



HAL
open science

A model driven engineering approach to build secure information systems

Thi Mai Nguyen

► **To cite this version:**

Thi Mai Nguyen. A model driven engineering approach to build secure information systems. Modeling and Simulation. Université Paris Saclay (COmUE), 2017. English. NNT : 2017SACLL001 . tel-01514651

HAL Id: tel-01514651

<https://theses.hal.science/tel-01514651v1>

Submitted on 26 Apr 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOCTORAT EN CO-ACCREDITATION
TELECOM SUDPARIS ET L'UNIVERSITE PARIS-SACLAY

Spécialité: Informatique

Ecole doctorale: Sciences et Ingénierie

Présenté par

Mme Thi-Mai Nguyen

Pour obtenir le grade de
DOCTEUR DE TELECOM SUDPARIS

A MODEL DRIVEN ENGINEERING APPROACH TO BUILD SECURE INFORMATION SYSTEMS

Soutenue le 13/01/2017

devant le jury composé de :

Directeurs de thèse :

Mme. Amel Mammar	Maître de conférences, HDR, TELECOM SudParis, France
Mme. Régine Laleau	Professeur, Université Paris-Est Créteil, France

Rapporteurs :

M. Yves Roudier	Professeur, Université Nice Sophia Antipolis, France
M. Christian Attiogbé	Professeur, Université de Nantes, France

Examineurs :

M. Pascal Poizat	Professeur, Université Paris Ouest Nanterre La Défense, France
M. Akram Idani	Maître de conférences, Université Grenoble Alpes , France
M. Paul Gibson	Maître de conférences, HDR, TELECOM SudParis, France

to my family

Acknowledgment



I would like to express my deepest appreciation and gratitude to my supervisors Dr. Amel Mammam and Professor Régine Laleau for their regular feedback, encouragement, and excellent advices throughout this research. Without their mentorship, I would have not been able to finish this dissertation.

I am also thankful to Dr. Akram Idani for his scientific advices and support. It has been a real pleasure to work with him.

I would like to thank all members of the jury. I thank Professor Christian Attiogbé and Dr. Ludovic Apvrille for accepting being my thesis reviewers and for their attention and thoughtful comments. I also thank Professor Pascal Poizat, Dr. Akram Idani and Dr. Paul Gibson for accepting being my thesis examiners.

I wish to thank all the members of the computer science department of Telecom SudParis, whose friendly company during my thesis was a great pleasure. I would like to thank Brigitte Houassine for her kind help and assistance. A special thank to my colleagues Karn, Nabila, and Monika for their moral support and the lovely moments we spent. I would like to express my deepest gratitude to Nathan for all his encouragement and support when I needed the most.

Next, I owe my deepest gratitude to my friends especially Son, Huy, Long, Hoa for their support. I would like to express my warmest affection to Minh for always being by my side, helping me unconditionally and for all the beautiful moments we shared in France.

I am forever thankful to my family: my father, my mother, my sister and my brothers who were always there for me with encouraging words. Your encouragement made me go forward and made me want to succeed. I would like to express my heartiest gratitude to my boyfriend, Luong. His love, encouragement, understanding and support helped me to get through all the difficult times. This is because of him that I can reach the goal. Thank you so much for having faith in me even when I doubted myself! I love you so much. I dedicate this thesis to all of you, my wonderful family.

Abstract

Nowadays, organizations rely more and more on information systems to collect, manipulate, and exchange their relevant and sensitive data. In these systems, security plays a vital role. Indeed, any security breach may cause serious consequences, even destroy an organization's reputation. Hence, sufficient precautions should be taken into account. Moreover, it is well recognized that the earlier an error is discovered, the easier and cheaper it is debugged. The objective of this thesis is to specify security policies since the early development phases and ensure their correct deployment on a given technological infrastructure.

Our approach starts by specifying a set of security requirements, i.e. static and dynamic rules, along with the functional aspect of a system based on the Unified Modeling Language (UML). Fundamentally, the functional aspect is expressed using a UML class diagram, the static security requirements are modeled using SecureUML diagrams, and the dynamic rules are represented using secure activity diagrams.

We then define translation rules to obtain B specifications from these graphical models. The translation aims at giving a precise semantics to these diagrams, thus proving the correctness of these models and verifying security policies with respect to the related functional model using the AtelierB prover and the ProB animator. The obtained B specification is successively refined to a database-like implementation based on the AOP paradigm. The B refinements are also proved to make sure that the implementation is correct with respect to the initial abstract specification. Our translated AspectJ-based program allows separating the security enforcement code from the rest of the application. This approach avoids scattering and tangling the application's code, thus it is easier to track and maintain.

Finally, we develop a tool that automates the generation of the B specification from UML-based models and the derivation of an AspectJ implementation from the refined B specification. The tool helps disburden developers of the difficult and error-prone tasks and improve the productivity of the development process.

Résumé

Aujourd’hui, les organisations s’appuient de plus en plus sur les systèmes d’information pour collecter, manipuler et échanger leurs données. Dans ces systèmes, la sécurité joue un rôle essentiel. En effet, toute atteinte à la sécurité peut entraîner de graves conséquences, voire détruire la réputation d’une organisation. Par conséquent, des précautions suffisantes doivent être prises en compte. De plus, il est bien connu que plus tôt un problème est détecté, moins cher et plus facile il sera à corriger. L’objectif de cette thèse est de spécifier les politiques de sécurité depuis les premières phases de développement et d’assurer leur déploiement correct sur une infrastructure technologique donnée.

Notre approche commence par spécifier un ensemble d’exigences de sécurité, i.e. des règles statiques et dynamiques, accompagnées de l’aspect fonctionnel d’un système basé sur UML (Unified Modeling Language). L’aspect fonctionnel est exprimé par un diagramme de classes UML, les exigences de sécurité statiques sont modélisées à l’aide de diagrammes de SecureUML, et les règles dynamiques sont représentées en utilisant des diagrammes d’activités sécurisées.

Ensuite, nous définissons des règles de traduction pour obtenir des spécifications B à partir de ces modèles graphiques. La traduction vise à donner une sémantique précise à ces schémas permettant ainsi de prouver l’exactitude de ces modèles et de vérifier les politiques de sécurité par rapport au modèle fonctionnel correspondant en utilisant les outils AtelierB prover et ProB animator. La spécification B obtenue est affinée successivement à une implémentation de type base de données, qui est basée sur le paradigme AOP. Les affinements B sont également prouvés pour s’assurer que l’implémentation est correcte par rapport à la spécification abstraite initiale. Le programme d’AspectJ traduit permet la séparation du code lié à la sécurité du reste de l’application. Cette approche permet d’éviter la diffusion du code de l’application, et facilite ainsi le traçage et le maintien.

Enfin, nous développons un outil qui génère automatiquement la spécification B à partir des modèles UML, et la dérivation d’une implémentation d’AspectJ à partir de la spécification B affinée. L’outil aide à décharger les développeurs des tâches difficiles et à améliorer la productivité du processus de développement.

List of Publications

1. Thi Mai Nguyen, Amel Mammar, Regine Laleau and Samir Hameg, *A tool for the generation of a secure access control filter*, IEEE 10th International Conference on Research Challenges in Information Science, RCIS 2016, Grenoble, France, June 01-03, 2016.
2. Amel Mammar, Thi Mai Nguyen and Regine Laleau, *Formal development of a secure access control filter*, IEEE 17th International Symposium on High Assurance Systems Engineering, HASE 2016, Orlando, Florida, USA, January 07-09, 2016.

Table of contents

1	Introduction	23
1.1	Research Motivations	24
1.2	Research Contributions	26
1.3	Thesis Structure	27
2	Background	29
2.1	Introduction	30
2.2	The B Method	30
2.2.1	Abstract machine	30
2.2.2	Refinement	34
2.2.3	Discussion	36
2.3	Model-Driven Engineering	37
2.3.1	An overview	37
2.3.2	Model-Driven Architecture	37
2.4	Role Based Access Control	38
2.4.1	Core RBAC	39
2.4.2	RBAC Constraints	40
2.4.3	SecureUML	41
2.5	RBAC in Database Management Systems	41
2.5.1	Database User	42
2.5.2	User-defined Database Role	42
2.5.3	User-Role assignment	42
2.5.4	Permission assignment	43
2.6	Aspect Oriented Programming	44
2.6.1	An overview	44
2.6.2	AspectJ	46
2.7	Conclusion	48
3	State of The Art	49
3.1	Introduction	50
3.2	Techniques for Security Specification	50
3.2.1	UML and OCL based approaches	50
3.2.2	Alloy-based Approaches	58
3.2.3	Z-based Approaches	63
3.2.4	A B-based Approach	68
3.2.5	Discussion	72
3.3	Support Tools for Access Control Policies	74

3.3.1	SecureMOVA	74
3.3.2	B4MSecure	77
3.3.3	Discussion	79
3.4	Implementation of An Access Control Specification	80
3.5	Enforcement of Access Control Policies	82
3.5.1	Java Authentication and Authorization Service	83
3.5.2	Annotation-based approaches	83
3.5.3	AOP-based approaches	84
3.5.4	Discussion	86
3.6	Conclusion	88
4	Formal Development of a Secure Access Control Filter	89
4.1	Introduction	90
4.2	The case study: a bank system	92
4.3	Graphical modeling of security requirements	93
4.3.1	SecureUML	93
4.3.2	Activity diagrams for dynamic security rules	94
4.4	Generation of a B specification	96
4.4.1	Overview of the B method	97
4.4.2	Translation of the functional model: the class diagram	97
4.4.3	Formalizing SecureUML in B	99
4.4.4	Translation of the secure UML activity diagrams into B	101
4.4.5	Putting all the security and functional constraints together	105
4.5	Verification and validation	107
4.6	Conclusion	109
5	A Tool for the Generation of a Secure Access Control Filter	111
5.1	Introduction	112
5.2	Overview of the tool	113
5.3	Overview of the B method	115
5.4	Graphical modeling of the application: case study	115
5.5	From graphical diagrams to B formal notations	121
5.5.1	Translation of the class diagram	121
5.5.2	Translation of the SECUREUML diagram	123
5.5.3	Translation of the secure UML activity diagram	124
5.6	The B specification of a secure filter	126
5.7	Conclusion	127

6	A Formal Approach to Derive an AOP-Based Implementation of a Secure Access Control Filter	131
6.1	Introduction	132
6.2	The case study: a purchase order system	134
6.3	A formal B specification of a secure filter	136
6.3.1	Translation of the class diagram into a B specification . .	136
6.3.2	Translation of the SecureUML diagram into a B specification	137
6.3.3	Translation of the secure UML activity diagrams into a B specification	139
6.3.4	Designing the secure filter	140
6.4	From an abstract B specification to a relational-like B implementation	141
6.4.1	Data refinement	141
6.4.2	Behavioral refinement	143
6.5	The AspectJ implementation of the application	146
6.5.1	Transformation rules of B into JAVA/SQL	148
6.5.2	Deployment of the class diagram	151
6.5.2.1	Definition of the tables and the associated JAVA methods and stored procedures	153
6.5.2.2	Translation of the operations of the class diagram	154
6.5.3	Deployment of the SecureUML diagram	155
6.5.4	Deployment of the secure activity diagrams	158
6.5.4.1	Definition of the log tables and the associated JAVA classes	158
6.5.4.2	Translation of the secure operations of the secure activity diagrams	160
6.5.4.3	Translation of the log operations	161
6.5.5	Deployment of the filter	161
6.6	Tool support	163
6.6.1	Translating the B specification of the class diagram . . .	163
6.6.2	Translating the B specification of the SecureUML diagram	164
6.6.3	Translating the B specification of the secure activity diagram	166
6.6.4	Translation of the access control filter	167
6.7	Conclusion	168
7	Conclusions and Future Work	171
7.1	Contributions	172
7.2	Future Work	173

List of Tables

2.1	The visibility of the SEES clause	33
2.2	The visibility of the INCLUDES clause	34
2.3	The visibility of the IMPORTS clause	36
3.1	Synthesis of formal-based approaches for security specifications .	75
4.1	Results of the proof phase	109
6.1	Type mappings table among B, JAVA, and SQL Server	149
6.2	Transformation of B expressions to SQL Server	150

List of Figures

2.1	Software development process in B	31
2.2	B machine	32
2.3	Core RBAC [1]	40
2.4	SecureUML metamodel [2]	41
2.5	Normal compilation process (left) and AOP compilation process (right)	45
3.1	A static model [3]	51
3.2	A functional model [3]	52
3.3	Collaboration Diagram: Permission Approval [3]	52
3.4	Conceptual class model for RBAC [4]	53
3.5	A RBAC class diagram template [5,6]	55
3.6	A RBAC class diagram for a banking system [6]	56
3.7	Violation of SSD Constraint [5,6]	56
3.8	Object Diagram for SSD Policies [6]	56
3.9	Use case: TeamWorker [7]	57
3.10	Model Driven Security [8]	58
3.11	Assurance Management Framework [9]	62
3.12	The transformation method [10]	63
3.13	An access control filter [11]	69
3.14	A dynamic access control rule in ASTD [11]	70
3.15	A secure operation	72
3.16	A dynamic operation [12]	73
3.17	An access control filter [12]	73
3.18	Validation/Verification activities supported by B4MSecure [13] .	77
3.19	The architecture of B4MSecure	78
3.20	UML to B translation	78
3.21	SecureUML to B translation	79
3.22	A secure operation	79
3.23	The AAC metamodel [14]	82
3.24	Aspect code for object-based access control [15]	85
3.25	An example of the JSAL implementation in AspectJ [16]	87
3.26	JSAL architecture [16]	88
4.1	The class diagram of a simplified banking system	92
4.2	Generic SecureUML model	94
4.3	Static access control model	94

4.4	A generic form of a secure activity diagrams	95
4.5	Example of dynamic security rules modeled by activity diagrams: Rule 3	96
4.6	Example of dynamic security rules modeled by activity diagrams: Rule 4	96
4.7	Machine <i>Context</i>	98
4.8	The static part of the functional B specification	98
4.9	The functional specification of Operation <i>validateDeposit</i>	99
4.10	The B context machine of the SecureUML diagram of Figure 4.3	101
4.11	The B machine managing the roles of users	101
4.12	B translation of a SecureUML permission (Figure 4.2)	102
4.13	B translation of a SecureUML operation (Figure 4.3)	102
4.14	Machine <i>ActionsHistory</i>	103
4.15	Translation of a secure operation of an activity diagram	104
4.16	Translation of the activity diagram of Figure 4.5	104
4.17	Translation of the activity diagram of Figure 4.6	105
4.18	The abstract specification of Machine <i>Secure_Filter</i>	106
4.19	The concrete specification of Machine <i>Secure_Filter</i>	107
4.20	The architecture of the B specification	108
5.1	Translation workflow	114
5.2	Editing a class diagram under the B4MSECURE platform	116
5.3	The SecureUML metamodel (adapted from [17])	117
5.4	Editing a SecureUML diagram under the B4MSECURE platform	118
5.5	The secure UML activity metamodel	119
5.6	Example of dynamic security rules modeled by activity diagrams: Rule 2	120
5.7	Example of dynamic security rules modeled by activity diagrams: Rule 4	120
5.8	Translation of the class diagram	122
5.9	Generation of the B operation <i>makePayment</i> of the class <i>HospitalStay</i>	122
5.10	The B context machine of the SecureUML diagram of Figure 5.4	124
5.11	The B modeling of the static secure operation corresponding to the operation <i>makePayment</i> of Figure 5.4	125
5.12	The B specification of the history execution of the operation <i>makePayment</i>	125
5.13	The B translation of the activity diagram of Figure 5.6	126
5.14	The B specification of the filter for the operation <i>makePayment</i>	127

5.15	The concrete specification of the filter of the operation <i>makePayment</i>	128
6.1	The class diagram of a simplified purchase order	135
6.2	A secure activity diagram modeling a dynamic security rule . . .	136
6.3	The B specification of the class diagram	137
6.4	The B translation of the SecureUML diagram	138
6.5	The B translation of the activity diagram in Figure 6.2	139
6.6	The B specification of the filter of the operation Receive	140
6.7	The B specification of a relational table	142
6.8	The B evaluation of a predicate	142
6.9	The B specification of a log table	144
6.10	The B implementation of the operation <i>receive</i>	145
6.11	The B implementation of a log operation <i>LogReceive</i>	145
6.12	The B implementation of the operation <i>ADReceive</i>	146
6.13	Derivation of the AspectJ implementation	147
6.14	AspectJ implementation principle	148
6.15	Transformation of a B operation to a stored procedure	149
6.16	Transformation of B operation to JAVA method	151
6.17	Transformation of B expressions into JAVA	152
6.18	Generation of a table	163
6.19	Generation of a stored procedure <i>NotReceive</i>	163
6.20	Generation of the associated JAVA method <i>NotReceive</i>	164
6.21	Generation of the stored procedure <i>receive</i>	164
6.22	Generation of the associated JAVA method <i>receive</i>	165
6.23	Generation of the security data	165
6.24	Generation of the class <i>SecureUMLJAVATrans</i>	166
6.25	Generation of a log class	167
6.26	Generation of a log table	167
6.27	Generation of the log method	167
6.28	Generation of the method <i>ADOp</i>	168
6.29	Generation of the aspect class	168

CHAPTER 1

Introduction

Contents

1.1	Research Motivations	24
1.2	Research Contributions	26
1.3	Thesis Structure	27

An information systems (IS) is any system for the collection, organization, storage and communication of information. The impact of ISs on our economic and social life is increasing significantly due to the spreading of the ubiquitous computing. In these systems, security plays a vital role and is a central issue in their construction and manipulation. ISs are required to be more secure in order to resist to potential attacks. Due to the importance of security in ISs, it has been receiving a large number of interests.

One of the most common security issues is access control. An access control policy regulates accesses to the protected resources of a system. There are several access control mechanisms available, such as Mandatory Access Control (MAC), Discretionary Access Control (DAC), and Role-Based Access Control (RBAC). In this thesis, we choose RBAC for its advancement over MAC and DAC in term of simplifying the permissions management and improving the organizational productivity. Moreover, a large number of products support RBAC directly, and others support close forms of the role concept, such as user groups [18]. For example, the notion of role is considered as a part of the SQL standard in some commercial database management systems, such as Oracle [19] and SQL Server [20].

An important and challenging task is to tackle security requirements in the early stages of the development process. That can avoid an ad-hoc integration

of security mechanisms after the system has been implemented. It is also easier to analyze security policies at the abstract level because security specifications contain less details about specific platforms and underlying technologies in these early stages.

UML is a de facto standard to model requirements of secure systems on top of its textual descriptions. Graphical models provide intuitive and synthetic views of a system that can facilitate the communication among stakeholders, such as developers, designers and users. However, their semantics are often blurred, and therefore their meaning can be ambiguous to the interpretation. This may end up in implementing some undesirable functionalities. OCL is then designed to minimize the ambiguity of UML models [21]. Likewise, formal languages, such as Alloy [22], Z [23], and B [24] can be alternative techniques for precisely representing a system. Combining graphical and formal notations in software development has been an interesting area. As such, it is able to take advantages of both techniques, and they can complement each other: graphical models offer the visualization of a system, but they are not precise enough for the reasoning; due to the precision of mathematical notations and automatic reasoning, formal methods can overcome the ambiguity of these graphical models and provide rigorous specifications of the system.

Security is seen as a crosscutting concern of an application since it impacts on nearly each component of the application. The classical methods often integrate security codes inside the components that they affect. That is a major cause of the code tangling and scattering. Consequently, it is remarkably difficult to maintain the program, especially in large systems. We believe that security policies should be enforced as crosscutting concerns. It means that the security logic is separated from the application program, but it should be able to automatically intercept method invocations. Regarding the separation of concerns, the aspect oriented programming (AOP) stands out to be the best technique. It allows the modularity of crosscutting concerns by introducing additional behaviors into an existing code without any modification. Indeed, it separately encodes these concerns within *aspects*, specifies which pieces of code are modified through *pointcuts*, and injects additional behaviors implemented in *advices* into the business program without cluttering it.

1.1 Research Motivations

The literature review shows that functional and security models of a secure information system are generally treated separately in the design step. However, security requirements have a large impact on the functional model. Thus, it is

necessary to consider both models at the same time in order to specify the effect of security concerns and to have a complete security analysis. There are a few attempts that specify the security aspect along with the functional model of a secure system. For example, the work of Basin et al. [2] allows to point out which elements of the functional model are protected in the security model. Our work is influenced by this direction.

The use of formal languages, such as Z, B, and Alloy on top of graphical models of secure systems has been investigated in several studies [10, 11, 25]. The combination of formal and graphical techniques can produce specifications which, on one hand, can be understood and then validated by participants (e.g. developers, designers, and users) and, on the other hand, can be formally verified using the different tools available for formal methods. We are interested precisely in the B method as it is a complete formal language, and it has been used in many industrial projects, especially in railway systems (Metro line 14, the Charles de Gaulle airport shuttle automated pilot, etc.). Moreover, it has reliable free tools (AtelierB [26], ProB [27]) to support the whole software development process.

Despite the benefits of combining graphical and formal methods, developers are still reluctant to apply such approaches. The reason is that using formal methods requires a well-prepared knowledge about mathematical basics, which is difficult and challenging for their users. Automating model transformations from security models to formal specifications is an obvious need of developers. It can also improve the quality of the produced system and the productivity of the development process. Several efforts are deployed to automatically translate UML models into B specifications [28–30]. Nonetheless, these works focus only on the functional aspect of a secure system. The B4MSecure platform [13] permits to extract various B specifications from functional and static security models, but it does not consider dynamic security requirements. We particularly complement this work by supporting the dynamic security aspect of secure information systems.

Once security and functional requirements are specified, the development process consists in implementing them. Our goal is to separate the security and functional codes so that the structure of the final system is clear and easy to maintain. That is why we choose an approach based on AOP. There are many efforts on applying this paradigm to enforce security policies [14–16, 31–33]. Notwithstanding, these works consider access control constraints in the implementation stage [15, 16, 31, 32] or generate security codes from not-yet-verified models [14, 33]. Our approach covers the whole development life-cycle of secure systems: it starts by graphically modeling the functional and security

requirements; the graphical models are then translated into formal specifications that are successively validated and verified; the proved specifications are in turn refined until it is possible to straightforwardly map them into an AOP-based program.

1.2 Research Contributions

The dissertation aims at providing a Model Driven Engineering (MDE) approach to build secure information systems. The following describes our contributions to the specification, model transformations, and enforcement of security policies involving the functional aspect of such systems:

- **Specification of security design models:** we propose to graphically express functional and security models by UML-based diagrams: UML class diagrams are used to represent the functional requirements, SecureUML models static security rules, and UML activity diagrams describe dynamic security rules related to ordering and history-based constraints. These visual models are then translated into B specification modules. This translation intends to have rigorous descriptions and to avoid multiple interpretations for the specifications. It also allows to validate and verify a security policy at the platform independent level. That means that the security policies are specified and validated before their implementation. Therefore, the security enforcement code can vary in different settings. We also derive an access control filter that integrates different specifications related to a secure operation: i.e. functional, static, and dynamic specifications. As such, the security policies can be verified with respect to the functional specification.
- **Enforcement of access control policies:** our security enforcement approach allows a separation of concerns. In other words, the security logic and the main functionalities of an application are separated. To do so, we devise translation rules from the B specification to an AOP-based application. Essentially, the verified B specification is refined until obtaining a relational-like B implementation that is straightforwardly mapped into an AspectJ-based program connected to a Relational Database Management System (RDBMS).
- **Automation of the languages transformations (from UML to B and from B to AspectJ/JAVA/SQL):** as stated above, the graphical models of a secure application are transformed into the B specifications.

