They consists of five function call, and similar to the previous functions four check for the user, data, and system attributes. Moreover, these functions call a function `roleCheck`, in which the current user activated role is checked against the role-based authorisations of the controlled object. If all these functions calls are true, then the `PRBACCheck` and `PRBACAMCheck` functions return *True* (i.e., access granted), or *false* (i.e., access must be denied).

4.3.5 Access Control Predicates

Within the last step, the predicates are generated and weaved around the *controlled object* and their related *operations*. Here Φ generates and weaves (i.e., through uppermost `PhiACPredGW` strategy) the *access control predicates* that guards the controlled objects and their related operations by calling the predicate handling functions (i.e., previous step) and sends the object identifier as an *input* parameter.

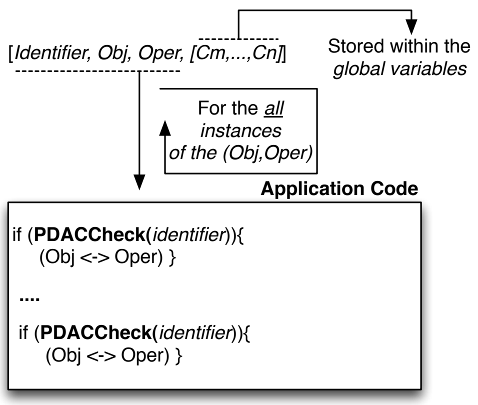
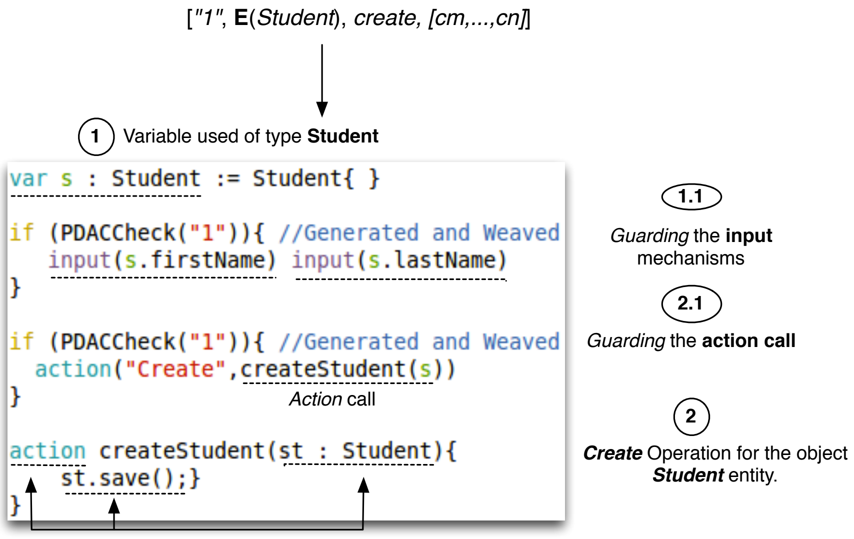

[Identifier, Obj, Oper, [Cm,...,Cn]]
Stored within the global variables
For the all instances of the (Obj,Oper)
Application Code
if (PDACCheck(identifier)){
(Obj <-> Oper) }
....
if (PDACCheck(identifier)){
(Obj <-> Oper) }

Figure 4.36: Simple illustration on weaving a function call (i.e., *PDACCheck*) around *all* the instances of the controlled object and its related operations

The rest of this part explains this generation and weaving step, based on the object and operation types, sorted from fine-grained to coarse grained, as following:

Entities and Properties: This part discusses weaving the predicate function call, based on the *entities* and *properties* and related operations, as shown below:

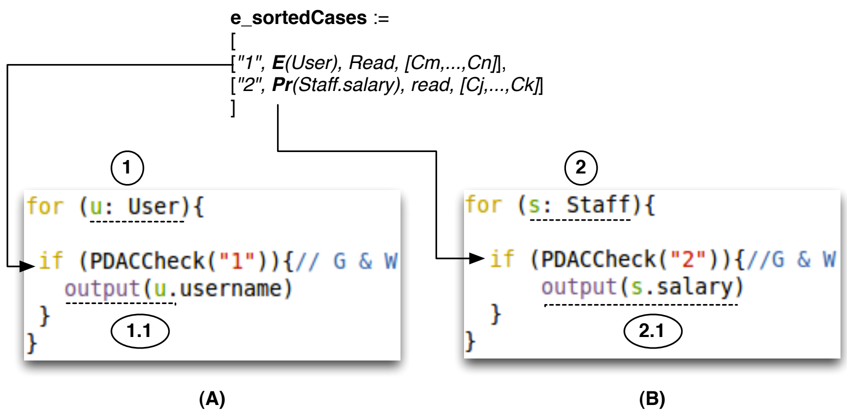
- *Create:* Within the WebDSL language the developer can *create* an entity. For guarding the entity and its related *create* operation, as Figure 4.37 shows, an *if statement* with the function call is generated and weaved around the entity `Student` and its related *create* operation. The if statement does not contain the *else* element, because the *unauthorised* user cannot see the operation (i.e., guard by the access control predicate) and does not need to know about the missing operation.


["1", E(Student), create, [cm,...,cn]]
1 Variable used of type Student
1.1 Guarding the input mechanisms
2.1 Guarding the action call
Action call
2 Create Operation for the object Student entity.

```
var s : Student := Student{ }

if (PDACCheck("1")){ //Generated and Weaved
   input(s.firstName) input(s.lastName)
}

if (PDACCheck("1")){ //Generated and Weaved
  action("Create",createStudent(s))
}

action createStudent(st : Student){
    st.save();}
}
```

Figure 4.37: Guarding the entity `Student` and its *create* operation by generating the function calls and weaving them around its input and action call

- *Read or Secret:* Within the WebDSL language, the element `output` is used to display the entities and their properties. For this as Figure 4.38 shows, Φ generate and weave the guards around *output* elements, that displays the instances of entity `User` (part A), or property `Staff.salary` (part B).

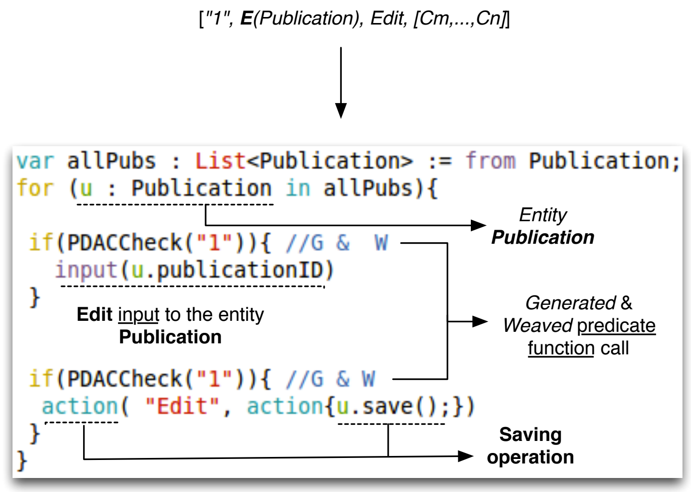

e_sortedCases :=
[
["1", E(User), Read, [Cm,...,Cn]],
["2", Pr(Staff.salary), read, [Cj,...,Ck]]
]
1
1.1
2
2.1
(A)
(B)

```
for (u: User){
  if (PDACCheck("1")){// G & W
    output(u.username)
  }
}
```

```
for (s: Staff){
  if (PDACCheck("2")){//G & W
    output(s.salary)
  }
}
```

Figure 4.38: Guarding the entity `User` (part A), and property `Staff.salary` and their related *output* elements

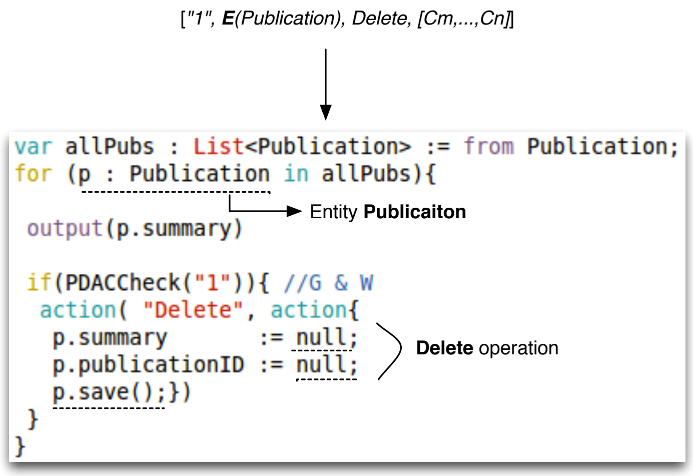
- *Update:* As Figure 4.39 shows, Φ generates and weaves the predicate function call (i.e., `PDACCheck`) around the *input* mechanism of the *properties* of an entity and their *action* code.


["1", E(Publication), Edit, [Cm,...,Cn]]
Entity Publication
Edit input to the entity Publication
Generated & Weaved predicate function call
Saving operation

```
var allPubs : List<Publication> := from Publication;
for (u : Publication in allPubs){
 if(PDACCheck("1")){ //G & W
   input(u.publicationID)
 }
 if(PDACCheck("1")){ //G & W
  action( "Edit", action{u.save();})
 }
}
```

Figure 4.39: Guarding the *edit* input mechanism and its related *action*, by generating and weaving predicate function call around them

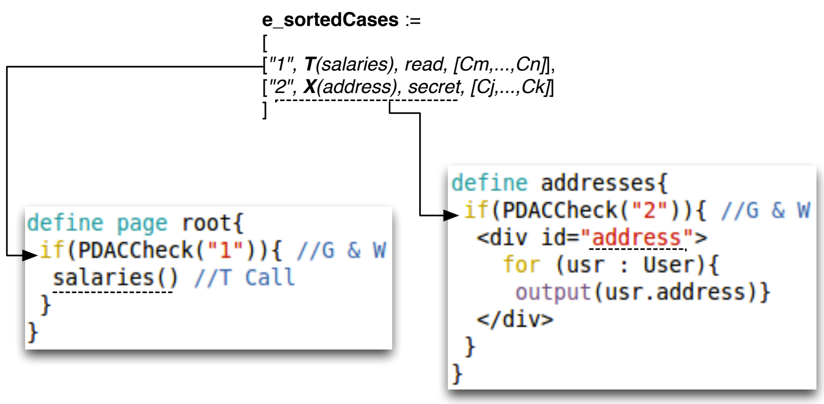
- *Delete:* Similar to the create and edit operations, for the *delete* operation related to a control object of type *entity*, Φ generates and guards the delete action (see Figure 4.40). Note that the assumption here is, an action that stores `null` for the instances of an entity is in fact the delete operation. This way there is a distinction between the *update* and *delete* properties.



```
var allPubs : List<Publication> := from Publication;
for (p : Publication in allPubs){
 output(p.summary)

 if(PDACCheck("1")){ //G & W
  action( "Delete", action{
   p.summary       := null;
   p.publicationID := null;
   p.save();})
 }
}
```

Figure 4.40: Guarding the *delete* operation related to the `Publication` entity

Template and XML: For the *Template* and *XML* objects, only the *read* and *secret* operation can be used. As Figure 4.41 shows, similar to guarding the *entities* and *properties* and their related *read* or *secret* operation, Φ generates the function call `PDACCheck`, and weaves them around the *template calls* and *XML nodes*.



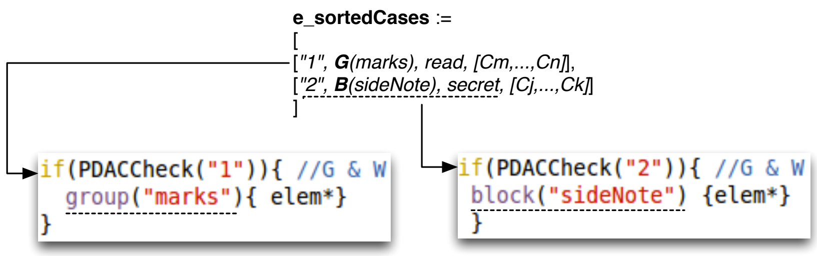
```
e_sortedCases :=
[
["1", T(salaries), read, [Cm,...,Cn]],
["2", X(address), secret, [Cj,...,Ck]]
]
```

```
define page root{
 if(PDACCheck("1")){ //G & W
  salaries() //T Call
 }
}
```

```
define addresses{
 if(PDACCheck("2")){ //G & W
  <div id="address">
    for (usr : User){
     output(usr.address)}
  </div>
 }
}
```

Figure 4.41: Guarding the the objects `salaries` template call and XML node `address` and their related *read* and *secret* operation

Group, and Block: Identical to the Template and XML objects, for guarding the *Group* and *Block* objects and their *read* or *secret* operation; Φ generates and weaves the `PDACCheck` function call around them as shown in Figure 4.42.



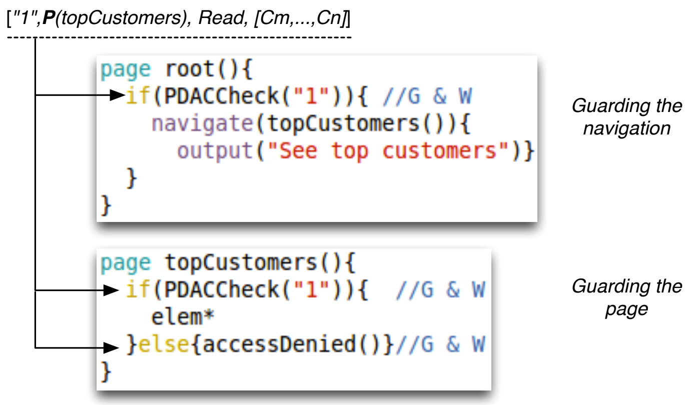
```
e_sortedCases :=
[
["1", G(marks), read, [Cm,...,Cn]],
["2", B(sideNote), secret, [Cj,...,Ck]]
]
```

```
if(PDACCheck("1")){ //G & W
  group("marks"){ elem*}
}
```

```
if(PDACCheck("2")){ //G & W
 block("sideNote") {elem*}
}
```

Figure 4.42: Guarding the the objects `marks` group and block `sideNote` and their related *read* and *secret* operation.

Page: For the controlled objects of type *page* and their related *read* and *secret* operation, Φ generated the function call to the related predicate handling function (e.g., `PDACCheck`) and weaved around the elements of the pages and their navigation links throughout the Web application code, as shown in Figure 4.43.



```
["1",P(topCustomers), Read, [Cm,...,Cn]]
```

```
page root(){
 if(PDACCheck("1")){ //G & W
   navigate(topCustomers()){
     output("See top customers")}
 }
}
```

```
page topCustomers(){
 if(PDACCheck("1")){  //G & W
   elem*
 }else{accessDenied()}//G & W
}
```

Figure 4.43: Φ guards the the page `topCustomers` and its navigation link, by the generating and weaving the function call.

4.3.6 Generating and Weaving Φ Panel

As Figure 4.22 shows, within the last transformation step, Φ (i.e., through upper-most `PhiPanelGW` strategy) enables the target Web application to have a Φ *panel* with the following functionalities:

Access Rights Activation: It enables the users of the application to activate a subset of their assigned *labels* and/or *roles* for the defined ΦMAC and ΦRBAC models. The ΦDAC constraints do not need to be activated as they are based on each *individual user* (i.e., discretionary-based constraints) or *attribute-based* constraints (i.e., user-, data- and system-based attributes).

Authorisation Management Rights Activation: Φ enables the users with the administration rights to activate a subset of their administrative rights by activating their assigned *labels* and/or *roles* based on the defined Φ*RBACAM* or Φ*MACAM* models. Similar to the ΦDAC models, for the administrative rights related to the defined ΦDACAMs models, no activation is required as the authorisation management rights are based on discretionary- and attribute-based constraints.

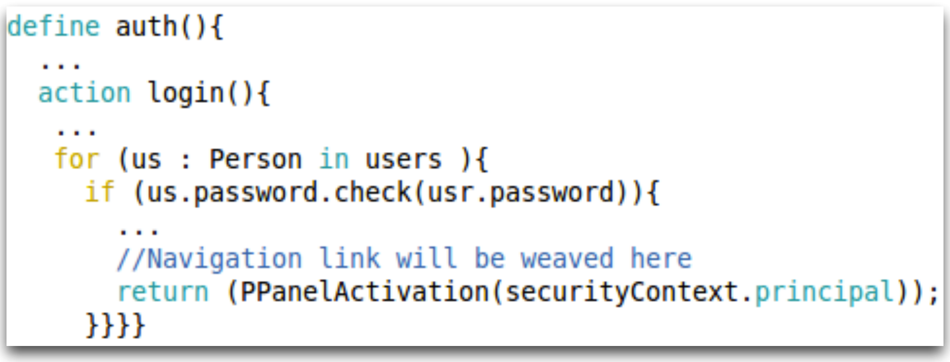
Access Control Assignment: When the authorised users of the *authorisation management* models activate a subset of their administrative rights, they can assign the related defined access control models to the users of the system.

The rest of this part discusses the mentioned functionalities of Φ panel, that are generated and weaved by Φ in more details.

4.3.6.1 Access Control and Authorisation Management Activations

The non-admin users of the system must be able to activate their access control rights after their authentication. Also, the admins of the application must be able to activate their administrative rights. For these functionalities the following elements are generated and weaved by the Φ (i.e., `PhiPanelGW` strategy):

Authentication Modification: In an application without access control and authorisation management mechanism, the application redirects the users' *successful* authentications to a page (e.g., `root page`). However, here the users need to get redirected to a page to *activate* their access or authorisation management rights. For this, as Figure 4.44 illustrates, Φ adds a navigation to the `PPanelActivation` page, by generating it and weaving it to the authentication template code [22]. So, as Figure 4.45 shows (i.e., part A), this is the entrance point for the Web applications that have defined Φ-based models.



```
define auth(){
 ...
 action login(){
  ...
  for (us : Person in users ){
    if (us.password.check(usr.password)){
      ...
      //Navigation link will be weaved here
      return (PPanelActivation(securityContext.principal));
    }}}}
```

Figure 4.44: Φ Modifies authentication template by adding a navigation

Authorisation Activation Page: The page `PPanelActivation` is generated and weaved by Φ (see part B in Figure 4.45) that provides the following functionalities to the users of the application.

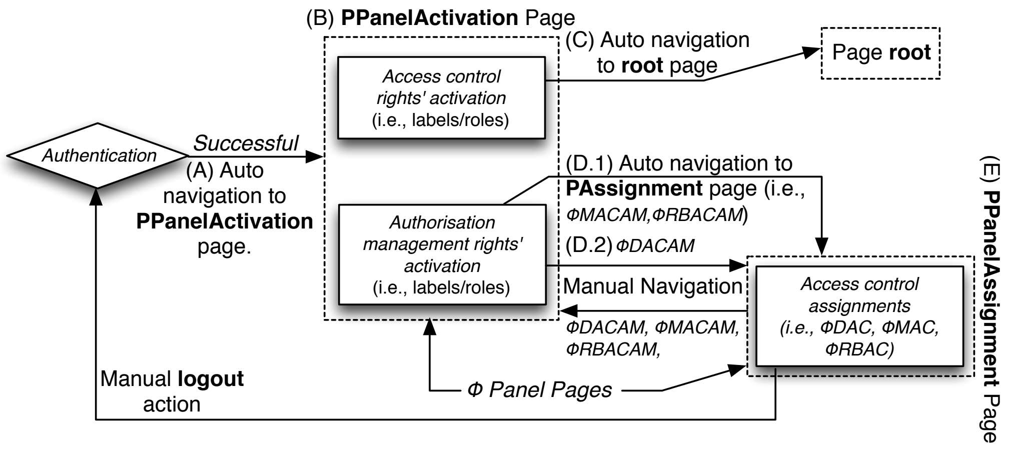

(B) **PPanelActivation** Page
(C) Auto navigation to **root** page
Page **root**
Access control rights' activation (i.e., labels/roles)
Authentication
Successful
(A) Auto navigation to **PPanelActivation** page.
(D.1) Auto navigation to **PAssignment** page (i.e., *ΦMACAM,ΦRBACAM*)
(D.2) *ΦDACAM*
Authorisation management rights' activation (i.e., labels/roles)
Manual Navigation
ΦDACAM, ΦMACAM, ΦRBACAM,
Access control assignments (i.e., ΦDAC, ΦMAC, ΦRBAC)
(E) **PPanelAssignment** Page
Manual **logout** action
Φ Panel Pages

Figure 4.45: Φ Panel Workflow

Access Control Activation. Users of the system must activate a subset of their rights through activating a set of *labels* or *roles* that are assigned to them through defined Φ-based access control models, or through authorisation management mechanisms that are generated by Φ. After activating their labels or roles users are automatically redirected to the `root` page (i.e., part C in Figure 4.45). Note that, in this way the activities related to the users with non-admin rights are separated than the users with the admin related rights.

Authorisation Management Activations. Similar to the previous point, the users with the admin related rights can activate their *labels* (in case of defined ΦMACAM model) and/or *roles* (in case of defined ΦRBACAM model). After activation the users are automatically redirected to the `PPanelAssignment` page to assign the access control rights to the users of the system. For this as Figure 4.45 illustrates within this page, Φ shows these mechanisms in case the user has the administrative rights through saved authorisation management rights, that are based on defined ΦMACAM and the ΦRBACAM models. In case there is only a ΦDACAM mechanism, Φ generates a navigation link to the `PAssignment` page (i.e., see D.1 in Figure 4.45). This is because there might be a case in which the user has the *both* access and authorisation management rights, and therefore, the user needs to choose between two sets of functionalities. After manual or automated navigation the users with the administrative rights will assign the related access control rights through the `PPanelAssignment` page.

4.3.6.2 Authorisation Management Assignments

The admins of the system must be able to *assign* access control rights to the users of the system. For this, Φ generates and weaves the `PPanelAssignment` page that can provide a subset of the following functionalities based on the defined *Φ-based authorisation management models*, their relation to the *defined Φ-based access control model*, and the *current user's administrative rights*:

Reading Assigned Access Rights. The users with the sufficient administrative rights can *read all* the defined access rights through the first group that holds the *permissions* (i.e., relation between object and a relation) and their *access control rights*. Moreover, if the admin of system activates different types of authorisation models (e.g., ΦMACAM and ΦRBACAM), the Φ's generated and weaved code, groups the related authorisation managements based on *authorisation management rights*.

Creating Access Control Assignments. The users with the required administrative rights can create *new* assignments based on the *users of the system* and underlying access control model related to the defined authorisation management models. The second group (i.e., `'Create Access Control Assignments'`) within this page holds all the create operation for the current users based on their authorisation management rights. Although all the defined authorisation management models allow authorised users to create new assignments to the users of the system for their underlying access control model (i.e., ΦDAC, ΦMAC, and ΦRBAC); but the generated Φ code checks for the following constraints:

Label Conflicts: The authorised users can not assign a *permission* (i.e., control object and its related operations) to a user that has the conflict with the assigned label based on its original access control rights.

Role Conflicts: Similar to the previous point, the assignment is unsuccessful if the admin tries to *create* a new assignment to a user that has a conflict with the role based on the defined *RBAC* characteristics within the defined ΦRBAC model.

Updating and Deleting Access Control Assignments. Based on the semantics of the basic MAC models, ΦMACAM *cannot* `update` or `delete` its underlying access control models. Furthermore, no authorisation management model allows `update` or `delete` operations on their underlying ΦMAC model. Therefore for the defined ΦRBACAM or ΦDACAMs the authorised users *can* `update` or `delete` the access control assignments of their underlying access control if they are ΦDAC or ΦRBAC models. Moreover, similar to the *assignment creation*, the generated Φ checks for any role conflicts based on the access control assignment rights and the target users' *current assigned roles.*

4.4 WebDSL Extension

This part discusses how the Φ compiler, attached to the WebDSL compiler. For this, the syntax definition related Φ models, Phi.ini, and Z3 formulas were added, as an import module, to the WebDSL's main syntax file (i.e., `WebDSL.sdf`). So, this way after rebuilding the WebDSL compiler, these added syntax became part of the WebDSL's tree grammar, signature and parser.

For adding the Φ's transformation rules, a directory created and added to the languages directory that consists of the master file (i.e., `Phi.str`) with the uppermost strategy (i.e., `Phi-Start`) that first calls the uppermost strategy for validating and verifying Φ models (i.e., located in `PhiVV.str` file) and then if successful it calls uppermost strategy `PhiGW` (located in `transformation.str` file) for executing the transformation phase.