We have done this transformation automatically by developing a tool. The generated B specifications are correctly checked using AtelierB [26] without any additional modification. Most proofs are automatically discharged: the abstract and the refinement specifications of the filter are automatically proved. The verification of these models can also be done by playing scenario using the ProB [27] animator. ProB implements an automatic model-checking technique to verify a B specification. Such a specification is moved through several state changes, starting from a valid initial state into a state that violates its invariant. As such, security flaws that the static check cannot detect can be recognized.

The initial B specification is successively refined into a relational-like implementation, which can be mapped into an AspectJ-based application connected to a DBMS. Using our tool, the executable code can be automatically generated from the formal implementation. The security code and the main program are produced separately, making the application easier to track and maintain. The JAVA/SQL program is derived from the functional specification, while the access control filter results in the aspect code. The generated SQL statements are correct with respect to the SQL Server syntax.

1.3 Thesis Structure

The thesis is organized in seven chapters:

- **Chapter 2: Backgrounds** presents a description of the different concepts needed for understanding the rest of the manuscript. It includes theoretical details of the B method. It also presents an overview of Model Driven Engineering, role based access control, and aspect oriented programming. SecureUML and RBAC features supported in database management systems are also covered in this chapter.
- **Chapter 3: The State of The Art** reviews the existing works related to the development of secure systems. We present techniques for the specification of access control requirements and supporting tools for specifying and validating RBAC models. This chapter also includes a number of approaches that deal with model transformations of RBAC policies to implementation codes. The state of the art in the security enforcement based on AOP is also considered in this chapter.

- **Chapter 4: Formal Development of a Secure Access Control Filter** describes graphical representations of a secure system using UML-based models and their translation into B. We present the mapping rules from UML class diagrams, SecureUML, and secure activity diagrams into B specifications, through which we can ensure the consistency of these models and validate the system. Finally, a proved filter, which permits to take into account different security rules, is formally derived using the B refinement technique.
- **Chapter 5: A Tool for the Generation of a Secure Access Control Filter** describes a tool that automates the translation presented in Chapter 4. The tool is built based on the Eclipse platform which integrates the TopCased modeling environment.
- **Chapter 6: A Formal Approach to Derive an Aspect Oriented Programming-Based Implementation of a Secure Access Control Filter** proposes an AOP-based approach for the security enforcement. It describes the translation rules from the presented B specifications into an AspectJ implementation connected to a SQL server. The automation of this transformation is also provided here.
- **Chapter 7: Conclusions and Future Works** sum up our claimed contributions along with future perspectives.

CHAPTER 2
Background

Contents

2.1	Introduction	30
2.2	The B Method	30
2.2.1	Abstract machine	30
2.2.2	Refinement	34
2.2.3	Discussion	36
2.3	Model-Driven Engineering	37
2.3.1	An overview	37
2.3.2	Model-Driven Architecture	37
2.4	Role Based Access Control	38
2.4.1	Core RBAC	39
2.4.2	RBAC Constraints	40
2.4.3	SecureUML	41
2.5	RBAC in Database Management Systems	41
2.5.1	Database User	42
2.5.2	User-defined Database Role	42
2.5.3	User-Role assignment	42
2.5.4	Permission assignment	43
2.6	Aspect Oriented Programming	44
2.6.1	An overview	44
2.6.2	AspectJ	46
2.7	Conclusion	48

2.1 Introduction

This chapter presents the basic knowledge and concepts to help the readers understand the different contributions described in the rest of the manuscript. We will start by the B method in Section 2.2. Afterwards, we will give an overview of Model Driven Engineering in Section 2.3. Role-based access control and its features supported in relational database management systems will be introduced in Sections 2.4 and 2.5 respectively. Finally, we will present aspect-oriented programming in Section 2.6.

2.2 The B Method

The B method is a formal language introduced by J-R Abrial. Based on *first-order logic*, it provides a means for “the central aspects of the software life cycle, namely: the abstract specification, the design by successive refinement steps, the layered architecture, and the executable code generation” [24]. In other words, the B method covers most of the life-cycle stages of the software development process: from the design phase to the implementation phase. That allows the early error detection of systems. It is commonly known that the earlier an error is detected, the easier and cheaper it is to fix

B formal representations express different levels of abstraction of a system, from an abstract specification to an executable code (Figure 2.1). The abstract specification describes the fundamental properties of the product. Conceptual details are added incrementally through the refinement process. The last result of the refinement step, called the implementation, should be close to the target programming language. As such, we may generate an executable code.

The benefit of using B is the availability of supporting tools. For instance, AtelierB [26] developed by ClearSy is an industrial tool for analyzing, checking types, and generating proof obligations of a B component. AtelierB provides an automatic prover with a predefined rule base and an interactive prover permitting users to define their own rules. Another tool, called ProB [27], allows the automatic animation of a B specification.

2.2.1 Abstract machine

An abstract machine is a fundamental concept in B. It is seen as a “pocket calculator” [24], which includes an invisible memory and a number of keys. The memory stores the state of the machine, while the keys are operations used for any modification of the state.

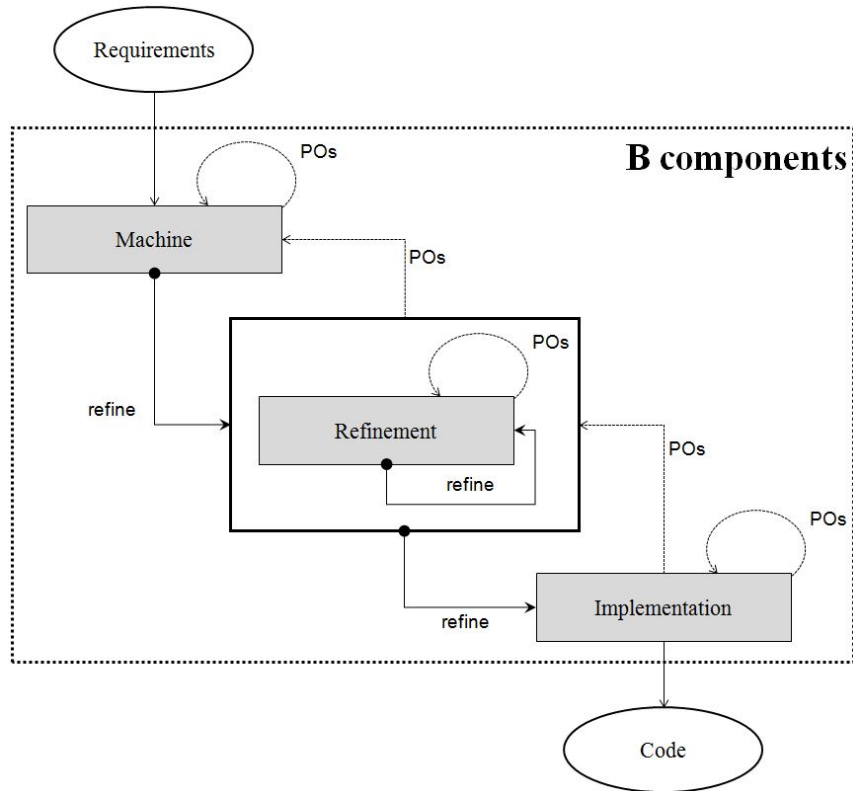


Figure 2.1: Software development process in B

Each abstract machine is composed of static and dynamic parts (See Figure 2.2):

- **The static part** declares the state of a system in terms of sets (the SETS clause), constants (the CONSTANTS clause), and variables (the VARIABLES clause). The definition of these declarations is specified by the *Predicate Calculus* and *Set Theory* languages. Predicates denoting the properties of the constants are defined in the PROPERTIES clause. The *INVARIANT* clause permits to type the variables and to define the properties that they must always verify.
- **The dynamic part** is expressed through the initialisation (the INITIALISATION clause) and the operations (the OPERATIONS clause). The initialisation aims to set initial values for the variables. The operations describe the evolution of a machine by modifying the state of that machine. An operation is specified as a non-executable pseudo-code, and it does not include any sequence or loop. This pseudo-code is formalized

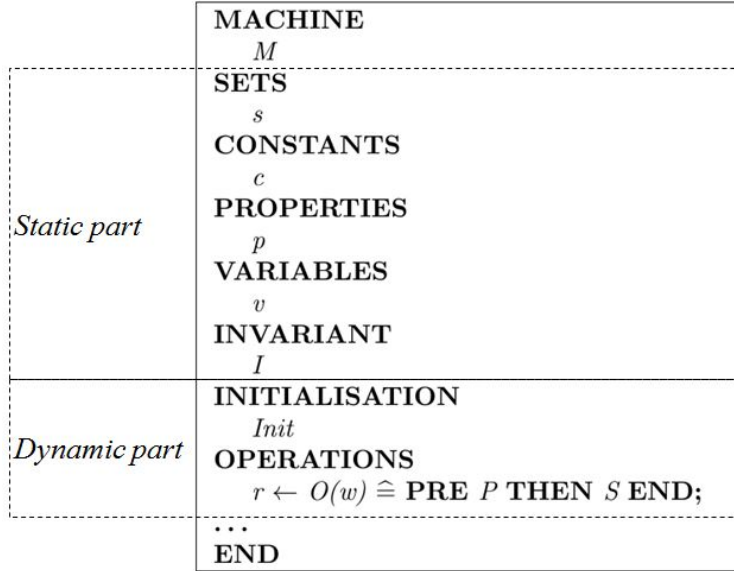


Figure 2.2: B machine

in the *Generalized Substitution Language*. Each substitution used in our work is an assignment statement that specifies variables to be modified. An operation is executed under a precondition (P) specifying the necessary and sufficient conditions that make the invariant satisfied after its execution (after executing S). An operation may also assume parameters (w) and return a value (r). The general form of a B operation can be defined as follows:

$$r \leftarrow O(w) \hat{=} \text{PRE } P \text{ THEN } S \text{ END};$$

Abstract machine correctness

Definition: “A proof obligation is a mathematical formula to be proven, in order to ensure a B component is correct” [24].

The proof obligations generated for an abstract machine cover:

- **the correctness of the initialisation** (2.1): the initialisation is correct if and only if the invariant (I) is true after executing the initialisation ($Init$):

$$[Init]I \tag{2.1}$$

Table 2.1: The visibility of the SEES clause

Objects (of seen machine)	Clauses (of seeing machine)		
	PROPERTIES	INVARIANT	OPERATIONS
sets	visible	visible	visible
constants	visible	visible	visible
variables	no	no	read-only
operations	no	no	no

- **the correctness of operations** (2.2): the proof obligations of an operation verifies that the invariant remains verified after the execution of that operation:

$$I \wedge P \Rightarrow [S]I \quad (2.2)$$

Modularity of an abstract specification

Modularity is an efficient solution to deal with the complexity of a system [34]. The concept of an abstract machine is very close to some well-known concepts, such as class and module in programming languages. This notion supports the modularity in software development, thus it allows us to construct abstract machines in a shared way.

At an abstract level, B introduces four clauses relating to the modularity. These clauses differ on the visibility rules. Here, we only present the clauses that are used in our work.

- The **SEES** clause addresses a list of machines sharing a part of data with the seeing components. The visibility of the elements of a seen machine within different clauses of the seeing machine is shown in Table 2.1. The rows correspond to the elements of the seen machine and the columns correspond to the elements of the seeing machine. The value *visible* indicates that the considered element of the seen machine is accessible in specific clauses of the seeing machine. The value *no* means that the considered element of the seen machine cannot be referenced in specific clauses of the seeing machine.

Table. 2.1 shows that the sets and the constants of the seen machine are fully visible within the seeing machine. The variables of the seen machine can not be referenced within the invariant of the seeing machine. Although they are visible in the operations of the seeing machine, they cannot be

Table 2.2: The visibility of the INCLUDES clause

Objects (of included machine)	Clauses (of including machine)		
	PROPERTIES	INVARIANT	OPERATIONS
sets	visible	visible	visible
constants	visible	visible	visible
variables	no	visible	read-only
operations	no	no	visible

modified. The operations of the seen machine are fully invisible within the seeing machine.

The SEES clause is non-transitive: if a machine M_1 sees a machine M_2 which, in turns, sees another machine M_3 , then M_1 has to see M_3 again if M_1 requires certain information of M_3 .

- The **INCLUDES** clause allows us to include other machines in a given machine. The visibility of the different elements of an included machine is shown in Table. 2.2.

As we can notice, the sets and the constants of an included machine are accessible from anywhere in the including machine. The variables can be only read by the operations of the including machine. The operations of the included machine can be called within the operations of the including machine.

The INCLUDES mechanism is transitive: if a machine M_1 includes a machine M_2 , then M_1 does not need to include again the machines already included in M_2 . These included machines of M_2 are implicitly included in M_1 .

2.2.2 Refinement

Refinement is a technique that is used to transform an abstract model into a more concrete one. Basically, refinement consists in weakening preconditions and replacing a parallel substitution by a sequence one. The last refinement step, called implementation, aims at obtaining data and substitutions close to those of a programming language such that the translation into the chosen target language of the data and control structures (used in this level) must be a straightforward task. Each refinement primarily concerns the two following notions:

- **VARIABLES:** abstract variables may be preserved, deleted, or changed the form within the *VARIABLES* clause of a refinement. A refinement can also introduce new variables. In this case, the invariants that links the abstract variables and the concrete ones must be specified.
- **OPERATIONS:** each abstract operation must be refined accordingly in refinements. To do so, we remove non-executable elements of each operation, such as precondition and choice. We can use more concrete and deterministic substitutions or call operations of other abstract machines imported within the implementation.

The refinement of an abstract machine is performed gradually in several steps rather than all at once. Details of the problem are added incrementally through each step. Note that the B refinement cannot change the signature of the operations or introduce new operations. The last level of the refinement is called implementation which should be close to a particular programming language, such as C, Ada, or Java.

Refinement correctness

To prove the correctness of a refinement, we have to establish:

- **The correctness of the initialisation (2.3):** it proves that the initialisation of a refinement ($Init_r$) establishes its invariant (I_r) without contradicting the initialisation of the refined component ($Init$).

$$[Init_r] \neg [Init] \neg I_r \quad (2.3)$$

- **The correctness of the operation (2.4):** assuming that S_r is the substitution of the concrete operation, I_r is the invariant of the refinement. A refinement operation is correct when it preserves the invariant without contradicting the specified operation. In other words, the effects of the refinement operation must not be in contradiction with the effects specified in the abstract operation. The aim of the proof obligation will therefore be built around a double negative.

$$I \wedge P \wedge I_r \Rightarrow [S_r] \neg [S] \neg I_r \quad (2.4)$$

Table 2.3: The visibility of the **IMPORTS** clause

Objects (of imported machine)	Clauses (of implementation)		
	PROPERTIES	INVARIANT	OPERATIONS
sets	visible	visible	visible
constants	visible	visible	visible
variables	no	visible	no
operations	no	no	visible

Implementation

Implementation is the last refinement. It cannot be refined further. As already mentioned, the implementation should be easily translated into a given programming language. The implementation data must be concrete and the abstract sets have to be valued. The precondition, parallel, choice, and ANY substitutions are not allowed in the implementation.

Modularity of an implementation

The implementation of a refinement may import some machines to implement its variables and operations (the **IMPORTS** clause). The imported machines offer their data and operations, which are either refined into an implementation or easily translated into the target language. Table 2.3 describes the visibility of the **IMPORTS** clause. As we can see, the sets and the constants of the imported machine are fully visible within the implementation. The variables of the imported machine are only accessible within the invariant of the implementation. This aims to express the relationship between the variables of the imported machine and those of the component refined by the implementation. The operations of the implementation are encoded by calling the operations of the imported machines.

2.2.3 Discussion

To summarize, the main advantages of the B method are:

- the modularization facilities.
- the code generation
- the availability of supporting tools.

The B method has been successfully used in industrial projects, especially in the domain of safety critical systems [35]. Among industrial large-scale projects developed in B, the METEOR project [36] is the first one that fully applies the formal process to develop the first automatic train operating system in Paris (line 14). In the establishment of the VAL shuttle for Roissy Charles de Gaulle airport [37], driverless light trains developed by Siemens Transportation Systems, B has been used for construction of the software.

2.3 Model-Driven Engineering

2.3.1 An overview

Model Driven Engineering [38], MDE for short, is a promising paradigm for software development. The goal of MDE is to increase the abstraction level in representing a system and the automation in constructing it. In MDE, *models* play an important role in software development. The automation in the production of the system can be achieved by *model transformations*.

Models are considered as central entities in MDE. A model is an abstract representation of a (part of) system. It shows a partial view of the system by simplifying the one that needs to be captured or automated. A model focuses on the relevant information of a (part of) system rather than details from a given viewpoint. Therefore, it often requires multiple models for a better representation of a system. Promoting the use of models intends to improve quality of the resulting software, because it is easier to understand, simulate, analyze, and validate abstract models than computer programs.

Model transformations allow a source model to be automatically transformed into a target model according to transformation rules. One can define mappings between models in the same or different languages. Transformation tools are essential to maximize the benefits of using models and minimize the effort of constructing the modeled system. Using supporting tools can reduce the burden of hand-coded tasks, that are tedious and error-prone. Thus, they can significantly improve the productivity of the software development.

2.3.2 Model-Driven Architecture

Model Driven Architecture (MDA) [39] is known as the best MDE initiative. It is a mission of the Object Management Group (OMG) in “solving integration problems through open, vendor-neutral interoperability specifications” [39]. The objective of MDA is to improve productivity, portability, interoperability, and

reusability by using models to describe a system. The core of MDA is multiple OMG's standards, including: the Unified Modeling Language (UML), Meta Object Facility (MOF), XML Metadata Interchange (XMI), and the Common Warehouse Metamodel (CWM).

The MDA framework includes the following three model types:

- **Computation Independent Model (CIM)** describes what a system is expected to do, but hides all information on how this system will be implemented.
- **Platform Independent Model (PIM)** describes the function and the structure of a system without any technical details. It provides a degree of independence to the features of a specific platform. Not surprisingly, all PIMs are expressed in UML.
- **Platform Specific Model (PSM)** represents the business model with technical concepts of a particular type of platform. Although it is still a UML model, it should be able to simulate how a system operates on the target platform.

Automating model transformations is a major goal of MDA. A transformation can be executed through tools to transform PIM to PIM, PIM to PSM, PSM to PSM, and PSM to code. In some restricted cases, the whole application can be generated: the more completely the PSM specifies the target platform environment, the more completely the code is generated. As a result, we can reduce the effort of hand programming and the cost of the models maintenance. In the transformation step, MOF is used to define languages and transformations between languages.

During the development, some tools can be used to model, verify, compare, transform, analysis, test, and simulate.

2.4 Role Based Access Control

Role based access control (RBAC) [40] is a de facto standard for controlling access to information systems. It has been standardized by the NIST (National Institute of Standards and Technology) and widely used in industrial projects. The idea is to regulate accesses to the resources of a system basing on the roles of the users. RBAC permits to reduce the complexity and cost of security management within enterprises [41]. Indeed in RBAC, a role is relatively persistent with respect to a job function or title within an organization. Individual users

are assigned to roles, which permissions are associated with. As a result, a user gains rights based on his roles.

There are two principal RBAC concepts, namely the core RBAC and constraints. These concepts are explained in the following subsections.

2.4.1 Core RBAC

Core RBAC is an essential part in any RBAC system. The static aspect of core RBAC addresses five basic data elements: users (*Users*), roles (*Roles*), permissions (*Permissions*), objects (*Objects*), and operations (*Operations*) executed on objects.

- *Users* are people who use the system.
- *Roles* are permanent job titles within enterprises.
- *Permissions* are the ability to perform operations on protected objects.
- *Objects* are potential resources to protect, that contain or receive information, such as files or directories in an operation system.
- *Operations* are actions through which users can perform on objects in the system. The type of these operations depends on the considered system. For example, within a database management system, operations include insert, select, delete, and update.

Figure 2.3 shows the core RBAC elements and their relations. The central of RBAC is the concept of role relations, including *User Assignment* and *Permission Assignment*. A user assignment shows that a user can play one or more roles, and a single role can be assigned to more than one user. Comparably, a permission assignment is a many-to-many relation. That means that a single permission can be assigned to several roles, and a single role can be associated to multiple permissions.

Core RBAC also introduces the concept of *Sessions*, which is associated with the dynamic aspect of RBAC. During a session, a user might activate one or many roles. At a given moment, the permissions which are available to a user are those associated to the roles activated during the user's session. A user may establish one or more sessions, but a session is established by a single user.

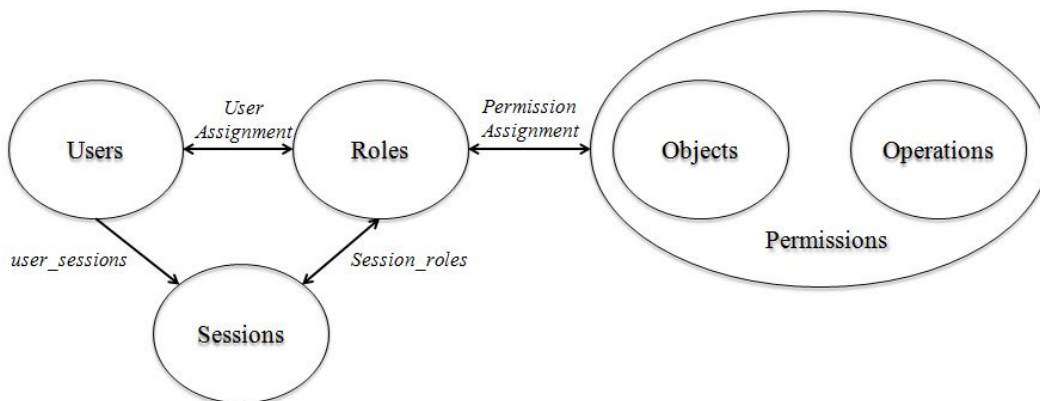


Figure 2.3: Core RBAC [1]

2.4.2 RBAC Constraints

It has been long recognized that collusions among users is a major security issue within an organization. To minimize such a fraud, additional constraints are addressed in RBAC models. For example, one may require different users to perform a delicate business function. This is a type of separation of duty constraints.

Separation of duty can be either static or dynamic [42]. The distinction between static separation of duty (SSD) and dynamic separation of duty (DSD) constraints is based on the moment they are established. A SSD constraint can be defined during the assignment of users to roles. For example, we may want to state that two conflict roles should not be assigned to the same user. On the other hand, a DSD policy should be verified during the execution of the system. For instance, a user is allowed to activate two exclusive roles independently but it is prohibited to activate them simultaneously.

History-based constraints can be seen as a special type of DSD constraints. This kind of constraints takes into account the history of the system resources access. In some situations, a coordination of permissions is necessary in order to prevent fraud. For example, within a shopping online system, a special order has to be received by the client who demanded it. That means we have to store the actions history in order to grant or deny other actions. On the other hand, a fraud prevention can be implemented by splitting a business functionality into a set of actions that are assigned to different users. For example, in an inventory system, we may want to state that a purchase order cannot be created and received by the same user.

2.4.3 SecureUML

SecureUML [2, 43] is a UML-based modeling language for access control policies. SecureUML extends UML models with concepts of RBAC, such as roles, users, and permissions. Using SecureUML, it is possible to visually represent security requirements based on RBAC along with the functional aspect of secure systems.

The metamodel encoding the abstract syntax of SecureUML is depicted in Figure 2.4. Essentially, this language expresses which roles are assigned to which users, which permissions are assigned to which roles, which actions and constraints are assigned to which permissions, and which resources are assigned to which actions. Actions related to permissions can be either atomic or composite. An atomic action denotes an actual operation from the modeled system, whereas a composite action groups lower-level actions to specify permissions for a set of actions.

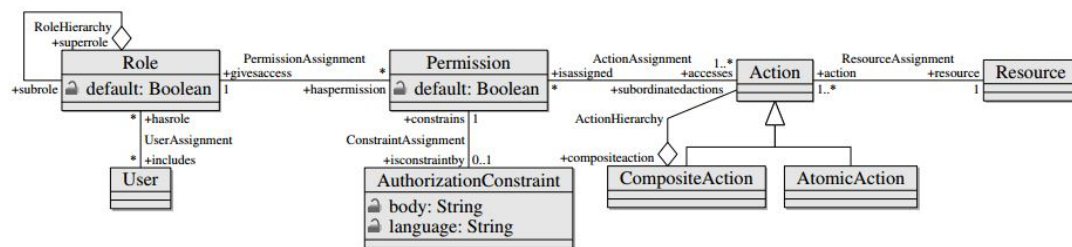


Figure 2.4: SecureUML metamodel [2]

SecureUML allows a separation of the functional and security aspects of the system to secure. The functional component is represented using UML diagrams, whereas the security component is designed using UML profile extensions like stereotypes, tagged values, and constraints. SecureUML can also be combined with other design modeling languages, such as ComponentUML and UML class diagrams. The goal is to automatically generate applications and their associated security infrastructures from those combined models.

2.5 RBAC in Database Management Systems

RBAC has received a considerable attention as an alternative approach for traditional mandatory and discretionary access controls, because it significantly simplifies the security management within organizations. Recently, most popular commercial database management systems support RBAC. In this section, we analyze RBAC features supported in Microsoft SQL Server (version 2014).

2.5.1 Database User

Each database has a list of database users. Every user is mapped to a login, which is an identity that the user uses to authenticate himself within a database instance. A login can be created, changed, and removed from a database. Similarly, we can create a new user and delete an existing user in a database.

Before creating a user in SQL Server, a login must be created. The **CREATE LOGIN** statement creates a new login that can be used to connect to a SQL Server instance. Because it is a high-rank privilege, only security administrators can have this right. A login name within a database must be unique. The following statement creates the *bobLogin* login with a password *bobPwd*.

```
CREATE LOGIN bobLogin  
WITH PASSWORD = bobPwd
```

After a login is created, it is then mapped to a user. To create a new user, we must use the **CREATE USER** statement. Only a security administrator has the right to execute this statement for it is a powerful privilege. For example, the following statement creates a user whose name is *bob* for the login *bobLogin*:

```
CREATE USER bob FOR LOGIN bobLogin;
```

2.5.2 User-defined Database Role

In SQL Server, a user-defined database role is created through the **CREATE ROLE** statement. The name of a role must be unique in the database. We can also specify the owner of the new role, i.e. a database user or role, by the **AUTHORIZATION** option. If no owner is specified, the new role will be owned by the user who executed the creation of that role. The following syntax creates the *manager* role.

```
CREATE ROLE manager;
```

2.5.3 User-Role assignment

Microsoft SQL Server supports a many-to-many relation between the users and the roles. That means that a role can have multiple users called members, and a user can play more than one role. To assign a role to a user, we use the **ALTER ROLE** statement together with the **ADD MEMBER** option. For example, the statement that assigns *bob* to the *manager* role is as follows:

```
ALTER ROLE manager ADD MEMBER bob;
```

2.5.4 Permission assignment

In SQL Server, database-level permissions regulate the use of specific commands that access certain objects within the scope of the specified database. An object in SQL Server can be a table, a view, a table-valued function, a stored procedure, etc. In the thesis, we consider solely the granting of permissions on stored procedures.

Stored procedures

A stored procedure is a compiled database object that contains one or more SQL statements. Its goal is to reuse the code that is called in many places within applications. Instead of writing a query each time needed, we can encode it as a part of a stored procedure and call it when needed. A stored procedure is created using the **CREATE PROCEDURE** statement. For example, a simple stored procedure that gets all the bills in the *OnlineShop* database is defined by:

```
CREATE PROCEDURE getBill
AS
SELECT * FROM OnlineShop.Bill
GO
```

The stored procedure offers several advantages, such as:

- As they are saved within the database, the syntax is checked in the database.
- Writing redundant code is avoided. Indeed, a stored procedure can be called as many times as needed from any machine that connects to the database.
- Grant permissions to execute a stored procedure can be assigned to users/roles.

Grant permissions

The owner of an object has the right to grant permissions on that object, and permissions can be assigned to roles or users. The syntax of granting permissions on objects is defined by:

```
GRANT <some_permission>  
ON <some_object>  
TO <some_user/some_login/some_role>
```

Regarding the stored procedure, the granting syntax implies:

- < *some_permission* >: *EXECUTE*.
- < *some_object* >: the name of the granted stored procedure.
- < *some_user* | *some_login* | *some_role* >: a database user, a log in, or a database role.

The following example grants the **EXECUTE** permission to the *manager* role on the stored procedure *getBill* within the *OnlineShop* database. The **OBJECT** phrase is optional if the schema name is specified. If the **OBJECT** phrase is used, the scope qualifier (::) is required.

```
GRANT EXECUTE  
ON OBJECT::OnlineShop.getBill  
TO manager;
```

Granting the execution of a stored procedure to a role also means that this role is allowed to performed actions encapsulated within the stored procedure. The below list details the actions of the objects, that can be included in the body of the stored procedure.

- Table actions: DELETE, INSERT, REFERENCES, SELECT, UPDATE.
- Column actions: SELECT, REFERENCES, UPDATE.
- ...

2.6 Aspect Oriented Programming

2.6.1 An overview

Aspect-Oriented Programming (AOP) [44] is a programming technique for improving the separation of crosscutting concerns in softwares. The goal is to help the engineer develop and maintain large applications. By doing so, it allows the developer to separately specify crosscutting concerns, and to insert them into the main logic without modifying the program itself.

Crosscutting concerns tend to affect several other modules in the system. Thus, their related implementations often span multiple implementation modules. This can cause *code scattering*: code is duplicated in several modules. Thus, changing a concern requires to modify all the affected modules. On the other hand, in the system, a module may interact simultaneously with different crosscutting concerns. This can cause *code tangling*: several concerns are implemented in the same modules. That means that changing a concern may cause unintended modifications of other tangled concerns. Examples of crosscutting concerns are security, logging, synchronization, persistence, and so on.

AOP attempts to solve the tangling and scattering problems related to crosscutting concerns by allowing the implementation of such concerns in stand-alone modules called *aspects*. The concept of an aspect is similar to a class: it can be abstract or concrete; it can extend other classes and aspects; it can contain attributes and methods. Applying AOP, the concern remains being a crosscutting to other modules, but localizing concerns is easier and clearer.

The compilation of a program developed using AOP differs from the compilation of an ordinary program in the way that, the aspect weaver tailors the aspect code with the main code before compiling into an executable code (Figure 2.5). The weaver accepts the core functionality and the aspect program as inputs, and produces the desired total system operation.

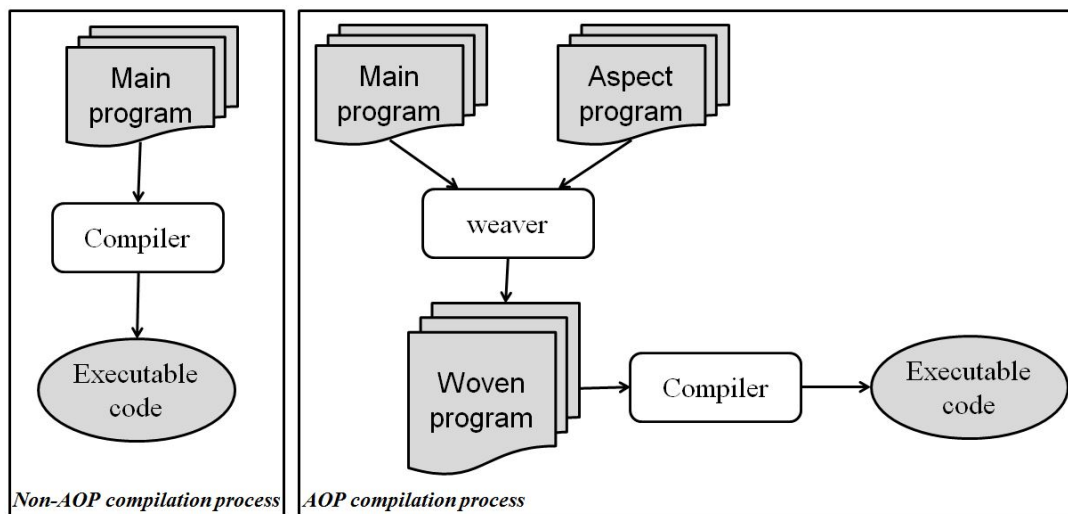


Figure 2.5: Normal compilation process (left) and AOP compilation process (right)

AOP languages define three main elements for the modularization of crosscutting concerns. These elements are explained in Subsection 2.6.2 introducing

AspectJ:

- Joinpoints.
- A means for identifying joinpoints.
- A means for specifying behaviors at joinpoints.

2.6.2 AspectJ

AspectJ [45] is a general-purpose aspect-oriented extension for Java. It is a de facto standard for AOP. AspectJ uses Java syntax for encoding crosscutting concerns. As a result, AspectJ code is compiled into standard Java bytecode.

Joinpoint

The central concept in AOP is the joinpoint. A joinpoint is a position in the execution of a program. AspectJ allows a diversity of locations where a joinpoint can refer to, including:

- *Method/Constructor call* at which a method (a constructor of a class) is called.
- *Method/Constructor execution* at which an individual method/constructor is invoked.
- *Field get* at which a field of a class, object, or interface is read.
- *Field set* at which a field of an object or class is set.

...

Pointcut

A pointcut (or a pointcut designator) is a collection of joinpoints and optional values in the execution context of the pointcut. A pointcut selects a number of joinpoints based on defined criteria. The criteria can be method names or method names specified by wild-cards. Pointcuts can be combined using *and* (&&), *or* (||), and *not* (!) operators.

Advices

The concept of advice is similar to that of function. Advices are used to specify code that should be executed at each joinpoint specified in a pointcut.

In AspectJ, there are three critical types of advices as follows:

- **Before advice:** the advice code starts execution when a joinpoint is reached but before the computation proceeds.
- **After advice:** the advice code starts execution after the computation of a joinpoint, but before exiting this point.
- **Around advice** is the most important and powerful advice. Such an advice surrounds the joinpoint, i.e. a part the advice code is executed before and another part is executed after the execution of the joinpoint. This advice can also choose to proceed the execution of the joinpoint or not.

Aspect

Aspects are modular units of crosscutting implementation. An aspect is a class that implements concerns that cut across other classes. In AspectJ, an aspect is declared by the keyword *aspect*. It includes pointcuts and advices. Only an aspect can define advices.

An example

To illustrate how AspectJ works, let us take a simple HelloWorld example. The method *sayHello* simply prints a hello message to a given name.

```
//HelloWorld.java
public class HelloWorld
{
    public static void sayHello(String name)
    {
        System.out.println ("Hello, " + name);
    }
}
```

The aspect in this example aims to add messages before and after executing the method *sayHello*. Before printing the greeting message "Hello, name", the program should print "A greeting message from aspect!". After printing the greeting message, it should print "See you soon!".


```
//HelloWorldAspect.java
public aspect HelloWorldAspect
{
    pointcut callSayHello() : call(public static void
        HelloWorld.sayHello(...));
    //A before advice
    before() : callSayHello()
    {
        System.out.println("A greeting message from aspect!");
    }
    //An after advice
    after() : callSayHello()
    {
        System.out.println("See you soon!");
    }
}
```

2.7 Conclusion

This chapter describes the basic concepts related to our work. First, we introduced the B method, a formal language that we use to formalize, validate, and verify security rules together with the functional requirements of an information system. Then, we presented an overview of model-driven engineering that is a software development methodology applied in our research. The security mechanism that we aim to treat, namely role-based access control, was carried on. Its features integrated in a commercial database management system was also studied. Finally, we specified the aspect oriented programming paradigm that we target to. In the next chapter, we will advance our investigations with existing works related to the topics that we are interested in.

CHAPTER 3
State of The Art

Contents

3.1	Introduction	50
3.2	Techniques for Security Specification	50
3.2.1	UML and OCL based approaches	50
3.2.2	Alloy-based Approaches	58
3.2.3	Z-based Approaches	63
3.2.4	A B-based Approach	68
3.2.5	Discussion	72
3.3	Support Tools for Access Control Policies	74
3.3.1	SecureMOVA	74
3.3.2	B4MSecure	77
3.3.3	Discussion	79
3.4	Implementation of An Access Control Specification	80
3.5	Enforcement of Access Control Policies	82
3.5.1	Java Authentication and Authorization Service	83
3.5.2	Annotation-based approaches	83
3.5.3	AOP-based approaches	84
3.5.4	Discussion	86
3.6	Conclusion	88

3.1 Introduction

In this chapter, we survey the state of the art on the specification and enforcement of access control policies together with a number of supporting tools in the security development. We start by examining approaches on security specification using semi-formal and formal languages in Section 3.2. Afterwards, we review supporting tools for modeling, analysis and model transformations of security policies in Section 3.3. A selection of approaches on security code generation is described in Section 3.4. Section 3.5 summarize a number of security enforcement techniques. Finally, we conclude the literature review in Section 3.6.

3.2 Techniques for Security Specification

This section reviews the literature on security specification techniques that employ RBAC variants [40]. These techniques fall into two categories: semi-formal techniques refer to approaches using UML and OCL, while formal techniques are based on formal languages, such as Alloy [22], Z [23], and B [24]. The benefits and limitations of the presented approaches are also discussed.

3.2.1 UML and OCL based approaches

UML has been a standard modeling language in the software industry for decades. Meanwhile, OCL [46] has been used for expressing and analyzing constraints in object oriented models as a standard constraints specification language. Due to the fact that UML and OCL are widely used in industrial environments, there are a large number of UML and OCL based approaches to specifying access control requirements. In this manuscript, we highlight a selection of them.

Shin et al. in [3] used different UML models to represent various views of RBAC. These views are as follows:

- **Static:** the static view embodies the conceptual structure of RBAC using a UML class diagram. Figure 3.1 depicts the static model of RBAC. Classes are used to represent the basic elements of RBAC, namely *User*, *Role*, *Permission*, *Constraint*, and *Session*. The *Role* and *Permission* classes may be specialized into two categories, i.e. user and administrative, depending on the level of users' qualification. A constraint is defined for a user, a role, a permission, or a session. In addition, the static model

has a special class called *Session Hour*. This class is used when a user has to establish a session to activate her/his roles. This notion is useful to express session-based constraints. For example, an organization may require that a session is only valid within one hour.

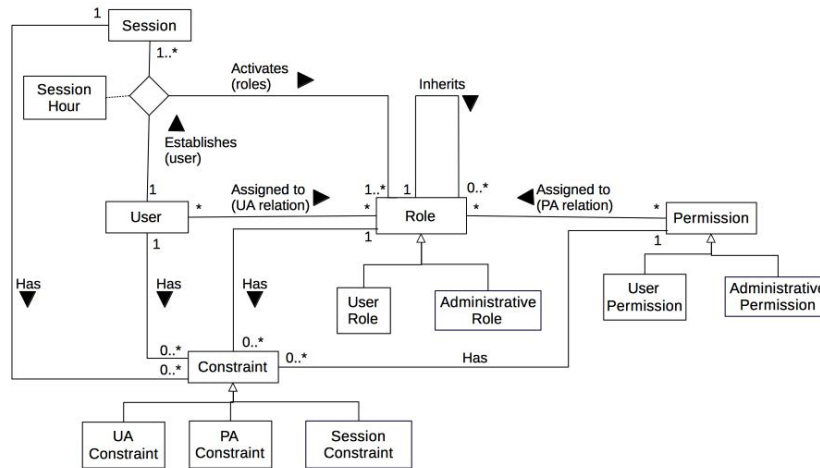


Figure 3.1: A static model [3]

- **Functional:** the functional view constructs more clearly functions that the RBAC system should provide by using UML use case diagrams. For example, this view can be used to describe roles within a system and their permitted actions. Figure 3.2 illustrates that the actor *User* is permitted to establish sessions, request permission approval, and close sessions.
- **Dynamic:** the dynamic view uses UML collaboration diagrams to refine the use cases of the functional view. The goal is to show interactions among elements of the use case diagram. The collaboration diagram for *Permission Approval* is illustrated in Figure 3.3. It requires that a session should be created to activate roles before approving the permission.

Although UML models are investigated to represent the different aspects of RBAC, this approach does not address the validation of these models. Also, it omits the notion of protected resources in RBAC. As a result, there is no interaction between security and functional requirements.

In [4], Ahn et al. proposed to use UML and OCL to express RBAC constraints at the design stage. In this work, the basic entities of RBAC and their relations are represented using a UML class diagram (Figure 3.4), whereas the

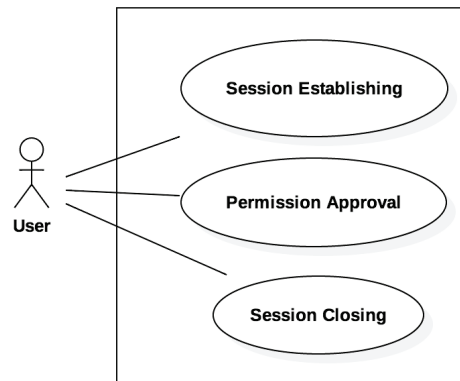


Figure 3.2: A functional model [3]

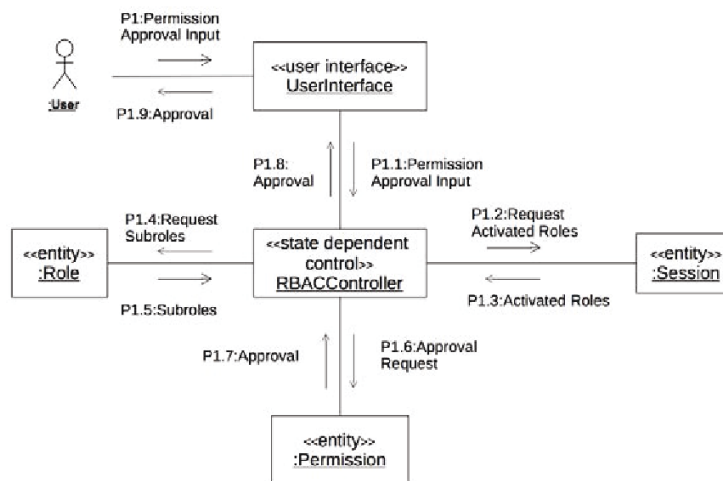


Figure 3.3: Collaboration Diagram: Permission Approval [3]

RBAC constraints are specified as OCL expressions. There are three types of constraints supported by this approach:

- **separation of duty constraints:** they ensure that conflicting roles cannot be assigned to the same user. For example, the following OCL expression makes sure that two conflicting roles *accounting* and *manager* are not assigned to the same user. To do so, this constraint identifies all the sets of mutually exclusive roles, checks all the roles assigned to each user, and enforces that the considered user cannot have more than one exclusive role.

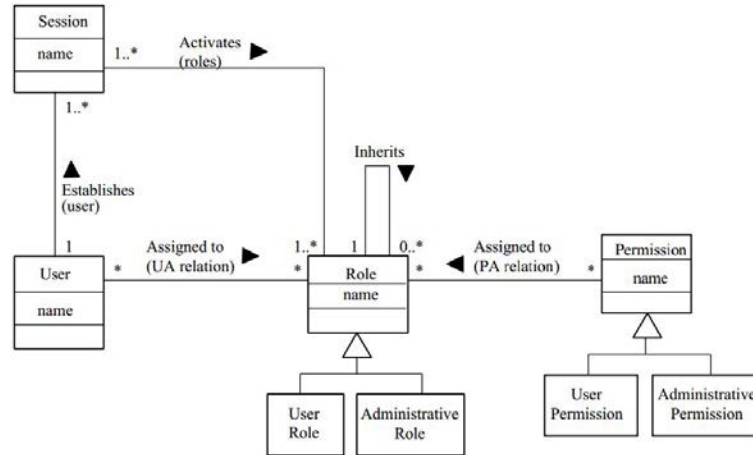


Figure 3.4: Conceptual class model for RBAC [4]

context *User* **inv:**

let $M : \text{Set} = \{\{accounting, manager\}, \dots\}$ **in**

$M \rightarrow \text{select}(m/\text{self.role} \rightarrow \text{intersection}(m) \rightarrow \text{size} > 1) \rightarrow \text{isEmpty}$

- **prerequisite constraints:** such constraints mean that a user can only play a certain role if he has already played another role. For example, a user can be a member of the *tester* role only if he is already a member of the *project_team* role. This constraint can be specified as follows:

context *User* **inv:**

$\text{self.role} \rightarrow \text{includes}(\text{'tester'})$ **implies**

$\text{self.role} \rightarrow \text{includes}(\text{'project_team'})$

- **cardinality constraints:** this kind of constraints defines a numerical limitation for roles. For example, the *chairman* role should be assigned to only one person in the organization. The OCL expression of this constraint is described as follows:

context *Role* **inv:**

$\text{self.user} \rightarrow \text{select}(u/\text{self.name} = \text{'chairman'}) \rightarrow \text{size} = 1$

In term of supporting tools, Ahn et al. also developed a tool, called RAE [47], for the validation of the RBAC model and constraints and the generation of a JAVA code from them. The validation component of the tool uses a set of system states and checks such states against authorization policies.

A pattern-based approach for modeling RBAC policies is introduced by Ray, Kim and their colleagues. [5,6]. They used UML diagram templates for defining reusable RBAC policies patterns. Figure 3.5 shows a class diagram template describing hierarchical RBAC with static and dynamic separation of duty constraints. The symbol (\square) is used to indicate parameters to be bound. Class templates represent essential RBAC concepts, namely *User*, *Role*, *Permission*, *Object*, *Operation*, and *Session*. Each class template is associated with attribute templates (e.g. \square *Name* : *String* of *Role*) and operation templates (e.g. \square *GrantPermission* of *Role*). An association template (e.g. \square *UserAssignment*) consists of the parameters indicating the association name and multiplicities. The RBAC model, i.e. the class diagram, of a specific application is obtained by instantiating the RBAC template with the values of the application. For example, Figure 3.6 shows the RBAC class diagram of a banking application. *BankRole*, *BankObject*, and *Transaction* are bound to the *Role*, *Object*, and *Operation* parameters respectively in the RBAC template diagram. RBAC constraints, such as separation of duty, prerequisite, and cardinality constraints are formalized in OCL. For example, the OCL expression formulating the SSD constraints is defined by:

```
//SSD constraints
context  $\square$ User inv:
self  $\square$ Role  $\rightarrow$  forAll(r1, r2 | r1  $\square$ SSD  $\rightarrow$  excludes(r2))
```

To gain a better understanding of RBAC constraint violations, they created UML object diagram templates. These patterns can be used to check the presence of constraint violations. Developers can instantiate these patterns as UML object diagrams to recognize violations of application-specific security constraints. For instance, static separation of duty policies prevent a user from being assigned to two conflicting roles. The template representing such constraints is shown in Figure 3.7. In the banking system, the following pairs of conflicting roles (*teller*, *accountant*), (*teller*, *loanOfficer*), (*loanOfficer*, *accountant*), (*loanOfficer*, *accountingManager*), (*customerServiceRep*, *accountingManager*) are visualized in Figure 3.8. The validation approach uses the violation patterns to identify policy conflicts, i.e. if the violation pattern occurs in an object diagram modeling the security policy of a particular application, then a conflict exists. However, checking the presence of a pattern within an object diagram requires to search subgraphs in an object diagram. This is known as the subgraph isomorphism problem that can be a difficult task [48].