Moreover the WebDSL's compiler pipeline (`dsl-to-core.str`) got extended by adding an extra step, which calls the uppermost strategy `Phi-Start`. After the WebDSL's compiler is build, WebDSL is able to support the Φ models to developers.

4.5 Summary and Conclusion

This chapter discussed the concepts behind the Φ pipeline that was divided into testing and transformation phases, and how it was implemented.

There are ten consecutive steps in the validating and verification phase that are checking the structure of the defined access control elements, based on their semantics and their target language. These checks are written within the Φ compiler and designed for efficiency and cheapness to execute the validation and verification before transformation phase.

The Φ's transformation phase was divided into five consecutive steps, to generate and weave all the required elements for proving the Φ-based mechanisms.

Φ compiler was developed as a part of the WebDSL. For this purpose, the syntax definition of Φ's models, Phi.ini file, and related Z3 formulas and their results were defined and added as a part of the WebDSL's syntax definition. Then for the transformation phase, the Φ's transformation phase was imported to the WebDSL's list of languages. Furthermore the Φ's step was added to the WebDSL's pipeline as a new stage, after the *model-to-model* transformation stage.

Chapter 5

Case Studies and Φ Evaluation

5.1 Introduction

This chapter shows the evaluation of the Φ language in terms of defining fine-grained access control and authorisation management models, through two case studies:

Academic Research Group: This case study covers the functionalities that are typically found in the Web pages of an academic research group. It was designed and implemented to evaluate ΦRBAC and its role-based authorisation management system in a realistic medium-sized Web application.

Social Networking: This case study focuses on the social networking aspect of Web applications. It enables the registered users of the system to create an arbitrary number of networks from their list of related users, and perform some of the usual social networking functionalities (e.g., write a status). The aim of this case study is to evaluate Φ in terms of defining multiple access and authorisation management models for a single Web application.

Each case study consists of an underlying Web application[1] that is developed using WebDSL, and, as an extension, a separate Φ model to define the access control component for the Web application. Then Φ during the compile-time generate and weave the target code of Φ-based model (discussed in Section 4.3) into the WebDSL code. The aim of using different case studies is to evaluate Φ with respect to the following criteria:

Effectiveness and developer efficiency: It is important to show that the provided functionalities are effective in terms of access control and authorisation management models and mechanisms. I use white-box testing techniques for testing the access

[1]http://www.philang.org/repos/case-studies

control and authorisation management models and the application code (see Chapter 4 for more details). Therefore it is critical for us to show the fact that each step is correct based on Φ semantics and the defined access control model. As mentioned in the previous chapter, I verify and validate the defined access control and authorisation management models both in isolation and in combination with their target application code. Therefore there is a wide range of errors that could occur during the code compilation. These errors must be reported to the developer in terms of the defined Φ models. It is important that the compiler gives the error messages on the right abstraction level.

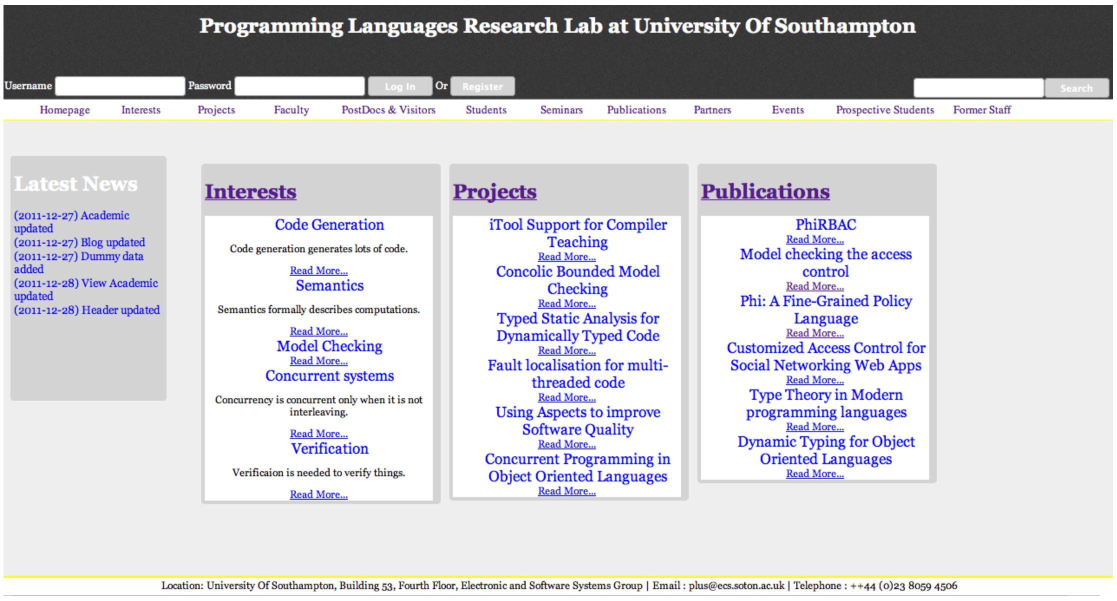
Ease of use: It is important to show that Φ has the right abstraction level and is easy to use during development phase; in particular, it should not add more complexity to the development. This is measured by the quantitative evaluation at the end of each case study.

I measure *separation of concerns*, *effectiveness*, and *ease of use* metrics computed from defined Φ models, their generated code, and the application characteristics (e.g., number of controlled objects, functionalities, etc.). For measuring the *correctness* and *error handing* of Φ I introduced a set of errors and checked if the errors can be discovered during the run-time by using Φ. The seeded errors are a subset of possible errors that are presented in Appendix B.

5.2 Academic Research Group

The first case study consists of a Web application for academics and other users (e.g., post graduates, post doctorates, etc.) in the Programming Languages Research Lab at University of Southampton (PLUS). I chose this case study because it is a real-world example where I are familiar with the intended policy regulations. It makes a good case study to evaluate Φ's role-based access control and authorisation management models (ΦRBAC and ΦRBACAM) because of its size (i.e., 22 entities and 198 properties over 16 pages) and crucial role of the *role-based* authorisation approach in an organisational-based system.

The rest of this section explains the application and its components in terms of its data model, page flows, access control, and statistical overview of the defined ΦRBAC model with respect to the generated access control and Web application's elements.



Programming Languages Research Lab at University Of Southampton

Latest News

(2011-12-27) Academic updated
(2011-12-27) Blog updated
(2011-12-27) Dummy data added
(2011-12-28) View Academic updated
(2011-12-28) Header updated

Interests

Code Generation
Code generation generates lots of code.
Read More...
Semantics
Semantics formally describes computations.
Read More...
Model Checking
Read More...
Concurrent systems
Concurrency is concurrent only when it is not interleaving.
Read More...
Verification
Verificaion is needed to verify things.
Read More...

Projects

iTool Support for Compiler Teaching
Read More...
Concolic Bounded Model Checking
Read More...
Typed Static Analysis for Dynamically Typed Code
Read More...
Fault localisation for multi-threaded code
Read More...
Using Aspects to improve Software Quality
Read More...
Concurrent Programming in Object Oriented Languages
Read More...

Publications

PhiRBAC
Read More...
Model checking the access control
Read More...
Phi: A Fine-Grained Policy Language
Read More...
Customized Access Control for Social Networking Web Apps
Read More...
Type Theory in Modern programming languages
Read More...
Dynamic Typing for Object Oriented Languages
Read More...

Location: University Of Southampton, Building 53, Fourth Floor, Electronic and Software Systems Group | Email : plus@ecs.soton.ac.uk | Telephone : ++44 (0)23 8059 4506

Figure 5.1: PLUS homepage

5.2.1 Data Model

The data model is divided into two categories of elements, *users* and *activities*. The full list of the data model entities, their properties and their descriptions is given at the Appendix C.

Users: The system needs to manage different activities for the different types of users in the application. The application code thus uses nine different entity types (see Figure 5.2). The type `Person` holds the general information regarding the users of the system, such as `username` and `lastname`. The remaining entities inherit from the `Person` entity. The entity `universityMember` is a super-type for the different types of university members such as `Academics`. It holds the general information regarding the university members, such as `universityID` and list of `Publications`. Other entities that inherit from `universityMember` entity store specific information about `academics`, `postdocs`, `postgraduates`, and project students (i.e., `MSc` and `BSc`) respectively. For example, the entity `Academic` holds the information about academics such as `biography` and his/her related postdocs and postgraduates. The entity `Partner` finally holds the information, such as `school name` or `duration` of collaboration about the external users of the system that collaborate with the users of PLUS.

Activities: There are a number of activities that are available to the users of the system, and the related entities store these activities' information. The entity `Event` is the super-type entity for the `internalEvent` and `Seminar` activities and events. It holds the general information about an event such as `Date` and `Time`. The `internalEvent` entity stores information about the any event occurring in the PLUS group, such

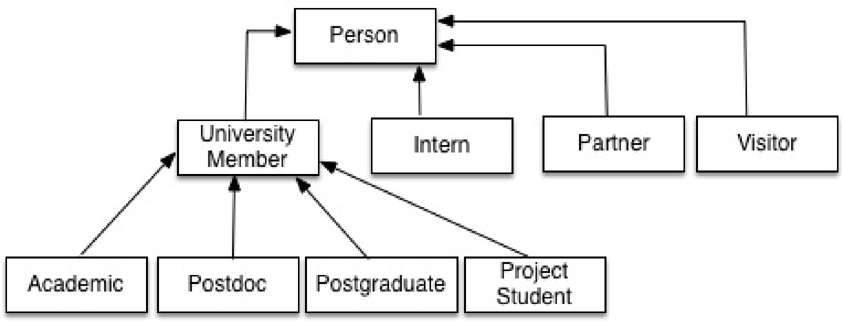


Figure 5.2: Different user types in the PLUS data model

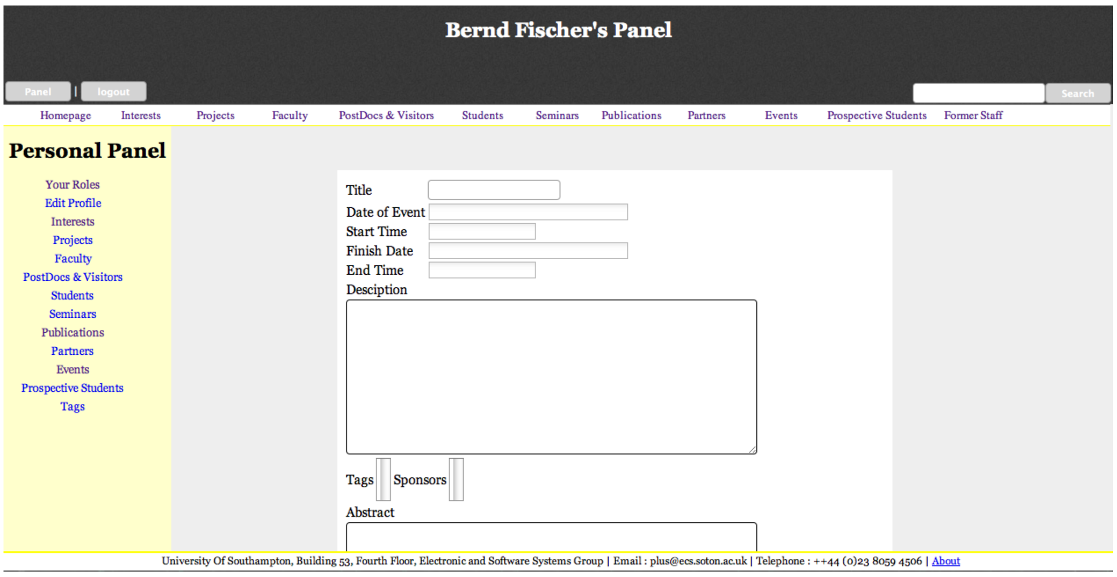
as the description of an open day for the research group. The `Seminar` entity stores information about a talk for the group, such as `speaker` information. There are nine other entities that hold the information about the other activities, such as `Publication` or `Blog`. I refer the reader to see Appendix C for these entities description.

5.2.2 Page Structures

This Web application consists of 20 main pages that can be categorised as *user-* and *activity-based* pages:

User-based: Any user of the system starts with the homepage. In this page the user can see a list of `interests`, `projects` and `publications`. Moreover the user can `register`, `login` (after registration), and `logout` (if logged in). In the `registration` page the user can enter his/her credentials (e.g., `username`, `firstname`) to be added to the system as a user. After registration, the user can then `login` to the system, to become an *internal user*. After successful authentication the user is then automatically redirected to the user panel page where can use the authorised operations. The other user-based pages are `Faculty` (shown in Figure 5.4), `PostDocs and Visitors`, `Students`, `Partners`, `Prospective Students` and `Former Staff`. Within each of these pages a list of users and the navigation to their profiles are shown. All activities related to each user are shown within his/her profile.

Activities-based: There are also a number of pages that reflect the activities related to the research group. They are (`Interests` (shown in Figure 5.5), `Projects`, `Publications`, `Seminars`, and `Events`. These pages list the related activity within the page, and contain a number of links for more related information. For example, as Figure 5.5 lists the interests within the group and `academics` and `projects`, and a navigation to the interests profile page.

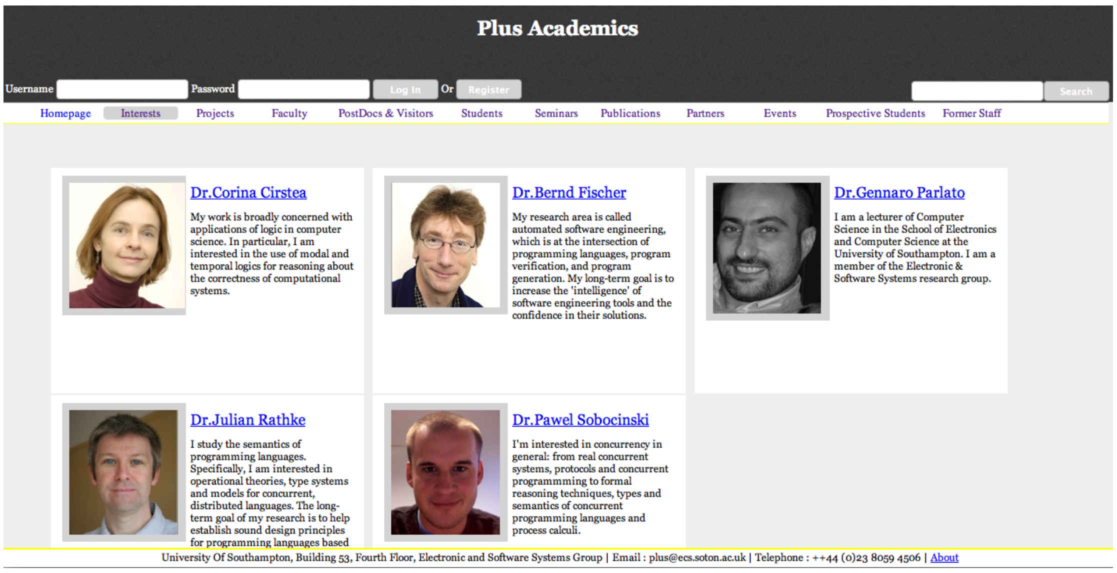


Personal Panel

Title
Date of Event
Start Time
Finish Date
End Time
Desciption
Tags Sponsors
Abstract

University Of Southampton, Building 53, Fourth Floor, Electronic and Software Systems Group | Email : plus@ecs.soton.ac.uk | Telephone : ++44 (0)23 8059 4506 | About

Figure 5.3: Adding an event through a user panel



Dr.Corina Cirstea

My work is broadly concerned with applications of logic in computer science. In particular, I am interested in the use of modal and temporal logics for reasoning about the correctness of computational systems.

Dr.Bernd Fischer

My research area is called automated software engineering, which is at the intersection of programming languages, program verification, and program generation. My long-term goal is to increase the 'intelligence' of software engineering tools and the confidence in their solutions.

Dr.Gennaro Parlato

I am a lecturer of Computer Science in the School of Electronics and Computer Science at the University of Southampton. I am a member of the Electronic & Software Systems research group.

Dr.Julian Rathke

I study the semantics of programming languages. Specifically, I am interested in operational theories, type systems and models for concurrent, distributed languages. The long-term goal of my research is to help establish sound design principles for programming languages based

Dr.Pawel Sobocinski

I'm interested in concurrency in general: from real concurrent systems, protocols and concurrent programmming to formal reasoning techniques, types and semantics of concurrent programming languages and process calculi.

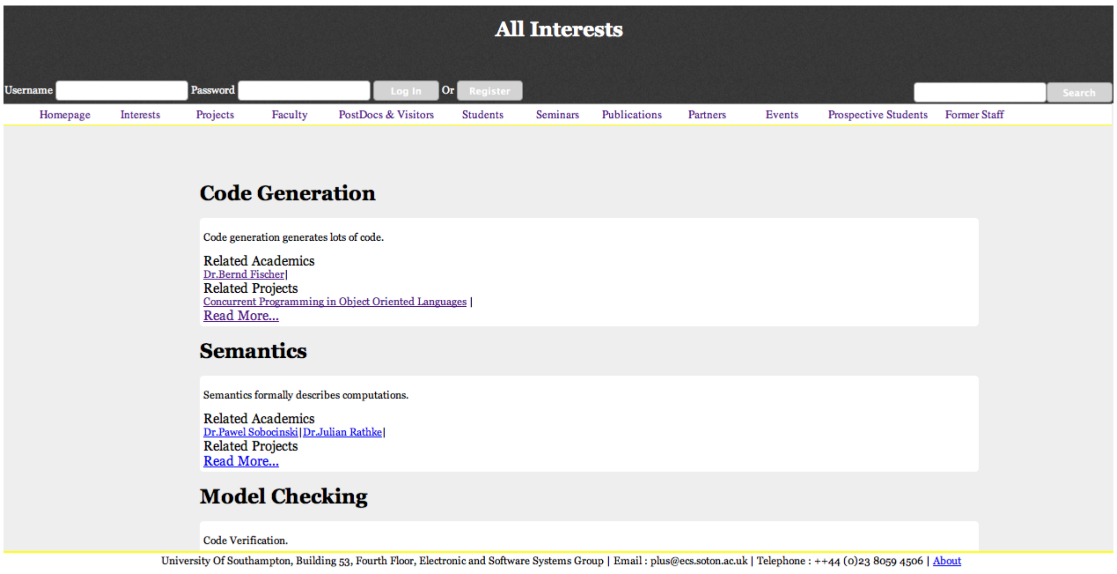
University Of Southampton, Building 53, Fourth Floor, Electronic and Software Systems Group | Email : plus@ecs.soton.ac.uk | Telephone : ++44 (0)23 8059 4506 | About

Figure 5.4: Academic page

Moreover, the users can manipulate the data from within their `profile` page. For example, as Figure 5.3 shows, the user can add an event to the system through his profile page. Furthermore as Figure 5.3 shows, based on the user authorisation rights he can do more operations on other aspects of the Web applications through his user panel.

5.2.3 Page flows and their related functionalities

This section discusses the functionalities that are available through the Web application pages. There following points are common in all the pages:



Code Generation

Code generation generates lots of code.

Related Academics
Dr.Bernd Fischer|
Related Projects
Concurrent Programming in Object Oriented Languages |
Read More...

Semantics

Semantics formally describes computations.

Related Academics
Dr.Pawel Sobocinski|Dr.Julian Rathke|
Related Projects
Read More...

Model Checking

Code Verification.

University Of Southampton, Building 53, Fourth Floor, Electronic and Software Systems Group | Email : plus@ecs.soton.ac.uk | Telephone : ++44 (0)23 8059 4506 | About

Figure 5.5: Interests page

Header Navigation Links: As it can be seen in Figures , authentication, registration, search, and other navigation links are part of the header. These links are present in all pages so that the users can navigate from one page to any other page. After *registration* the user can authenticate and go to his/her panel.

Search: There are two search features in this Web applications. First, a general search functionality is present on all the pages in the top right corner. In this case the user can search the content of the Web application and retrieve a set of results, based on the user's respective authorisation rights. There is also a specific "search event" mechanism that enables any user of the system to search for the events that were held or will be hold in the research group. Here, if the user is not registered, he/she can only see the general available events, but if she/he is registered, she can see the internal events such as seminars as well.

Pages and Functionalities. As mentioned, I have a header navigation link that redirects users to all the pages except the *user panel*. These pages, such as `interests` provide a general information about all the interests of the research group to the user. For example, the `Interest` entity has a `blurb` property. This property is used within the `Interests` page to give an idea about the group's interests. Furthermore, the user can navigate from the list of interests to the each interest page and see more information about the interest. This sort of flow was also implemented for the other entities of the Web application (e.g., `project`, `academic`, `publications`, etc.).

Pages and CRUD Operations. I do not have different pages for the `CRUD` operations of each entity and its properties. In terms of access rights and operations on the controlled objects I can divide all pages into `User Panel` and `others`. Within the `user panel` page the authorised user, can *create* and *delete* the controlled objects. However, for reading or editing the page the authorised user should navigate to the lists of objects at the same type. For example, an academic can *only* create a `project` when he/she is in his/her `user panel` page. However, to edit the description of the project, the academic needs to navigate to the `Projects` page. This is due to the fact that editing of the data should be coherent with the rest of the entries of the same type, so the user can see the effect of editing of an object based on itself and overall structure of other entries.

5.2.4 Defined ΦRBAC with ΦRBACAM authorisation models

This part of the thesis discuss the defined access and authorisation management models for the academic research group. Following this part I quantitatively evaluate the defined models with respect to the Web application.

Access Control Model

This case study is based on an organisation with a number of different roles and a hierarchy within the organisation, so that it is natural to define a ΦRBAC model. This part discusses the access control model based on the defined *roles*, their relations (e.g., *hierarchy*, SSOD) and the objects and operations they are protecting.

Roles and their relations: Figure 5.6 shows the used roles and the inheritance relation in the defined ΦRBAC access control model. In this model there are 14 roles divided into two main categories of *University Members* and *external members* (i.e., `Intern`, `Visitor`, `External Examiner`, and `Partner`). There are 6 roles that inherit from the `University Members` roles, such as `Secretary`. Furthermore, the are three roles that inherit the `Academic` role, such as `Supervisor`.

As Figure 5.7 shows, there are 30 *static separation of concerns* (SSOD) within the defined roles. As mentioned in Chapter 2 this relation means they can not be assigned to the same user *any time* during the system run-time. These relations separates the concerns within the university members and external users at all times.

Moreover, there are four *Dynamic Separation of Duty* (DSOD) relations between `Internal Examiner`, `Supervisor` and `Advisor`, `Academic` and `Head of Group`, `Advisor` and `Supervisor`.

Protected Objects: All the controlled objects in this case study are based on the defined entities and properties.

Policy Terms: There are 14 different policies for each roles with additional time attribute to enforce users to use the system during the office hours.

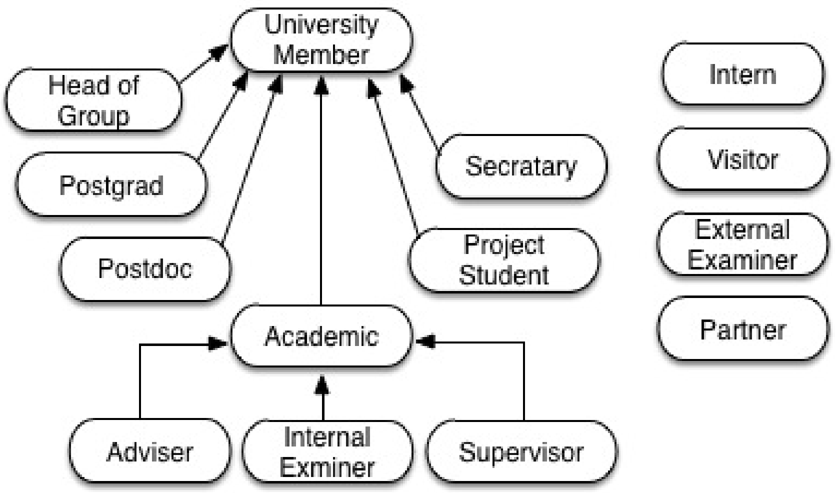


Figure 5.6: Defined roles and their inheritance relation for the PLUS web application

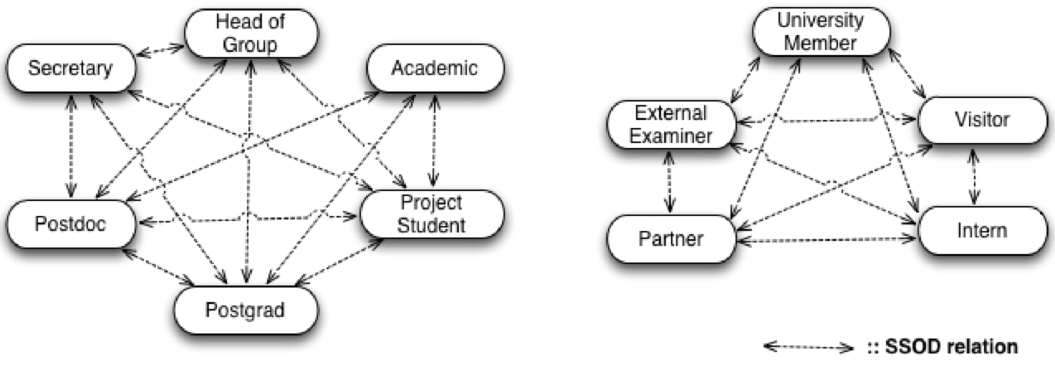


Figure 5.7: Defined SSOD relations for the defined roles

Policy Cases: As shown in Appendix C there are 15 number of policy cases that each defines a unique access control policy that all cover the controlled objects.

Authorisation Management Model

ΦRBACAM (see Appendix C) is used to define the authorisation management system for the defined access control model. It is based on the following elements:

Roles and their relations: There are three roles in this authorisation management model. The role *admin* with cardinality of one, represents the admin of the system. The roles *EmUsrEditor* and *EmActEditor* are also two roles within the authorisation management model that represent the emergency editor of the *user-* and *activity-based* data. These two roles can be assigned to a number of the users within the system for a period of time to deal with unwanted changes by the users of the system.

Authorised Managed Objects: Based on the defined roles, the authorised managed objects are divided into three parts. First object is the `Person` entity and its properties, second is the set of *user-based* entities such as `UniversityMember`; and third is a set of *activity-based* entities, such as `News`.

Policies and Policy cases: The authorisation management polices are based on the mentioned three roles and five *user attributes*. These user attributes representing the five academics that created the PLUS research group. Moreover for each policy term there is a policy term (eight in total). The first case states that the user with the role *EmUsrEditor* is able to *read* and *update* the access control policies for the *user-based* entities. The second case states that the user with the role *EmActEditor* is able to *read* and *update* the access control policies for the *activity-based* entities. The last six cases belongs to the user with the role *admin* or an academic who created the PLUS research group. Each of these cases are able to *read* and *update* *all* the defined authorisation rights within the system.

5.2.5 Defined Φ Model

This part presents the defined access and authorisation management model for the first case study.

As Listing 5.1 shows, because of the nature of the academic research group, both the access control and authorisation management model is *role-based* models (ΦRBAC and ΦRBACAM). The defined ΦRBAC is constructed based on the following elements:

Basic RBAC elements: As Listing 5.1 shows, there are 15 defined roles, such as *HeadOfGroup*. Moreover, there is 9 inherency relations within the university-based roles, such as the role *Advisor* inherits from the role *Academic*. Furthermore there are 25 SSOD relations between the defined roles, for instance, a *Postdoc* can not be the head of group. These relations are within the university-based roles and between the university and external users. Furthermore, there are 3 DSOD relations, such as between the *HeadOfGroup* and *academic* roles.

```
PhiRBAC{

 roles{ ProjectManager(*), UniversityMember(*), HeadOfGroup(1),
        PostGraduate(*), Intern(*), Academic(*), Advisor(*),
        InternalExaminer(*), Supervisor(*), Visitor(*), Postdoc(*),
        ProjectStudent(*), Secretary(*), Partner(*), ExternalExaminer(*)}

 hierarchy{UniversityMember -> (HeadOfGroup, Academic, Postdoc, Postgrad,
        ProjectStudent, Secretary), Academic -> (Advisor,
        InternalExaminer, Supervisor)}

ssod{ //Between university-based and external roles. Also within university-based roles
    UniversityMember <-> (Partner, Advisor, ExternalExaminer, Visitor,
    Intern), Partner <-> (Advisor, ExternalExaminer, Visitor, Intern),
    Advisor <-> (ExternalExaminer, Visitor, Intern),
    ExternalExaminer <-> (Visitor, Intern), Visitor <-> Intern,
    HeadOfGroup <-> (Postdoc, Postgrad, ProjectStudent, Secretary),
    Postdoc <-> (Postgrad, ProjectStudent, Secretary),
    Postgrad <-> (ProjectStudent, Secretary), ProjectStudent <->
    Secretary}

dsod{ HeadOfGroup <-> Academic, InternalExaminer <-> (Supervisor,
     Advisor)}

objects( E(Person), Pr(Person.[nickname, dob, persTel, persAddress,
   persEmail, profPicture, seminars]), E(Partner), E(Project),
   E(Publication), Pr(Visitor.[title, firstname, lastname]),
   E(Seminar), E(Calendar), E(Intern, Academic, Postdoc, Postgrad,
   Project, Interest), E(ProjectStudent, Viva), Pr (Postgrad.[
   startingDate, finishingDate, nineMSubDate, nineMReview,
   transSubDate,transReview, thesisSubDate, thesisReview, goesNominal,
   reason, extendedUntil, internalExaminer, externalExaminer],
   E(Project, Interest, Calendar), Pr(Postgrad.[researchTitle,
   researchSummary]), E(Intern),Pr(Intern.[researchTitle,
   researchSummary]), E(Visitor, Intern, ProjectStudent, Blog, Post,
   Comment, Tag, Interest), Pr(Academic.[uniAddress, uniTel,
   bio, mscProject, phds, postDocs, visitors, interns,
   allocatedViva, onlineBlurb, fullProfile, interests,
   firstAWrite]), Pr(Postdoc.[researchTitle, researchAbstract,
   startingDate, finishingDate)]), E(Visitor,Intern, ProjectStudent,
   Blog, Post, Comment, Tag(author), Publication, Interest),
   E(Partner, Intern, Blog, Post, Comment, Tag),
   E(Visitor, Intern, ProjectStudent, Blog, Post, Comment,
   Tag, Publication, Interest), E(Viva),
   Postgrad.internalExaminer.username, Pr(Postgrad.[nineMReview,
   transferReview]), Pr(ProjectStudent.ProjectReview), Pr(Postgrad.[
   startingDate, finishingDate, nineMSubDate, nineMReview,transSubDate,
   transReview, thesisSubDate,internalExaminer, externalExaminer]),
   E(Visitor), Pr (Viva.[depName, schoolName]), E(ProjectStudent),
   Pr(ProjectStudent.[researchTitle, researchSummary]), E(Event,
   InternalEvent, Seminar, News, Calendar), Pr(Postgrad.thesisReview),
   E(Partner), Pr (Partner.description), E(Postdoc), Pr(Postdoc.
   [researchTitle, researchSummary])))

  policies(
   Self.username == This.Person, ProjectManager, Self.username ==
   This.Project.manager, UniversityMember, Self.username ==
   This.UniversityMember, HeadOfGroup, PostGraduate, Self.username ==
   This.PostGraduate, Intern, Self.username == This.Intern,
    Academic, Self.username == This.Academic, Advisor, Self.username ==
    This.PostGrad.advisor ,InternalExaminer, Self.username ==
    This.Postgrad.InternalExaminer, This.ProjectStudent.examiner,
    Supervisor, Self.username == This.Postgrad.supervisor,
    Visitor, Self.username == This.Visitor, ProjectStudent,
    Self.username == This.ProjectStudent, Secretary, ExternalExaminer,
    Self .username == This.Postgrad.externalExaminer, Partner,
    Self.username == This.Partner ,Postdoc, Self.username == This.Postdoc)


  cases{
    // Any insider user can update and delete his/her info. Also read his/her general info.
    (+,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i)  ->
    ([ud],r,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i),

    // User with role project manager can use CRUD for Partner and CUD his/her Project.
    (?,+,+,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i)  ->
    (i,i,[crud],[cud],i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i),

    // User with role university member can read registered users general info.
    (?,?,?,+,+, i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i) ->
    (i,r,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i),

    // User with role head of group can create or delete a set of entities such as Academic,84 read
postgrad info, and update a set of entities such as Interest.
    (?,?,?,+,−,+, i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i) ->
    (i,i,i,i,i,i,i,i,[cd],r,u,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i),

    // User with the role Postgraduate can see his/her info and update his/her research title and
summary
    (?,?,?,+,−,−,+, i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i) ->
    (i,i,i,i,i,i,i,i,i,r,i,u,i,i,i,i,i,i,i,i,i,i,,i,i,i,i),

    // User with role Intern can read their info and update their research title and summary
    (?,?,?,−,?,?,?,?,+,+, i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i) ->
    (i,i,i,i,i,i,i,i,i,i,i,i,r,u,i,i,i,i,i,i,i,i,i,i,i,i,i),

    // User with the role Academic can create a set of entities such as Visitor, and update
his/her97 info and related user (e.g., postdoc) and activity (i.e., blog) information. Moreover the
user98 can read a set of entities such as Partner and delete a set of entities such as visitor.
    (?,?,?,+,?,?,?,?,?,?,+,+,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i) ->
    (i,i,i,i,i,i,i,i,i,i,i,i,r,u,i,i,i,i,i,i,i,i,i,i,i,i,i,i),

    // User with the role advisor can use CRUD operation on the Viva when the user is103
advising a student
    (?,?,?,+,?,?,?,?,?,?,+,−,+,+, i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i) ->
    (i,i,i,i,i,i,i,i,i,i,i,i,i,i,[crud],i,i,i,i,i,i,i,i,i,i,i,i,i),

    //User with the role Internal examiner can update the nine and transfer review of the
postgraduate student
    (?,?,?,+,?,?,?,?,?,?,+,−,−,−,+,+,?,?,?,?,?,?) -> (i,i,i,i,i,i,i,i,i,
i,i,i,i,i,i,u,i,i,i,i,i,i,i,i,i,i,i,i),

    //User with the role Internal examiner can update the project review of a project student the user is examining
    (?,?,?,+,?,?,?,?,?,?,+,−,−,−,+,?,+,?,?,?,?,?,?) −> (i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,u,i,i,i,i,i,i,i,i,i,i,i),

    //User with the role supervisor can read and update the info on his/her postgraduate student
    (?,?,?,+,?,?,?,?,?,?,+,−,−,−,−,?,?,+,+,?,?,?,?) −> (i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,[ru],i,i,i,i,i,i,i,i,i,i),

    //User with the role visitor can read his/her info and update his/her depart name and school name
    (?,?,?,−,?,?,?,?,?,?,−,?,?,?,?,?,?,?,?,+,+) −> (i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,r,u,i,i,i,i,i,i,i,i),

    //User with the role Project student can read his/her info and update his/her research title and summary
    (?,?,?,+,?,?,?,?,?,?,−,?,?,?,?,?,?,?,?,?,?,+,+) −>
    (i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,r,u,i,i,i,i,i,i)

    //User with the role secretary can use CRUD on a set of entities such as Event
    (?,?,?,+,?,?,?,?,?,?,−,?,?,?,?,?,?,?,?,?,?,?,?,+) −>
    (i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,[crud],i,i,i,i,i),

    //User with the role external examiner can read and update a postgrad's thesis review is the user is external examiner of the postgrad student.
    (?,?,?,−,?,?,?,?,?,?,−,?,?,?,?,?,?,?,?,?,?,?,?,?,+) −>
    (i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,[ru],i,i,i,i),

    //User with the role partner can read his/her info and update his/her description
    (?,?,?,−,?,?,?,?,?,?,−,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,+,+,?,?) −>
    (i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,r,u,i,i),

    //User with the role postdoc can read his/her info and update his/her research title and summary.
    (?,?,?,+,?,?,?,?,?,?,i,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,+,+) −>
    (i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,i,r,u)

  PhiRBACAM{
    //Split the emergency between the user- and activity-based data
    roles {admin, EmUsrFix, EmActFix}
    hierarchy{(EmUsrFix, EmActFix) −> Admin}

    objects(
      E(Person), E(UniversityMember, Academic, Postdoc, Postgrad),
      E(News, Calendar, Event, InternalEvent, Seminar))

    policies (
      EmUsrFix, EmActFix, Admin, Self.username == "cc",
      Self.username == "bf", Self.username == "jr",
      Self.username == "gp", Self.username == "ps")

    cases{ //Read and update the policies.
      (+,?,?,?,?,?,?,?) −> (i    , [r, u],   i ),
      (−,+,?,?,?,?,?,?) −> (i    ,   i  , [r, u]),
    (+,+,+,?,?,?,?,?) -> ([r, u], [r, u], [r, u]),
    (?,?,?,+,?,?,?,?) -> ([r, u], [r, u], [r, u]),
    (?,?,?,?,+,?,?,?) -> ([r, u], [r, u], [r, u]),
    (?,?,?,?,?,+,?,?) -> ([r, u], [r, u], [r, u]),
    (?,?,?,+,?,?,+,?) -> ([r, u], [r, u], [r, u]),
    (?,?,?,+,?,?,?,+) -> ([r, u], [r, u], [r, u])
}
```

Listing 5.1: Defined access control and authorisation management model for the academic research group

Controlled Objects: Listing 5.1 shows the controlled objects within the system. These objects are based on the defined entities and properties of the academic research group that discussed in the last two parts of this Appendix. The objects are grouped and ordered based on their related policies that are used within the policy cases.

Policies and Policy cases: Listing 5.1 shows the defined policy terms and Listing 5.1 shows the policy cases. The policy terms consists of the defined roles and a set of user attribute-based policy terms. The cases are based on the access control requirements that were discussed in details in Appendix C . For clarity there is a comment on the top of each case that shows the relation between the *policy cases*, *policy terms* and the *controlled objects* (shown in Listing 5.1).

Authorisation management model: Listing 5.1 also shows the defined authorisation management model (i.e, 18 LOC) for the research group.

5.2.6 Quantitative Evaluation

This section gives an quantitative overview on the Web application, the defined ΦRBAC model, and provides a set of quantitative data that compares the defined model with respect to the generated access elements and applications' access control objects.

Web Application. The data model of this application consists of 22 entities and 189 properties, in which I have 120 basic type and 69 entity types properties. The core WebDSL compiler then generated 48 tables within the MySQL database. The size of the application is 4500 LOC that consist of 14 unique pages, and 16 hyperlinks and 637 buttons (i.e., CUD operations, panel, search, searchEvent, and logout).

ΦRBAC. The defined access control model is defined in 145 LOC, and has the following quantitative characteristics:

- **RBAC elements:** It has 15 roles (e.g., `academic`, `supervisor`, etc.), 25 SSOD relations (e.g., `supervisor` and `postdoc`), 3 DSOD relations (e.g., `supervisor` and `advisor`), it has 9 inherency relations (e.g., between `academic` and `supervisor`).

- **Controlled Objects:** It enforce access control directly on 20 entities, and 64 properties that are scattered through out the application, such as `Seminar` and `postgraduate.viva.feedback` property.
- **Defined Policies:** It has 23 policy terms in which I have 14 roles (e.g., `advisor`, `internalExaminer`) and 9 are user context constraints (e.g., `Self.username == This.Person`). Based on these policy terms I have 30 cases that cover the 84 objects.

Validation and Verification: Before the code transformation phase, Φ on a MACBook Air with 8GB RAM and 1.7 GHZ CPU, validate and verified the defined model (Listing 5.1) in 6.1 seconds.

Generated Access Control Elements. 508 LOC was generated based on the defined ΦRBAC model, based on the following quantitative characteristics:

- **Predicates:** There are 236 predicates that cover the 84 used objects and 252 buttons directly and 211 objects and 633 buttons indirectly. These predicates were generated in 472 LOC and weaved throughout the application.
- **Data Model Definition:** 8 LOC were generated for defining data model for storing the RBAC elements (i.e., `role`, `activated roles`, `role assignment`).
- **Access rules:** 4 LOC were generated for defining the access control rules for creating open point cuts for `pages`, `templates`, and `actions`.
- **Data Model Initialisation:** 160LOC were generated to initialise the RBAC characteristics defined in the ΦRBAC model.
- **Authorization Management System:** 30 LOC were generated for role assignment and activation functionality for the Web application.

Conclusion. The efficiency of Φ is calculated based on the following percentage: *((targetcode-sourcecode)/targetcode)*100.* By doing so based on the above information, I see that in this case study, with a medium size Web application, the efficiency of using the ΦRBAC model is 76 percent.

5.3 Case Study Two: A Social Networking System

Social Web applications are collaborative, and multi-use software systems. Such systems require flexible security elements to guard the Web applications' objects in an efficient manner. Therefore to cover different security aspects of such systems, it is necessary to give the developer the ability of *simultaneously* defining a number of fine-grained access control and authorization management models. This case study is designed to check the efficiency of using Φ for defining and enforcing ΦDAC, and ΦRBAC access control, and ΦDACAM, ΦRBACAM authorisation management models on a single Web application. First, this part of the thesis discusses the available functionalities of this case study. Second, it explains the required access control and authorisation models and mechanisms; and finally evaluates the efficiency of Φ in social Web application domain.

Available Functionalities. This case study is a social networking system so that the registered users can do the following main tasks:

Relationship: The users can create an arbitrary number of relations to the other users of the system. The user A and B are related *iff* user A request to be related to the user B and user B accept the request, or vice versa.

Network: A user can create an arbitrary number of networks and add other users by accepting their *request*. Moreover, other users of the system can see the network and can join the network. Furthermore, the authorised users of a network can contribute to the network page by adding *topics* or *comments*.

General Social Activities: The user can also perform usual social activities by setting a new *status* or writing a *Wall Post* on his/her profile, and his/her related users' profile page.

Access Control: Each user has the ability to set the security constraints for his/her profile page and each his/her created networks.

5.3.1 Data Model

Same as previous case study, the data model for this case study constructed in two steps. First, the hand-written data model elements, that are representing the data structure of the social networking system. The second set of data model elements will be generated to store the data related to the access control and authorisation management elements. The following give an overview of the data structure of this case study (see Appendix C for their description and their access control requirments).

User: the entity `User` stores the data related to the registered users of the system. Its properties hold the general information about the users such as `username` and `nickname` as well as their virtual networking life such as their latest `status` and their `networks`.

WallPost: This entity holds the data instances of the data related to the `wall posts` such as post's `author` and its `time-stamp`.

Network: This entity holds the information related to each network such as `name` and its `user list`. Note that during the compilation the generated access control entities will be linked to the network entity, so that during the run-time the network creators can assign appropriate access rights to the members of a network.

Status: This entity holds the information related to the `status` messages that each user adds to his/her profile page. It also holds other general information such as `time-stamp` of each status message, that are filled by the Web application itself.

Topic and Comment: The instances of this entity hold the information regarding to the topics that were discussed within each network, such as the `author` and its related `comments`. Similarly the entity `Comment` holds the information about each comment related to a `Post`.

5.3.2 Page Structures, flows and their related functionalities

The pages of the social networking system are divided into `homepage`, `profile`, `network`, and `topic` pages. The rest of this section explains the related functionalities of this case study with respect to these pages.

Homepage: In this page (shown in Figure 5.8), the outside user can register herself by providing the general information of herself and choosing a `username` and `password`. Each registered user can login to the system by entering his/her `username` and `password`.

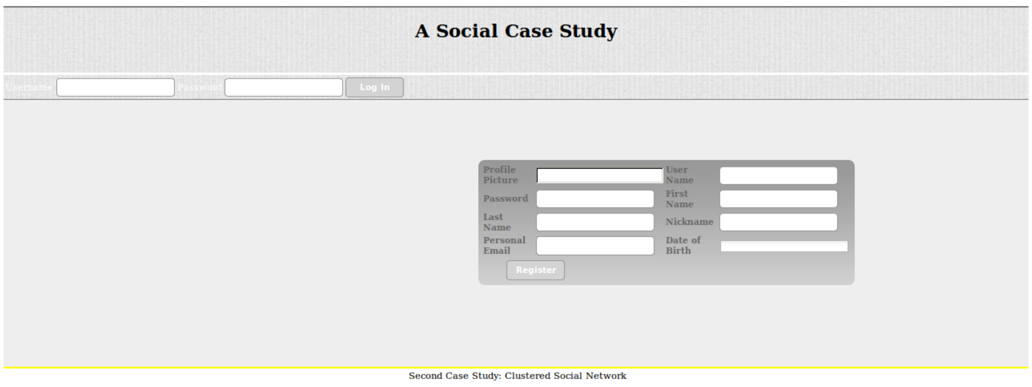


Figure 5.8: Social networking homepage

User Profile: After the user authentication, the user gets directed to his/her profile page. In this page the user can do the following tasks:

- **Profile Settings:** Each registered user has a default profile settings. The user can do edit the personal information on her profile or delete his/her own profile on the system.
- **Relations:** Each user can send a *relationship request* to the other users of the system. Also the user can *accept* or *deny* a relationship request from the other users.
- **WallPosts:** A user of the system can *add*, *delete* or *edit* a `wall post` on his/her `profile` and on her profile page and anywhere else he/she is the author of the Wall post.
- **Networks:** Any user of the system can `add` and then `edit` or `delete` a set of networks. The user can add arbitrary number of related users to each created networks. Moreover, also other users of the system can *request* to join a network. For example, if a user `A` defines a network for Web application development (e.g., `WebAppDev`), other users of the system can see the network name but they can't contribute to the network; because it is access controlled. Therefore for any user who wants to contribute to the network, he/she needs to send a *request* to the user `A` to be the part of the `WebAppDev` network.
- **Search:** An insider user can `search` for other users in the system based on their general information, such as `firstname`. Search results are based on the user privileges with respect to the other users' information.

Network: As mentioned each user can create an arbitrary number of networks and, by accepting their *request*,add arbitrary number of users to those networks. There are the following functionalities available for each network:

- *Add/Delete a user:* As mentioned the added users to the system can be *added* or *deleted* from the network, by the *authorised users.*
- *Topics and their comments:* The authorised users of a network can *add* or *delete topics* to the network page. Moreover, other authorised users can add *comments* to these topics.
- *Join Request:* Any user of the system can send a `Join request` to the creator of a network.

Topic: Each topic within any network has its own unique page. The authorised user can *add* or *delete* comments related to each topic page.

5.3.3 Access Control and Authorisation Management

The main aim of this case study is to demonstrate that the developer has the ability to easily define multi-declarative fine-grained access control models with their related authorisation management model, for a single social Web application. The defined data, access, and authorisation management models related to this case study and their description can be seen at Appendix C.

User profile

This part discusses the defined *access* and *authorisation* management model for the *user profile* page. ΦDAC and ΦDACAM were used to define the access and authorisation management model for the *User* entity because for each user of the system, the data that he/she is adding on the profile page is very important. Therefore the system requires to consider each user as an administrator on the data he/she is putting on the profile page. Therefore, ΦDAC defined for authorising access on each user profile page. Moreover, the *authorisation management* of each user page is given to the user so that he/she can manage the users who are contributing to his/her page by *adding* wall posts, and could have access to his/her related users.

The following shows the ΦDAC access control elements (see the defined model in Appendix C):

Controlled objects: The controlled objects are based on the defined entities that were used within the *user panel* page. They are namely `User`, `Status`, and `Wallpost` entities.

Policy and policy cases: There are four different policy terms based on the *ownership* of the user profile (the person who created the profile), *related users* and the *author* of a Wallpost. There are also four unique policy cases that state the *access rights* of four sets of authorised and not authorised users. The model and its description is shown in Appendix C.

The following shows the ΦDACAM elements:

Managed Objects and operations: The authorisation management is on the instances of the entities used within the user profile (i.e., `User`, `Status`, and `Wallpost`)

Policy and policy cases: Within this model, only the owner of the profile is able to *read* and *update* the authorisation rights defined within the access control model.

Network

This part explains the defined *access* and *authorisation* management model for the *Network* data model. Each network can have large number of users, and they are designed to be used as discussion boards of the Web application. So, there are different groups of people who are contributing to each network. Therefore to enforce a set of access on a group of contributes ΦRBAC is used to control set of operations and objects of the network. Moreover ΦRBACAM is defined to manage the controlled objects' authorisation rights, because there are a number of connected management-based roles.