In [7], Breu et al. presented the use of UML use case elements and OCL for specifying user rights. The main idea of the user rights models is that actors are

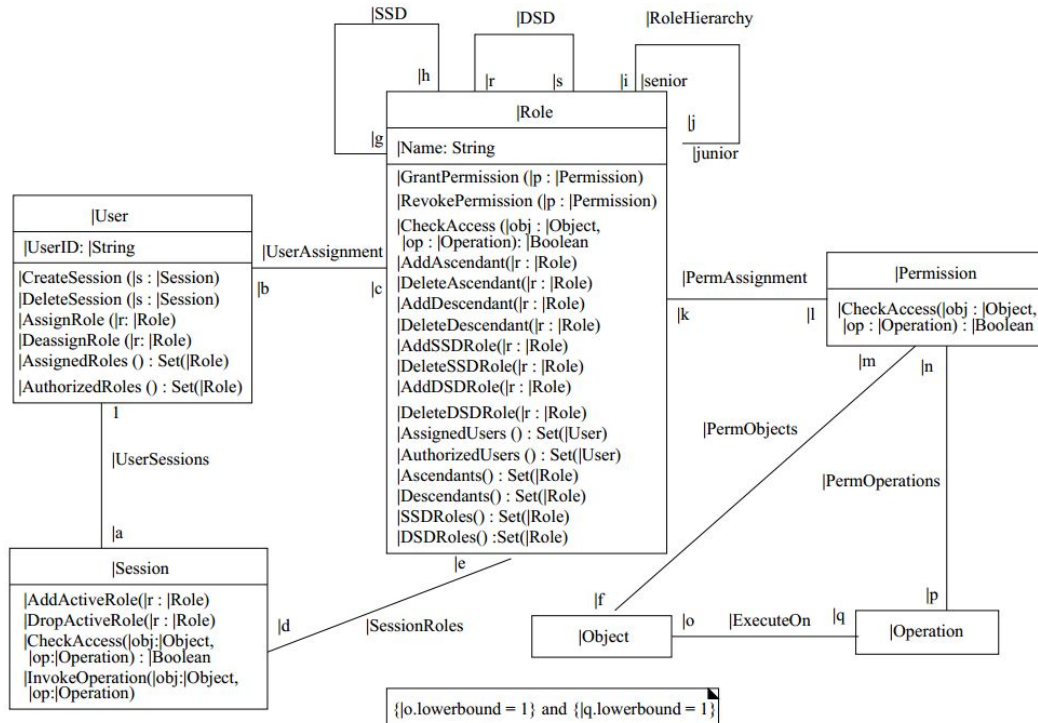


Figure 3.5: A RBAC class diagram template [5, 6]

assigned to permissions linked to objects of the class diagram (the functional model). Each use case diagram corresponds to a coherent interaction between a particular actor and the system. The basic concepts of the use case model are actors who interact with the system, use cases, and objects involved in the use cases. Actors in such a model stand for the roles that the system users play. For example, the use case diagram in Figure 3.9 depicts permissions of the *TeamWorker* role. The left hand of the model is the actor representing all the users who play the *TeamWorker* role, while the right hand includes objects appearing in the class diagram that the actor is authorized to access (the class diagram is omitted here for simplicity). In this work, OCL is used to formalize the use case model, i.e. the role-permission assignment. Considering the following OCL operation that allows a team worker to read his own accountings through the *getAccountingdate()* method. Precisely, the *ACTeamWorker* actor is mapped to the *Accounting* class. The associated predicate verifies that the actor who calls *getAccountingdate()* should be a team worker.

```

context Accounting :: getAccountingDate()
perm (act : ACTeamWorker) : self.user = act.map()

```

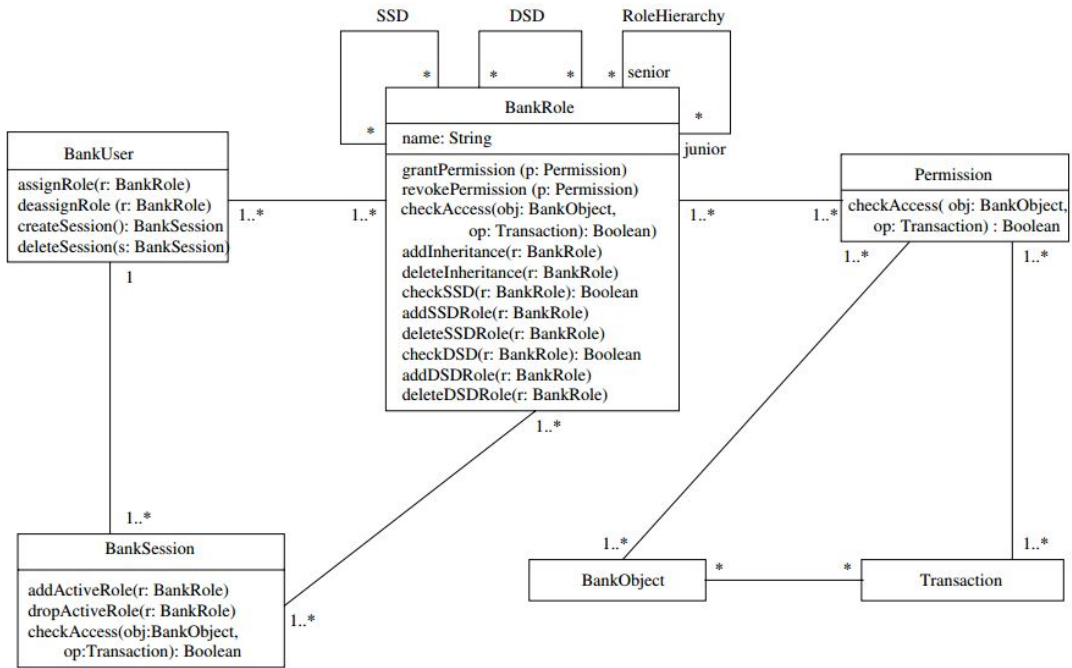



Figure 3.6: A RBAC class diagram for a banking system [6]

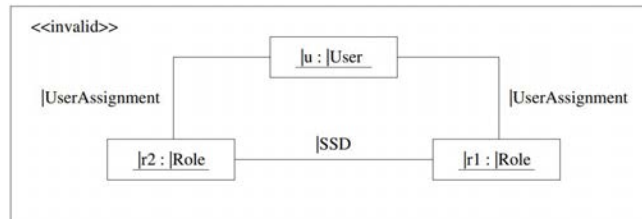


Figure 3.7: Violation of SSD Constraint [5, 6]

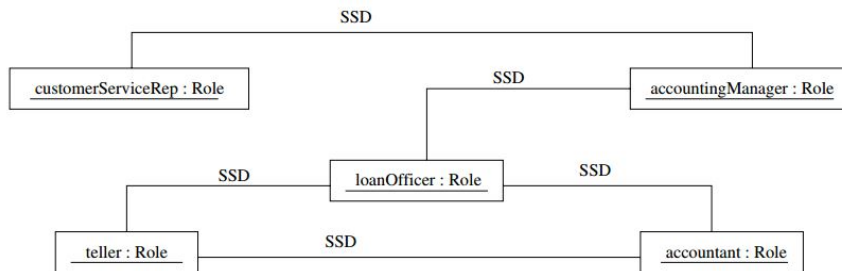


Figure 3.8: Object Diagram for SSD Policies [6]



Figure 3.9: Use case: TeamWorker [7]

To enhance the manipulation of OCL for specifying and validating RBAC extensions, Sohr et al. [49] investigated the USE system [50]. In their work, authorization constraints, e.g. static SoD and prerequisite roles restrictions, are formulated in OCL. Let's consider an example of RBAC constraint as follows:

context *User* **inv** *Prerequisite Role*:

self.role \rightarrow includes(*engineer*) **implies** **self**.role \rightarrow includes(*employee*)

This OCL expression describes a prerequisite constraint stating that a user can be assigned to the *engineer* role only if he/she is already assigned to the *employee* role. Due to the fact that OCL expressions consider only the current system states, it is unsuitable to represent constraints that refer to distinct instants of time. Therefore, the authors employed TOCL (Temporal OCL) [51], which is an OCL extension with temporal elements, for specifying history-based authorization constraints. Dynamic object-based SoD is an example of such constraints. It states that a user is not allowed to execute an action acting upon an object if he/she has previously performed a certain action acting upon the same object. To specify this constraint in TOCL, they used two predicates introduced in [52], namely *auth(u,op,obj)* and *exec(u,op,obj)*. The *auth* predicate means that the user *u* is permitted to performed the operation *op* on the object *obj*, while the *exec* predicate means that the user *u* executes the operation *op* on the object *obj* at the present state. The following TOCL specification expresses object-based dynamic SoD constraints.

context *Object* **inv** *ObjDSoD*:

Operation.allInstances \rightarrow forAll(*op,op1* | *User*.allInstances \rightarrow forAll(*u* /
(*Exec(u,op,self)* **and** *op1* <> *op*) **implies** *always not*
Auth(u,op1,self)))

Finally, this paper demonstrated how to exercise the USE tool to validate and test access control policies expressed in UML and OCL. Using this tool, a policy designer can detect conflicting and missing authorization constraints.

Basin et al. [8,53] introduced a MDA-based approach for the modeling of se-

curity requirements and the generation of security infrastructures, called Model Driven Security (MDS) (Figure 3.10). They combined a UML-based design language (e.g. ComponentUML) with a security modeling language (i.e. SecureUML [43]) by defining a *dialect*. The role of the dialect is to identify particular elements of the design language as the protected resources of the security language. Such a combined model is also the base of model transformations that produce an access control infrastructure. Analyzing access control decisions is performed upon the combined model through OCL queries. These queries formalize questions about the relationships between users, roles, permissions, and actions. For example, the operation `User::allAllowedActions():Set(Action)` returns the collection of actions that are permitted for the given user.

context `User::allAllowedActions():Set(Action)` *body:*
self.hasrole.allPermissions().allActions()->asSet()

The analysis process is automated by using a tool called SecureMOVA [2] (more details in Sect. 3.3.1).

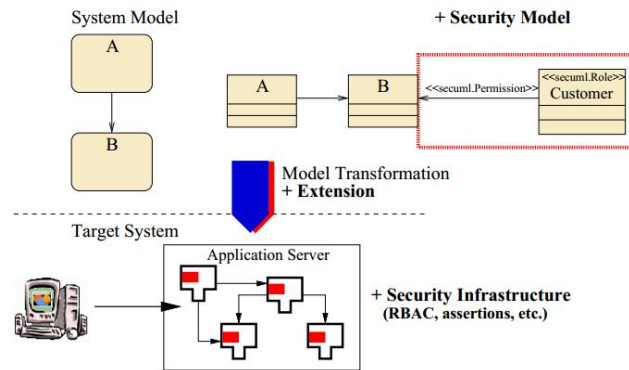


Figure 3.10: Model Driven Security [8]

3.2.2 Alloy-based Approaches

Alloy is a textual modeling language based on first-order logic. It is 1) a lightweight language and yet capable to express a useful range of structural properties, 2) precise enough to formally express complex constraints, and 3) amenable to a fully automatic semantic analysis [22]. In the following, we present a list of the existing works that apply Alloy to specify RBAC requirements.

Schaad et al. [54] introduced a methodology dealing with RBAC extensions. In particular, they described the transformation of different access control variants into Alloy, including:

- **from the RBAC96 model [18] specifying access control policies to Alloy:** the initial structure of the RBAC96 model can be easily specified in Alloy. Essentially, the *domain* paragraph describes RBAC objects with the *fixed* keyword indicating that they are drawn from a specified pool of objects. The *state* paragraph describes the relations between these objects. If the cardinality of the relation is not specified, then it is automatically assumed to be zero or more, otherwise the (!) and (+) symbols indicate cardinalities of one and one or more respectively.

```

model RBAC96{
  domain {fixed User, fixed Role, fixed Session, fixed
    Permission}
  state {
    ur_assignment: User -> Role
    rp_assignment: Role -> Permission
    us_assignment: User! -> Session
    sr_assignment: Session -> Role+
    rr_hierarchy: Role -> Role
  }
}

```

- **specifying SoD constraints in Alloy:** to enforce SoD constraints, they expanded the (*model RBAC96*) specification by adding the following new relation *r_exclusive*. This relation defines pairs of conflicting roles.

```
r_exclusive: Role -> Role
```

The static and dynamic SoD invariants verify that all pairs of conflicting roles are not assigned to the same user or the same user within a session respectively.

```

//Static SoD Invariant:
inv static_sod {all r1, r2:Role | r1 in r2.r_exclusive
  ->no(r1.~ur_assignment & r2.~ur_assignment)}
//Dynamic SoD Invariant:
inv dynamic_sod {all r1, r2:Role | r1 in r2.r_exclusive
  ->no(r1.~sr_assignment & r2.~sr_assignment)}

```

By using the Alloy Analyzer, it is able to automatically check conflicts that may arise after executing the administrative operations with respect to separation of duty constraints. Yet, the validity of the RBAC models is not addressed. The *Operation* and *Object* concepts of RBAC are also ignored in this work.

In [55], Zao et al. reported the use of Alloy to analyze the internal consistency of their RBAC schema and verify the correctness of its implementation. Similar to Schaad et al. in [54], they also defined two paragraphs in the Alloy specification of RBAC96: the *domain* paragraph specifies RBAC entities, and the *state* paragraph describes the relations between entities. This specification also includes formulas expressing RBAC constraints, such as SoD (the *conflictRoleRule* invariant).

```

model rbac96 {
  domain {Users, Roles, fixed Operations, Objects, Sessions}
  state {
    userRole : Users -> Roles
    permissions : Operations -> Objects
    rolePermis [Roles]: Operations -> Objects
    userRoleExt: Users -> Roles
    objectOprToRole[Objects]: Operations -> Roles
    objectOprToRoleExt [Objects]: Operations -> Roles
    roleObjToOperation[Roles]: Objects -> Operations
    conflictRoles : Roles -> Roles //Role-centric SoD
    conflictObjects : Objects -> Objects //Permissions-centric SoD
    conflictUsers : Users -> Users //User-centric SoD
    inherits : Roles -> Roles + //Role inherits
  }
  //SoD Rule: User-role conflicts
  // - Enforces "inherited conflicts "
  // ex. r1 conflicts r2 (no user can be both r1, r2)
  // r3 >= r2 -> r1 conflicts r3
  inv conflictRoleRule {
    no r | r in r. conflictRoles // irreflexive
    all u | no r1, r2, r3 |
    r1 in u.userRole &&
    r1 in r2. conflictRoles &&
    r2 in r3. inherits &&
    r3 in u.userRole
  }...
}

```

This work provided a more complete RBAC specification than that defined in [54]. In addition, it demonstrated the capacity of Alloy in (1) verifying the correctness of different RBAC implementations, (2) checking the consistency among entities, relations and constraints of the RBAC schema, and (3) searching for a plausible instance of the schema. These verifications are done using the Alloy Constraint Analyzer.

Hu and Ahn [9] presented a framework, so-called Assurance Management Framework (AMF) (Figure 3.11), for the formal verification and the test cases generation of RBAC models. In AMF, core RBAC and constraints are formalized in Alloy. Essentially, basic RBAC elements and relations are defined as sets, including a set of roles, a set of users, a set of permissions, a set of user-role, a set of role-permissions, etc. The primary representation of the RBAC model in Alloy is defined as follows:

```

module RBAC
  sig User {}
  sig Role {}
  sig Operation {}
  sig Object {}
  sig Permission {Operation, Object}
  sig Session {}
  sig URA {
    ura: User->Role}
  sig PRA {
    pra: Permission->Role}
  sig US {
    us: User!->Session}
  sig SR {
    sr: Session->Role}
  sig PB {
    pb: Operation->Object}

```

The framework also supports SoD constraints in the context of conflicting roles. The following Alloy definitions express these constraints, where *conflict_role* is a set of roles conflicting to each others, and *cardinality* is the maximum number of roles in the exclusive roles set that a user can be assigned.

```

sig SCR {
  conflict_role : set Role,
  cardinality : Int}

```

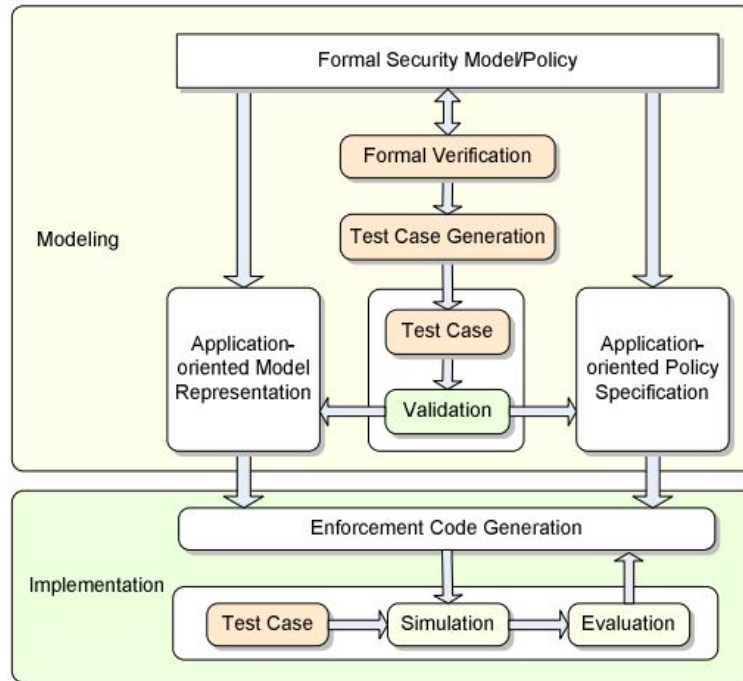


Figure 3.11: Assurance Management Framework [9]

By running the Alloy Analyzer, one can verify the compliance of the access control specifications with respect to the selected security properties. The conformance testing is then automatically derived from the security formal specification. The obtained test cases are used to compare the actual system implementation to the expected result derived from the formal specification. In addition to the above features, the framework also supports the automatic generation of the security enforcement code in the Java language using the RAE tool [47].

In their work [10], Toahchoodee et al. focused on the transformation of a UML class diagram and its OCL specification into an Alloy model. Figure 3.12 sketches the transformation function proposed in this approach. They utilized a class diagram to merge the functional and security models of a system. Classes of this combined model are simplified by removing attributes that are not relevant to security reasoning. Furthermore, each class can be constrained by location and time. A class is mapped to an object declaration in the *sig* structure in Alloy. The *sig User*, *sig Role*, *sig Permission*, *sig Task* (denotes secure operations), *sig Location* and *sig Time* are the basic signatures of the model. Each *Property* of a class is translated into a field of the mapped signa-

ture. For example, the *Role* class has a property (*location: Location*), which is then converted to the field (*location: one Location*) of *sig Role* (i.e. *sig Role {location: one Location,...}*). The permission-role assignment and the static SoD constraints realized in OCL expressions are transformed into *Predicates* in Alloy. The following describes the Alloy predicate of a SoD constraint. It states that a user who plays the roles (*r1* and *r2*) is not allowed to execute two conflicting tasks (*ELEVEN* and *FIFTEEN*).

```

pred SoD[self: User]{
  all r1, r2: self.roles |
  ((ELEVEN in r1.tasks) => (FIFTEEN !in r2.tasks)) &&
  ((FIFTEEN in r1.tasks) => (ELEVEN !in r2.tasks))}

```

Regarding the automation of the transformation, the authors made use of the UML2Alloy tool [56] for automatically converting a class diagram and its OCL specification into an Alloy model, which is subsequently analyzed by the Alloy Analyzer.

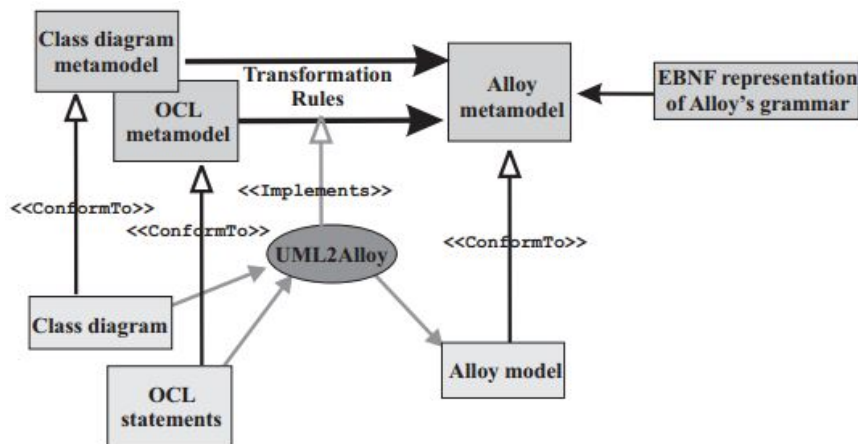


Figure 3.12: The transformation method [10]

3.2.3 Z-based Approaches

Boswell [57] demonstrated the use of Z to develop a security policy model for NATO AIR Command and Control System. Nonetheless, this work covers only the mandatory and discretionary access controls, but it does not support the RBAC model. On the other hand, the validation of the security model is performed manually to produce informal arguments which could ensure a suitable

level of confidence. In [58], Morimoto et al. proposed a Z-based technique for specifying and verifying security properties with respect to the ISO/IEC 15408 common criteria. They also developed a verification process for security properties by using the theorem prover Z/EVES. The process consists of four steps: 1) select the formalized criteria (defined in the ISO/IEC 15408 common criteria) required in a target system; 2) formalize the specifications of the target system in Z; 3) instantiate the selected criteria templates; 4) verify the formalized specifications against the instantiated criteria. Using this process, it is possible to check whether the specifications meet the security criteria of the Common Criteria.

In [59], Abdallah and Khayat attempted to specify a variety of state-based Flat RBAC models in Z. The formalization starts by specifying a core RBAC model. The state of the core RBAC model comprises a set of principals (*Principals*), a set of roles (*Roles*), and a set of valid tasks (*Tasks*). This model also includes two relations: *PrincipalRoles* that assigns roles to principals; and *RolePermissions* that relates tasks to roles. The whole model can be described in Z using the following scheme:

Core_RBAC

Roles : \mathbb{P} *ROLE*

Principals : \mathbb{P} *PRINCIPAL*

Tasks : \mathbb{P} *TASK*

PrincipalRoles : *PRINCIPAL* \leftrightarrow *ROLE*

RolePermissions : *ROLE* \leftrightarrow *TASK*

Principals \subseteq dom *PrincipalRoles*

ran *PrincipalRoles* \subseteq *Roles*

Roles \subseteq dom *RolePermissions*

ran *RolePermissions* \subseteq *Tasks*

Then, the core RBAC model is refined to the Flat RBAC model by adding two additional components: *Operations* that represent a set of valid operations, and *Objects* that denote a set of protected objects. The goal is to refine the type *TASK* into a cross product of two other types *OPERATION* and *OBJECT* ($TASK == OPERATION \times OBJECT$). Hence, a task is defined as a pair of an operation and an object. The expanded Flat RBAC model can be formalized as follows:

<i>Flat_RBAC</i> <i>Core_RBAC</i> <i>Objects</i> : \mathbb{P} <i>OBJECT</i> <i>Operations</i> : \mathbb{P} <i>OPERATION</i>
$\text{dom } \textit{Tasks} \subseteq \textit{Operations}$ $\text{ran } \textit{Tasks} \subseteq \textit{Objects}$

The refinement may include constraints involving the cardinality and separation of duty. For this purpose, they defined a new relation named *ActiveRole* representing the current role of a principal. To activate a role for a principal, the role must be already assigned to the principal. That means that this pair is a value of *PrincipalRoles*. The specification of *ActiveRole* is defined by:

<i>ActiveRole</i> <i>Core_RBAC</i> <i>ActiveRole</i> : <i>PRINCIPAL</i> \leftrightarrow <i>ROLE</i>
$\textit{ActiveRole} \subseteq \textit{PrincipalRoles}$

A SoD constraint prohibits a principal to assume two conflicting roles at the same time. This kind of constraint is defined by the *ConflictRoles* declaration storing pairs of mutual exclusive roles, and a predicate ensuring that the active roles set of any principal does not contain conflicting roles.

<i>SoD</i> <i>ActiveRole</i> <i>ConflictRoles</i> : <i>ROLE</i> \leftrightarrow <i>ROLE</i> $\forall p : \textit{Principals} \bullet$ $(\textit{ActiveRoles}(p) \times \textit{ActiveRoles}(p)) \cap \textit{ConflictRoles} = \emptyset$
--

Although this approach offers a formal representation of RBAC variants, it does not address the validation and verification of the RBAC model.

The work in [60] coped with the specification and verification of a state-based RBAC model. It used the Z language to express the RBAC state model. This model is a tuple $[USERS, ROLES, OPS, OBJS, SESSIONS, UA, PA, RH, CC]$, where *USERS*, *ROLES*, *OPS*, *OBJS*, *SESSIONS* are the set of users, roles, operations, objects and sessions respectively; *UA* and *PA* are the assignment relations of users to roles and permissions to roles respectively; *RH* is the role hierarchy relation; *CC* is a set of constraints. Essentially, the basic RBAC

concepts, namely users, roles, operations, objects, permissions, and sessions are defined as power sets in Z . The relationships between these elements are specified as relations. For instance, the user-role assignment permits to assign a set of roles to a user. Moreover, the Z specification of RBAC also contains four functions that express the following RBAC constraints: (1) the $pmutex(p)$ function represents all the permissions conflicting to the given permission p ; (2) $RSSoD(r)$ and (3) $RDSO(r)$ identify a set of roles in which each role has respectively a static/dynamic mutual exclusive roles relationship with a given role r ; (4) the $Cardinality(r)$ function restricts the maximum number of users being assigned to a given role r . The RBAC elements, relations between the elements, and constraints are included in the RBAC schema.

RBAC

Users : $\mathbb{P} \text{USERS}$
Roles : $\mathbb{P} \text{ROLES}$
Ops : $\mathbb{P} \text{OPS}$
Objs : $\mathbb{P} \text{OBJS}$
Perms : $\mathbb{P} \text{PERMS}$
Sessions : $\mathbb{P} \text{SESSIONS}$
assigned_roles : $\text{USERS} \rightarrow (\mathbb{P} \text{ROLES})$
authorized_roles : $\text{USERS} \rightarrow (\mathbb{P} \text{ROLES})$
assigned_perms : $\text{ROLES} \rightarrow (\mathbb{P} \text{PERMS})$
assigned_users : $\text{ROLES} \rightarrow (\mathbb{P} \text{USERS})$
user_sessions : $\text{USER} \rightarrow (\mathbb{P} \text{SESSIONS})$
pmutex : $\text{PERMS} \rightarrow \mathbb{P} \text{PERMS}$
RSSoD : $\text{ROLES} \rightarrow \mathbb{P} \text{ROLES}$
RDSO : $\text{ROLES} \rightarrow \mathbb{P} \text{ROLES}$
Cardinality : $\text{ROLES} \rightarrow N$

Based upon well-constructed security theorems, their verification guarantees the consistency of the RBAC model, i.e. it makes sure that the system always remains in the secure states. This process is automated by using the Z/EVES theorem prover.

In [25], Qamar et al. presented a contribution in specifying and validating security-design models based on the Z notation. Their proposal took into account both functional and security requirements. The functional specification is automatically generated through the translation from the UML class diagram to Z by using their developed tool, called RoZ [61]. The security aspect is based upon a reusable security kernel which defines RBAC elements in Z . In this work, the concept of role represents job titles in the modeled system like doctors and

patients; users are individuals using the system, and they connect to the system by establishing sessions based on their identities; permissions are a list of operations introduced in the class diagram, e.g. *readMedicalRecord* and *updateDoctor*; such operations are classified into different types of abstract actions, such as *EntityRead* and *EntityUpdate*; these operations affect on protected resources which are classes in the class diagram; The presented RBAC basics are introduced in Z as the enumerated sets. The values of these sets are extracted from the SecureUML diagram of a particular application to instantiate the security kernel.

RBACSets

```

role :  $\mathbb{F}$  ROLE
user :  $\mathbb{F}$  USER
uid :  $\mathbb{F}$  USERID
permission :  $\mathbb{F}$  PERMISSION
abs_action :  $\mathbb{F}$  ABS_ACTION
atm_action :  $\mathbb{F}$  ATM_ACTION
resource :  $\mathbb{F}$  RESOURCE

```

In addition to the essential concepts, the security kernel also specifies their relationships through functions. For example, the permission assignment *perm_Asmt* is defined as a relation between (user identity, user, role) and (permissions, atomic actions, resources). The *session_User* function links a session to a user who activates a set of roles during the session by using the *session_Role* function.

Relations

```

perm_Asmt : (USERID  $\times$  USER  $\times$  ROLE)  $\leftrightarrow$ 
            (PERMISSION  $\times$  ATM_ACTION  $\times$  RESOURCE)
session_User : SESSION  $\rightarrow$  USER
session_Role : ROLE  $\leftrightarrow$  SESSION
...

```

The *SecureOperation* operation checks whether a user identified by his id, acting with a given role during a given session, is allowed to perform a given action on a resource. *SecureOperation* is then included in the secure operations of the functional model as a precondition before their actual execution.

SecureOperation

session? : *SESSION*

resource? : *RESOURCE*

atm_action? : *ATM_ACTION*

role? : *ROLE*

user? : *USER*

uid? : *USERID*

permission? : *PERMISSION*

$(session?, user?) \in sessio_User$

$(role?, session?) \in session_Role$

$((uid?, user?, role?), (permission?, atm_action?, resource?)) \in perm_Asmt$

Using the Jaza tool [62], the validation is performed by animating the model. The animation of the specifications is able to check the execution right of the user who requests an action. Such a process is based on the evaluation of queries about the access control rules and the animation of user-defined scenarios.

3.2.4 A B-based Approach

Despite the fact that the B method is useful for formalizing safety-critical systems, and there are a number of commercial supporting tools, the use of B for the security of information systems has not been notably investigated. The work of Milhau et al. [11] is one of the B-based approaches for formalizing access control requirements that strongly inspires our proposal in the thesis.

Figure 3.13 gives an overview of the access control filter proposed in [11]. They started by visually representing functional and security requirements using UML-based languages. In particular, they introduced the functional aspect of an information system with a UML class diagram, static access control policies using SecureUML diagrams, and dynamic access control constraints with ASTD [63]. The graphical models are then translated into a set of B components. Finally, they defined an access control filter that combines different B components in order to make the final access control decisions.

Here, we focus on the use of ASTD for modeling dynamic constraints, while the representation of the functional and basic authorization requirements using a class diagram and SecureUML respectively will be presented in Chapter 5. ASTD extends Harel's Statecharts [64] by using EB^3 operators [65] to specify a sequence of actions in information systems. For example, in a hospital system, we could have an access control rule such as *"If a patient has left the hospital, only doctors belonging to the hospital during the patient's stay will keep read*

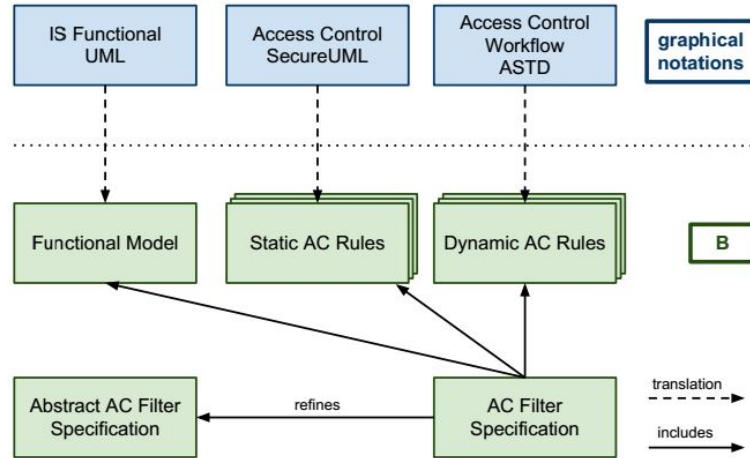


Figure 3.13: An access control filter [11]

access to his medical record". This rule considers a set of ordered actions, such as $JoinHospital(d, h)$ describing that a doctor is assigned to a hospital, $leaveHospital(d, h)$ representing that a doctor leaves the hospital, etc. In [11], the authors used the ASTD notation to specify this kind of rule (Figure 3.14). Each action of the ASTD model is represented by a notation $\langle \vec{s}, a(\vec{p}) \rangle$, where \vec{s} is a list of parameters (e.g. user and role), a is an action of the system, and \vec{p} is its parameters. As we can notice, some actions define \vec{s} through wildcards ($-$), while others state specific values of \vec{s} . The wildcard represents that the specification accepts any values of \vec{s} . The ASTD model also uses EB^3 operators. For example, the quantified choice operator ($| h : Hospital$) indicates that the hospital h of the class *Hospital* is associated to the doctor d of the class *Doctor*. The action $JoinHospital(d, h)$ assigns a doctor to a hospital when the doctor joins the hospital, whereas the action $LeaveHospital(d, h)$ removes the link of d and h when the doctor leaves the hospital. The Kleene closure operator (denoted by $*$) allows the iteration of the sub-ASTD. That means that after leaving a hospital, a doctor can join another one, creating a new link between d and h (see Section 3.2 in [11]).

Afterwards, the graphical models, i.e. class diagram, SecureUML, and ASTD, are translated into B specifications. This work applied the separation of concerns methodology. It specified the functional, static security, and dynamic security parts in isolation.

- A single B machine is created for the functional model. It contains the data denoted by sets, system states denoted by variables, and states evolution denoted by operations. This B component is the formal representation

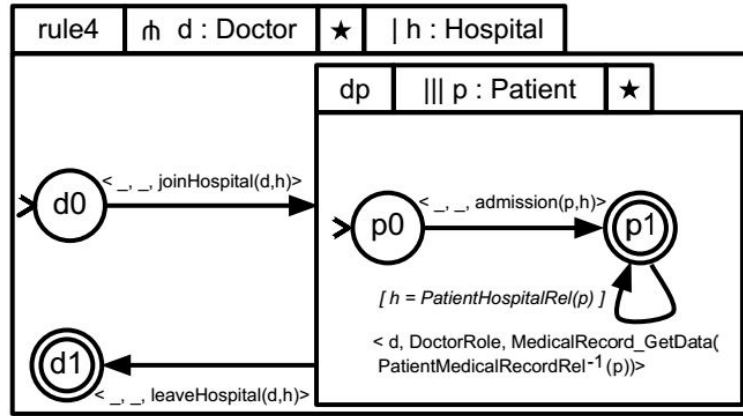


Figure 3.14: A dynamic access control rule in ASTD [11]

of the UML class diagram. The translation rules from the UML class diagram into B are adapted from several existing works [28, 30, 66]. Let's take a simple library system as an example. Assuming that there is a class *Book* with an attribute *series* denoting the id of a book and two operations: *borrow* and *return*. The functional B specification of this example is defined as follows:

MACHINE

Library

SETS

BOOK

VARIABLES

book, series

INVARIANT

$book \subseteq BOOK \wedge series \in book \rightsquigarrow NAT$

INITIALISATION

$book := \emptyset \parallel series := \emptyset$

OPERATIONS

borrow = ...

return = ...

- The second B machine formalizing the static access control policies is the result of the translation of the SecureUML model. It represents the SecureUML elements as enumerated sets. For example, *Jack* and *Member* represent respectively a user and a role in the library system. The static

security specification of the library example is defined by:

SETS

$USERS = \{Jack, \dots\};$
 $ROLES = \{Member, \dots\};$
 $PERMISSIONS = \{MemberPermission, \dots\};$
 $ACTIONS = \{borrow, return, \dots\};$
 $ENTITIES = \{Book, \dots\}$

...

The relations between SecureUML elements are also introduced in the static security specification. For instance, *roleOf* returns a set of roles that a user can play, and *isPermitted* computes the set of authorized functional operations for each role.

$$roleOf \in USERS \rightarrow \mathbb{P}(ROLES)$$

$$isPermitted \in ROLES \leftrightarrow ACTIONS$$

This machine also contains a secure operation for each functional operation needed to be protected. A secure operation ensures that only authorized users can call the referred functional operation (Figure 3.15). It adds two parameters *user* and *role* corresponding respectively to the user who is trying to invoke the operation and one of his roles ($role \in roleOf(user)$). The predicate $operation \in isPermitted[\{role\}]$ verifies whether *operation* is granted to *user* acting on *role*. Such a secure operation returns a value that grants or denies the invocation.

- The dynamic security component includes operations that constrain the functional operations with dynamic security properties. In [12], the authors focused on history-based constraints. Such constraints can refer to different system states. Let's go back once again to the example in Figure 3.14. Figure 3.16 describes the B operation for the action *Dynamic_MedicalRecord_GetData*. Conditions $user \in Doctor$ and $role = DoctorRole$ check whether the user *user* who is trying to perform *MedicalRecord_GetData* is a doctor and is assigned to the *DoctorRole* role. The condition $StateQChoice(user) = PatientHospitalRel(pp)$ makes sure that the doctor *user* already joined the hospital where the patient *pp* stays. The condition $StateAutomaton_DoctorHospital(user) = dp$ verifies that the doctor *user* is still working in the hospital by checking that the ASTD remains in the state *dp*. The condition $StateAutomaton_dp(user, pp) = p1$ ensures that the patient *pp* admitted to the hospital after the joining of the


```

answer ← secure_operation(..., user, role) ≐
PRE
  ...
  user ∈ USERS ∧
  role ∈ ROLES
THEN
  IF
    role ∈ roleOf(user) ∧
    operation ∈ isPermitted[{role}] ∧
    ...
  THEN
    answer := granted
  ELSE
    answer := denied
  END
END

```

Figure 3.15: A secure operation

doctor *user*. *StateAutomaton_dp* and *StateAutomaton_DoctorHospital* are partial functions representing the state of the inner and upper automaton respectively, where *dp* is a place.

The separate development method allows to avoid tangling different concerns of an application, thus developers and security engineers can focus on their own tasks. In the end, the presented separate components are put together in an access control filter. The goal is to check security requirements before the actual execution of the secure operations. This weaver tailors the functional logic and its associated access control rules, including both static and dynamic constraints, in the same language. Figure 3.17 describes an access control decision algorithm proposed in [12]. Essentially, The filter grants the execution of a secure operation only if both static and dynamic access control grant it.

3.2.5 Discussion

In this section, we surveyed the existing works on specifying access control policies using semi-formal (UML+OCL) and formal languages. The formal techniques supporting the verification and validation of access control specifications are based on Z, Alloy, and B. Table 3.1 summarizes our analysis of these techniques.

```

answer ← Dynamic_MedicalRecord_GetData(Instance, user, role) $\hat{=}$ 
PRE
  Instance ∈ MedicalRecord ∧
  user ∈ USERS ∧
  role ∈ ROLES
THEN
  IF
    user ∈ Doctor ∧
    role = DoctorRole ∧
    StateQChoice(user) = PatientHospitalRel(pp) ∧
    StateAutomaton_DoctorHospital(user) = dp ∧
    StateAutomaton_dp(user, pp) = p1
  THEN
    StateAutomaton_dp(user, pp) := p1 ||
    answer := granted
  ELSE
    answer := denied
  END
END

```

Figure 3.16: A dynamic operation [12]

```

answer, executed ← Filter_operation(..., user, role) $\hat{=}$ 
VAR static, dynamic, functional IN
  static ← Secure_operation(..., user, role);
  IF static = granted THEN
    dynamic ← Dynamic_operation(..., user, role);
    IF dynamic = denied THEN
      answer := denied; executed := NotExecuted
    ELSE
      functional ← operation(...);
      IF functional = ok THEN
        answer := granted; executed := ok
      ELSE
        answer := granted; executed := NotExecuted
      END;
    ELSE
      answer := denied; executed := NotExecuted
    END
  END
END

```

Figure 3.17: An access control filter [12]

Combining UML and OCL can, on the one hand, graphically model security requirements and, on the other hand, analyze various security properties of a secure system. However, the literature review shows that the automated analysis of security models has not been widely investigated. SecureMOVA and USE are major contributions on modeling access control policies in UML diagrams and automatically analyzing RBAC models realized in OCL.

Models written in Alloy can be automatically analyzed using the Alloy Analyzer [67]. However, the discussed solutions mainly focus on the static aspect of RBAC. Dynamic access control policies are partially covered in some proposals. Moreover, since the verification techniques of Alloy are based on model-checking, it is not efficient for analyzing large models. The explosion of the system state space is the main issue of the model-checking techniques that has to be tackled.

In contrast, the solutions based on the B method can cope with the problem of the state space explosion thanks to a theorem proving technique. The work of Milhau et al. [11] based on the B method covers various types of access control requirements, including static and dynamic rules. These requirements also involve the functional aspect of the application. Nonetheless, the use of ASTD only for visualizing security requirements may be too complicated. We believe that the visual representations of a system should be easily understandable/readable to disburden the communications among participants, i.e. the designer, the developer, and the end-user.

The other formal techniques reviewed in this section are based on the Z notation. Z has been mostly used to write precise application specifications. One of the major drawback in using Z is the lack of tools supporting the analysis of the formalized model.

3.3 Support Tools for Access Control Policies

In this section, we review tools supporting the modeling and model transformations of access control policies. The first tool is SecureMOVA that allows to model and validate the combination of SecureUML and ComponentUML models. The other tool is B4MSecure that provides a mean to transform UML-based models to formal specifications.

3.3.1 SecureMOVA

Developed by Basin et al. [2], SecureMOVA is a modeling and analysis tool for *security-design models*. This kind of model is a composition of SecureUML and ComponentUML: SecureUML is a modeling language for RBAC policies;

Table 3.1: Synthesis of formal-based approaches for security specifications

Studied solutions	Authorization	SSD	DSD	V/V	w.r.t functional logic	Tools
<i>Alloy-based approaches</i>						
Schaad et al. [54]	Y	Y	Y	N		Alloy Analyzer
Zao et al. [55]	Y	Y	N	N		Alloy Analyzer
Ahn, Hu et al. [9,47]	Y	Y	Y	Y (via test cases)		Alloy Analyzer
Toahchoodee et al. [10]	Y	Y	N	Y		Alloy Analyzer
<i>Z-based approaches</i>						
Yuan et al. [60]	Y	Y	Y	N		
Morimoto et al. [58]	Y	Y	Y	N		Z/EVES
Qamar et al. [25]	Y	N	N	Y		Jaza
<i>B-based approaches</i>						
Milhau et al. [11]	Y	Y	Y	Y		AtelierB, ProB

Y = Yes, N = No, V/V = Validation/Verification, w.r.t = with respect to, SSD = Static Separation of Duty, DSD = Dynamic Separation of Duty

whereas, ComponentUML is used for modeling component-based systems that contains a subset of UML class elements (i.e. entities, associations, and attributes/methods of entities). SecureMOVA provides facilities for modeling functional requirements in UML class diagrams, formalizing different access control information in OCL, and evaluating OCL queries on a given model. These queries concern the relationships between users, roles, permissions, and actions. They can refer to elements of the functional model.

SecureMOVA allows an automated analysis of security-design models. In particular, it supports reasoning authorization constraints in a security scenario (a system state) involving entities that take part in the system model. To this end, the authors implemented a *dialect* metamodel which merges the SecureUML metamodel with the ComponentUML model. Hence, one is capable to write and analyze OCL queries on the snapshots of this combined metamodel.

Using SecureMOVA one can ask questions about the basic authorization constraints. The following is a list of such queries:

- Given a role, what are the atomic actions that a user playing this role can perform?
- Given an atomic action, which roles can perform this action?
- Given a role and an atomic action, under which circumstances a user playing this role can perform this action?
- Are there two roles with the same set of atomic actions?
- Given an atomic action, which roles allow the least set of actions, including the atomic action?
- Do two permissions overlap?
- Are there overlapping permissions for different roles?
- Are there atomic actions that every role, except the default role, may perform?

SecureMOVA also allows to analyze access control decisions on a snapshot (an object diagram) of the ComponentUML model. In other words, such constraints query about the actions that a given user or a given role are permitted to perform at a given system state (so-called *security scenario*). Some examples of this kind of queries are listed below:

- Given an action on a concrete resource, which roles are to be assigned to a given user to be able her to perform the action in the given scenario?
- Are there actions on the concrete resources that every user can perform in the given scenario?

Notwithstanding, all the reported access control decisions that SecureMOVA supports depend on the static access control information, i.e. user-role and role-permission assignments. This tool does not address dynamic security constraints. The analysis is executed merely on snapshots.

3.3.2 B4MSecure

Existing works in the use of formal methods for security mainly focus on the verification of security policies without addressing the functional aspect of an application [54, 55, 58]. Idani et al. [13] have made a further step by taking into account both functional and access control models (Figure 3.18), which are then formally validated and verified. They introduced the B4MSecure platform that allows the translation of the functional model (i.e. UML class diagram) and its associated security model (i.e. SecureUML diagram) into formal B specifications. This subsection gives an overview of the B4MSecure tool. The architecture of the tool is shown in Figure 3.19.

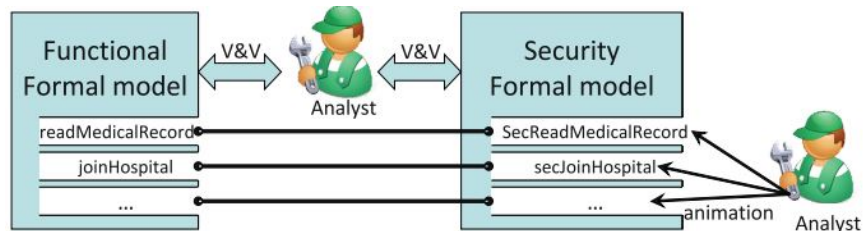


Figure 3.18: Validation/Verification activities supported by B4MSecure [13]

With the B4MSecure tool, one can present the functional logic of an application using a UML class diagram and access control policies with SecureUML models. SecureUML is a UML profile that includes RBAC elements such as user, role and permission. The foundation of B4MSecure’s modeling function is the Eclipse Topcased environment. On this editing environment, it is possible to manually define B invariants and preconditions for operations. That is done by annotating the operations of the functional model with additional B preconditions and substitutions. This function bridges the gap between B and the modeling language.