The defined ΦRBAC is constructed based on the following elements:

Roles and their relations: There are five roles in the defined ΦRBAC model for the networks of the system. They are namely *NetworkOrganizer*, *Admin*, *LeadContributor*, *Contributor*, and `NetworkUser`. More over there are three inheritance relation. For example, the the role *NetworkOrganizer* inherits from *Admin* and *LeadContributer* roles.

Controlled Objects: The controlled objects are based on three entities of `Network`, `Topic`, and `Comment`.

Policies and Policy cases: The defined roles create the policies of this model. Also there are four cases that each represents the authorisation rights of a role. For example the user with the role *NetworkUser* can only read the content of the Network. Note that the full description of the model is discussed in Appendix C.

Also, the defined ΦRBACAM is based on the following elements:

Roles and their relations: There are two disjoint roles. They are *NetworkManager* and *NetworkMonitor* roles.

Controlled Objects: Also the authorisation management is on the access controlled objects (i.e., `Network`, `Topic`, and `Comment` entities).

Policies and Policy cases: In addition with the mentioned two roles there is also a user attribute-based policy term which is based on the *creator* of a `Network` entity. Moreover for each of these policy terms there is a policy case. For example the *creator* of the `Network` entity is able to read and update all the authorisation rights related to the `Network` entity.

5.3.4 Defined Φ Model

his section shows and discuss the defined Φ-based access and authorisation management models for the social networking Web application. In this case study the following models were defined:

ΦDAC with ΦDACAM: A discretionary-based access and authorisation management model were defined, for the user profile page. As discussed in Chapter 5 because every user needs to protect her own data discretionary-based models were the most efficient model. As Listing 5.2 shows, the access control model is based on the following elements:

Controlled Objects: The controlled object in this model consists of the properties within the `User`, `Status`, and `Wallpost` entities.

Policies and policy cases: There are three polices and 4 policy cases. The following explains the polices and their cases:

- *Profile owner:* Each user who creates a profile (after registration) become the owner of the profile. The first policy term and case says the owner of a profile `(Self == user, +)` is able to use all the available operations on the related instance of the entity `User` and all its related properties.
- *Related users:* The related users of each profile are able to read the following properties: `nickname`, `title`, `firstname`, `lastname`, `wallPosts`, `writtenWP`, and `allStatus`. Moreover they are able to *create*, *update*, or *delete* a wallpost. The second and third policy case states the mentioned criteria for related users.
- *Non related users:* The unrelated users are able to see the `group` *addRequest* that consist of a name of a user (i.e., instances of `firstname` and `lastname` properties) and its related `Join request` action. The fourth policy in Listing 5.2 represent this description.

Authorisation Management. The authorisation management model simply has one policy case, that states the *profile owner* is able to *read* and *update* all the mentioned policies related to the entities (i.e., `User`, `Status`, `Wallpost`) that are used within the user profile page.

```
PhiDAC{

  objects(E(User), Pr(User.[nickname,title ,firstname, lastname,
    wallposts, writtenWP, allstatus]), Pr(User.wallPosts),G(addRequest))

  policies( Self == User, Self == relatedUser, Self.username == WallPost.
  author)

  cases{ (+,−, −, −) −> ([crud], i, i, i),
    (−, +, −, −) −> (i, r, c, i), //Related but didn't create any wallpost
    (−, +, +, −) −> (i, r, [r, u, d], i),
    (−, −, −, −) −> (s, s, s, r)}

  PhiDACAM{ objects(E(User, Status, Wallpost))
    policies(Self == User)
    cases{ (+) −> ([r,u])}
  }
}

PhiRBAC{

  roles{NetworkOrganizer, Admin, LeadContributor,Contributor,
      NetworkUser}

  hierarchy{ (Admin, LeadContributor) -> NetworkOrganizer,
     Contributor -> LeadContributor, NetworkUser ->  Contributor}

  objects( E(Network), E(Topic), E(Comment))

  policies( NetworkOrganizer, LeadContributor, Contributor, NetworkUser)

  cases{ (+,+,+,+)   -> ([c, r, u, d], i, i),
         (-,+,+,+ )  -> (i, [c, r, u, d], i),
         (-,-,+,+)   -> (i, i, [c, r, u, d]),
         (-,-,-,+)   -> (r, i, i)}
  PhiRBACAM{
   roles{ NetworkManager, NetworkMonitor}

   objects(Network, Topic, Comment)

   policies( Self.username == Network.creator, NetworkManager,
 NetworkMonitor)

   cases{(-,+,-) -> (i,[ru],i),
      (-,-,+) -> (i,i,[ru]),
      (+,?,?) -> ([ru],i,i)}
 }
}
```

Listing 5.2: Defined ΦDAC model with discretionary-based authorisation management for the user profiles

ΦRBAC with ΦRBACAM: As mentioned before for the ΦRBAC with ΦRBACAM were used to define the access control and authorisation models. The ΦRBAC access control model consists of the following elements:

- *Roles and their hierarchy:* As mentioned in prior to this section, and as Listing 5.2, there are five roles within the system. They are namely *NetworkOrganizer, Admin, LeadContributor, Contributor*, and *NetworkUser*. Moreover as shown in Listing 5.2 there are set of inheritance relation between these roles, for example, *NetworkOrganizor* inherits from *Admin* and *LeadContributer*.
- *Controlled Objects:* The controlled objects consists of the *Network*, *Topic*, and *Comment* entities and their properties.
- *Policies and policy cases:* The first policy case states that the user with the role *NetworkOrganizer* is able to use the *CRUD* operations on the `Network` entity

and its properties. Moreover the second case states that the users with the role *LeadContributor* are able to use the *CRUD* operations on the entity `Topic`. Furthermore, the users with the role *Contributor* are able to use the *CRUD* operations on the `Comment` entity. Finally the users with the role *NetworkUser* are able to *read* the content of the `Network` entity.

Authorisation Management Model. As Listing 5.2 shows, there are two disjoint roles. They are namely *NetworkManager* and *NetworkMonitor*. As the first two policy cases within the ΦRBACAM shows, the user with the role *NetworkManager* is able to *read* and *update* the authorisation rights related to the entity `Topic`; and the user with the role *NetworkMonitor* is able to use *read* and *update* the authorisation rights related to the `Comment` entity. Furthermore, as the third policy case states,the creator of the a network is able *read*, and *update* all the authorisation rights related to the entity `Network` and is related properties.

5.3.5 Quantitative Evaluation

This part of the report discusses the quantitative overview of the social web application, the defined Φ models and gives a set of quantitative data that compares the defined models with respect to the generated access and authorisation management elements and authorised objects.

Web Application. The data model of this application consists of 5 entities and 37 properties, in which I have 23 basic type and 14 entity types properties. The core WebDSL compiler then generated 15 tables within the MySQL database. The size of the application is 2000 LOC that consist of 3 unique pages, and 10 hyperlinks and 145 buttons (i.e., CUD operations, panel, search, searchEvent, and logout).

Defined Model. As mentioned, three types of access control and authorisation management models were used for *user profile*, *relations*, and *networks*. The defined model is 38 LOC based on the following elements:

Access rights: Overall there are 10 roles (for enforcing role-based access and management rights on networks), and $m$amount of DAC rights for each user in the system where m is the number of the related users. Overall the Φ models defined in 45 LOC.

Controlled Objects: Overall these Φ models directly control the defined data model (i.e., 37 properties in 5 entities), except the created network names as they are public to all the user.

Validation and Verification: Φ on a MACBook Air with 8GB RAM and 1.7 GHZ CPU, validate and verified the defined model (Listing 5.1) in 2 seconds.

Defined Policies: For the networks I have 7 access and 6 authorisation management policy terms. For the user profile, 4 number of DAC-based policy terms as the user needs to link each of the related users to a set of objects and operations. Therefore over all I have 23 unique policies that enforces defined authorisation rights through out the application.

Generated Elements: 373 LOC was generated based on the defined Φ models. They have the following quantitative characteristics:

- **Predicates:** From 23 unique policies, 144 predicates were generated to cover the properties of the defined models.
- **Data Model Definition:** 25 LOC were generated for defining data model for storing the RBAC(i.e., `role`, `activated roles`, `role assignment`), `activated roles`, `label assignment`), and DAC elements (i.e., assigning users to permissions).
- **Access rules:** 4 LOC were generated for defining the access control rules for creating open point cuts for `pages`, `templates`, and `actions`.
- **Data Model Initialization:** 80LOC were generated to initialise the RBAC, and DAC characteristics defined in the Φ models.
- **Authorization Management System:** 120 LOC were generated for role assignment, role activation, label assignment, label activation, and a discretionary assignment (i.e., (object, oper) pair to each user) functionalities for the Web application.

Conclusion. Similar to the evaluation of case study one (see 5.2.6), I calculate the efficiency of Φ language by the following formula: *((targetcode-sourcecode)/targetcode)*100.* Here the defined model is 38 LOC while the generated code is 373 LOC, and therefore the efficiency of Φ is 89%.

5.4 Φ Evaluation

This part summaries the findings and weaknesses of the Φ language in defining multi-declarative fine-grained access control models for a single Web application.

5.4.1 Findings

Run-Time Access Constraints: The first issue that was realised using Φ was the fact that I need to provide an access control mechanism that can be fully or partially configured during the Web application run-time. For example, in this case study, only during the run-time the user can assign the *Φ*DAC constrains, through a network, to a set of users.

Design: During this case study, regardless of the security perspectives, I realised the fact that using Φ is helping the design of the application. For this case study I spread 11 unique functionalities through just 3 pages. This was only because I had the ability to enforce access control on fine-grained objects and their related operations regardless of their location within the application.

5.4.2 Weaknesses

The following points show the weaknesses that became clear during the development of the second case study:

Automated abuse intervention mechanism: In this case study I give each user the ability to set their profiles authorisation rights for the other users through different networks. In this case, if a user, by mistake or by choice, assigns a set of authorisation rights to a set of users that shouldn't have any right; those users could abuse their power to operate on a set of controlled objects. This highlights the fact that as the authorisation settings move from development phase towards run-time phase the application needs to have a mechanism to pick abusing behaviours and stop such users to use the system (e.g., change their password automatically and stopped the abusers to authenticate to the system).

Data History: Also these two case studies taught me the fact that, the application needs to have a mechanism to *record* the history of the data for auditing purposes. For example, consider a case that user `A` give the user `B` the full read access right for a network, friends. In this case user `B` could find the information related to the other users in the network friends and use their data. In this case user `A` does not know about the actions that user `B` are doing based on granted read access rights. If I had *an automated mentoring mechanism* in the system; the user A could see this fact and take away the user `B`'s *read* access rights to the friends network.

5.5 Summary and Conclusions

This chapter presented two case studies, to check the efficiency (see 5.2.6 and 5.3.5 sections) of the *access control*, *authorisation management models*, *architectures* and *mechanisms* that were introduced in this research. Moreover after each case study Φ was evaluated quantitatively based on its usage.

In case study one, the Web application implemented for the research group. The used access control model was *Φ*RBAC model with the centralised authorisation management system (the default setting). In case study two, the Web application was implemented for a social networking system. The access models used in this case study were ΦDAC with ΦDACAM, and ΦRBAC with ΦRBACAM. Moreover at the end, the weaknesses and shortcomings of Φ were discussed.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

This research presented the design and implementation of a declarative and fine-grained policy language (i.e., Φ) for the Web application domain. This chapter first summarises the contributions of this research, then it addresses the raised research questions that discussed in the introduction, and finally provides the potential future path of this research.

6.1.1 Summary of contributions

The following points are the contributions of this research:

Design: The work area of this research is an intersection of access control, authorisation management, Web application, language design and software testing. Each of these domains and the subsection of their unions have different set of research background in which they tackle declaratively and fine-granularity of authorisation- based models in different set of solutions and abstraction levels. Therefore, throughout the course of this research the following design decisions were vital to create the Φ's characteristics.

- *Right Abstraction Level:* Policy languages such as Ponder [61] claim that they are declarative, however the abstraction level of their syntax are far away from authorisation notions (i.e., roles, SOD, etc.). In Φ, and consequently in this research, there is no difference between standardised authorisation notions (e.g., NIST RBAC [145]) in literature and Φ models.
- *Fine-Granularity:* This research presented fine-grained access and authorisation management models, not just in terms of enforcing access and management on

fine-grained objects; but also the policy of Φ models are fine-grained themselves. Therefore, the developer can define a set of models based on fine- grained access and authorisation management policies on a set of fine-grained controlled objects.

- *Covering different aspects:* Web applications can be developed in such a way that cover different aspects of an organisation. To support this core concept, Φ allows the developer to define, up to three different types of access control models (i.e., ΦDAC, ΦMAC, and ΦRBAC), and also for each access control model, up to three different types of authorisation management models (i.e., ΦDACAM, ΦMACAM, and ΦRBACAM). Therefore, Φ gives the capability to the developer to decide on the types of access and authorisation management models for the users of a Web application.
- *Coverage:* Only the developer of the application knows about the usage requirement of access control and authorisation management models. Therefore, Φ provides the coverage concept for both access and authorisation management models, to the developer, to define a set of sufficiency checks for Φ models. So, during the compiler time if this criteria is not fulfilled the Φ compiler will terminate the compilation.

Architecture: The practical outcome of this research is Φ. As mentioned earlier, Φ's compiler has a pipeline architecture that is divided into testing and transformation phases. The novelty of the pipeline is that, Φ extensively tests the defined models and the target application before the transformation phase; by using a set of written strategies and an SMT solver during the compile time. Then, if the testing phase is successful, Φ generates and then weaves the access control and authorisation management mechanisms within the target application.

A Ground Work for further research: Φ is not just a concept but it is developed as an add-on language to a domain-specific language WebDSL. This enables a tool for the researchers to use and do the further experimental research in the following domains:

- *Language-based solutions:* This research provides an in-depth discussion for designing and implementing a declarative and fine-grained policy language for the Web application domain. This research in its core is a language based solution for the complexity of implementing authorisation-based models. As a result, Φ can be imported to the other languages such as Scala [94] or Web development frameworks such as Grails [8] or Play [13].
- *Extensions:* Φ provides declaratively and fine-granularity just for the access control and authorisation management models. Φ is used as a ground work for other security-based solutions such as data-history and provenance.

Publications: I did publish two papers [79, 78], as the first author, in the domain of software security. Both focuses on variations of ΦRBAC, its validation and verification and transformation stages. The first publication [79], focuses on ΦRBAC and its usage in the domain of Web application; and why it is essential to have a declarative- and fine-grained policy language. The second publication [78], continues with this discussion by adding the attribute-based constrains (i.e., user-, data-, and system-based) to the ΦRBAC model.

6.1.2 Answers to Research Questions

This section discusses the raised questions introduced in the first chapter.

Research Question 1 *How can I provide a fine-grained access control modelling mechanism to the developer?*

The main aim of this research was to provide a policy language (i.e., Φ) that is declarative and fine-grained. In Φ, both access control and their authorisation management models are abstracted and separated from the Web application code. Afterwards the Φ generator generates the low level code (e.g., data mode, pages, etc.) and then weaves them into the target application. By using this method Φ achieves the following points:

- *Productivity and Maintainability:* Φ was designed in a way that handles the maintainability such that the authorisation-based changes are only through Φ models. This is an advantage for the Φ compared to the current access control modelling in WebDSL. This is due to Φ' *separation of concerns* characteristic (see Chapter 5 for examples). For example, if the developer needs to change something with the current settings, the developer needs to change the definitions of access control rules as well as the data model and other related templates.
- *Reliability:* Φ models, as any other software element, need to be validated and verified. As shown in Chapter 4 (see Section 4.2), because of using a high abstracted notion within a matrix structure, the semantical validated and verified of the access control models and their authorisation management definitions were formulated into a first order logic formula and validated and verified with an SMT solver during the compiler time.

Research Question 2 *Is it possible to define the notion of fine-grained access control models outside the context of their implementation approach?*

Yes, by using Φ, the developer defines authorisation-based models outside any language terminologies (i.e., functional, agent oriented, etc.) but with respect to the authorisation based terminologies (i.e., DAC-, MAC-, or RBAC-based). Moreover, Φ can be applied to the other programming languages by importing its syntax and rewrite strategies with the compiler of the host language. Important to note that, a set of language-based justifications need to be considered, such as generating the host language code through the Φ's rewrite strategies.

Research Question 3 *What elements need to be considered during the testing phase of a fine-grained access control model and its mechanism?*

Φ tests both the defined models and the target application before the transformation phase. Φ tests the models based on the set of controlled objects they used (i.e., object scope validation), the structure of the access control elements (i.e., for ΦMAC and ΦRBAC); and if the policy and coverage cases of Φ models are overlapping or incomplete. Moreover, Φ checks for the dead authorisation code within the target application based on the location of the objects. The way that Φ handles object scope and Web application validations, builds its compiler's pipeline based on first testing and then transformation phases.

Research Question 4 *What are the benefits and constraints of the fine-grained access control models and their mechanisms?*

Φ models enforce access and authorisation management mechanisms to a range of objects with different fine-granularity levels, such as pages or a property of an entity. Moreover, the policy terms of the model are fine-grained as well. Φ also considers the authorisation definitions during the transformation phase. This fine-granularity provides a very flexible environment for the developer to enforce declarative and fine-grained access and authorisation management models. Note that, the fine-grained authorisation management mechanisms helps the authorised user of the application to do the managerial work. However, the constraints are related to the characteristic of the access control models themselves. Access control models deal with how to enforce access restrictions to different objects of the system. If the researcher wants to explore other security related areas, they are needed to extend the Φ models. For example, if they want to tackle the issue of SQL attack, or use a data history in provenance-based solutions, they need to extend Φ syntax and its generated components to deal with saving the user actions during the system run-time.

6.2 Future Work

The following points are the related future work to the Φ language.

Provenance: On the top of the mentioned contributions I can add a provenance module. Provenance uses the data history that is gathered by the generated Φ mechanism automatically to test the user behaviours through their interactions with the application.

Explore Main Stream Languages: Currently Φ generates WebDSL code. WebDSL code itself is a domain specific language. To provide Φ for more developers, it will be an interesting step to generate new modern languages such as Scala [94], or importing it to the other Web frameworks such as Play [13].

Further Validation and Verification on Financial Regulations: Policy regulations are used in financial sector. These policy regulations need to be implemented in software systems. It will be a good idea to see the challenges that Φ will face in such circumstances, and if necessary, update the Φ to be adoptable in the financial sector.

For my future plan after the PhD, I have started my position as a lecturer at PSU (Prince Songkla University), Phuket, Thailand since June 2013.

Appendix A

Syntax Definition

This part shows and discusses the defined syntax[1] of Φ *language*, *Z3 formalism*, and Φ*'s configuration file* (i.e., `Phi.ini`). For the semantical meanings of the Φ language please refer to Chapter 3.

A.1 Φ Language

The Φ's syntax definition (written in SDF [90]) is discussed here based on the *controlled objects and their related operations*, *policy terms and their signs*, *access* and *authorisation management models*.

A.1.1 Lexical Syntax

This part explains the defined *lexical syntax*, that are defining the following elements within the Φ language:

Object related operations: As Listing A.1 shows (lines 2 and 3), Φ supports all the data manipulation operations (*create*, *read*, *update*, and *delete*), with additional *secret*, and *ignore* operations. The *create* (i.e., `c`), *update* (i.e., `u`), and *delete* (i.e., `d`) can be used *only* in relation with the defined *entities* and *properties*. The *read* operation (i.e., `r`), *secret* (i.e., `s`), and *ignore* (i.e., `i`) can be used in relation with all the supported objects.

Policy Signs: Listing A.1 shows three types of policy related signs (see line 6). The *+* indicates the related policy term holds, and - indicates it does not hold; and the *?* states the *don't care* state, that is, the related policy term does not effect the policy case.

[1]Available at `http://philang.org/repos/SyntaxDefinition/Phi.sdf`

Comparison Signs: These signs are used within the policy terms of type *discretionary-* or *attribute-based* (i.e., *user*, *data*, *system*, or *user and data comparisons*). Listing A.1 shows the list of available comparison signs (see lines 9-11).

```
   %%Object-related operations
2  "c"     -> PhiOOper    "r"     -> PhiOOper    "u"     -> PhiOOper
3  "d"     -> PhiOOper    "s"     -> PhiOOper    "i"     -> PhiOOper

5  %%Policy-related signs
6  "+"     -> PhiPOper    "-"     -> PhiPOper    "?"     -> PhiPOper

   %%Comparison signs
9  "==" -> CompSign {cons("EQSign")}   "!=" -> CompSign {cons("NotEQSign")}
10 ">"  -> CompSign {cons("GtSign")}   ">=" -> CompSign {cons("GtEqSign")}
11 "<"  -> CompSign {cons("SmSign")}   "<=" -> CompSign {cons("SmEqSign")}

   %% Role cardinality: a number || as many (wild card)
14 Int  -> CardNum         "*"     ->  CardNum
```

Listing A.1: The defined lexical syntax

Role Cardinality: As discussed before, *role cardinality* needs to be initiated for each defined role in a ΦRBAC or ΦRBACAM model. As Listing A.1 shows (line 14), the role cardinality is either defined as a *number* or a wild card (i.e., `*`) the wild card shows that, there is no limitation in *assigning* the role to the registered users within the Web application.

A.1.2 Objects

This part presents the syntax definition of the supported object, that are sorted based on their granularity levels.

Properties: The most fine-grained object supported object within the Φ language and WebDSL [83] is the properties of the entities (e.g., `User.username`). Φ enables developers to define the properties of the entities as a controlled object based in the two different following ways:

- *Single Property:* There might be a case in which *only* one property of an entity needs to be used within the access control or the authorisation management model. In this case, as Listing A.2 shows, the developer can use a path to the property the same way that the WebDSL [22] supports (e.g., `Pr(UniversityMember.id)`.
- *Multiple Properties:* There might be a situation in which the developer wants to use a set of properties of an entity as one controlled object within the access and authorisation management models. In this case, as Listing A.2 shows

(lines 12), the developer can use brackets to declare a set of properties of an entity (e.g., `Pr(User.[address, phone, email])`).

XML Nodes: As Listing A.2 shows (See line 8), the developer can use *XML* nodes within the set of objects used in the Φ's access and authorisation management models.

Entity: A set of entity names can be used, as an object, after the `E` keyword, as shown in Listing A.2.

Group: A set of *group* names can be used, as an object, after the `G` keyword, as shown in Listing A.2.

```
"objects" "(" {CObj "," }+  ")"  -> PhiCObjs    {cons("PhiCObjs")}
"Pr"     "(" {PhiPath ","}+ ")"  -> CObj        {cons("PhiPr"), prefer}
"P"      "(" {Id ","}+      ")"  -> CObj        {cons("PhiPg"), prefer}
"T"      "(" {Id ","}+      ")"  -> CObj        {cons("PhiT" ), prefer}
"B"      "(" {Id ","}+      ")"  -> CObj        {cons("PhiB" ), prefer}
"G"      "(" {Id ","}+      ")"  -> CObj        {cons("PhiG" ), prefer}
"E"      "(" {Id ","}+      ")"  -> CObj        {cons("PhiE" ), prefer}
"X"      "(" {Id ","}+      ")"  -> CObj        {cons("PhiX" ), prefer}

Exp                             -> PhiPath     {cons("NormPath"), prefer}
Id                              -> PhiPath     {cons("PhiPath")}
Id "." "[" PathContent "]"      -> PhiPath     {cons("NodesList")}
{ Id "," }+                     -> PathContent {cons("PhiLevelOne")}
Id "." "[" PhiPath  "]"         -> PathContent {cons("PhiNested")}
```

Listing A.2: Used controlled objects within Φ language

Block: Listing A.2 shows that, a set of *block* names can be used, as an object, after the `B` keyword.

Page: A set of *page* names can be used, as an object, after the `P` keyword, as shown in Listing A.2.

Template: Listing A.2 shows that, a set of *template* names can be used, as an object, after the `T` keyword.

A.1.3 Policy terms

As mentioned throughout the thesis, the policy terms within the Φ models are defined after the `policies` keyword (see Listing A.3). The syntax definition of policy terms (Listing A.3) that can be used with the Φ's access and authorisation management models, are described as follows:

Discretionary: The discretionary constraints can be used within the ΦDAC and ΦDACAM models. This constraints are based on the value of the user identifiers defined within the Web application. For this reason, Φ allows developers to use `Self` keyword, followed by a comparison sign (discussed earlier) and value, to indicate the discretionary-based requirements.

```
%% Defined Policies.
"policies" "(" { PhiPTerm ","}+ ")" -> PPolicies {cons("PhiPolicies")}

%% Role || Label.
Id                                   -> PhiPTerm   {cons("RoleOrLabel"), prefer}
%% Attribute-based comparisons.
PhiAtt CompSign Exp                  -> PhiPTerm   {cons("VCond")}
PhiAtt CompSign PhiAtt               -> PhipTerm   {cons("AttCompAtt")}

%% Discretionary constraints.
"Self"                               -> PhiAtt     {cons("UserIden")}
%% User and data attributes.
"Self" "." DMObjPath                 -> PhiAtt     {cons("UserAtt")}
"This" "." DMObjPath                 -> PhiAtt     {cons("DataAtt")}

%% System attributes (time).
"Sys"  "." "time" TimeRange          -> PhiAtt     {cons("TimeAtt"), prefer}
CompSign TimeDigit                   -> TimeRange  {cons("TimeRange")}
"(" CompSign TimeDigit ","
        CompSign TimeDigit ")"       -> TimeRange  {cons("TimeLimRange")}

%% System attributes (date).
"Sys"  "." "date" DateRange          -> PhiAtt     {cons("DateAtt"), prefer}
CompSign DateDigit                   -> DateRange  {cons("DateRange")}
"(" CompSign DateDigit ","
        CompSign DateDigit ")"       -> DateRange  {cons("DateLimRange")}

"Sys"  "." "time" CompSign Exp       -> PhiAtt     {cons("TimeAtt"),   prefer}
Int    ":" Int                       -> TimeDigit  {cons("TimeDigit"), prefer}
Int "/" Int "/" Int                  -> DateDigit  {cons("DateDigit"), prefer}

%% For avoiding ambiguities with the WebDSL's original identifier
"Self"                               -> Id {reject}
"This"                               -> Id {reject}
```

Listing A.3: Policy terms in Φ

Attribute-based policies: As mentioned through out the thesis, the supported attribute policies are based on *users*, *data*, and system (date and time) constraints. These policies are:

- *User- and Data-based attributes:* As Listing A.3 shows (lines 11-15), the `Self` keyword, followed by a *property path*, a *comparison sign* and a *value* can be used to define a *user-based* policy term (e.g., `Self.age > 18`). Also, for the *data-based* attributes, the `This` keyword can be used, followed by a *property path*, a *comparison sign*, and a *value*, as defined in Listing A.3 (e.g., `This.username := "John"`). Moreover, the comparison between the *user* and *data* attributes can be used as a policy term. For this, as Listing A.3 shows, the `Self` keyword followed by the *property path*, a *comparison sign*, and the `This` keyword followed by the *property path*, can be used to define such comparisons (e.g., `Self.salary == This.salary`).

- *System-based Attributes:* Moreover, the *system-based* attributes can be used as a policy term. The system-based attributes defines a *range* of time or dates with *lower* and *upper bounds*. Listing A.3 shows, for defining the time-based constraints, the developer uses the `Sys.time` keyword and then within its brackets define the lower and upper time bounds (e.g., `Sys.time(9:00, 17:00)`). Within the Φ syntax, the developer uses `Sys.date` keyword, and within its brackets defines the date bounds, Φ (Listing A.3). Furthermore, as shown in the Listing A.3, the comparison signs can be used for the system-based attributes's bounds limitations (e.g., `Sys.date(>=1/1/2010, <= 1/1/2020)`).

Labels and Roles: As shown in Listing A.3, the WebDSL *identifier* is used to declare *labels* for the Φ's mandatory-based (i.e., ΦMAC and ΦMACAM) and *roles* for role-based models (i.e., ΦRBAC, and ΦRBACAM).

A.1.4 Policy and Coverage Cases

The syntax definition of the policy and coverage cases are discussed in this section (See Listing A.4 and A.5).

Policy Cases. For each access or authorisation management model within the Φ language has a set of policy cases defined within curly brackets of the `cases` keyword (see Listing A.4 line 1).

```
"cases" "{" { PhiCase ","}+ "}"    -> PhiCases {cons("PhiCases")}

%% Each policy case
"(" { PhiPOper ","}+  ")" "->" "(" { ObjOpers ","}+ ")"
                                   -> PhiCase  {cons("PhiCase")}
PhiOOper                           -> ObjOpers {cons("SingleOper")}
"[" { PhiOOper ","}+ "]"           -> ObjOpers {cons("MultOper")}
```

Listing A.4: Policy cases syntax definition

As Listing A.4 shows, on the left hand side of each policy case the set of *policy signs* are used. These policy signs are related to the defined policy terms based on their order. As mentioned before, the policy signs give a *logical meaning* (i.e., *true* or *false*) to their related policy terms. Then after the `->` keyword on the left hand side the object-related operations are used individually or as *a set* within the brackets (see Listing A.5).

Coverage Cases. The coverage cases can be defined by the developers for each access or authorisation management model. Listing A.5 shows the syntax definition of the coverage cases. Similar to the policy cases, on the left hand side of each policy case, a set of policy signs (i.e., `+`, `-`, `?`) can be used separated by comma. On the right hand side as shown in

Listing A.5 a set of object related operations can be used *individually* or as a *set* and their related percentage can be assigned to them.

```
"cases" "{" {CovCase ","}+ "}" -> CovCases {cons("AllCovCases")}
"(" {PhiPOper ","}+ ")"  "->"
"(" { PercRange ","}+ ")"    -> CovCase  {cons("CovCase")}

%% r < 10 or r > 90
ObjOpers CompSign Int
  -> PercRange {cons("OperPerc")}

%% Range for one operation ( <10, >80 ).
ObjOpers "("CompSign Int "," CompSign Int ")"
  -> PercRange {cons("OperPercRange")}

%% Single Value [[c,r,u].(<10)].
"[" "[" {ObjOpers ","}+ "]""." "("CompSign Int ")" "]"
  -> PercRange {cons("OpersPerc")}

%% Lower and upper bound value range for list of operations [[cr].(<90)]
"[" "[" {ObjOpers ","}+ "]""." "("ComSign Int "," CompSign Int ")" "]" ->
     PercRange {cons("OpersPercRange")}
```

Listing A.5: Syntax definition of coverage cases

A.1.5 Access Control Models

This part discusses the syntax definition of the Φ-based access control models. They are as follows:

ΦDAC: Listing A.6 shows the syntax definition of the ΦDAC model that consists of *objects* (discussed in A.2), *policies* and their *cases* (discussed in A.3 and A.4). Moreover, for the *coverage* as shown in Listing A.6, after the `coverage` keyword, with its brackets, the *objects*, *policies*, and coverage cases can be used. As Listing A.6 shows, an authorisation management model is defined for the ΦDAC model.

```
"PhiDAC"  "{" PhiCObjs PPolicies PhiCases
  DACCoverage? PhiAM "}"          -> PhiDACModel {cons("PhiDAC")}
"coverage" "{" PhiCObjs PPolicies CovCases "}"
                                  -> DACCoverage {cons("DACCoverage")}
```

Listing A.6: The ΦDAC defined syntax

ΦMAC: Listing A.7 shows that, a ΦMAC model consists of the following elements:

- *Labels:* As the syntax definition shows (see lines 4 and 5), the user defines a set of *confidentiality*- and/or *integrity-based* labels. Please refer to Chapter 3 and 4 for their semantics and examples.

- *Objects:* Similar to ΦDAC, the discussed objects can be used. However, for the ΦMAC model, the defined *confidentiality* and/or *integrity* labels are attached to the objects.

```
"PhiMAC" "{" ConLabels? IntLabels? MACObjects PPolicies
  PhiCases MACCoverage? PhiAM* "}"    -> PhiMACModel {cons("PhiMAC")}

"conLabels" "{" {Id "->" }+ "}"       -> ConLabels   {cons("ConLables")}
"intLabels" "{" {Id "->" }+ "}"       -> IntLabels   {cons("IntLables")}

%% Used objects and their related secrecyand/or integrity labels
"objects" "(" { MACObjComp ","}+ ")" -> MACObjects  {cons("MACObjects")}
CObj "." "label"  "(" Id  ")"        -> MACObjComp  {cons("MACObjElem")}
CObj "." "labels" "(" Id "," Id  ")" -> MACObjComp  {cons("MACObjElems")}

"coverage" "{" MACObjects PPolicies CovCases "}"
                                      -> MACCoverage {cons("MACCoverage")}
```

Listing A.7: ΦMAC defined Syntax

- *Policies and Cases:* The syntax definition for the policy cases is the same as ΦDAC model. As mentioned before, because of the MAC semantics, the use of *create* and *delete* operations are treated as an error during the compilation of the ΦMAC model (see Chapter 4).
- *Coverage:* Same as ΦDAC model, the coverage consists of *objects*, *policies* and coverage cases. The difference with the ΦDAC is that, in the coverage of the ΦMAC model, as Listing A.7 shows, the *confidentiality* and/or *integrity* labels needs to be declared for each object.
- *Authorisation Management:* Similar to the ΦDAC model, as Listing A.7 shows, the developer defines an authorisation management model (i.e., ΦDACAM, ΦMACAM, ΦRBACAM).

ΦRBAC: As Listing A.8 shows, the ΦRBAC syntax definition is based on the following elements:

- *Roles:* The developer defines a set of roles and their cardinalities within the brackets of the `roles` keyword.
- *Roles' relations:* The developer also defines a set of *hierarchy*, *ssod* and *dsod*, relations after `hierarchy`, `ssod`, and `dsod` keywords, for and between the defined roles.
- *Objects:* All the defined objects can be used as the controlled objects within the ΦRBAC models.
- *Policy and Coverage Cases:* The developer uses *roles* and attribute-based policy terms as a set of policies. Identical to ΦDAC and ΦMAC the policy cases consist of the policy signs and object-related operations.

- *Authorisation Management:* Identical to the ΦDAC and ΦMAC, the developer uses a Φ-based authorisation management model for the ΦRBAC model.

```
"PhiRBAC" "{" PhiRoles RHierarchy? SSOD? DSOD? PhiCObjs PPolicies
PhiCases DACCoverage? PhiAM "}" -> PhiRBACModel {cons("PhiRBAC")}

%% Roles and cardinalities
"roles" "{"  {PhiRole ","}+ "}"  -> PhiRoles {cons("PhiRoles")}
Id "(" CardNum ")"               -> PhiRole  {cons("RoleWithCardinality")}

%%Hierarchy relations
"hierarchy" "{" HierElems "}"    -> RHierarchy  {cons("hierarchy")}
{ HierElem "," }*                -> HierElems
"(" {Id ","}+ ")" "->" "(" {Id ","}+ ")"
                                 -> HierElem    {cons("RoleHier")}
%%SSOD and DSOD relations.
"ssod" "{" { SODElems ","}+   "}"   -> SSOD     {cons("SSOD")}
"dsod" "{" { SODElems ","}+   "}"   -> DSOD     {cons("DSOD")}
"(" { SODElem ","}+  ")" "<""-"">" "(" {SODElem ","}+  ")"
                                    -> SODElems {cons("sepElem")}
Id        -> SODElem     {cons("sepNorm")}
"&&" "(" {Id ","}+ ")"            -> SODElem  {cons("sepAnd")}
```

Listing A.8: ΦRBAC defined syntax

A.1.6 Authorisation Management Models

This part discusses the syntax definition of the Φ-based authorisation management models (Listing A.9) are as follows:

ΦDACAM: This model consists of *objects*, policy terms, policy cases and coverage identical to the ΦDAC model (discussed in section A.6).

ΦMACAM: This model is defined within the curly brackets of the keyword `PhiMACAM`. It *syntactically* uses all the elements of the ΦMAC model (i.e., *lables*, *objects*, *policies and their cases*, and *coverage*) except the authorisation management elements.

```
%% PhiDACAM model
"PhiDACAM"  "{" PhiCObjs PPolicies PhiCases DACCoverage?  "}"
                                          -> PhiAM {cons("PhiDACAM")}

%% PhiMACAM model
"PhiMACAM"  "{" ConLabels? IntLabels? MACObjects
   PPolicies PhiCases MACCoverage? "}" -> PhiAM {cons("PhiMACAM")}

%% PhiRBACAM model
"PhiRBACAM" "{" PhiRoles RHierarchy? SSOD? DSOD? PhiCObjs PPolicies
  PhiCases DACCoverage? "}"  -> PhiAM {cons("PhiRBACAM")}
```

Listing A.9: Φ-based authorisation management models

ΦRBACAM: Listing A.9 shows that, the syntax definition of the ΦRBAC model consists of *roles* and their *cardinalities*, their *optional relations*, *objects*, *policies and their cases* and *coverage*. All these elements described in the ΦRBAC model.

A.1.7 Attachment to WebDSL's Syntax

After defining the syntax of the Φ language, these definitions need to be *included and attached* within the WebDSL's syntax definition, so that it can be the part of the WebDSL's parser (i.e., `WebDSL.tbl`). For this purpose all the Φ-based access and authorisation management models are attached to the WebDSL language by adding them to the WebDSL's `Definition` (Listing A.10). Therefore the Φ-based models are defined *outside* the context of the Web application's page or template elements (page, templates, functions, etc.), as a set of *standalone* models.

```
PhiDACModel     -> Definition
PhiMACModel     -> Definition
PhiRBACModel    -> Definition
PhiAM           -> Definition
```

Listing A.10: Φ models are attached to the WebDSL's definition element

A.2 Z3 Formalism

The Listing A.11 shows all the syntax definition which added to the WebDSL's syntax for generating and the pretty-printing the Z3 formulas and their results.

A.3 Phi.ini

As mentioned before, the file `Phi.ini` needs to be initiated by the developer to cover the following elements:

Path to Model Checker: First the developer needs to define the path to the Z3 model checker, while the Φ compiler, *validates* and *verifies* the defined Web application based on the Z3 (Listing A.12 (See line 1)).

Path to Z3 Parser: The developer should also declare the path to the Z3 parser after the keyword `Z3Parser` (see Listing A.12).

Warning Termination: As mentioned in Chapter 4, Φ for the last two steps of the *validation and verification* phase, can either terminate or continue upon the occurrences of *incompletness* or *coverage* issues. As Listing A.12 shows (lines 3,4,

```
%%Boolean variable
":extrafuns" "("   VElem+   ")"      -> Z3VarDecl      {cons("Z3VarDecl")}
 "(" Id Z3BoolVar    ")"              -> VElem          {cons("VElem")}


%%Z3 Individual Formula

":formula" "(" "("  Z3LogCon
   LBlock* IBlock* ")" ")"           -> Z3IndFormula   {cons("OneFormula")}

"(" Z3LogCon   LogicalElem+   ")"    -> LBlock         {cons("LogicalBlock")}

":formula" "("   Z3FullBody   ")"    -> Z3IndFormula   {cons ("OneFormula")}

"(" Z3LogCon LBlock*  ")"            -> Z3FullBody     {cons ("FormBody")}
"(" Z3LogCon      LogicalElem+ ")"   -> LBlock         {cons ("LogicalBlock")}
"(" "implies"  LogicalElem LBlock ")"
                                     -> LBlock         {cons ("impliesBlock")}

%%LogicalElem Block

"(" Id ")"                           -> LogicalElem    {cons ("PLogicalElem")}
"(" Z3Not Id ")"                     -> LogicalElem    {cons ("PNegLElem")}
Id                                   -> LogicalElem    {cons ("LogicalElem")}
Z3Not Id                             -> LogicalElem    {cons ("NegLElem")}

Z3Formula                            -> Atom           {cons("Z3Check")}

%%For the answer

PreStart*                            -> Start          {cons ("startSymb")}
MCElem* FGMCAns                      -> PreStart       {cons ("preStart")}
FGMCAnsTwo                           -> PreStart

"(" CC ")"                           -> MCElem         {cons ("MCModel")}
"define" Id FGMCBool                 -> CC             {cons ("CheckerC")}

Start                                -> Atom           {cons("ModelChecker")}
```

Listing A.11: Defined syntax related to Z3

6 and 7) the developer can declare the termination in case of a warning (i.e., by using `Terminate`) or continuation (i.e., by using the `Continue`) keyword, after the `Incomp` and/or `Coverage` keywords.

```
"Z3" "@ FPath                   -> Z3Path         {cons ("Z3Path")}
"Z3Parser" "@" FPath            -> Z3Parser       {cons ("Z3Parser")}
"Incomp"   "=" PhiOption        -> IncompOption {cons("IncompOption")}
"Coverage" "=" PhiOption        -> CovOption      {cons ("CovOption")}
"/" {exp "/"}+                  -> FPath          {cons ("FPath")}
"Terminate"                     -> PhiOption      {cons("Terminate")}
"Continue"                      -> PhiOption      {cons("Continue")}
```

Listing A.12: Defined syntax for initialisation file

Appendix B

Validation and Verification in Φ

This Appendix provides all the possible errors and warnings that a developer can trigger based on the defined Φ-based access and authorisation management models.

B.1 Error Handling

This part provides all the types of errors[1] that are handled by Φ during its eight consecutive steps for handling the errors (discussed in 4.2.2).

B.1.1 Multiple Access Control Validation

In this step, *three* types of errors can be find by Φ, based on defined access control models, categorised as follows:

ΦDAC: Multiple ΦDAC models are not allowed in a single Web application, and leads to an error of type 1.1.

ΦMAC: Multiple ΦMAC models are not allowed in a single Web application, and in this case, they triggers the error of type 1.2.

ΦRBAC: Multiple ΦRBAC models are not allowed in a single Web application, that results in the error of type 1.3

B.1.2 Object Scope Validation

In this step, Φ discovers all the errors related to the object scope validation. As following shows, there are *three* main types of errors related to this category:

[1]philang.org/repos/errors

Out of Scope: At first, each used object within the *access control* models are checked against all the object defined within the application code. If any of controlled objects (i.e., *page*, *template*, *group*, *block*, *XML*, *entity*, and *property*) is not used within the application an error is given to the developer.

Authorisation Uniqueness: Φ does not allow objects to be shared within different access control models. Therefore, if two access control models share same object, *directly* or *indirectly* an error will be given to the developer.

Shared Scope: Here Φ validates if the used object within an access control and its authorisation management model are in fact directly or indirectly related. Table B.1 shows all types of errors related to this sub-step.

Table B.1: Different types of errors related to the shared object scopes within elements of access and authorisation management models

Error	**Defined Model**
2.2.1	Φ*DAC and its coverage (2.2.1.1)* Φ*DAC and Authorisation Management (2.2.1.2)*
2.2.2	Φ*MAC and its coverage (2.2.2.1)* Φ*MAC and Authorisation Management (2.2.2.2)*
2.2.3	Φ*RBAC and its coverage (2.2.3.1)* Φ*RBAC and Authorisation Management (2.2.3.2)*
2.2.4	Φ*DACAM and its coverage*
2.2.5	Φ*MACAM and its coverage*
2.2.6	Φ*RBACAM and its coverage*

B.1.3 Validation of Object-Operation Relation

In this step, Φ checks if there is any relation between data manipulative operations and non-data model objects. Table B.2 shows the error types related to the location of the illigal relation within diffrerent Φ-based models.

Table B.2: Different types of errors related to the object-operation relation validation within elements of access and authorisation management models

Error	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**	**3.6**
Defined Model	**ΦDAC**	**ΦMAC**	**ΦRBAC**	**ΦDACAM**	**ΦMACAM**	**ΦRBACAM**
	3.1.1 coverage (3.1.2)	*3.2.1 coverage (3.2.2)*	*3.3.1 coverage (3.3.2)*	*3.4.1 coverage (3.4.2)*	*3.5.1 coverage (3.5.2)*	*3.6.1 coverage (3.6.2)*

B.1.4 Policy Terms Correctness Validation

This step checks the correctness of each policy term within each model as follows:

Out of Context Validation: As mentioned here, Φ checks for the the policy terms that are not related to the access control semantics, such as using a *role* within the policy terms of an ΦDAC model. Table B.3 shows, the error types are categorised based on the defined access or authorisation management model and its semantical scope.

Table B.3: Different types of errors related to the out of context validation within elements of access and authorisation management models

Error	**4.1.1**	**4.1.2**	**4.1.3**	**4.1.4**	**4.1.5**	**4.1.6**
Defined Model	**ΦDAC**	**ΦMAC**	**ΦRBAC**	**ΦDACAM**	**ΦMACAM**	**ΦRBACAM**
	4.1.1.1 coverage (4.1.1.2)	*4.1.2.1 coverage (4.1.2.2)*	*4.1.3.1 coverage (4.1.3.2)*	*4.1.4.1 coverage (4.1.4.2)*	*4.1.5.1 coverage (4.1.5.2)*	*4.1.6.1 coverage (4.1.6.2)*

Duplicaitons: As mentioned before, here Φ checks for repetitions within the policies of the access control models, their coverage and authorisation management and their coverage. Table B.4 shows the error types that are categorised based on the defined access or authorisation management model.

Table B.4: Different types of errors related to the duplication validation within elements of access and authorisation management models

Error	**4.2.1**	**4.2.2**	**4.2.3**	**4.2.4**	**4.2.5**	**4.2.6**
Defined Model	**ΦDAC**	**ΦMAC**	**ΦRBAC**	**ΦDACAM**	**ΦMACAM**	**ΦRBACAM**
	4.2.1.1 coverage (4.2.1.2)	*4.2.2.1 coverage (4.2.2.2)*	*4.2.3.1 coverage (4.2.3.2)*	*4.2.4.1 coverage (4.2.4.2)*	*4.2.5.1 coverage (4.2.5.2)*	*4.2.6.1 coverage (4.2.6.2)*

User Identifier Validation: Φ checks the correctness of *user identifier-based* policy terms, within ΦDAC and ΦDACAM models.

Attribute-based Validation: Here Φ validates the attribute-based terms. As Table B.5 shows, similar to the previous steps, the error types are categorised based on the Φ model they are defined within.

Table B.5: Different types of errors related to the validation of the out of *attribute-based* policy terms within the access and authorisation management models

Error	**4.4.1**	**4.4.2**	**4.4.3**	**4.4.4**	**4.4.5**	**4.4.6**
Defined Model	**ΦDAC**	**ΦMAC**	**ΦRBAC**	**ΦDACAM**	**ΦMACAM**	**ΦRBACAM**
	4.4.1.1 coverage (4.4.1.2)	*4.4.2.1 coverage (4.4.2.2)*	*4.4.3.1 coverage (4.4.3.2)*	*4.4.4.1 coverage (4.4.4.2)*	*4.4.5.1 coverage (4.4.5.2)*	*4.4.6.1 coverage (4.4.6.2)*

B.1.5 Access Element Correctness Validation and Verification

As mentioned before, at this step Φ validate and verifies the ΦMAC, ΦMACAM, ΦRBAC, and ΦRBACAM models. This part, categorises this type of error and give a number of examples.

Role and Label Duplication Validation: Table B.6 categorises different types of errors in the ΦRBAC, ΦRBAC, ΦMAC, ΦMACAM, based on duplications in role and label definitions.

Table B.6: Different types of errors related to the duplication validation within ΦMAC, ΦMACAM, ΦRBAC, and ΦRBACAM models

Error	**5.1.1**	**5.1.2**	**5.1.3**	**5.1.4**
Defined Model	**ΦMAC**	**ΦRBAC**	**ΦMACAM**	**ΦRBACAM**
Duplication Validation	*5.1.1.1 (Condidentiality labels)* *5.1.1.2 (Integrity labels)*	*5.1.2.1 (roles)* *5.1.2.2* (hierarchy) *5.1.2.3 (SSOD)* *5.1.2.4 (DSOD)*	*5.1.3.1 (Condifentiality labels)* *5.1.3.2 (Integrity labels)*	*5.1.4.1 (roles)* *5.1.4.2* (hierarchy) *5.1.4.3 (SSOD)* *5.1.4.4 (DSOD)*

Shared Label: Label names can not be shared between the confidentiality and integrity labels within the MAC-based models. These errors are divided into two categories based on the type of the model (see Table B.7).

Table B.7: Two types of errors related to the validation of shared labels within the defined *confidentiality* and *integrity* labels

Error	**5.2.1**	**5.2.2**
Defined Model	**ΦMAC**	**ΦMACAM**
Shared Labels	*5.2.1*	*5.2.2*

Object Label Errors: This relates to the object label errors and duplications.