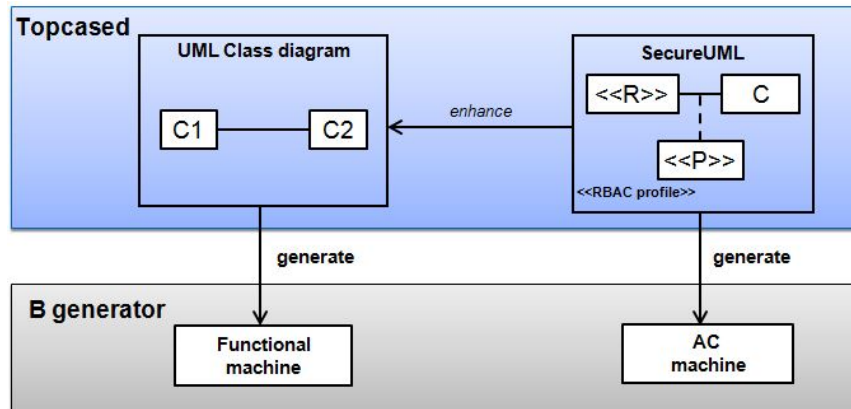


Figure 3.19: The architecture of B4MSecure

The tool produces B specifications from the functional and security models. The transformation follows the MDE paradigm, i.e. it is based on metamodels which encode semantics of the models used in the framework, namely UML, SecureUML, and B metamodels. The transformation rules are defined as mappings between these metamodels. The resulting B specifications include:

- The B specification of the functional model describes data and functionalities of the system. The B elements are generated from classes, attributes, and associations between classes (Figure 3.20).

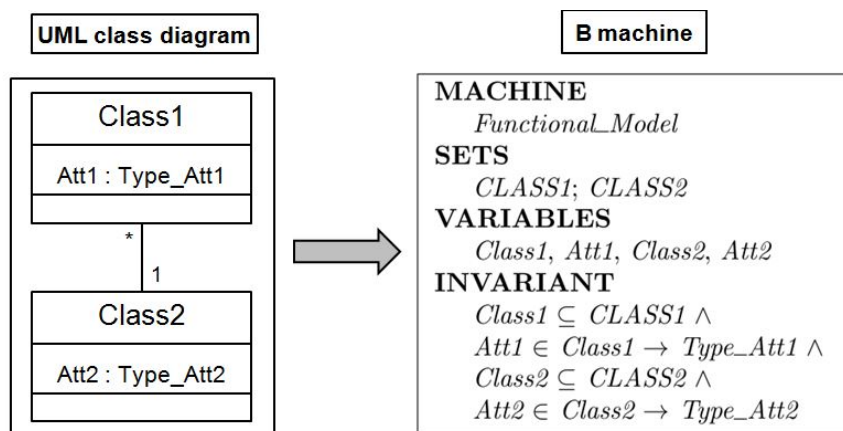


Figure 3.20: UML to B translation

- The formal specification derived from SecureUML gathers secure operations that encapsulate functional operations (Figure 3.22) and the essential

RBAC concepts (Figure 3.21).

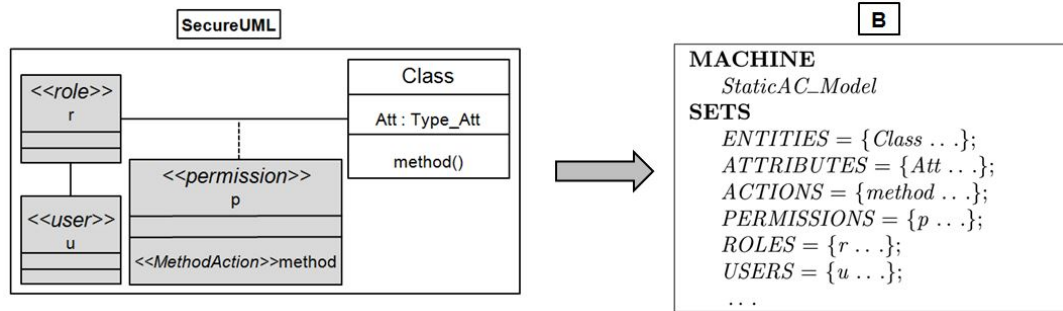


Figure 3.21: SecureUML to B translation

A secure operation (Figure 3.22) is used to check if the user (i.e. the currently logged-in user) requesting the corresponding functional operation has the right to execute it or not: the execution proceeds if the current role of this user is authorized to perform the operation.

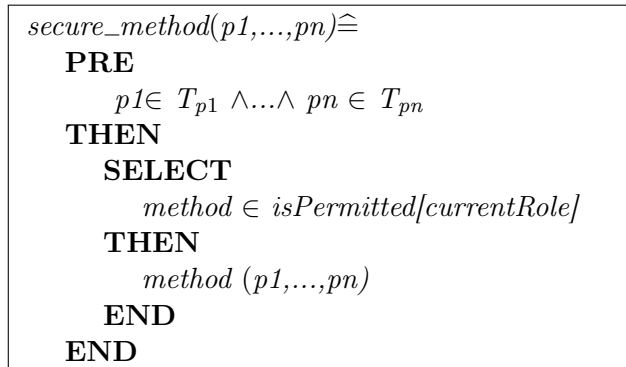


Figure 3.22: A secure operation

The validation of the resulting B specifications can be performed in two ways: proof-based and animation techniques. Taking benefit of the AtelierB tool [26], one can validate the consistency of the B specification based on the generated proof obligations. The ProB tool [27] supports the validation by animating the secure operations.

3.3.3 Discussion

In this section, we presented tools that support RBAC, namely SecureMOVA and B4MSecure tools. Both tools provide a capability of combining a security

language and a functional modeling language. In the case of SecureMOVA, it allows the composition of SecureUML and ComponentUML. B4MSecure allows the integration of SecureUML and UML class diagrams. The validation and the verification of models within SecureMOVA are performed by verifying OCL properties, while B4MSecure supports these activities based on the B method. However, in both cases, they only consider static access control rules. These contributions do not address dynamic security requirements. In our work, we take into account both static and dynamic aspects of an access control along with the functional model. Our goal is to formally verify the correctness and the consistency of these models.

3.4 Implementation of An Access Control Specification

The literature review has shown that security policies can be integrated into system design models. Such an integration can be seen as a basis for generating systems along with their security infrastructures. This section surveys a number of techniques on transforming the RBAC policies into enforcement codes.

In [8], Basin et al. presented the generation of applications and their security infrastructures from UML-based models. They demonstrated model transformations from two combinations of security languages and modeling languages. The first combination is between UML class diagrams and SecureUML, from which they generate access control infrastructures of EJB and .NET systems. The second combination is based on statecharts and SecureUML, which produces access control infrastructures for web applications. However, their generated code does not respect the separation of concerns principle. That means that the generated access control logic is not separated from the application program.

The work in [33] stressed on formalizing a compilation process that automatically generates an AOP-based enforcement code from role-slices access control policies. A role slice is a UML extension which specifies roles as specialized class diagrams. Such diagrams contain classes of the functional model and their methods. The role is granted to the methods appearing in its specialized class diagram. The idea is to gather in a package all the class methods that a role is authorized to perform. Once an access control policy is modeled by using role slices, they are automatically translated to the enforcement code through a code generator. The outputs of the code generator include a *policy database* and an *access control aspect*. The policy database contains the access control policy

and an authorization schema storing users and their assigned roles. The access control aspect intercepts all the calls to the protected classes and grants/denies accesses according to permissions defined in the policy database. Nonetheless, the generated security code is only an abstraction of a real AOP program: i.e. a pointcut is a declaration referring to a specific method, and it has no attribute; an advice represents only the *around* construct of the associated pointcut, and it does not contain any implementation. The approach proposed in this thesis generates a more complete aspect-oriented program for access control enforcement.

Braga [14] proposed a MDS approach that supports code generation from RBAC policies. Towards this end, the author used SecureUML to model RBAC policies and introduced a transformation from SecureUML to aspects. The transformation is based on the three following metamodels:

- **The SecureUML metamodel** defines roles, permissions, resources (i.e. entities, attributes, and methods), authorization constraints, and their assignments. An authorization constraint is essentially a predicate over the state of its associated entity, represented as an OCL boolean expression.
- **The Aspects for Access Control (AAC) metamodel (Figure 3.23):** an instance of the AAC metamodel represents an aspect program of a SecureUML policy. In particular, for each entity in a SecureUML policy, there exists an abstract class (*ResClass*) and an aspect (*Aspect*): the abstract class is actually an interface which is constituted by its attributes (*ResAttribute*) and methods (*ResMethod*); the aspect controls the access to the abstract class. For each method of the abstract class, there exists a pointcut (*Pointcut*) and an advice (*Advice*). The advice relates to SecureUML's authorization constraints associated with the method. The body of an *Advice* is essentially a sequence of conditions (i.e. authorization constraints). It may return successfully if the user has the appropriate *Role* and fulfills the authorization constraints. Otherwise it returns an error (for instance, by raising an exception). Each role in a policy corresponds to a role instance of the aspect (*RoleClass*).
- **The merged metamodel of SecureUML and AAC** defines classes that are given by the disjoint union of SecureUML's and AAC's classes. The relations of this combined metamodel are given by the disjoint union of the relations on each metamodel. The role of such a metamodel is to validate the transformation process by specifying a set of invariants. For instance, an implementation of the transformation from SecureUML to

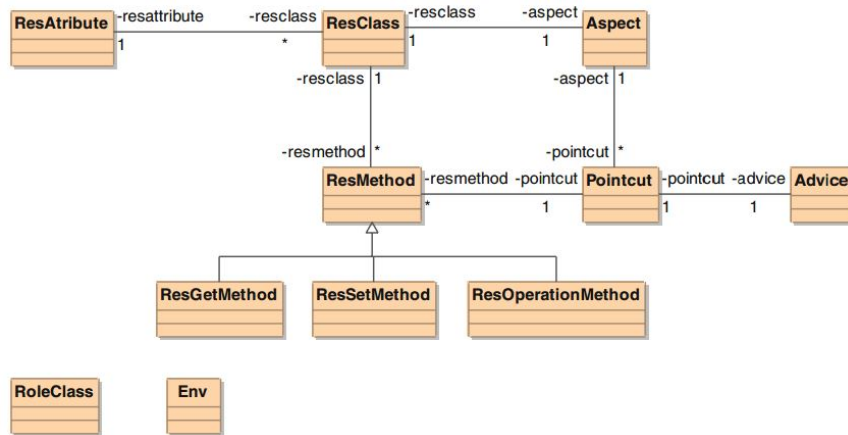


Figure 3.23: The AAC metamodel [14]

AAC must ensure that each entity in a SecureUML model must have an abstract class in the corresponding AAC model.

To sum up, this work is similar to ours in the way they use SecureUML to graphically express access control policies and AOP to enforce these policies. But it differs in the validation and model transformations. In this approach, the validation is restricted to syntactical verification of the different models. Moreover, they do not address the effect of access control models on the functional model, neither do they provide a formal verification of the interaction between security and functional requirements. In our work, we design access control models along with the related functional model. These models are then mapped into B specifications in order to formally reason about their correctness and consistency. In other words, our security policies are verified and validated with respect to the functional specification. The verified B specification is successively refined until its implementation can be straightforwardly mapped into an AspectJ-based program.

3.5 Enforcement of Access Control Policies

Application-level access control policies enforcement has been studied in great details. In this section, we highlight a selection of contributions in this domain. First, we present a practical security library developed by Sun Microsystems in Section 3.5.1. We then discuss the use of annotations for security in Section 3.5.2. Section 3.5.3 reviews a number of the existing works based on AOP for the access control enforcement.

3.5.1 Java Authentication and Authorization Service

RBAC is now available in the standard enterprise software development environments. For instance, Sun Microsystems has released a security framework and library, called Java Authentication and Authorization Service (JAAS) [68], which enhances the Java platform with access control capabilities. In JAAS, the authentication is used to check who is running the system (called *subject*). Both users and computing services can represent subjects. Once authentication has been verified, JAAS provides an authorization check based on privileges (principles) associated with the authenticated subject. JAAS also allows enforcing access controls upon roles/groups just as they are with any principle. Although the goal of JAAS is to isolate the user authentication module from the application code and treat it independently, the developer still needs to write code within the functional program in order to use methods of JAAS. As a result, the final program is tangled and scattered, thus it is difficult to maintain.

3.5.2 Annotation-based approaches

Another notable approach which applies annotations in Java programs for security enforcement is explored in [69, 70]. In particular, protected elements, such as classes, interfaces, and methods in the Java source code, are annotated with roles. Only the users who have at least one of the roles annotated on a method (resp. a class or an interface) are permitted to access that method (resp. that class or interface). The enforcement of RBAC policies is performed dynamically by inserting runtime checks to verify that the current user has the adequate roles when the annotated method is called.

For example, the following code defines a class *Order*, which has two methods *Order* and *approve*. The *Order* method is the constructor creating an object of the class. The *approve* method is used to approve an order. Assuming that the *Accounting* role is granted to execute the *approve* method. To express this rule, one must explicitly annotate the method with the role by writing *@Accounting* above the *approve* definition (the annotation in Java is denoted by the @ symbol). Whenever *approve* is invoked, the system verifies that the currently logged in user is a member of the *Accounting* role. If it is not the case, then the invocation is denied.

```
public class Order {  
    Order(List<Items> items) { ... }  
    @Accounting  
    void approve() { ... }
```

}

In this approach, the user-role assignment can be implemented explicitly, in a XML file for example. Annotating roles on applications can enforce authorization policies at run-time. However, annotations are placed on protected elements in all over the program. It is remarkably difficult to know where a specific annotation should go, especially in large programs.

3.5.3 AOP-based approaches

The problem about scattered and tangled code of the above approaches can be overcome by using AOP. Indeed, this paradigm allows to express separately multiple concerns and automatically merge them together into working systems. The use of AOP for security concerns has been intensively investigated at both modeling [71–73] and implementation levels [15, 16, 31–33, 74]. The following reviews the existing works based upon AOP for enforcing security.

Viega et al. have pioneered an AOP-based approach for enforcing security policies [31]. They developed an aspect-oriented extension of the C programming language for specifying security concerns and implementing a weaver that merges security code into C programs. There are several types of locations that the weaver can operate on, including *function calls*, *function definitions*, and *pieces of functions*. Once the location is identified, one can insert a code before or after the join point, or replace a code at this point. Although their aspect is used to deal with various crosscutting concerns, such as performing error-checking, implementing the buffer overflow protection, and logging data, it does not report how they can be used to enforce access control policies.

AOP-based security enforcement on Java applications was first explored by De Win et al. [15, 32, 74]. The authors introduced a framework based on AspectJ for handling access control to distributed systems. This framework consists of generalized aspects for security requirements, including the *Identification*, *Authentication*, and *Authorization* aspects (Figure 3.24). The *Identification* aspect is used to tag the entities that must be authenticated. It also contains a field *Subject* that stores the identity information. The *Authentication* aspect defines the *authenticationCall* pointcut to specify all the places where the method of an application is invoked. Before the actual execution of a method, the identity information from the *Identification* aspect is copied to a local field of this aspect so that the authentication information can be passed to the access control mechanism. Finally, the *Authorization* aspect verifies access based on the identity information received through the *Authentication* aspect. The verification is performed for every execution of the method by using JAAS. In brief, this

```

public aspect Identification of eachobject(instanceof(Client)){
    public Subject subject null;
}
public aspect Authentication of eachflowroot(authenticationCall())
{
    private Subject subject;
    pointcut serviceRequest() : calls (ServerInterface, * service (..));
    pointcut authenticationCall() :
        hasaspect ( Identification ) && serviceRequest();
    before (Object caller) : instanceof(caller) && authenticationCall(){
        final Identification id = Identification .aspectOf( caller );
        if (id.subject == null){
            <login>;
            subject = id.subject;}
    }
    public Subject getSubject (){
        return subject;}
}
public aspect Authorization{
    pointcut checkedMethods () : executions(* service (..));
    before () returns Object : checkedMethods(){
        Authentication au = Authentication.aspectOf();
        Subject subject = au.getSubject();
        boolean allowed = <check access control>;
        if (allowed)
            return proceed();
        else
            throw new Exception("Access denied");}
}

```

Figure 3.24: Aspect code for object-based access control [15]

approach mainly focused on authentication and authorization, in comparison to our approach that defines an aspect including various types of crosscutting concerns, such as history-based security constraints. Moreover, their approach supports user-based access control, whereas ours is based on role-based access control.

In [16], Huang et al. described the principle and the architecture of a security

aspect library called JSAL (Figure 3.26). This library is built in AspectJ based on the Java security packages JAAS and JCE [75]. It contains four independent aspect components, namely *Encryption/Decryption*, *Authentication*, *Authorization*, and *Security audit*. They created for each component an abstract aspect, which includes common operations. Such an abstract aspect defines codes invoking Java security packages, such as JAAS and JCE. Developers can define a concrete aspect for a specific application by extending the abstract aspect according to the security policy of the application. The extension is done by defining or overriding pointcuts. For example, the abstract encryption aspect *AbstractDESApect* (Figure 3.25) contains two abstract pointcuts, namely *encryptOperations* and *decryptOperations*, and two *around advices* implementing the pointcuts. The advices make calls to the JCE package to use its security operations, such as *encrypt* and *decrypt*. When it is used, the developer needs to extend the abstract encryption aspect and define the concrete pointcuts within the extended aspect. e.g. *sendMsg* and *recvMsg* are the concrete pointcuts in the simple example *main*. Whenever these methods (*sendMsg* and *recvMsg*) are called, the concrete aspect *MyDESApect* will intercept them with their corresponding advices code. In summary, the approach provided reusable security aspects in AspectJ as a practical software component. However, their aspects are based only on popular Java security packages. That might not be enough for the security of a system.

3.5.4 Discussion

In comparison, aspect-oriented security is more flexible than annotation-based approaches and JAAS. Regarding the annotation-based security techniques, security properties are declaratively expressed within the application, but the annotations cause code tangling and scattering. In the case of JAAS, application developers are free from implementing security mechanisms and leave the definition of security properties to security experts at first step. Yet, this solution does not allow the separation of concerns since it requires to mix the security and application codes in the final system. On the other hand, AOP-based solutions allow to implement independently the security code and then weave it into the application code during the compile time or the runtime. No modification of the functional program is required to introduce security properties in this approach.

To our best knowledge, none of the studies based on AOP consider dynamic security requirements (e.g. history-based and order-based constraints). In this thesis, we propose a security enforcement approach that includes both static

```

/*Abstract aspect*/
public abstract aspect AbstractDESAspect {
    public abstract pointcut encryptOperations(String msg);
    public abstract pointcut decryptOperations(String msg);
    public void around(String msg): encryptOperations(msg) {
        DesCipher enc = new DesCipher();
        enc.savekey("Deskey");
        //Encrypt
        String encryptedMsg = enc.encrypt(msg);
        proceed(encryptedMsg);
    }
    public void around(String msg): decryptOperations(msg) {
        DesCipher enc2=new DesCipher("Deskey");
        //Decrypt
        String decryptedMsg = enc2.decrypt(msg);
        proceed(decryptedMsg);
    }
}
/*Concrete aspect*/
public aspect MyDESAspect extends AbstractDESAspect {
    public pointcut encryptOperations(String msg):
        call (String sendMsg(String)) && args(msg);
    pointcut decryptOperations(String msg):
        call (String recvMsg(String)) && args(msg);
}
/*A simple example*/
public static void main(String[] args) {
    String text = "Hello world!";
    // send messages, needs encryption here
    String set = sendMsg(text);
    System.out.println ("The send messages are : " + set);
    String dec = recvMsg(set);
    // receive messages, needs decryption here
    System.out.println ("The send messages are : " + dec);
}

```

Figure 3.25: An example of the JSAL implementation in AspectJ [16]

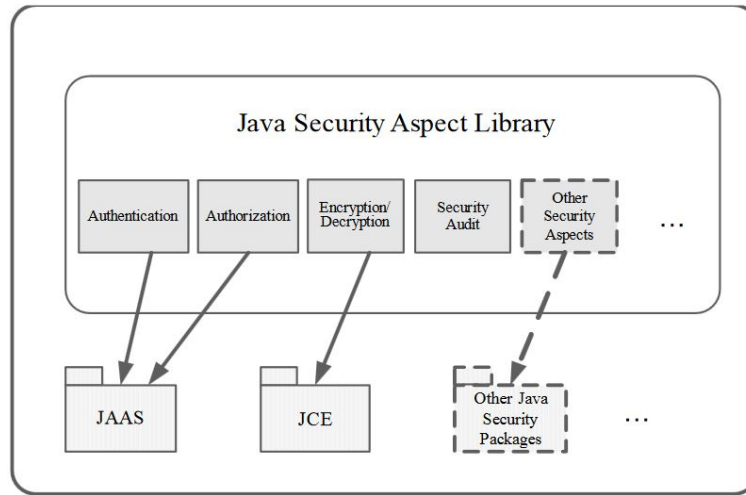


Figure 3.26: JSAL architecture [16]

and dynamic security properties of an information system. The proposed aspect checks the permission of authenticated users as well as history-based and order-based conditions associated to each method called by the users. Moreover, the security program is generated from a proved formal specification.

3.6 Conclusion

Access control policies protect the resources of software systems by controlling who has rights to access to them. RBAC has become the standard authorization model used in the industrial applications because it simplifies the access control management. Thus, there are a large number of studies dealing with RBAC. In this chapter, we have presented existing works on the specification and enforcement of access control policies. We have also reviewed tools that support the modeling and the formal specifications of such requirements. In our work, we propose a solution based on the MDE paradigm to handle various security requirements from the early design step to the implementation step.

Formal Development of a Secure Access Control Filter

The contents of this chapter are adapted from our paper [76]. My main contributions in this paper are the modeling of different security requirements of an information system in UML-based diagrams and their transformations into B specifications together with the functional model of the system. The paper is submitted and accepted in the 17th IEEE International Symposium on High Assurance Systems Engineering took place in Orlando, Florida, USA on 7th-9th January 2016.

Contents

4.1	Introduction	90
4.2	The case study: a bank system	92
4.3	Graphical modeling of security requirements	93
4.3.1	SecureUML	93
4.3.2	Activity diagrams for dynamic security rules	94
4.4	Generation of a B specification	96
4.4.1	Overview of the B method	97
4.4.2	Translation of the functional model: the class diagram	97
4.4.3	Formalizing SecureUML in B	99
4.4.4	Translation of the secure UML activity diagrams into B	101
4.4.5	Putting all the security and functional constraints together	105
4.5	Verification and validation	107
4.6	Conclusion	109

With the advent of the Internet, most organizations offer more and more access to their information systems in order to increase their benefits. However, such an opening may cause security issues if sufficient precautions are not taken. An adequate solution to secure access to information systems consists in (1) defining the sufficient security policies and (2) ensuring their correct deployment on a given technological infrastructure. The present chapter deals with the first point by introducing a formal approach that permits to develop a secure filter for an information system that respects different kinds of security rules: functional, static and dynamic rules. The proposed approach uses the SecureUML language [77] to express the static rules and adapts the UML activity diagrams for dynamic ones while the structure of the manipulated data and the functionalities are expressed using a UML class diagram. Starting from these graphical notations, the approach consists in mapping them into a B formal specification to ensure their consistency and validate the system. Finally, a proved filter, which permits to take into account different security rules, is formally derived using the B refinement technique [78].