Table B.8: Two types of errors related to the validation of the defined objects within the ΦMAC and ΦMACAM models

Error	**5.3.1**	**5.3.2**
Defined Model	**ΦMAC**	**ΦMACAM**
Object-Label Error	*5.3.1*	*5.3.2*

Hierarchy Verification: This type (see TableB.9) of error relates to the hierarchy verification within ΦRBAC and ΦRBACAM.

Table B.9: Two types of errors related to the roles' hierarchy verification within the ΦRBAC and ΦRBACAM models

Error	**5.4.1**	**5.4.2**
Defined Model	**ΦRBAC**	**ΦRBACAM**

B.1.6 Validation and Verification for the Correctness of Each Case

This phase checks the correctness of each policy and coverage case within the Φ's access and authorisation management models. This part discusses the sub-steps of this validation and verification phase based on the *type* of the Φ's models.

ΦDAC and ΦDACAM Models

This step verifies each defined *policy* and *coverage* case of the ΦDAC and ΦDACAM models, based on the following consecutive sub-steps:

User Identifier Verification: This sub-step verifies within each case, if two user identifiers are *true* at the same time. As Table B.10 shows, two errors are given to the developer based on the type of the model and the location of the error (i.e., *policy* or *coverage* case).

Table B.10: Four errors related to the user identifier verification within the ΦDAC and ΦDACAM models

Error	**6.1**	**6.4**
Defined Model	**ΦDAC**	**ΦDACAM**
User Identifier Error	*6.1.1 (policy)* *6.1.2 (coverage)*	*6.4.1 (policy)* *6.4.2 (coverage)*

Attribute-based Verification: This sub-step verifies each case based on the attribute-based policy terms and their signs based on the following types of attributes:

- *User-based Attribute Verification:* Here the user-based attributes are checked within each case. Table B.11 shows the errors based on the type of the model and their location.

Table B.11: Two types of errors related to the user-based attributes verification within the ΦDAC and ΦDACAM models

Error	**6.1**	**6.4**
Defined Model	**ΦDAC**	**ΦDACAM**
User-based attributes Error	*6.1.3 (policy)* *6.1.4 (coverage)*	*6.4.3 (policy)* *6.4.4 (coverage)*

- *Data-based Attribute Verification:* Here the data-based attributes policy terms and their related policy signs are checked, and in the case of an error, the error message is given to the developer based on the type of the model and the location of the error, as shown in Table B.12.

Table B.12: Four different error messages related to the data-based attributes verification within the ΦDAC and ΦDACAM models

Error	**6.1**	**6.4**
Defined Model	**ΦDAC**	**ΦDACAM**
Data-based attributes Error	*6.1.5 (policy)* *6.1.6 (coverage)*	*6.4.5 (policy)* *6.4.6 (coverage)*

- *User- and data-based Comparison Verification:* Here each policy term, which is the comparison between the user- and data-based attributes and their policy signs, are checked, and in the case of an error, the error message is given to the developer based on the type of the model and the location of the error (see Table B.13).

Table B.13: Four different error messages related to the comparisons between user- and data-based attributes within the ΦDAC and ΦDACAM models

Error	**6.1**	**6.4**
Defined Model	**ΦDAC**	**ΦDACAM**
User-based attributes Error	*6.1.7 (policy)* *6.1.8 (coverage)*	*6.4.7 (policy)* *6.4.8 (coverage)*

- *System-based Verification* This part checks the system-based policy terms (i.e., *time* or *date*) and their related policy signs, and in the case of an error, the error message is given to the developer based on the types of the model and the location of the error as shown in Table B.14.

Table B.14: Eight different error messages related to the system-based attributes verification within the ΦDAC and ΦDACAM models

Error	**6.1**	**6.4**
Defined Model	**ΦDAC**	**ΦDACAM**
System-based attributes Error	*6.1.9 (policy (time)) 6.1.10 (coverage (time)) 6.1.11 (policy (date)) 6.1.12 (coverage (date))*	*6.4.9 (policy (time)) 6.4.10 (coverage (time) 6.4.11 (policy (date)) 6.4.12 (coverage (date)*

ΦMAC and ΦMACAM Models

This step checks the correctness of each defined policy and coverage case within the ΦMAC and ΦMACAM models, based on the following consecutive sub-steps:

Allowed Operation Validation: Here the not allowed operations (i.e., *update* and *delete*) are checked within the ΦMAC and ΦMACAM models, and the appropriate error is given to the developer as categorised in Table B.15.

Table B.15: Different error messages related to the un allowed operations within ΦMAC and ΦMACAM models

Error	**6.2**	**6.5**
Defined Model	**ΦMAC**	**ΦMACAM**
Not allowed Operation Error	*6.2.1 (policy)* *6.2.2 (coverage)*	*6.5.1 (policy)* *6.5.2 (coverage)*

MAC Property Verification: Within this sub-step, the information flow of both *confidentiality* and *integrity* dominance levels are checked based on *policy signs* within each policy or coverage case. The error is based on the fact, that the upper levels have + signs while their lower levels have - sings. In this case, the error message is given to the developer based on the type of the model and the location they are in, as shown in Table B.16.

Table B.16: Different error types related to the MAC property verification within ΦMAC and ΦMACAM models

Error	**6.2**	**6.5**
Defined Model	**ΦMAC**	**ΦMACAM**
Not allowed Operation Error	*6.2.3 (policy - conf)* *6.2.4 (coverage - conf) 6.2.5 (policy - Integ)* *6.2.6 (coverage - Integ)*	*6.5.3(policy - Conf)* *6.5.4 (coverage - Conf) 6.5.5(policy - Integ)* *6.5.6 (coverage - Integ)*

Strict-Property Validation: After checking the policy signs related to the information flow and the dominance levels, this sub-step checks for the *strict-property* validation. Strict-property validation checks if for an object the related operations is a set of *Create and Read* with in a policy case, then the related policies within that case *must* be equal to the assigned labels to the object. In case of an error, the error message is given to the developer based on the type of the model and the location of the error; as illustrated in Table B.17.

Confidentiality Verification: In this sub-step the *confidentiality-related* properties are checked, as following:

Table B.17: Different error types related to the strict-property validation within ΦMAC and ΦMACAM models

Error	**6.2**	**6.5**
Defined Model	**ΦMAC**	**ΦMACAM**
Strict-property verification	*6.2.7 (policy - conf)* *6.2.8 (coverage - conf) 6.2.9 (policy - Integ)* *6.2.10 (coverage - Integ)*	*6.5.7(policy - Conf)* *6.5.8 (coverage - Conf) 6.5.9 (policy - Integ)* *6.5.10 (coverage - Integ)*

- *Read-down property Verification:* Here the read-down concept within the defined ΦMAC or ΦMACAM is checked, and in the case of an error the error message is given to the developer based on the type of the model and the location of the error (see Table B.18).

Table B.18: Different error types related to the *read-down* property within ΦMAC and ΦMACAM models

Error	**6.2**	**6.5**
Defined Model	**ΦMAC**	**ΦMACAM**
Read-down property verification error	*6.2.11 (policy)* *6.2.12 (coverage)*	*6.5.11 (policy)* *6.5.12 (coverage)*

- *Write-up Property Verification:* Here the write-up concept within the ΦMAC and ΦMACAM is checked, and in the case of an error, based on the location of the error and the model it is in, an error message is given to the developer. All the sub-types of these errors are categorised within the Table B.19.

Table B.19: Different error types related to the *write-up* property within ΦMAC and ΦMACAM models

Error	**6.2**	**6.5**
Defined Model	**ΦMAC**	**ΦMACAM**
Write-up property verification error	*6.2.13 (policy)* *6.2.14 (coverage)*	*6.5.13 (policy)* *6.5.14 (coverage)*

Integrity Verification: In this sub-step the *integrity-related* properties are checked, as following:

- *Read-up property Verification:* The read-up concept with the defined policy cases are checked, and in the case of an error, an error message is given to the developer based on the type of the model and the location of the error (see Table B.20).

Table B.20: Different error types related to the *read-up* property within ΦMAC and ΦMACAM models

Error	**6.2**	**6.5**
Defined Model	**ΦMAC**	**ΦMACAM**
Not allowed Operation Error	*6.2.15 (policy)* *6.2.16 (coverage)*	*6.5.15 (policy)* *6.5.16 (coverage)*

- *Write-down property Verification:* The write-down property of the defined ΦMAC and ΦMACAM models are checked here based on the defined policy or coverage cases and their related objects and their security clearance labels. In case of an error, as Table B.21 shows, the error message is given to the developer based on the type of the model and the location of the error.

Table B.21: Different error types related to the *write-down* property within ΦMAC and ΦMACAM models

Error	**6.2**	**6.5**
Defined Model	**ΦMAC**	**ΦMACAM**
Not allowed Operation Error	*6.2.17 (policy)* *6.2.18 (coverage)*	*6.5.17 (policy)* *6.5.18 (coverage)*

Attribute-based Verification: Identical to the ΦDAC and ΦDACAM models, the attribute-based policies and their related policy signs are checked; and in case of an error, the error message is given to the developer based on their location as categorised in Table B.22.

ΦRBAC and ΦRBACAM Models

This step validates and verifies the correctness of each defined policy and coverage case within the ΦRBAC and ΦRBACAM models, based on the following consecutive sub-steps:

Hierarchy and SSOD/DSOD Verification: As mentioned in Chapter 4 , at first the logical state of each policy and/or coverage case is checked based on the defined roles' relations (i.e., hierarchy, SSOD, DSOD) and the policy signs. In the case of an error, as Table B.23 categorised, the error message is given to the developer based on the location of the error.

Attribute-based Verification: Identical to the ΦDAC, ΦDACAM, ΦMAC, and ΦMACAM, the attribute-based policy terms and their related policy signs are verified as shown Table B.24 an error is given to the developer based on the location of the error.

Table B.22: Different error types related to the attribute-based policy terms and their signs within ΦMAC and ΦMACAM models

Error	**6.2**	**6.5**
Defined Model	**ΦMAC**	**ΦMACAM**
User-based attributes Error	*6.2.19 (policy) 6.5.20 (coverage)*	*6.5.19 (policy) 6.5.20 (coverage)*
Data-based attributes Error	*6.2.21 (policy) 6.5.22 (coverage)*	*6.5.21 (policy) 6.5.22 (coverage)*
User data attributes Comparisons	*6.2.23 (policy) 6.5.24 (coverage)*	*6.5.23 (policy) 6.5.24 (coverage)*
Sys-based attributes Error	*6.2.24 (policy (time)) 6.2.25 (coverage (time)) 6.2.26 (policy (date)) 6.2.27 (coverage (date))*	*6.2.24 (policy (time)) 6.2.25 (coverage (time) 6.4.26 (policy (date)) 6.4.27 (coverage (date)*

Error	**6.3**	**6.6**
Defined Model	**ΦRBAC**	**ΦRBACAM**
Not allowed Operation Error	*6.3.1 (policy) 6.3.2 (coverage)*	*6.6.1 (policy) 6.6.2 (coverage)*

Table B.23: Different error types related to the relation and policy signs checks within ΦRBAC and ΦRBACAM models

B.1.7 Overlapping Validation and Verification

As discussed in Chapter 4, this step checks the overlapping within the Φ-based models. the following three parts shows the categorisation of the error messages within different types of the Φ models, in regarding to the *overlapping* validation and verification checks.

ΦDAC and ΦDACAM Models

This step validates the overlapping cases within ΦDAC and ΦDACAM models' policy and coverage cases. As mentioned in the Chapter 4, these checks are carried out by Φ compiler without using the model checker for these models. In case of an error, the error message is given to the developer based on the location of the error as illustrated in Table B.25.

Table B.24: Different error types related to the attribute-based policy terms and their signs the ΦRBAC and ΦRBACAM models

Error	**6.3**	**6.6**
Defined Model	**ΦRBAC**	**ΦRBACAM**
User-based attributes Error	*6.3.3 (policy)* *6.3.4 (coverage)*	*6.6.3 (policy)* *6.6.4 (coverage)*
Data-based attributes Error	*6.3.5 (policy)* *6.3.6 (coverage)*	*6.6.5 (policy)* *6.6.6 (coverage)*
User data attributes Comparisons	*6.3.7 (policy)* *6.3.8 (coverage)*	*6.6.7 (policy)* *6.6.8 (coverage)*
Sys-based attributes Error	*6.3.9 (policy (time))* *6.3.10 (coverage (time))* *6.3.11 (policy (date))* *6.3.12 (coverage (date))*	*6.6.9 (policy (time))* *6.6.10 (coverage (time)* *6.6.11 (policy (date))* *6.6.12 (coverage (date)*

Table B.25: Different error types related to *overlapping* cases within ΦDAC and ΦDACAM models

Error	**7.1**	**7.4**
Defined Model	**ΦDAC**	**ΦDACAM**
Overlapping cases	*7.1.1 (policy)* *7.1.2 (coverage)*	*7.4.1 (policy)* *7.4.2 (coverage)*

B.1.8 ΦMAC and ΦMACAM Models

This step checks the overlapping cases within the ΦMAC and ΦMACAM models. As discussed before, for finding these errors, the Φ language turns the defined models into *first-order logical formulas* and then check for their *satisfiability* through the model checker Z3 [62]. In case of *SAT* an error is given to the developer, based on the error and the location of it. The Table B.26 categorises all these errors based on the type of the model.

Table B.26: Different error types related *overlapping* cases within ΦMAC and ΦMACAM models

Error	**7.2**	**7.5**
Defined Model	**ΦMAC**	**ΦMACAM**
Overlapping cases	*7.2.1 (policy)* *7.2.2 (coverage)*	*7.5.1 (policy)* *7.5.2 (coverage)*

B.1.9 ΦRBAC and ΦRBACAM Models

Within this phase, the ΦRBAC and/or ΦRBACAM models are verified for the *overlapping* cases. As discussed in Chapter 4, similar to the Φ's MAC-based models, Φ checks the overlapping cases of these models through a model checker by transforming these models, paring every two cases (i.e., one for policy and one for coverage cases) and checks for the *satisfiability* through the model checker Z3 [62]. In case of *SAT*, an error is given to the developer in terms of the defined model, based on the location of an error. The Table B.27 categorises all these errors based on the type of the model.

Table B.27: Different error types related *overlapping* cases within ΦRBAC and ΦRBACAM models

Error	**7.3**	**7.6**
Defined Model	**ΦRBAC**	**ΦRBACAM**
Overlapping cases	*7.3.1 (policy)* *7.3.2 (coverage)*	*7.6.1 (policy)* *7.6.2 (coverage)*

B.1.10 Dead Authorisation Code Verification

This step checks the *dead authorisation code* within the Φ's access and authorisation management models.As discussed in Chapter ?? the overall approach for these errors within the Φ's access control and authorisation management models are similar. For the concept of the dead authorisation code within the access control models please refer to Chapter 4. Table B.28 shows three types of categorises of these errors based on the location of these errors.

Table B.28: Different types of errors with the Φ's access and authorisation management models

Error	**8.1**	**8.2**	**8.3**	**8.4**	**8.5**	**8.6**
Defined Model	**ΦDAC**	**ΦMAC**	**ΦRBAC**	**ΦDACAM**	**ΦMACAM**	**ΦRBACAM**
	8.1	*8.2*	*8.3*	*8.4*	*8.5*	*8.6*

B.2 Warning Handling

This part provides all the types of warnings[2] that are handled by Φ during its two consecutive steps of the warning handling phase.

[2]philang.org/repos/test-warnings

B.2.1 Completeness Verification

As discussed within the validation and verification phase, this step checks the completeness of the policy cases within Φ's access and authorisation model, by transforming these models into *first-order logic* formulas and gives a set of warning and provides the missing cases to the developer. The table B.29 shows this warning categorises into six types of error messages, based on the type of the model.

Table B.29: Different categories of warnings related to the completeness verification within Φ models

Warning	**9.1**	**9.2**	**9.3**	**9.4**	**9.5**	**9.6**
Defined Model	**ΦDAC**	**ΦMAC**	**ΦRBAC**	**ΦDACAM**	**ΦMACAM**	**ΦRBACAM**
	9.1	*9.2*	*9.3*	*9.4*	*9.5*	*9.6*

B.2.2 Coverage Verification

As mentioned in Chapter 4 the developer has the option to define a set of *coverage* cases for the defined access and/or authorisation management model. So, during the last step of the validation and verification phase within the Φ language, the defined access and/or authorisation management models are checked against the coverage cases. As the Table **??** shows, this error can occur within each defined Φ-based model and therefore it is categorised into six different error messages based on the type of the model.

Table B.30: Six different categories of warning related to the coverage verification within Φ models

Warning	**10.1**	**10.2**	**10.3**	**10.4**	**10.5**	**10.6**
Defined Model	**ΦDAC**	**ΦMAC**	**ΦRBAC**	**ΦDACAM**	**ΦMACAM**	**ΦRBACAM**
	10.1	*10.2*	*10.3*	*10.4*	*10.5*	*10.6*

Appendix C

Defined Data, Access, and Authorisation Management Models

This appendix describes the defined data models for the two case studies that were presented in Chapter 5.

C.1 Academic Research Group

This part of the thesis, first describes the defined data model for the research group case study, and the fine-grained access control requirements for the defined entities and their properties. Second, based on all the requirements a Φ-based access control model and its authorisation management model is constructed. The data model is divided into *user-* and *activity-based* models, so the next two parts explain these models, and their fine-grained access control requirements. Moreover at the end of this part the defined access control and authorisation management model is presented.

C.1.1 User-based data models and their access control requirements

This part of the thesis discusses the data model for different types of users (see Figure C.1) within the research group. The general information about each user is stored in the `Person` entity and also other type of users have their own representatives. The rest of this section presents the defined entities, their properties and their access control requirements.

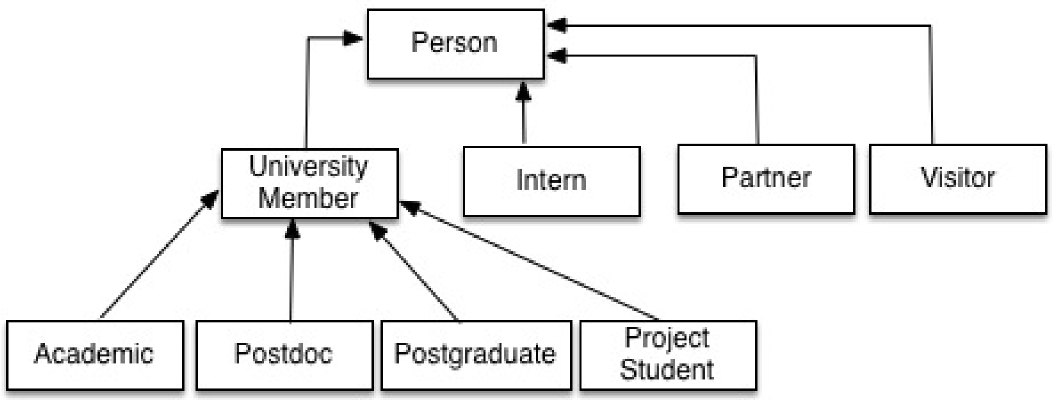


Figure C.1: Different user types in the PLUS data model

C.1.1.1 Person

The entity `Person` holds the general and common information among the users of the web application, such as `username` and `password`. The information is filled by the user who likes to register herself. Some users might register to the system to apply for a project they are interested in and therefore I have, a property `registerNote`, so that the user can explain why he/she is registering for the membership. The personal information such as personal telephone number and email are optional, so the user can choose not to fill them without interfering with the user registration.

```
entity Person {
  username      :: String        (id)
  password      :: Secret
  nickname      :: String        (name)
  title         :: String
  firstname     :: String
  lastname      :: String
  dob           :: Date           (optional)
  persTel       :: String        (optional)
  persAddress   :: String        (optional)
  persEmail     :: String        (optional)
  registerNote  :: WikiText
  profPicture   :: URL
  seminars      -> Set<Seminar>  (optional)
}
```

Listing C.1: Person data model

Access Control Requirements. Each user of the system must be able to use *read*, *update*, and *delete*) operations on the all the *properties* of the entity `Person` that are related to herself. Note that the *create* operation on the `Person` entity is *not* access controlled as any *outside* user can register to the system. Moreover the users with the role *UniversityMember* can read the following properties: `dob`, `persTel`, `persAddress`, `persEmail`, `Seminars`. Furthermore, the *outside* users can *read* the following properties: `title`, `firstname`, `lastname`, `profilePicture`.

C.1.1.2 Partner

The entity `Partner` represents the partner that are collaborating the members of the system. It inherits from the entity `Person` and therefore can use its properties as well. The partners can be a member and/or a sponsor of a set of projects.

```
entity Partner : Person {

 partnerID      :: String
 memberOf       -> Set<Project>
 sponserOf      -> Set<Project>
 description    :: WikiText
}
```

Listing C.2: Partner data model

Access control requirements. The users with the following roles can operate on this entity:

Project Manager: A user with a role *ProjectManager* can use CRUD operations on this entity and its properties.

Partner: The user with the role *Partner* can *update* his/her description. Also the user can *read* an instance of the entity partner and its properties.

Academic: Only the users with the role *Academic* can *read* the instances of this entity and its properties.

C.1.1.3 Visitor

The entity `Visitor` represents the visitors of the research group. As Listing C.3 shows, the properties of this entity hold the information regarding the visitor users such as the user they want to visit (i.e., `visiting`), and by whom they were invited (i.e., `invitedBy`).

Access control requirements. The users with the following roles can operate on this entity:

Academic: The users with the role *Academic* can use the *CRUD* operations on the `Visitor` entity and its properties.

```
entity Visitor : Person {

  visitorID     :: String (name)
  from          :: Date
  until         :: Date
  visiting      -> Set<Person>
  invitedBy     -> Set<Academic>
  depName       :: String
  schoolName    :: String
  viva          -> Set<Viva> (optional) //Not all visitors are examiners!
}
```

Listing C.3: Visitor data model

University Member: The users with this role can *read* the name of the visitors (i.e., `Visitor.title`, `Visitor.firstname`, `Visitor.lastname` properties) and their visit duration (instances of `from` and `until` properties).

Visitor: A user with the role *Visitor* are able to *read* all their instances and *update* their department and school names (i.e., `depName`, `schoolNames` properties). Note that the users with the role `ExternalExaminer` are not able to operate on the property `viva` and their access described within the `Viva` entity.

C.1.1.4 Intern

The `Intern` entity represents a set of users that are doing internship with academics. So its properties represent the information for this type of user, such as `researchTitle` and `managers`.

```
entity Intern : Person {

  managers          -> List<Academic>
  researchTitle     :: String
  researchSummary   :: WikiText
  startingDate      :: Date
  finishingDate     :: Date
}
```

Listing C.4: Intern data model

Access Control Requirements. The users with the following roles can operate on the properties of this entity:

Head of Group or Academic: A user with the role *HeadOfGroup* or *Academic* are able to use the *CRUD* operation on the `Intern` entity and its properties.

Intern role with user and data attributes: The user with the role *Intern* is able to *read* all the properties of this entity when they are related to herself. Moreover, the user is able to *update* his/her research title (i.e., `researchTitle` property) and summary (i.e., `researchSummary` property).

C.1.1.5 University Member

The entity `uniMemeber` represents the users who are university members and stores their information. In our system a university member is a person that has an university ID and email from both `university` and `school` that he/she is involved in. Also, the member can do a set of activities such as events and vivas. The properties in the `uniMember` entity represent the set of data that are linked to the information and activities that a university member can have such as `publications`.

```
entity UniMember : Person {

  uniID          :: String (name)
  uniEmail       :: Email
  schoolEmail    :: Email
  firstWrote    -> Set<Publication>  (inverse = Publication.firstAuthor)
  publications  -> Set<Publication>  (optional)
  events        -> Set<Event>        (optional)
  vivas         -> Set<Viva>         (optional)
  projects      -> Set<Project>      (optional)
  workAddress   -> Address           (optional)
  ecsEmail       :: Email
  homePage       :: URL
}
```

Listing C.5: Data model related to the university members

Access control requirements. The users of the system with the following roles can operate on the `uniMember` entity and its properties:

Secretary: Each user with the role *Secretary* is able to *create*, and *update all* the properties related to the university members. Moreover, the user is able to *read* the `vivas` property.

Head of Group: The user with the role *HeadOfGroup* is able to *delete* an instance of this entity and therefore forever suspend a university member from the system. Moreover, the user is able to *read* the *vivas* property.