4.1 Introduction

An Information System (IS) is the part of an organization responsible for collecting and manipulating all its relevant and sensitive data. Nowadays, it is at the heart of most companies and constitutes then a critical element that needs an adequate attention regarding security issues. Indeed, an information system often interacts with humans or other systems by exchanging information and any security breach may cause serious and even irreversible consequences. To avoid such risks, a common way is to control access to an information system by defining some security rules. Roughly speaking, a security rule specifies, for an authenticated user, which actions are allowed/forbidden according to his/her current role and context. To ensure the security of a system, many types of rules may be required. These rules can be classified into two main classes: static and dynamic. Static rules refer to a given single moment of the system whereas dynamic ones require to take the execution history of the system into account, that is the actions already performed in the system in general or by a given user in particular. If we consider the case of a hospital, a static rule will be, for instance, “*only a person with the role Doctor can make a diagnosis*”, whereas a dynamic rule will be, for example, “*the person who performs a laboratory test cannot validate it*”. In addition to these kinds of rules, we have also to consider the usual functional constraints like, for instance, the maximum

number of patients each doctor can treat. In face of this rule diversity, several languages may be needed to cope with them. In this chapter, we propose to use three UML-based languages: a class diagram to describe the structure of the data manipulated in the system together with their functional constraints, **SecureUML** [77] to deal with static rules and activity diagrams [79] for dynamic ones.

Even if the use of a graphical notation to express functional/security rules argues for an intuitive, synthetic and visual presentation of the considered system, it is often a source of ambiguities. Moreover, we need a unifying language in which all the functional/security rules can be expressed in order to be coordinated to verify the consistency of the different security rules with respect to the functional aspects of the system. To this aim, we suggest to translate the obtained graphical modeling into a formal B specification that can be formally verified using the different associated tools. We have chosen the B method because it is based on concepts that are easy to learn. Moreover, it has a reliable free tool (AtelierB [80]) that supports all the development stages. The obtained B specification is then taken as a basis for the development of a correct access control filter that ensures that the execution of each action of the system can happen only if all the specified functional/security rules are fulfilled. Such an approach permits to consider the different functional/security constraints from the first design phases on and therefore to reduce the global development cost.

The main contributions of the present chapter can be summarized as follows:

1. A set of generic rules to map a **SecureUML** diagram, modeling static security rules, into a B specification,
2. A set of generic rules to map a UML diagram, representing a dynamic security rule, into a B specification,
3. A generic specification of a filter that permits to coordinate all the specified security rules.

The rest of the chapter is organized as follows. The case study used throughout the chapter to illustrate the proposed approach is introduced in the next section. Section 4.3 shows the modeling of the functional and security requirements using UML-based notations namely, a class diagram, **SecureUML** and activity diagrams. The translation of these diagrams into a B specification is presented in Section 4.4. The verification and the validation of these specifications are illustrated in Section 4.5. The last section concludes and presents some future work.

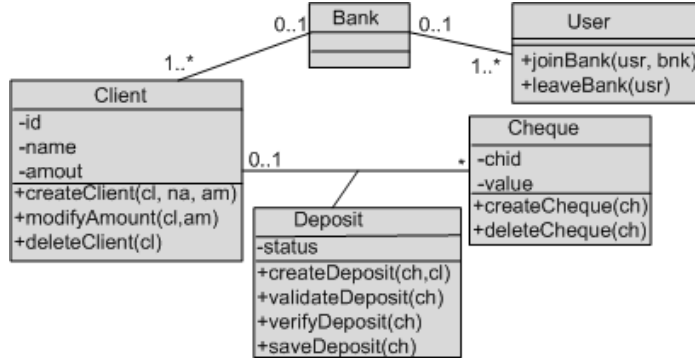


Figure 4.1: The class diagram of a simplified banking system

4.2 The case study: a bank system

To illustrate the proposed approach, we reuse the case study introduced in [81]. It is about a cheque deposit use case of a banking system whose class diagram is depicted in Figure 4.1. This use case involves two classes, *Client* and *Cheque*, which are linked by an association class *Deposit*. Each class or association class is described by a set of attributes and defines some operations to create new instances or delete the existing ones. We make the assumption that the first attribute of each class denotes its key. As specified by the multiplicities, each cheque may be deposited by one client at most (multiplicity 0..1), whereas each client may deposit zero to several cheques (multiplicity 0..*). In addition, the bank manages a set of users that can join/leave the bank at any moment.

The deposit of a cheque by a client is executed by a bank's employee and launches the following ordered actions:

- (1) creating a new deposit to link the client to the cheque (operation *createDeposit*)
- (2) validating the cheque by an employee of the branch (operations *validateDeposit*)
- (3) saving the cheque (operation *saveDeposit*)
- (4) verifying the cheque if its amount exceeds a given limit (operation *verifyDeposit*)

To ensure the security of this deposit process, a set of security rules have been identified. For the sake of concision, only the following rules are presented:

- Rule 1.** Only **Tellers** and **Advisors** are authorized to make a deposit (operation *createDeposit*),
- Rule 2.** Only **Tellers** are permitted to validate a deposit (operation *validateDeposit*),
- Rule 3.** A deposit should not be validated by the same user who created it,
- Rule 4.** The validation of a cheque (operation *validateDeposit*) should be executed by a user who belongs to the bank at the time the deposit is created.

Analyzing the above rules gives:

- (i) Rules 1 and 2 specify which roles the user must play to perform the corresponding actions,
- (ii) Rule 3 denotes a dynamic separation of duty since the same user cannot play different roles for the same deposit. The separation of duty allows to restrict the roles that a user can play depending on the actions he/she has/has not performed in the past.
- (iii) Rule 4 is a dynamic rule since the permission of a cheque validation depends on an action executed in the past.

The following section presents the graphical modeling of these rules using SecureUML and UML activity diagrams.

4.3 Graphical modeling of security requirements

Before describing how we use the SecureUML and UML activity diagrams to describe both static and dynamic security rules, we give a brief introduction of these notations.

4.3.1 SecureUML

Based on UML graphical notations, SecureUML allows to extend a UML functional model with concepts of role-based access control models (RBAC [82]) in order to specify the different roles that a user can play and their associated permissions on the resources of the system. Basically, a SecureUML diagram depicts some classes of the class diagram that are linked to classes representing roles (stereotype << *role* >>). These links are association classes (stereotype

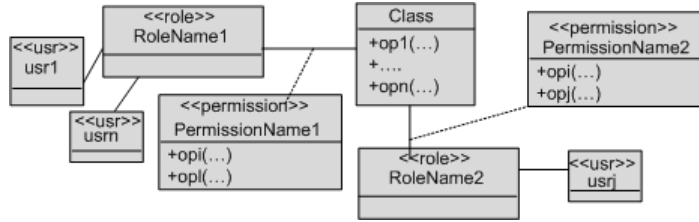


Figure 4.2: Generic SecureUML model

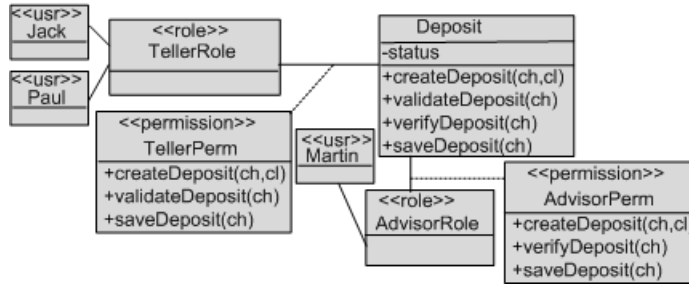


Figure 4.3: Static access control model

`<< permission >>`) denoting the permissions of the role to execute some operations of the class/association class. Finally, specific users can be attached to a given role to state that they can play it (See Figure 4.2).

Figure 4.3 can be seen as an instantiation of Figure 4.2 to model the rules 1 and 2 of the case study: two roles are defined `TellerRole` and `AdvisorRole`; `TellerPerm` states that users with Role `TellerRole` are permitted to create (operation `createDeposit`), validate (operation `validateDeposit`) and save (operation `saveDeposit`) a deposit; `Jack` and `Paul` both can play Role `TellerRole`.

4.3.2 Activity diagrams for dynamic security rules

In [83], the ASTD notations [84] are used to model dynamic security rules. Even if ASTD has a rich power of expressing, they are not always simple to use and the translation of such a modeling into B gives a very complex specification with very difficult proof obligations. This is why in this chapter, we suggest to use the UML activity diagrams to model dynamic security rules.

UML activity diagrams are used to describe the dynamic aspects of systems. They aim at depicting the control flow from one activity to another by including sequencing, branching, parallel flow, swimlanes, etc. They are usually used to describe business processes. In a UML activity diagram, an activity cannot start its execution before the completion of all the activities that precede it.

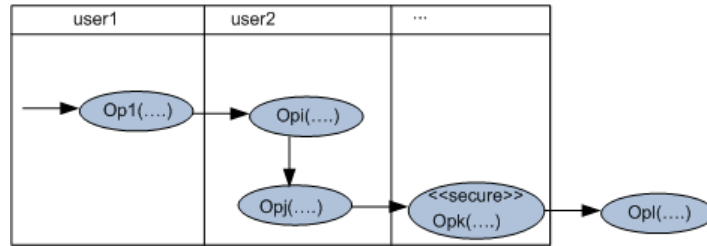


Figure 4.4: A generic form of a secure activity diagrams

To model a dynamic security rule with an activity diagram, we consider that activities refer to the operations of classes/associations, whereas swimlanes denote the actors executing them. Moreover, a dynamic security rule is modelled by a partial activity diagram to show, on the one hand, the execution order of the operations and, on the other hand, the constraints on the users executing them. Basically, this is achieved as follows (See figure 4.4):

- if an operation Op_j should be executed after an operation Op_i , then we link the activity nodes related to them by a transition from Op_i to Op_j ,
- using the stereotype `<<secure>>`, we indicate that the operation Op_k is the operation to secure, that means that any execution constraints will be put on this operation,
- if the users that perform the operations Op_i and Op_k should be different (resp. the same), then we put both operations in different (resp. the same) swimlanes.
- the operation Op_i that should not happen before the operation to secure is placed after it using a control flow.

Figure 4.5 and 4.6 show the UML activity diagrams associated with Rules 3 and 4. As stated before, these rules are dynamic since they take the history of the system into account. Rule 3 states that the users executing operations *createDeposit* and *validateDeposit* should be different. Both rules implicitly express that Operation *createDeposit* is executed before Operation *validateDeposit*.

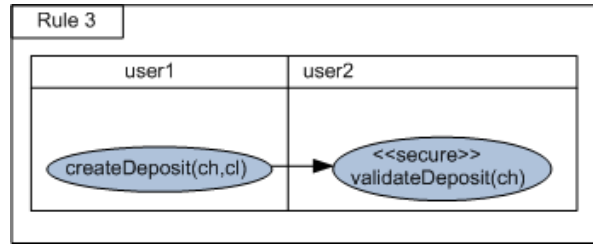


Figure 4.5: Example of dynamic security rules modeled by activity diagrams: Rule 3

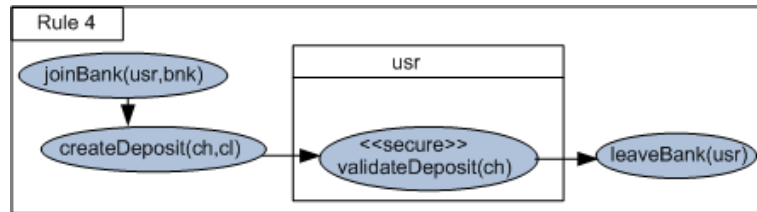


Figure 4.6: Example of dynamic security rules modeled by activity diagrams: Rule 4

4.4 Generation of a B specification

The graphical modeling achieved in the previous section offers a visual, synthetic and user-friendly view of the system. However as stated before, it may be source of ambiguity and does not permit any formal verification. On the other hand, formal methods permit to build precise models on which it is possible to verify a large range of properties and even to generate a correct implementation of the system. This is why much research work has investigated the combination of graphical notations with formal methods in order to take advantage of both notations [85–89]. In that direction, we have proposed formal rules to translate UML notations into a B specification [90, 91]. This work mainly stressed the functional aspects of a system and did not consider any security constraints. This section briefly recalls the defined translation rules, then it shows how the obtained B specification can be completed with the translation of the security diagrams, that are the SecureUML and UML activity diagrams.

4.4.1 Overview of the B method

Introduced by Jean-Raymond Abrial [78], B is a formal method dedicated to developing safe systems. B specifications are organized in abstract machines. Each machine contains state variables on which operations act, and an invariant constraining them. Operations are specified in the Generalized Substitution Language. A substitution is like an assignment statement. It allows us to identify which variables are modified by the operation without mentioning the variables not modified. A preconditioned substitution is the form (**PRE** P **THEN** S **END**) where P is a predicate, and S a substitution. When P holds, the substitution is executed, otherwise nothing can be ensured. For instance, the substitution S might not terminate or might violate the invariant. The B refinement is the process of transforming, by successive steps, a specification (the variables and the operations) into a less abstract one. The last refinement step, called implementation, aims at obtaining data and substitutions close to those of a programming language such that the translation into the chosen target language of the data and control structures (used in this level) must be a straightforward task. Both specification and refinement steps give rise to proof obligations. At the abstract level, proof obligations ensure that each operation maintains the invariant of the system, whereas at the refinement level, they ensure that the transformation preserves the properties of the abstract level.

4.4.2 Translation of the functional model: the class diagram

The translation rules defined in [90,91] generate a B specification from a UML class diagram as follows. For each class C , we define a given set S_C and a variable V_C to respectively represent the set of all possible instances and the existing ones at each moment. Each attribute is mapped into a function from V_C to its type, the function modeling the key of the class is an injection. An association involving two classes C_1 and C_2 is mapped into a B variable defined as a relation between V_{C_1} and V_{C_2} . According to its multiplicities, this association becomes partial function (\rightarrow), total function (\rightarrow), an injection (\hookrightarrow), etc. To ease the integration of the security rules, this B specification is structured as follows: all the derived given sets are included into a machine called *Context* that can be seen (Clause **SEES**) by any machine to have access in read to them (See Figure 4.7). The variables and the operations are defined in a separate machine that sees the machine *Context*. Figure 4.8 shows a part of the B machine generated for the class *Cheque* and the association *Deposit* (See Figure 4.1).

The machine *Functional_Requirement* specifies a set of operations of the

```

MACHINE Context
SETS Clients; Cheques;
      Status= {deposited, validated, saved};
      ExecutionResult={Ok, Ko}
END

```

Figure 4.7: Machine *Context*

```

MACHINE Functional_Requirement
SEES Context
VARIABLES   cheques, chid, value, deposit, status
INVARIANT
  cheques  $\subseteq$  Cheques  $\wedge$  chid  $\in$  cheques  $\mapsto$  NAT  $\wedge$ 
  deposit  $\in$  cheques  $\rightarrow$  clients  $\wedge$  status  $\in$  deposit  $\rightarrow$  Status
  ...
END

```

Figure 4.8: The static part of the functional B specification

classes/associations. Such operations permit, for instance, to add/delete an instance and also to update the attribute values. Contrary to the translation rules defined in [90, 91], a defensive strategy is adopted in the present chapter for the generation of these operations since we take all the possible parameter values into account. Basically, only the type of the input parameters is assumed in the precondition of an operation, all the other conditions required for its correct execution are checked in its body using an **IF** substitution: if the conditions are fulfilled then the state of the system evolves otherwise it remains unchangeable. In both cases, a value is returned to inform the user about the result of the execution of the operation. Figure 4.9 depicts the B specification of the operation *validateDeposit* defined on the association *Deposit*. This operation assumes the type of its input parameter ($ch \in Cheques$), then it verifies if the related cheque has been deposited: if so ($ch \in dom(deposit)$), the status of the deposit is updated to *validated* (using the substitution $status(ch, deposit(ch)) := validated$) and the user is informed that the state of the system has been updated ($result := Ok$). Otherwise, nothing is done and the value *Ko* is returned.

```

result ← validateDeposit(ch) ≐
PRE   ch ∈ Cheques THEN
  IF ch ∈ dom(deposit) THEN
    status(ch, deposit(ch)) := validated ||
    result := Ok
  ELSE result := Ko
  END
END;

```

Figure 4.9: The functional specification of Operation *validateDeposit*

4.4.3 Formalizing SecureUML in B

To our best knowledge, research work on the translation of **SecureUML** into formal specifications is rather narrow. In [92], the authors present an Event-B model for OrBAC policies with the purpose of deploying a security policy. Our goal is to integrate security policies into functional requirements from the early design phases.

An interesting idea of integrating security requirements in the software development process is introduced in [93] using Alloy, a formal method based on a model checking technique. Due to the state space explosion problem which is intrinsic to model checking, such an approach is not suitable for information systems where applications are data intensive. The closest work to ours seems to be that presented in [94, 95] where formal mapping rules are defined to derive B specifications from **SecureUML** diagrams. Basically, these rules consists in defining a secure operation *secureOp* for each operation *Op* whose execution is permitted to a given role *r*. Operation *secureOp* has two additional input parameters, the user executing the operation and his/her current role. After verifying that the role has the permission to execute *op*, then the operation *Op* is called:

```

SecureOp(..., usr, role) ≐
PRE usr ∈ Users ∧ role ∈ PermittedOfUser(usr) ∧
  Op ∈ Permitted(role) ∧ ... THEN
  Op(...)
END

```

The main drawback of such approaches is that the proposed translation is not faithful to the semantics of the **SecureUML** diagram. Indeed, the semantics of a **SecureUML** diagram only states which role can execute a given operation but

does not mean the actual execution of the operation. Indeed, the actual execution of the operation can require other conditions like dynamic security rules. Moreover, since the permission of the role to execute the operation Op is checked in the precondition, the user still can call the operation even if his/her role is not allowed to do that! In this chapter, we propose an alternative of this translation that overcomes these drawbacks. We translate the **SecureUML** diagram of Figure 4.2 as follows by defining a new context machine *SecureUMLContext* that includes:

- 3 given sets: $Roles = \{RoleName1, RoleName2\}$ represents the different roles existing in the system, $Operations = \{op_1, \dots, op_n\}$ to denote the operations defined in classes and associations, and $Users = \{usr1, usn, usrj\}$ to represent the users of the system,
- 2 constants: $Permissions = \{RoleName1 \mapsto opi, RoleName1 \mapsto opl, RoleName2 \mapsto opi, RoleName1 \mapsto opj\}$ to model the permitted operations of each role, and $PermittedUsersRoles = \{usr1 \mapsto RoleName1, usrn \mapsto RoleName1, usrj \mapsto RoleName2\}$ to store the roles that a user can play.

Applying these rules to the **SecureUML** diagram of Figure 4.3 gives the B specification of Figure 4.10.

The above machine *SecureUMLContext* is included into a new machine that defines a variable *CurrentRole* that gives the current role of each user and an operation *ConnectUser*(*user*, *role*) that permits User *usr* to connect to the system using Role *role* (See Figure 4.11).

To introduce the translation of **SecureUML** into B, let us consider the diagram of Figure 4.2. Each operation Op_i , that can appear in several permissions $PermissionName_j$, is mapped into a single B operation *SecureOp_i* with a parameter *usr* to denote the user that wants to execute Op_i . The operation checks whether the current role of the user has the permission to execute op_i . According to the result, the system informs the user that the execution is granted or not (See Figure 4.12). These generated operations are defined in one or several new machines that see Machines *SecureUMLContext* and *SecureUMLTranslation*.

For instance, Figure 4.2 states that Operation $op1$ is permitted for any user that has either Role *RoleName1* or Role *RoleName2*. In B, this is translated as depicted in Figure 4.12.

By instantiating the above generic operation, the B specification of Operation *SecureValidateDeposit* is as in Figure 4.13.

These generated operations are included into Machine *SecureUMLTranslation*.

```

MACHINE SecureUMLContext
SETS Roles = { TellerRole, AdvisorRole };
       Operations = { createDeposit, validateDeposit,
                     saveDeposit, verifyDeposit };
       Users = { Paul, Jack, Martin };
       Access = { granted, denied };
CONSTANTS   Permissions, PermittedUsersRoles
PROPERTIES
       Permissions = { TellerRole  $\mapsto$  createDeposit,
                       TellerRole  $\mapsto$  validateDeposit,
                       TellerRole  $\mapsto$  saveDeposit,
                       AdvisorRole  $\mapsto$  createDeposit,
                       AdvisorRole  $\mapsto$  verifyDeposit,
                       AdvisorRole  $\mapsto$  saveDeposit };
       PermittedUsersRoles = { Paul  $\mapsto$  TellerRole,
                                Jack  $\mapsto$  TellerRole, Martin  $\mapsto$  AdvisorRole }
END

```

Figure 4.10: The B context machine of the SecureUML diagram of Figure 4.3

```

MACHINE SecureUMLTranslation
SEES SecureUMLContext
VARIABLES CurrentRole
INVARIANT
       CurrentRole  $\in$  Users  $\leftrightarrow$  Roles  $\wedge$ 
       CurrentRole  $\subseteq$  PermittedUsersRoles
OPERATIONS
ConnectUser(usr, role) =
PRE usr  $\in$  Users  $\wedge$  role  $\in$  Roles  $\wedge$ 
       role  $\in$  PermittedUsersRoles[{usr}] THEN
       CurrentRole(usr) := role
END

```

Figure 4.11: The B machine managing the roles of users

4.4.4 Translation of the secure UML activity diagrams into B

Let us recall that we use UML activity diagrams to express dynamic security rules. Such rules often depend on the execution history of the different system's actions. In other words, we need to know, at each moment,

```

access ← SecureOp1(usr) ≐
PRE usr ∈ Users THEN
  IF usr ∈ dom(CurrentRole) ∧
    CurrentRole(usr) ∈ {RoleName1, RoleName2} THEN
    access := granted
  ELSE access := denied
  END
END

```

Figure 4.12: B translation of a SecureUML permission (Figure 4.2)

```

access ← SecureValidateDeposit(usr) ≐
PRE user ∈ Users THEN
  IF user ∈ dom(CurrentRole) ∧
    CurrentRole(user) ∈ {TellerRole} THEN
    access := granted
  ELSE access := denied
  END
END

```

Figure 4.13: B translation of a SecureUML operation (Figure 4.3)

which actions have actually been executed in the system. This is why for each operation $op(p_1, \dots, p_n)$, we define two additional variables $historyOp$ and $OrderExecutionOp$ that store its corresponding execution occurrences together with the user that executes it and the execution time:

```

historyOp ∈ Typep1 × ... × Typepn → Users
OrderExecutionOp ∈ Typep1 × ... × Typepn → NAT

```

Such variables are included into a new machine *ActionsHistory* that defines an operation $ExecutionOp$ for each operation Op of the functional model. Regarding Op , $ExecutionOp$ has an additional parameter that denotes the user who is executing Op ; it also checks the same functional constraints (using the **IF** substitution). If the functional constraints required for the execution of op are fulfilled, it calls the operation Op and updates the variables $historyOp$ and $OrderExecutionOp$ using the value of the clock of the system $currentOrder$. Of course, to make the call to Operation Op possible, Machine *ActionsHistory* includes (**INCLUDES** of B) Machine *Functional_Requirement*:

```

result ← ExecutionOp(usr, p1, . . . , pn) ≐
PRE usr ∈ Users ∧ precondition_of_Op THEN
  IF same_conditions_as_Op THEN
    result ← op(p1, . . . , pn) ||
    historyOp(p1, . . . , pn) := usr ||
    OrderExecutionOp(p1, . . . , pn) := currentOrder
  ELSE result := Ko
END
END

```

Instantiating these mapping rules gives the following B specification for Operation *ValidateDeposit* (See Figure 4.14).

```

MACHINE ActionsHistory
SEES SecureUMLContext, Context
INCLUDES Functional_Requirement
VARIABLES
  historyValidateDeposit, OrderExecutionValidateDeposit
INVARIANT
  historyValidateDeposit ∈ cheques → Users
  OrderExecutionValidateDeposit ∈ cheques → NAT
OPERATIONS
result ← ExecutionValidateDeposit(usr, ch, cl) ≐
PRE usr ∈ Users ∧ ch ∈ Cheque ∧ cl ∈ Clients THEN
  IF ch ∈ dom(deposit) THEN
    result ← validateDeposit(ch) ||
    historyValidateDeposit(ch) := usr ||
    OrderExecutionValidateDeposit(ch) := currentOrder
  ELSE result := Ko
END
END

```

Figure 4.14: Machine *ActionsHistory*

The history of executed actions being memorized, each secure activity of an activity diagram is translated into a B operation as follows. Let us consider a secure operation *Opk* preceded by a sequence of activities $Op_i(p_{i1}, \dots, p_{in})_{i=1..n}$ and followed by an activity $Op_l(p_1, \dots, p_n)$ (See figure 4.4). Then, *Opk* is translated as illustrated by Figure 4.15.