Academic Each user with the role *Academic* is able to use CUD operations on the following properties: `firstWrote`, `publications`, `vivas`, `workaddress`, and `homepage`. Moreover the user with the role *Academic* is able to read the instances of the `vivas` property. Note that other properties are readable throughout the application by any type of user, so the academics are able to read them before they use the CUD operations.

University Members and their own data: The users with the role *UniversityMember* are able to *update* their homepage address (i.e., `homepage` property).

C.1.1.6 Academic

The entity `Academic` belongs to academic users and holds their information. The properties related to this entity represents the set of tasks that the academic can do such as managing a set of postdocs or supervising a set of postgraduates. The property `onlineBlurb` shows a short introduction of user among other academics. The property `bio` holds the whole introduction of the academic.

```
entity Academic : UniMember {

  staffID       :: String
  uniAddress    :: WikiText
  uniTel        :: String
  bio           :: WikiText
  contractFrom  :: Date
  contractDue   :: Date               (optional)
  mscProject    -> Set<Msc>           (inverse = Msc.supervisor)
  phds          -> Set<Postgrad>      (inverse = Postgrad.supervisor)
  postdocs      -> Set<Postdoc>       (inverse = Postdoc.manager)
  visitors      -> Set<Visitor>       (inverse = Visitor.invitedBy)
  interns       -> Set<Intern>        (inverse = Intern.managers)
  allocatedViva -> Set<Viva>          (inverse = Viva.internalExaminer)
  onlineBlurb   :: WikiText           //Specific for PLUS academics
  fullProfilo   :: WikiText           //Specific for PLUS academics
  interests     -> Set<Interest>      (inverse = Interest.relatedAcademics)
  position      :: String
  superProjects -> Set<Project>
  firstAWrote   -> Set<Publication>   (inverse = Publication.firstAAuthor)
}
```

Listing C.6: Academic data model

Access control requirements. The users with the following roles can operate on this entity:

Head of Group: A user with a *HeadOfGroup* role is able to use the *CUD* operation on the following properties: `contractFrom`, `contractDue`, `position` , and which project they are supervising (`superProject`). Moreover, the user with role is able to *read* the following properties: `contractFrom`, and `contractDue`.

Academic: A user with the role *Academic* is able to use the *CUD* operations on the following properties: `uniAddress`, `uniTel`, `bio`, `mscProject`, `phds`, `postDocs`, `visitors`, `interns`, `allocatedViva`, `onlineBlurb`, `fullProfilo`, `interests`, and `firstAWrite`.

Note that at this level some properties of the entity `Academic` are not access controlled, because they are fine-grainulary contrlloed at their own level. For example, any user with any permission is able to read the name of the Ph.D. students of an academic. So here instead of controlling the property `phds` of type `Postgrad` entity, the fine-grained access controlled within the entity `Postgrad`.

C.1.1.7 Postdoc

The information related to the post doc user, such as manager, is stored in the entity `Postdoc`. Even though there might be the case that the research title is not necessary, the postdoc entity has a property `researchArea` that is devoted to the research direction of the postdoc user.

```
entity Postdoc : UniMember {

  manager           -> Academic
  researchTitle     :: String
  researchAbstract  :: WikiText  (optional)
  startingDate      :: Date
  finishingDate     :: Date       (optional)
}
```

Listing C.7: Defined data model for postdocs

Access control requirements. The users with the following roles can operate on the properties of the entity `postdoc`:

Head of Group: A user with the role *HeadOfGroup* is able to use the *create* and *delete* operations on the `Postdoc` entity. Moreover, the user as the head of the group, must agree on each postdoc manager. Therefore the user with this role is able to *update* the entity `manager`.

Academic: A user with the role *Academic* must be able to use *t*he *update* operationbg on the following properties of a postdoc: `reasearchTitle`, `researchAbstract`, `startingDate`, and `finishingDate`.

Postdoc and related data: Also the user with the role *postdoc* must be able to *read* all the properties of this entity when they are related to him/herself. Moreover, the user can *update* her research title (i.e., `researchTitle`) and her research abstract (i.e., `researchAbstract`).

C.1.1.8 Postgraduate

The entity `Postgrad` stores the information about the postgraduate user. The information stored in its properties are related to a set of general information that are required for postgraduates, such as their `supervisor` and `adviser`. There are a set of properties that are optional as they do not need to be filled during adding the postgraduate record.

```
entity Postgrad : UniMember {

  supervisor        -> Academic
  adviser           -> Academic  (optional)
  researchTitle     :: String
  researchAbstract  :: WikiText
  startingDate      :: Date
  finishingDate     :: Date
  nineMSubDate      :: Date      (optional)
  nineMReview       :: WikiText  (optional)
  transSubDate      :: Date      (optional)
  transReview       :: WikiText  (optional)
  thesisSubDate     :: Date      (optional)
  thesisReview      :: WikiText  (optional)
  goesNominal       :: Date      (optional)
  reason            :: WikiText  (optional)
  extendedUntil     :: Date      (optional)
  internalExaminer -> Academic
  externalExaminer -> Visitor
}
```

Listing C.8: Defined data model for postgraduate students

Access control requirements. The users with the following roles are able to operate on the properties of the entity `postgrad`:

Head of Group: A user with the role *HeadOfGroup* must be able to use the *create* and *delete* operation on the entity `Postgrad`. Moreover the user is able to use the *update* operation on the following properties: `supervisor`,`adviser`, and `goesNominal`. Furthermore the user with this role can *read* the property `thesisReview`. Moreover the user with this role can *read* the following properties: `startingDate`, `finishingDate`, `nineMSubDate`, `nineMReview`, `transSubDate`, `transReview`, `thesisSubDate`, `thesisReview`, `goesNominal`, `reason`, `extendedUntil`, `internalExaminer`, `externalExaminer`.

Supervisor: A user with the role *Supervisor* must be able to use the *update* operation on the following properties: `startingDate`, `finishingDate`, `nineMSubDate`, `nineMReview`, `transSubDate`, `transReview`, `thesisSubDate`, `internalExaminer`, and `externalExaminer`. Moreover the user with this role can *read* the property `thesisReview`. Moreover, the user with this role has the *same read* authorisation rights with the user with the role *HeadOfGroup*.

External Examiner: The external examiner is able to *read*, *update*, and *delete* the property `thesisReview`.

Postgraduate: A user with the role *postgraduate* can *update* the `researchTile` and `researchSummary` properties. Moreover, the user can read the following properties: `startingDate`, `finishingDate`, `nineMSubDate`, `nineMReview`, `transSubDate`,`transReview`, `thesisSubDate`, `thesisReview`, `goesNominal`, `reason`, `extendedUntil`, `internalExaminer`, `externalExaminer`.

Note that any user with or without any permission can *read* the following properties: `supervisor`, `advisor`, `researchTitle` and `researchAbstract`.

C.1.1.9 Project Student

The entity `ProjectStudent` represents the projects that are taken by the undergraduate or graduate students. So, its properties include the information related to projects such as title (i.e., `projectTitle`) and `supervisor`.

```
entity ProjectStudent: UniMember {

  projectTitle    :: String (name)
  summary         :: WikiText
  startingDate    :: Date
  finishingDate   :: Date
  supervisor      -> Academic
  examiner        -> Academic
  projectReview   :: WikiText
  projectMark     :: Float
}
```

Listing C.9: Defined data model for master degree students

Access control requirements. The users with the following roles can operate on the properties of this entity:

Head of Group: The user with the role *HeadOfGroup* is able to *read* all the instances of the entity `ProjectStudent` and its properties.

Academic: In this system, any academic is able to supervise an Msc or an undergraduate project. So a user with the role *Academic* is able to use the *create*, *delete* operation on the entity `PorojectStudent`; and also *update* its properties. Moreover the user with the role *Academic* can read *all* the instances of this entity.

Msc or Bsc Student: A user with the role *MscStudent* or *BscStudent* is able to *update* the summary (i.e., `summary` property) and title of his/her project (i.e., `projectTitle` property).

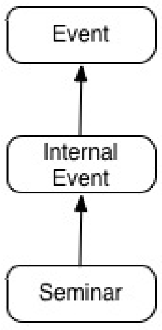


Figure C.2: Event hierarchy in the research group data-model

C.1.2 Activity-based data models and their access control requirements

This part shows and discusses the activity-based data models related to the research group. Moreover it provides the fine-grained authorisation of each activity-based entity and its related properties.

C.1.2.1 External and Internal Events and Seminars

As Listing C.10 shows, the entity `Event` represents the general information about the events, such as a starting date. The property `endDate` is optional because there are some events for a day and within a period of time. I also have properties `startTime` and `finishTime` to present the time of the event. The property `blurb` is a very short introduction that briefly represents the event. Based on the `blurb` property the user can go to the part of the application that represents the event and see the full description that is stored in the property `eventNote`.

Moreover, there are some events that are hold internally such as the entity `InternalEvent`, which inherits from `Event` (Figure C.2), represents these events and its properties store the data related for such events such as `abstract` of an event. The internal events have a set of organisers and sponsors of university member and partner entities. The invitees are from set of university members, however I use the value `0` to indicate the event is open for public. Furthermore, The entity `Seminar` inherits from `InternalEvent` entity and represents stores the speaker information of person entity.

Access control requirements. The users with the following roles can operate on the above entities (`Event`, `InternalEvent`, and `Seminar`):

Secretary: The secretary of the research group has a task to organise *all* the events that are related to the research group. So a user with the role *secretary* is able

```
entity Event {

    title        :: String (name)
    link         :: URL
    startDate    :: Date
    endDate      :: Date
    startTime    :: Time
    endTime      :: Time
    blurb        :: WikiText
    eventNote    :: WikiText
}

entity InternalEvent : Event {

    abstract     :: WikiText
    organisers   -> Set<UniMember>
    sponsors     -> Set<Partner>
    invitees     -> Set<UniMember>
    location     -> Address
}

entity Seminar: InternalEvent {
    speaker -> Person
}
```

Listing C.10: Defined data model for internal and external events and seminars

to use the *create* and *delete* operations on the `Event`, `InternalEvent`, `Seminar` entities; and *update* their properties.

University Members: The users with the role *universityMember* are able to *read* all the properties related to the `InternalEvent`, `Seminar` entities. Note that all the instances of the entity `Event` are *readable* by any inside or outside user.

C.1.2.2 Blog, Post, and Comment

The `Blog` entity holds the information about the created blogs within the system. Moreover, each blog has a post, in which the entity `Post` stores the information about the blog posts. Furthermore, the authorised users of the system can add a comment or a tag to a post. For these the entity `comment` and `tag` holds the information about each post of every created blog within the system.

Access control requirements. The users with the following roles can operate on the entities listed in Listing C.11:

Academic: For the efficiency of the user generated content of this application to attract more collaboration with the research group, *only* the academics are able to have a blog. So, the users with the role *Academic* are able to use the *create* and *delete* operations on the `Blog`, `Post`, `Comment`, and `Tag` entities. Moreover these users are able to use the *Update* operation on the properties of these entities.

```
entity Blog {

 name        :: String        (name)
 posts      -> Set<Post>      (inverse = Post.belongTo)
 public      :: Bool
 desc        :: WikiText
}

entity Post {

 postID        :: String        (id)
 belongTo     -> Blog
 title         :: String        (name)
 author       -> Person
 content       :: WikiText
 whenWritten   :: DateTime
 comments     -> Set<Comment>  (inverse = Comment.relatedPost)
 tags         -> Set<Tag>      (inverse = Tag.relatedPost)
}

entity Comment{

 commentID      :: String        (id,name)
 content        :: WikiText
 writtenBy     -> Person
 whenWritten    :: DateTime
 relatedBlog   -> Set<Post>
}

entity Tag{
   tagID    :: String  (id,name)
   tagName  :: String
   relatedPost -> Set<Post>
}
```

Listing C.11: Defined data models for Blog, Post, Comment, and Tag

University Member: To join the discussion within the blog posts, the users with the *universityMember* role can use the *create* operation on the entity `comment`.

Also any inside or outside users are able to *read* the instances of the `Blog`, `Post`, `Comment`, and `Tag` entities.

C.1.2.3 Project

This entity represents the projects that the members of the system are involved. The properties of this entity represent the information about each project such as `members` and its `description`.

Access control requirements. First, the users with the following roles can operate on the entity `Project` (Listing C.12) and its related properties:

```
entity Project {

  title            :: String           (name)
  members          -> Set<Academic>    (inverse = Academic.projects)
  extUsers         -> Set<Partner>     (inverse = Partner.memberOf)
  publicContact    -> Academic
  homepage         -> Link(optional)
  publications     -> Set<Publication >(inverse = Publication.relatedProjects)
  sponsoredBy      -> Set<Partner>     (inverse = Partner.sponserOf)
  description      :: WikiText
  relatedIntrests  -> Set<Interest>    (inverse =  Interest.projects)
}
```

Listing C.12: Project data model

Head of Group or Project Manager: All the projects are solely related to the research group, therefore a user with the role *HeadOfGroup* or *ProjectManager* is able to use the *Create* and *Delete* operations on the entity `Project`; and use the *Update* operation to update its properties.

Moreover, any other user with or without roles, are able to *read* the content of the instances of the entity `Project` and their properties.

C.1.2.4 Publication

This entity stores the information about the publication of the university members. Its properties are self explanatory, and they are a standard knowledge about each publication.

```
entity Publication{

   publicationID        :: String    (id)
   relatedProjects      -> Set<Project>
   title                :: String    (name)
   summary              :: WikiText  (optional)
   acquisitionDate      :: Date
   publishedDate        :: Date
   pubAddress           :: URL         (optional)
   confName             :: String
   firstAAuthor         -> Academic
   firstAuthor          -> UniMember
   restOfAuthors        -> Set<UniMember>
   journal              :: Bool
   paper                :: Bool
   book                 :: Bool
   chapter              :: Bool
   dblpLink             :: URL
}
```

Listing C.13: Defined data model for publication

Access control requirements. The users with the following roles can do the following operations on the `Publication` entity (Listing C.13) and its related properties:

University Member: The users with the role *UniversityMember* are able to *create* a publication and add it to system.

Academic: Moreover, in case of a duplication or error, the users with the role *Academic* are able to *update* all the properties of this entity or *delete* an instance of the `Publication` entity

Same as the `Project` entity, all the instances of the `Publication` entity are not access controlled, and are available to any inside and outside users of the system.

C.1.2.5 Viva

This entity and its properties represents the viva that can hold within the students and the academics of the system. The access rights for each properties are decided based on the defined access control which will be discussed later.

```
entity Viva {

  vivaID            :: String (id,name)
  student          -> Postgrad
  supervisor       -> Academic
  internalExaminer -> Academic
  externalExaminer -> Visitor
  dateOfViva        :: Date
  timeOfViva        :: Time
  plaveOfViva       :: String
  outcome           :: WikiText
}
```

Listing C.14: Defined data model for viva

Access control requirements. The users with the following roles can operate on the properties of the entity `Viva` (Listing C.14):

Supervisor or Advisor: The user with the roles *supervisor* or *Advisor* are able to use the *CRUD* operations on the entity `Viva` and its properties.

External Examiner: A user with the role *External Examiner* is able to *update* the property `outcome` property of this entity. Also the user is able to read all the properties of `Viva` entity.

Postgraduate: Moreover, the user with the roleand *Postgraduate* can *read* the content of this entity *iff* the instance of the entity relates to the user.

Head of Group: The user with the role *HeadOfGroup* is able to read all the instances of the `Viva` entity.

```
entity Interest {

  name                   :: String
  fp                     :: WikiText
  overallDescription     :: WikiText
  relatedAcademics      -> Set<Academic>
  projects              -> Set<Project>
}
```

Listing C.15: Defined data model for users' interests

C.1.2.6 Interests

Access control requirements. The users with the following roles can operate on the properties of the entity `Interest` (C.15):

Head of Group or Academic: The users with the role `HeadOfGroup` or *create* or *delete* an instance of the entity `Interest`. Moreover they are able to *Update* all its properties.

Moreover similar to the `Project`, and `Publication` entities, the instances of the entity `Interest` are available to the public, therefore any inside or outside user can read the instances of this entity and its properties.

C.1.2.7 News

There is a news component to this Web application that shows the latest news through homepage. So, the entity `News` and its properties represent the `title` and `description` of each news. Same as the entity Comment the author and time stamp is automatically filled and stored.

```
entity News {

  title        :: String
  whoWrote    -> Person
  whenWritten  :: Date
  desc         :: WikiText
}
```

Listing C.16: Defined data model for news

Access control requirements. The users with the following roles can operate on the properties of the entity `News` (C.16):

Secretary: As mentioned before the secretaries of the research group have a task to organise a set of events such as seminars. Therefore they should be able to announce these events through the news functionalities of the Web application. Therefore the users with the roles *Secretary* are able to *create* or *delete* an instance of the `News` entity. Moreover they are able to *update* its properties.

Moreover, other *inside* or *outside* users of the system are able to *read* the instances of the entity *News* from the homepage and their page.

C.1.2.8 Address

This entity represents the information about the address of the user.

```
entity Adderss {

  country      :: String
  city         :: String
  street       :: String
  building     :: String
  floor        :: String
  section      :: String (optional)
  room         :: String (optional)
  bay          :: String (optional)
}
```

Listing C.17: Defined data model for address

Access control requirements. The entity address is related to a set of entities within the data model, such as *Person*. The access control on this entity is based on those entities, because the Web application does not have any interface for just adding an instance of the `Address` entity to the system.

C.1.2.9 Calendar

This entity holds a set of notes based on the dates within the calendar. Therefore the users can see a set of notes (i.e., instances of `note` property) based on their chosen dates.

```
entity Calendar {
  noteDate :: Date
  note     :: String
}
```

Listing C.18: Calendar data model

Access control requirements. The users with the following roles can operate on the entity `Calendar` (Listing C.18):

Head of Group or Secretary: The users with the role *Secretary* or *HeadOfGroup* are able to *create* or *delete* an instance of the entity `Calendar`. Moreover they can update `noteDate` and `note` properties.

University Members: As calendar is used for internal purposes, only the users with the role *UniversityMember* are able to *read* the instances of the entity `Calendar`.

C.2 Social Networking

This section first presents the data model created for the second case study, *social networking system* and their access control rights. Second it presents the defined access and authorisation models for this case study.

C.2.1 Data model and their access rights

This part of the Appendix discuss the defined entities and properties of this case study based on used data within the *User Profile* and *Network* Pages.

C.2.1.1 User Profile

The data instances that are operate on the *User profile* are the following:

User: The entity `User` stores the general information about the users (e.g., `username`) the status and wall posts they are writing on their and other profiles. Moreover it stores the networks they are created (i.e., `createdNetworks`), and the relationships they are added (i.e., `addedUsers`) to their profiles of the system.

Status: Each user can add a new *status* on his/her profile page. This entity stores this information, such as when it was written (e.g., `dateofPost`).

Wall post: Each user can add a wall post to his/her profile or on related users. The entity `Wallpost` stores these information.

Access control requirements. As mentioned in Chapter 5, the entity `User` stores the general and specific information about each user and their activities such as writing status for their profiles. Therefore the access control on the `user`, `status`, and `Wallpost` entities, should be discretionary-based (ΦDAC). For this each user (i.e., *Self*) *must* be able to use the *CRUD* operations on these entities and their properties

C.2.1.2 Network

As Listing C.20 shows, there are the following entities that are used to store the data related to the networks of the system and their related functionalities:

```
entity User {
  nickname          :: String (name)
  username          :: String (id)
  password          :: Secret
  title             :: String
  firstname         :: String
  lastname          :: String
  persTel           :: String
  persAddress       :: String
  persEmail         :: String
  occupation        :: String
  dob               :: Date
  profPicture       :: URL
  requestedUsers    -> List<User>     // Users who want to be related.
  requestedTo       -> List<User>     // Users the user wants to be related.
  createdNetworks   -> List<Network> (inverse = Network.creator)
  joinedNetworks    -> List<Network>
  relatedUsers      -> List<User>
  registeredAt      :: DateTime        // Auto filled.
  //Wall posts related to the user profile.
  wallPosts         -> List<WallPost>(inverse = WallPost.userHost)
  //Written Wall posts on the other profiles.
  writtenWP         -> List<WallPost>(inverse = WallPost.author)
  allStatus         -> List<Status>(inverse = Status.author)
  writtenTopics     -> List<Topic>(inverse = Topic.author)
  writtenComments   -> List<Comment>(inverse = Comment.author)
}

entity Status {

  author            -> User // Automatically filled.
  msg               :: Text
  dateOfPost        :: Date //Auto filled.
  timeOfPost        :: Time //Auto filled.
}

entity WallPost {

  author            -> User //Auto filled.
  post              :: WikiText
  userHost          -> User //Auto filled.
  dateOfPost        :: Date //Auto filled.
  timeOfPost        :: Time //Auto filled.
}
```

Listing C.19: Defined entities for the user, status, and wall post

Network: The entity `Network` stores the general information about each network that is created in the system, such as its `name` and the `creator` of it.

Topic: The entity `Topic` stores all the instances of the created topics within each network. Its properties stores the general information about each topic, such as their `author` and its related network.

Comment: The entity `Comment` stores the instances of a comment which is related to a topic. Its properties store the general information such as the `content` of the comment and its related topic (i,.e., via `relatedTopic` property).

```
entity Network {
  creator          -> User
  name             :: String  (name, id)
  Members          -> List<User>
  topics           -> List<Topic> (inverse = Topic.relatedNetwork)
}

entity Topic {

  name             :: String  (name)
  author           -> User      //Auto filled.
  relatedNetwork   -> Network   //Auto filled.
  whenWritten      -> DateTime  //Auto filled.
  comments         -> List<Comment>(inverse = Comment.relatedTopic)
}

entity Comment {

  content          :: WikiText
  author           -> User      //Auto filled.
  relatedTopic     -> Topic     //Auto filled.
  whenWritten      -> DateTime  //Auto filled.
}
```

Listing C.20: Defined Model for the network, topics, and their related comments

Access control requirements. There are the following four roles that do a set of operations on the `Network` entity and its related properties:

Network Organiser: The users with the role *NetworkOrganizer* are able to use the *CRUD* operations on the entity `Network` and its properties.

Admin: First, the users with this role are able to use the the users with the role `contributor` and `leadContributer` role.

Lead Contributor: The users with the role `leadContributer` are able to use *create* a topic, and then able to *read*, *update*, and *delete* any topic within the network. So these users are able to use the *CRUD* operations on the `Topics` property of the `Network` entity. Also they inherit the `Contributor` rights.

Contributor: The users with this role are able to *create* comments (i.e., `comments` property) within each network and if necessary *update* or *delete* their own comments. Also they inherit the rights of the role `NetworkUser`.

Network User: The users with the role, `NetworkUser`, are able to use the *read* operation, on the properties of the `Network` entity.

References

[1] Apache Tomcat. Available online: http://tomcat.apache.org/.

[2] Binder, a logic-based security language. Available online: http://research.microsoft.com/apps/pubs/default.aspx?id=69917.

[3] Django. Available online: http://www.djangoproject.com/.

[4] ebay. Available online: http://www.ebay.com/.

[5] eXtensible access control markup language (XACML) version 3.0. Available online: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-multiple-v1-spec-cd-03-en.html.

[6] facebook. Available online: https://www.facebook.com/.

[7] Google developers. Available online: https://developers.google.com/appengine/?csw=1.

[8] GRAILS. Available online: http://grails.org/.

[9] Java platform, enterprise edition (java EE), Technical documentation. Available online: http://docs.oracle.com/javaee/.

[10] Jetty://. Available online: http://jetty.codehaus.org/jetty/.

[11] MISTA: Model-based integration and system test automation (a.k.a. ISTA). Available online: http://www.homepages.dsu.edu/dxu/research/MBT.html.

[12] Open source- Facebook developers. Available online: http://developers.facebook.com/opensource/.

[13] play. Available online: http://www.playframework.com/.

[14] Ponder: A policy language for distributed systems management. Available online: http://www-dse.doc.ic.ac.uk/Research/policies/ponder.shtml.

[15] Role engineering and RBAC standards. Available online: http://csrc.nist.gov/groups/SNS/rbac/standards.html.

[16] Ruby on rails. Available online: http://www.rubyonrails.org/.

[17] SlashGrid: transparent grid access to HTTP(S) servers. Available online: http://www.gridsite.org/slashgrid/.

[18] Twitter. Available online: https://twitter.com/.

[19] Twitter libraries. https://dev.twitter.com/docs/twitter-libraries.

[20] The virtual private database in Oracle9iR2: An Oracle technical white paper. Available online: http://www.cgisecurity.com/database/oracle/pdf/VPD9ir2twp.pdf.

[21] Wakanda. Available online: http://www.wakanda.org/.

[22] WebDSL. Available online: http://webdsl.org/home.

[23] *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, 2006.

[24] Martín Abadi. Access control in a core calculus of dependency. In *ICFP*, pages 263–273, 2006.

[25] Martín Abadi. Variations in access control logic. In *DEON*, pages 96–109, 2008.

[26] Martín Abadi. Logic in access control (tutorial notes). In *FOSAD*, pages 145–165, 2009.

[27] Martín Abadi, Michael Burrows, Butler W. Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, 1993.

[28] Sergei I. Adian and Anil Nerode, editors. *Logical Foundations of Computer Science, 4th International Symposium, LFCS'97, Yaroslavl, Russia, July 6-12, 1997, Proceedings*, volume 1234 of *Lecture Notes in Computer Science*. Springer, 1997.

[29] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *21st IEEE Int. Conf. Data Engineering*, Tokyo, 5-8 April 2005.

[30] G. J. Ahn. *The RCL 2000 language for specifying role-based authorization constraints*. PhD thesis, Goerge Mason University, Fairfax, VA, 1999.

[31] G. J. Ahn and R. S. Sandhu. The RSL99 language for role-based separation of duty constraints. In *ACM Workshop on Role-Based Access Control*, 1999, pp. 43–54.

[32] A. H. Anderson. A comparison of two privacy policy languages: EPAL and XACML. In *3rd ACM Workshop on Secure Web Services*, 2006, pp. 53–60.

[33] N. Antunes and M. Vieira. Enhancing penetration testing with attack signatures and interface monitoring for the detection of injection vulnerabilities in web services. In *IEEE Int. Conf. Services Computing*, Washington, 4-9 July 2011, pp. 104–111.

[34] M. Backes, M. Dürmuth, and G. Karjoth. Unification in privacy policy evaluation-Translating EPAL into Prolog. In *5th IEEE Int. Workshop on Policies for Distributed Systems and Networks*, 7-9 June 2004, pp. 185–188.

[35] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

[36] R. W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *IEEE Computer Society Symp. Research in Security and Privacy*, Oakland, 7-9 May 1990, pp. 116–132.

[37] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *26th Int. Conf. Software Engineering*, 23-28 May 2004, pp. 625–634.

[38] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE, 1973.

[39] B. Bellur. Certificate assignment strategies for a PKI-based security architecture in a vehicular network. In *IEEE Global Telecommunications Conf.*, New Orleans, 30 Nov. 2008-4 Dec. 2008, pp. 1–6.

[40] A. Belokosztolszki and D. M. Eyers. *Research directions in data and applications security, IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security, Cambridge, 28-31 July 2002*, chapter Shielding RBAC infrastructures from cyberterrorism. Kluwer, 2003.

[41] P. N. Benton, Gavin M. Bierman, and Valeria de Paiva. Computational types from a logical perspective. *J. Funct. Program.*, 8(2):177–193, 1998.

[42] K. Berket, A. Essiari, and A. Muratas. PKI-based security for peer-to-peer information sharing. In *4th IEEE Int. Conf. Peer-to-Peer Computing*, Zurich, 25-27 Aug. 2004, pp. 45–52.

[43] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM T. Database Syst.*, 23(3):231–285, 1998.

[44] E. Bertino, A. Kamra, E. Terzi, and A. Vakali. Intrusion detection in RBAC-administered databases. In *21st Annual Computer Security Applications Conf.*, Tucson, 5-9 Dec. 2005.

[45] E. Bertino and R. Sandhu. Database security-concepts, approaches, and challenges. *IEEE T. Depend. Secure*, 2(1):2–19, 2005.

[46] J. Bhattacharya and S. K. Gupta. *EPAL Based Privacy Enforcement Using ECA Rules.* 1st Int. Conf. Information Systems Security, LNCS 3803. Springer-Verlag, Berlin, 2005, pp. 120-133.

[47] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE, 1977.

[48] M. Bishop. What is computer security? *IEEE Secur. Priv.*, 1(1):67–69, 2003.

[49] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984.

[50] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: components for transformation systems. In *The 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Charleston, 9-10 Jan. 2006, pp. 95–99.

[51] G. Buday. Logic in computer science: Modelling and reasoning about systems. *J. Funct. Program.*, 18(3):421–422, 2008.

[52] W. E. Burr, D. F. Dodson, and W. T. Polk. *Electronic authentication guideline.* National Institute of Standards and Technology, Technology Administration, U.S. Dept. of Commerce, Gaithersburg, 2006.

[53] S. Castano, M. G. Fugini, G. Martella, and P. Samarati. *Database security.* Addison-Wesley & ACM Press, 1995.

[54] S. Casteleyn, F. Daniel, P. Dolog, and M. Matera. *Engineering web applications.* Data-centric systems and applications. Springer, 2009.

[55] R. Chandramouli. Application of XML tools for enterprise-wide RBAC implementation tasks. In *5th ACM Workshop on Role-Based Access Control*, 2000, pp. 11–18.

[56] K. Chen and C. M. Huang. *A Practical Aspect Framework for Enforcing Fine-Grained Access Control in Web Applications.* 1st Int. Conf. Information Security Practice and Experience, LNCS 3439. Springer-Verlag, Berlin, 2005, pp. 156-167.

[57] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symp. Security and Privacy*, 1987, pp. 184–195.

[58] J. Crampton and G. Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Trans. Inf. Syst. Secur.*, 6(2):201–231, 2003.

[59] K. Czarnecki and U. W. Eisenecker. *Generative programming: Methods, tools, and applications.* ACM Press/Addison-Wesley Publishing Co., New York, 2000.

[60] I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Trans. Internet Techn.*, 12(1), 2012.

[61] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. *The Ponder policy specification language*. Workshop on Policies for Distributed Systems and Networks, LNCS 1995. Springer-Verlag, 2001, pp. 18-38.

[62] L. de Moura and N. Bjørner. *Z3: An efficient SMT solver*. 14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems, LNCS 4963. Springer-Verlag, Berlin, 2008, pp. 337-340.

[63] B. de Win, F. Piessens, W. Joosen, and T. Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. In *ACSA Workshop on the Application of Engineering Principles to System Security Design*, Boston, 6-8 Nov. 2002.

[64] M. A. C. Dekker, J. Crampton, and S. Etalle. RBAC administration in distributed systems. In *13th ACM Symp. Access Control Models and Technologies*, 2008, pp. 93–102.

[65] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5), 1976.

[66] L. Desmet, P. Verbaeten, W. Joosen, and F. Piessens. Provable protection against web application vulnerabilities related to session data dependencies. *IEEE Trans. Software Eng.*, 34(1):50–64, 2008.

[67] John DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[68] S. De C. di Vimercati, S. Foresti, P. Samarati, and S. Jajodia. Access control policies and languages. *Int. J. Computational Science and Engineering*, 3(2):94–102, 2007.

[69] A. Erradi, P. Maheshwari, and V. Tosic. WS-policy based monitoring of composite web services. In *5th IEEE European Conf. Web Services*, Halle, 26-28 Nov. 2007, pp. 99–108.

[70] G. Faden. RBAC in UNIX administration. In *4th ACM Workshop on Role-Based Access Control*, 1999.

[71] D. F. Ferraiolo and D. R. Kuhn. Role-based access control. In *15th National Computer Security Conf.*, 1992, pp. 554–563.

[72] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM T. Inform. Syst. Se.*, 4(3):224–274, 2001.

[73] R. Ferrini and E. Bertino. Supporting RBAC with XACML+OWL. In *14th ACM Symp. Access Control Models and Technologies*, Stresa, 3-5 June 2009, pp. 145–154.

[74] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated designs for object-oriented systems.* Springer London, 2005.

[75] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[76] Y. L. Gall, A. J. Lee, and A. Kapadia. PlexC: A policy language for exposure control. In *17th ACM Symp. Access Control Models and Technologies*, Newark, 2012, pp. 219–228.

[77] Valerio Genovese and Deepak Garg. New modalities for access control logics: Permission, control and ratification. In *STM*, pages 56–71, 2011.

[78] S. H. Ghotbi and B. Fischer. *Fine-grained role- and attribute-based access control for web applications.* Communications in Computer and Information Science (CCIS). Springer-Verlag, 2013.

[79] S. H. Ghotbi and B. Fischer. A declarative fine-grained role-based access control model and mechanism for the web application domain. In *7th Int. Conf. Software Paradigm Trends*, Rome, 24-27 July 2012, pp. 80–91.

[80] U. Goltz. *Petri nets: Central models and their properties*, chapter Synchronic distance. Advances in Petri Nets 1986, Part I Proc. An Advanced Course, LNCS 254. Springer, 1987, pp. 338-358.

[81] G. S. Graham and P. J. Denning. Protection: Principles and practice. In *Spring Joint Computer Conf.*, 1972, pp. 417–429.

[82] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Softw. Test., Verif. Reliab.*, 15(3):167–199, 2005.

[83] D. M. Groenewegen, Z. Hemel, L. C. L. Kats, and E. Visser. WebDSL: A domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nashville, 19-23 Oct. 2008, pp. 779–780.

[84] D. M. Groenewegen, Z. Hemel, L. C. L. Kats, and E. Visser. When frameworks let you down, Platform-imposed constraints on the design and evolution of domain-specific languages. In *8th OOPSLA Workshop on Domain Specific Modelling*, Nashville, Oct. 2008, pp. 64–66.

[85] D. M. Groenewegen and E. Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In *8th Int. Conf. Web Engineering*, pages 175–188, 14-18 July 2008, pp. 175–188.

[86] M. V. Gundy and H. Chen. Noncespaces: Using randomization to defeat cross-site scripting attacks. *Computers & Security*, 31(4):612–628, 2012.

[87] D. A. Haidar, N. Cuppens-Boulahia, F. Cuppens, and H. Debar. An extended rbac profile of XACML. In *3rd ACM Workshop on Secure Web Services*, 30 Oct-3 Nov. 2006, pp. 13–22.

[88] A. Y. Halevy. Answering queries using views: A survey. *The Int. J. Very Large Data Bases*, 10(4):270–294, 2001.

[89] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. On protection in operating systems. In *5th ACM Symp. Operating Systems Principles*, 1975, pp. 14–24.

[90] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF-reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[91] Z. Hemel, L. C. L. Kats, and E. Visser. *Code generation by model transformation. A case study in transformation modularity.* 1st Int. Conf. Model Transformation, Theory and Practice of Model Transformations, LNCS 5063. Springer, 2008, pp. 183-198.

[92] M. Hitchens and V. Varadharajan. *Issues in the design of a language for role based access control.* Information and Communication Security, LNCS 1726. Springer-Verlag, Berlin, 1999, pp. 22-38.

[93] M. Hitchens and V. Varadharajan. *Tower: A language for role based access control.* Policies for Distributed Systems and Networks, LNCS 1995. Springer-Verlag, Berlin, 2001, pp. 88-106.

[94] C. Hortsmann. *Scala for the impatient.* Addison-Wesley Professional, 2012.

[95] G. Hsieh, K. Foster, G. Emamali, G. Patrick, and L. M. Marvel. Using XACML for embedded and fine-grained access control policy, Fukuoka, 16-19 March 2009, pp. 462–468.

[96] J. Y. Huang, I. E. Liao, and H. W. Tang. A forward authentication key management scheme for heterogeneous sensor networks. *EURASIP J. Wirel. Comm.*, 2011.

[97] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. Securing web application code by static analysis and runtime protection. In *13th Int. Conf. World Wide Web*, New York, 17-22 May 2004, pp. 40–52.

[98] T. Jaeger and A. Prakash. Requirements of role-based access control for collaborative systems. In *ACM Workshop on Role-Based Access Control*, 1995.

[99] R. Jagadeesan. *From authorization logics to types for authorization.* 6th Asian Symp. Programming Languages and Systems, LNCS 5356. Springer, Berlin, 2008, p. 255.

[100] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM T. Database Syst.*, 26(2):214–260, 2001.

[101] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symp. Security and Privacy*, Oakland, 4-7 May 1997, pp. 31–42.

[102] J. Jin and G. J. Ahn. Role-based access management for ad-hoc collaborative sharing. In *11th ACM Symp. Access Control Models and Technologies*, 2006, pp. 200-209.

[103] J. Jin, G. J. Ahn, and M. Singhal. *ShareEnabler: Policy-driven access management for Ad-Hoc collaborative sharing.* Current Trends in Database Technology- EDBT Workshops, LNCS 4254. Springer-Verlag, Berlin, 2006, pp. 724-740.

[104] N. D. Jones and R. Ry. Hansen. *The Semantics of "Semantic Patches" in Coccinelle: Program transformation for the working programmer.* 5th Asian Symp. Programming Languages and Systems, LNCS 4807. Springer, Berlin, 2007, pp. 303-318.

[105] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *IEEE Symp. Security and Privacy*, Berkeley/Oakland, 21-24 May 2006.

[106] J. Julliand, P. A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *3rd Int. Workshop on Automation of Software Test*, 2008, pp. 41–44.

[107] J. Jürjens. Model-based security testing using UMLsec: A case study. *Electronic Notes in Theoretical Computer Science*, 220(1):93–104, 2008.

[108] P. A. Karger. Limiting the damage potential of discretionary trojan horses. In *IEEE Symp. Security and Privacy*, 1987, pp. 32–37.

[109] D. A. Kindy and A. K. Pathan. A detailed survey on various aspects of SQL injection: Vulnerabilities, innovative attacks, and remedies. *CoRR*, 2012.

[110] P. Klint, T. van der Storm, and J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *9th IEEE Int. Working Conf. Source Code Analysis and Manipulation*, 20-21 Sept. 2009, pp. 168–177.

[111] Miroslaw Kurkowski and Jerzy Pejaś. Artificial intelligence and security in computing systems. chapter A Propositional Logic for Access Control Policy in Distributed Systems, pages 175–189. Kluwer Academic Publishers, USA, 2003.

[112] L. G. Lawrence. The role of roles. *J. Computers and Security*, 12(1), 1993.

[113] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. J. DeWitt. Limiting disclosure in hippocratic databases. In *13th Int. Conf. Very Large Data Bases , Toronto, Canada, August 31 - September 3 2004*, Toronto, 31 August-3 Sep. 2004, pp. 108–119.

[114] H. Liang and J. Dingel. *A practical evaluation of using TXL for model transformation.* 1st Int. Conf. Software Language Engineering, LNCS 5452. Springer, Berlin, 2008, pp. 245-264.

[115] J. D. Lim, S. K. Un, J. N. Kim, and C. Lee. *Implementation of LSM-based RBAC module for embedded system.* 8th Int. Workshop Information Security Applications, LNCS 4867. Springer, Berlin, 2007, pp. 91-101.

[116] S. Liu, R. Zhang, D. Wang, H. Sun, Y. Chen, and L. Li. Implementing of gaussian syntax-analyzer using ANTLR. In *Int. Conf. Cyberworlds*, Hangzhou, 22-24 Sep. 2008, pp. 613–618.

[117] M. Lorch, D. B. Adams, D. G. Kafura, M. S. R. Koneni, A. Rathi, and S. Shah. The PRIMA system for privilege management, authorization and enforcement in grid environments. In *4th Int. Workshop Grid Computing*, 17 Nov. 2003, pp. 109–116.

[118] M. Lorch and D. G. Kafura. The PRIMA grid authorization system. *J. Grid Computing*, 2(3):279–298, 2004.

[119] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah. First experiences using XACML for access control in distributed systems. In *The 2003 ACM Workshop on XML security*, 2003, pp. 25–37.

[120] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *8th Int. Conf. Information and Communications Security*, 2006, pp. 139–158.

[121] M. Masi, R. Pugliese, and F. Tiezzi. Formalisation and implementation of the XACML access control mechanism. In *4th Int. Conf. Engineering Secure Software and Systems*, 2012, pp. 60–74.

[122] A. Masood, R. Bhatti, A. Ghafoor, and A. P. Mathur. Scalable and effective test generation for role-based access control systems. *IEEE T. Software Eng.*, 35(5):654–668, 2009.

[123] C. J. McCollum, J. R. Messing, and L. Notargiacomo. Beyond the pale of MAC and DAC-defining new forms of access control. In *IEEE Computer Society Symp. Research in Security and Privacy*, Oakland, 7-9 May 1990, pp. 190–200.

[124] G. McGraw and B. Potter. Software security testing. *IEEE Secur. Priv.*, 2(5):81–85, 2004.

[125] G. L. Millán, Ma. G. Pérez, G. Ma. Pérez, and A. F. Gómez-Skarmeta. PKI-based trust management in inter-domain scenarios. *Computers & Security*, 29(2):278–290, 2010.

[126] L. Montrieux, M. Wermelinger, and Y. Yu. Tool support for UML-based specification and verification of role-based access control properties. In *19th ACM Symp. Foundations of Software Engineering and 13th European Conf. Software Engineering*, Szeged, 4-9 Sep. 2011, pp. 456–459.

[127] Amihai Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *5th Int. Conf. Data Engineering*, Los Angles, 6-10 Feb. 1989, pp. 339–347.

[128] P. G. Neumann. Computer system- Security evaluation. In *National Computer Conf.*, 1978.

[129] A. C. O'Connor and R. J. Loomis. Economic analysis of role based access control. Technical report, National Institute of Standards and Technology, 2010.

[130] S. Osborn. Mandatory access control and role-based access control revisited. In *2nd ACM Workshop on Role-based Access Control*, 1997, pp. 31–40.

[131] S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM T. Inform. Syst. Se.*, 3(2):85–106, 2000.

[132] J. S. Park and R. S. Sandhu. RBAC on the web by smart certificates. In *4th ACM Workshop on Role-Based Access Control*, 1999, pp. 1–9.

[133] A. Pretschner, T. Mouelhi, and Y. L. Traon. Model-based tests for access control policies. In *1st Int. Conf. Software Testing, Verification, and Validation*, Lillehammer, 9-11 April 2008, pp. 338–347.

[134] C. D. P. K. Ramli, H. R. Nielson, and F. Nielson. XACML 3.0 in answer set programming. *CoRR*, 2012.

[135] E. Reshetova and J. E. Ekberg. Mandatory access control for mobile devices. Available online: `http://research.nokia.com/files/tr/NRC-TR-2008-010.pdf`, 2008.

[136] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *ACM Int. Conf. Management of Data*, Paris, 13-18 June 2004, pp. 551–562.

[137] A. Roichman and E. Gudes. Fine-grained access control to web databases. In *12th ACM Symp. Access Control Models and Technologies*, 2007, pp. 31–40.

[138] D. Rosenblum. What anyone can know: The privacy risks of social networking sites. *IEEE Secur. Priv.*, 5(3):40–49, 2007.

[139] A. Rosenthal and E. Sciore. View security as the basis for data warehouse security. In *Int. Workshop on Design and Management of Data Warehouses*, Stockholm, 5-6 June 2000, pp. 8.1–8.8.

[140] P. Samarati and S. C. de Vimercati. *Access Control: Policies, Models, and Mechanisms.* Foundations of Security Analysis and Design, LNCS 2171. Springer-Verlag, Berlin, 2000, pp. 137-196.

[141] R. S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[142] R. S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In *4th European Symp. Research in Computer Security*, 1996, pp. 65–79.

[143] R. S. Sandhu. The typed access matrix model. In *IEEE Computer Society Symp. Research in Security and Privacy*, Oakland, 4-6 May 1992, pp. 122–136.

[144] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[145] R. S. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for role-based access control: Towards a unified standard. In *5th ACM Workshop on Role-based Access Control*, 2000, pp. 47–63.

[146] R. S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Commun. Mag.*, 32(9):40–48, 1994.

[147] D. Sanz, P. Diaz, and I. Aedo. Implementing RBAC policies in a web server. In J. A. Carvalho, A. C. Hubler, and A. A. Baptista, editors, *Technology Interactions, 6th Int. Conf. Electronic Publishing*, Karlovy Vary, 6-8 Nov. 2002, pp. 297–306.

[148] F. Satoh and N. Uramoto. *Validating Security Policy Conformance with WS-Security Requirements.* 5th Int. Workshop on Security, Advances in Information and Computer Security, LNCS 6434. Springer, Berlin, 2010, pp. 133-148.

[149] I. Schieferdecker, J. Grossmann, and M. Schneider. Model-based security testing. In *Workshop on Model-Based Testing*, 2012, pp. 1–12.

[150] T. Scholte, D. Balzarotti, and E. Kirda. Have things changed now? an empirical study on input validation vulnerabilities in web applications. *Computers & Security*, 31(3):344–356, 2012.

[151] G. J. Simmons. The practice of authentication. In *Workshop on the Theory and Applications of Cryptographic Techniques*, Linz, 1985, pp. 261–272.

[152] K. Sohr, T. Mustafa, X. Bao, and G. J. Ahn;. Enforcing role-based access control policies in web services with UML and OCL. In *Annual Computer Security Applications Conf.*, 2008, pp. 257–266.

[153] Corporate IEEE Computer Society Staff, editor. *IEEE standard portable operating system interface for computer environments.* 1003.1. IEEE Press Piscataway, NJ, 1988.

[154] R. Steele and K. Min. HealthPass: Fine-grained access control to portable personal health records. In *24th IEEE Int. Conf. Advanced Information Networking and Applications*, 2010, pp. 1012–1019.

[155] M. Stonebraker and E. Wong. Access control in a relational data base management system by query modification. In *ACM Annual Conf.*, 1974, pp. 180–186.

[156] W. V. Sujansky, S. A. Faus, E. Stone, and P. F. Brennan. A method to implement fine-grained access control for personal health records through standard relational database queries. *J. Biomed. Inform.*, 43, 2010.

[157] M. R. Thompson, A. Essiari, and S. Mudumbai. Certificate-based authorization policy in a PKI environment. *ACM T. Inform. Syst. Se.*, 6(4):566–588, 2003.

[158] I. A. Tondel, M. G. Jaatun, and J. Jensen. Learning from software security testing. In *IEEE Int. Conf. Software Testing Verification and Validation Workshop*, Lillehammer, 9-11 April 2008, pp. 286–294.

[159] M. G. J. van den Brand. Applications of the ASF+SDF meta-environment. In *Int. Conf. Generative and Transformational Techniques in Software Engineering*, 2005, pp. 278–296.

[160] S. D. Vermolen and E. Visser. *Heterogeneous coupled evolution of software languages).* 11th Int. Conf. Model Driven Engineering Languages and Systems, LNCS 5301. Springer, Berlin, 2008, pp. 630-644.

[161] E. Visser. *Syntax definition for language prototyping.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 1997.

[162] E. Visser. *WebDSL: A case study in domain-specific language engineering.* Int. Summer School Generative and Transformational Techniques in Software Engineering II, LNCS 5235. Springer, Berlin, 2007, pp. 291-373.

[163] L. Wang, E. Wong, and D. Xu. A threat model driven approach for security testing. In *3rd Int. Workshop on Software Engineering for Secure Systems*, 2007, pp. 10–17.

[164] R. Watanabe, Y. Nakano, and T. Tanaka. Single sign-on techniques with PKI-based authentication for mobile phones. In *Security and Management*, Las Vegas, 12-15 July 2010, pp. 152–156.

[165] H. Wolfe. Encountering encryption. *Computers & Security*, 22(5):388–391, 2003.

[166] D. Xu, L. Thomas, Mi. Kent, T. Mouelhi, and Y. L. Traon. A model-based approach to automated testing of access control policies. In *17th ACM Symp. Access Control Models and Technologies*, 2012, pp. 209–218.

[167] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *ACM SIGOPS 22nd Symp. Operating Systems Principles*, 2009, pp. 291–304.

[168] C. N. Zhang and C. Yang. An object-oriented RBAC model for distributed system. In *Working IEEE/IFIP Conf. Software Architecture*, Amsterdam, 28-31 August 2001.

[169] Y. Q. Zhu, J. Li, and Q. H. Zhang. A general attribute based RBAC model for web service. In *IEEE Int. Conf. Services Computing*, Salt Lake City, 9-13 July 2007, pp. 236–239.