Regarding the initial operation (which is defined in the class diagram), this operation has as additional parameters (p_1, \dots, p_n) all the parameters of the


```

access ← ADSecOpk( $p_1, \dots, p_n, \mathbf{usr}$ )  $\hat{=}$ 
PRE  $usr \in Users \wedge$  typing of the input parameters THEN
  IF  $\bigwedge_{i=1..n} (p_{i1}, \dots, p_{in}) \in dom(historyOp_i) \wedge$ 
     $\bigwedge_{i=1..n-1} OrderExecutionOp_i(p_{i1}, \dots, p_{in}) \leq$ 
       $OrderExecutionOp_{i+1}(p_{i+11}, \dots, p_{i+1n}) \wedge$ 
     $(p_1, \dots, p_n) \notin dom(historyOp_l) \wedge$ 
    additional conditions
  THEN  $access := granted$ 
  ELSE  $access := denied$ 
  END
END

```

Figure 4.15: Translation of a secure operation of an activity diagram

activities that appear in the activity diagram. Moreover, we have a parameter usr denoting the user executing it. The operation returns *granted* if the previous operations Op_i are executed in the right order and the following Op_l action is not executed yet. Additional conditions may be included in order to translate constraints related to the users executing the operations. These conditions are of the form:

$$historyOp_i(p_{i1}, \dots, p_{in}) \text{ op } historyAOp_j(p_{j1}, \dots, p_{jn})$$

where $op = "="$ (resp. $op = "\neq"$) if the operations Op_i and Op_j should be executed by the same user (resp. different users). For instance, the activity diagrams of Figure 4.5 and 4.6 modeling Rules 3 and 4 are translated into B in Figure 4.16 and 4.17 respectively:

```

access ← ADValidateDeposit(ch, cl, usr)  $\hat{=}$ 
PRE  $usr \in Users \wedge ch \in Cheques \wedge cl \in clients$ 
  THEN IF  $ch \mapsto cl \in dom(historyCreateDeposit) \wedge$ 
     $usr = historyCreateDeposit(ch \mapsto cl)$ 
  THEN  $access := granted$ 
  ELSE  $access := denied$ 
  END
END

```

Figure 4.16: Translation of the activity diagram of Figure 4.5

The operations generated from the activity diagrams are defined in a new B machine *AD_Translation* that sees *Context*, *ActionsHistory* and *SecureUMLContext*.

```

access  $\leftarrow$  ADValidateDeposit(ch, cl, usr, bnk)  $\hat{=}$ 
PRE usr  $\in$  Users  $\wedge$  ch  $\in$  Cheques  $\wedge$ 
      cl  $\in$  Clients  $\wedge$  bnk  $\in$  Banks THEN
  IF ch  $\mapsto$  cl  $\in$  dom(historyCreateDeposit)  $\wedge$ 
      usr  $\in$  dom(historyJoinBank)  $\wedge$ 
      ch  $\mapsto$  cl  $\in$  dom(historyCreateDeposit)  $\wedge$ 
      usr  $\notin$  dom(historyLeaveBank)  $\wedge$ 
      OrderExecutionJoinBank(usr, bnk)  $\leq$ 
      OrderExecutionCreateDeposit(ch, cl)
  THEN access := granted
  ELSE access := denied
  END
END

```

Figure 4.17: Translation of the activity diagram of Figure 4.6

4.4.5 Putting all the security and functional constraints together

As we can remark, the B specification we have built makes a separation between functional and static/dynamic security modeling. The advantage of such a separation is twofold. New security or functional requirements can be integrated to the system without altering the existing B specification. The same remark applies for the deletion of existing security/functional requirements. In both cases, the modification is not very important since it would only affect a particular part of the specification. In addition, software designers having different qualifications can be assigned to specific parts of the development.

Nevertheless, an additional step is needed to put all these parts together in order to take a decision about the permission/prohibition of an access to a specific user. Let us suppose, e.g., that a user has to validate a deposit. The question is now which operation the user has to invoke:

- *validateDeposit* is not suitable since it does not take into account the security rules at all,
- *SecureValidateDeposit* is not suitable since it does not take into account either the functional constraints or the dynamic security rules,
- *ADValidateDeposit* is not suitable since it does not take into account either the functional constraints or the static security rules,

Thus, we have to specify a new operation that permits/denies a user to execute an operation by taking all the security/functional constraints into account. We call such an operation a *filter* which we develop using the B refinement.

The filter is defined in a new machine *Secure_Filter* that sees both *Context* and *SecureUMLContext*. It also includes *ActionsHistory* (See Figure 4.18). At the abstract level, the filter defines an operation *FilterOp* for each operation *Op* defined in a class or an association. Compared with the operation *Op*, this operation has an additional parameter *usr* to denote the user who is invoking *Op*. It returns two values *access* and *result*: (*access = granted*) means that all the static/dynamic security rules are fulfilled, otherwise *denied* is returned; *result = Ok* means that the operation *Op* has been successfully executed, otherwise *Ko* is returned. Of course, the values of *access* and *result* should imply the following property that states that: the operation *Op* can be executed only if the access is granted for the user:

$$result = Ok \Rightarrow access = granted$$

```

MACHINE
  Secure_Filter
SEES Context, SecureUMLContext
INCLUDES ActionsHistory
OPERATIONS
  result, access ← FilterValidateDeposit(usr, ch, cl, bnk) ≐
PRE usr ∈ Users ∧ ch ∈ Cheques ∧
  cl ∈ Clients ∧ bnk ∈ Banks THEN
  CHOICE
    result ← ExecutionValidateDeposit(usr, ch, cl) ||
    access := granted
  OR result := Ko || access := denied
  END
END;
END

```

Figure 4.18: The abstract specification of Machine *Secure_Filter*

At the abstract level, the filter states that when Operation *validateDeposit* is invoked by a user, two options are possible: either the access is denied and the operation is not called at all (*result = Ko*) or the access is granted and the result is equal to what the operation call returns. More precisely, the result depends on the functional requirements. At this level, we do not specify how

the access rights are defined for a user. It is the purpose of the refinement that details the steps to follow to do that.

To give access to the resources of the system, the filter has to check both static and dynamic security rules. If both rules are satisfied, the access is granted and the user can safely invoke the operation, otherwise the access is denied and nothing is done (See Figure 4.19).

```

result, access ← FilterValidateDeposit(usr, ch, cl, bnk) ≐
VAR staticRights IN
  staticRights ← SecureValidateDeposit(usr);
  IF staticRights = granted THEN
    VAR dynamicRights IN
      dynamicRights ← ADValidateDeposit(ch, cl, usr);
      IF dynamicRights = granted THEN
        result ← ExecutionValidateDeposit(usr, ch, cl, bnk)
      ELSE result := Ko
      END;
      access := dynamicRights ;
    END
  ELSE access := denied ; result := Ko
  END
END

```

Figure 4.19: The concrete specification of Machine *Secure_Filter*

Figure 4.20 gives the global architecture of the B development. It shows how, in a first step, different aspects of the system are separately modeled. These parts are then combined to coordinate them and ensure the correctness of the global specification.

4.5 Verification and validation

To verify the correctness of the obtained B specification, a set of proof obligations have been generated using the Proof Obligations Generator of AtelierB (GOP). Table 4.1 gives the statistics of the proof phase where:

1. *PO*: denotes the number of proof obligations generated for each machine,
2. *AutoDischarged PO*: denotes the number of proof obligations automatically discharged by the provers of AtelierB without any human intervention,

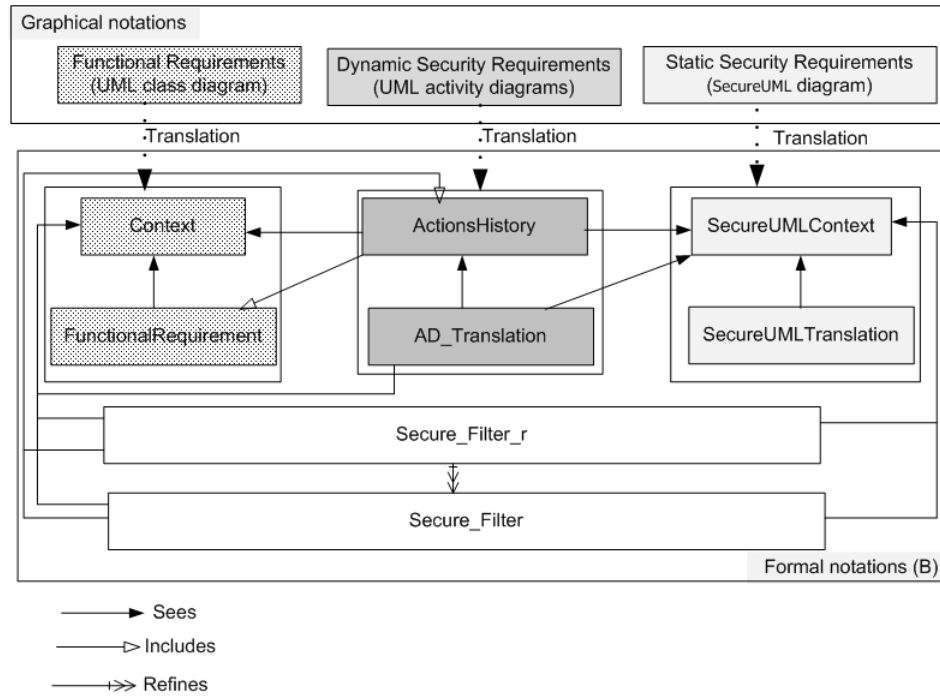


Figure 4.20: The architecture of the B specification

3. *PO InterDischarged*: denotes the number of proof obligations interactively discharged. The intervention of the user is necessary for some proof obligations to help the prover find the rules to apply to discharge them. For the running case study, the prover fails to automatically discharge only one proof obligation (for the machine *SecureUMLTranslation*) that we have interactively proved.

Let us note that these proof obligations ensure that all the properties expressed as invariants are satisfied and the development of the filter by refinement is also correct. However, we need to check that the dynamic security rules, we have specified using UML activity diagrams, are also verified. For that purpose, we animate the specification using the ProB [96] animator by applying several scenarios like the following one to validate Rule 4:

- (1) a deposit for a cheque *ch* is performed in a bank *bk*;
- (2) a new user *usr* joins Bank *bk*;
- (3) User *usr* connects to the system as a teller (his/her current role is *TellerRole*);

	PO	PO AutoDischarged	PO InterDischarged
<i>Context</i>	0	0	0
<i>FunctionalRequirement</i>	7	7	0
<i>SecureUMLContext</i>	0	0	0
<i>SecureUMLTranslation</i>	4	3	1
<i>ActionsHistory</i>	9	9	0
<i>AD_Translation</i>	2	2	0
<i>SecureFilter</i>	1	1	0
<i>SecureFilter_r</i>	6	6	0

Table 4.1: Results of the proof phase

(4) the user *usr* tries to validate Cheque *ch*.

As expected, the last action fails since the access is denied. Indeed, User *usr* joins the bank after the creation of the deposit. So, we have reversed actions (1) and (2) and in that case the cheque has been successfully validated.

4.6 Conclusion

In this chapter, we present a formal approach to integrate functional and security requirements from the first design stages in the domain of Information Systems. The approach is based on the modeling of the different aspects of the system using UML-based notations, namely a class diagram to describe the structure of the data, **SecureUML** to model static security rules and adapted UML activity diagrams to deal with dynamic ones. Even if it is well established that graphical notations are in favor of an intuitive and visual view of systems, there is no tool that would permit to validate them. This is why we define a set of formal translation rules to map them into a B specification on which verification/validation tools are used to ensure their correctness and consistency. The use of a unifying language (B in our case) also permits to coordinate all the requirements specified in different diagrams by designing a secure filter that checks all these requirements before giving access to the resources of the system.

To make our approach workable, we have developed a tool that automates all the translation rules and also the design of the filter. The tool has been implemented as an eclipse plugin and uses the TOPCASED environment in order to edit the different UML diagrams. As a next step, we will define refinement rules to deploy the application as a relational application.

A Tool for the Generation of a Secure Access Control Filter

The contents of this chapter are reproduced from our paper [97]. My contribution in this paper is the development of the tool that generates the B specification for a secure information system from its different UML-based models. The paper is submitted and accepted in the 10th IEEE International Conference on Research Challenges in Information Science took place in Grenoble, France on 1st-3rd June 2016.

Contents

5.1	Introduction	112
5.2	Overview of the tool	113
5.3	Overview of the B method	115
5.4	Graphical modeling of the application: case study	115
5.5	From graphical diagrams to B formal notations	121
5.5.1	Translation of the class diagram	121
5.5.2	Translation of the SECUREUML diagram	123
5.5.3	Translation of the secure UML activity diagram	124
5.6	The B specification of a secure filter	126
5.7	Conclusion	127

Currently, it is well recognized that coupling graphical and formal notations offers several advantages. Indeed, even if a graphical representation permits to

design a visual, synthetic and user-friendly view of the system, it may be source of ambiguity and does not permit any formal verification. Formal methods help to remedy these shortcomings by giving a precise semantics to graphical notations such that it becomes possible to verify a large range of properties and even to generate correct implementations. Nevertheless, users cannot take a full advantage of the benefits of such a combination if it is not supported by an automatic tool that liberates them from the tedious translation activity. Following this direction, the present chapter describes the main functionalities of a tool that automatically generates a formal secure access control filter for information systems. The goal of the filter is to regulate the access to data of an information system according to a set of static and dynamic rules. Data are described using a UML class diagram, whereas the static and dynamic rules are modeled using SECUREUML and UML activity diagrams respectively. Basically, the tool automatically generates the B formal specification corresponding to these diagrams and the filter.

5.1 Introduction

Despite the well-recognized advantages that formal methods offer in terms of precision and possibilities of correctness proof, their use are rather restricted to critical systems involving human lives. The developer's reluctance to use this kind of methods can be explained (partly) by the strong mathematical skills they require especially if formal specifications have to be written by hand. Moreover even if there is no human risk, some systems, like information systems, need to be secure. Indeed, an information system often interacts with humans or other systems by exchanging information and any security breach may cause serious and even irreversible consequences. The present chapter describes a tool, in support of the use of formal methods that automatically generates a secure filter for information systems. A secure filter permits to restrict access to a system only to the allowed users according to a given security policy. Roughly speaking, a security rule specifies, for an authenticated user, which actions are allowed/forbidden according to his/her current role and context. These security rules range from static rules related to a single system's state to dynamic ones involving its execution history. Furthermore, functional requirements should be also taken into account.

To develop a secure filter for information systems, we adopted a coupling-based approach that combines formal notations and graphical ones in order to take advantage of both notations as demonstrated in several previous research work [85, 98–101]. On the one hand, graphical notations permit to design in-

tuitive and visual models and, on the other hand, formal methods bring the precision and reasoning possibilities that are missing in graphical notations. Basically, we use a UML class diagram to describe the data of a system together with its functional requirements, SECUREUML and UML activity diagrams to describe the static and dynamic rules respectively. Then a set of transformation rules permits to translate these diagrams into a B specification. The goal of the current chapter is to present a tool that automates such transformations in order to make our approach workable and free software designers from a tedious and error-prone activity.

The sequel of the chapter is structured as follows. The next section gives an overview of the functionalities of the tool. Section 5.3 briefly introduces the B method. The modeling of the different diagrams is presented in Section 5.4 throughout a case study. The B specification produced by the tool on these diagrams is presented in Section 5.5. In Section 5.6, we describe how the generated B specification is combined to generate a secure filter that regulates the access to information of a system by given access exclusively to authorized users. Finally, we conclude and present some future work.

5.2 Overview of the tool

A team from the French LIG laboratory has developed an Eclipse platform tool, called B4MSECURE [102], to extract B specifications from functional UML models enhanced by an RBAC access control policy [82]. With this tool data are described by a class diagram and static secure rules by a SECUREUML diagram. However, the following weaknesses can be raised:

1. translation into B specifications of the UML association class concept is not supported.
2. translation of SECUREUML diagrams follows an offensive style which may cause security issues if the adequate conditions are not fulfilled.
3. dynamic security rules are not considered at all.

The present chapter suggests to extend this tool by adapting certain translation rules and introducing new ones in order to overcome the above limitations. Figure 5.1 depicts the workflow followed by the extended tool to generate a secure filter from UML notation-based diagrams describing the data and the security rules of a system. Mainly, four phases are distinguished:

1. constructing the class diagram to describe the data and the functional requirements of a system.
2. generating a B specification from the previous class diagram.
3. constructing the SECUREUML and activity diagrams that model both the static and dynamic security constraints; the tool checks the consistency of the SECUREUML and activity diagrams with respect to the previous B specification, that is, each operation used in such diagrams must refer to a B operation of the class diagram.
4. generating a B specification from the previously checked diagrams.

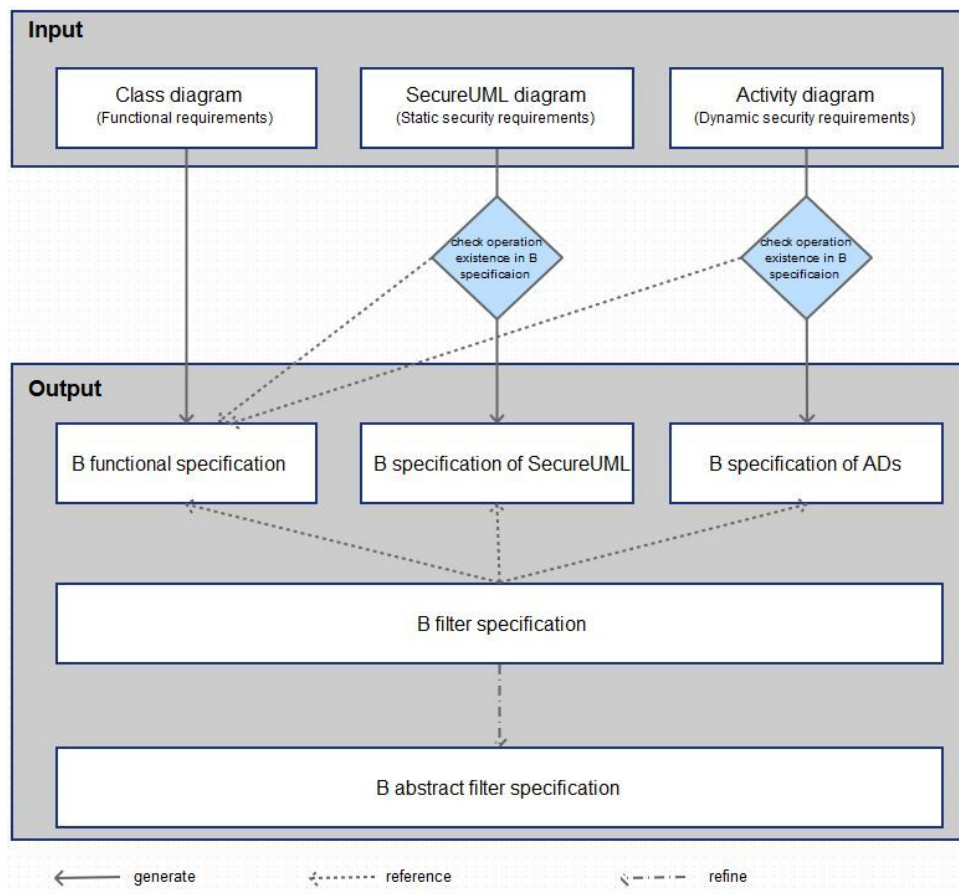


Figure 5.1: Translation workflow

The rest of the chapter illustrates the use of the tool throughout a case study presented in the next section.

5.3 Overview of the B method

Introduced by Jean-Raymond Abrial [78], B is a formal method dedicated to developing safe systems. B specifications are organized in abstract machines. Each machine contains state variables on which operations act, and an invariant constraining the variables. Operations are specified in the Generalized Substitution Language. A substitution is like an assignment statement. It allows us to identify which variables are modified by an operation without mentioning the variables not modified. A preconditioned substitution is of the form (**PRE** P **THEN** S **END**) where P is a predicate, and S a substitution. When P holds, the substitution is executed, otherwise nothing can be ensured. For instance, the substitution S might not terminate or might violate the invariant. The B refinement is the process of transforming, by successive steps, a specification (variables and operations) into a less abstract one. The last refinement step, called implementation, aims at obtaining data and substitutions close to those of a programming language such that the translation into the chosen target language of the data and control structures (used in this level) must be a straightforward task. Both specification and refinement steps give rise to proof obligations. At the abstract level, proof obligations ensure that each operation maintains the invariant of the system, whereas at the refinement level, they ensure that the transformation preserves the properties of the abstract level.

5.4 Graphical modeling of the application: case study

To illustrate the use of the tool to generate a secure filter, we consider a case study, from the medical domain, that deals with a set of hospitals where patients can be treated by doctors. Each patient, identified by his/her social security number, is described by an address and has a unique medical record that contains information related to the patient. Medical records are shared by all hospitals and are thus independent from any hospital. Patients may make several stays in different hospitals. Each hospital stay is stored with its entry date. During these stays, patients receive treatments by doctors. Figure 5.2 depicts the edition of the class diagram under the B4MSECURE platform. As we can remark, unfortunately, the parameters of the operations of an association class cannot be displayed; they can be found in the outline menu of the associated class association (See Figure 5.2).

When a patient arrives in a hospital, a secretary registers his/her admission

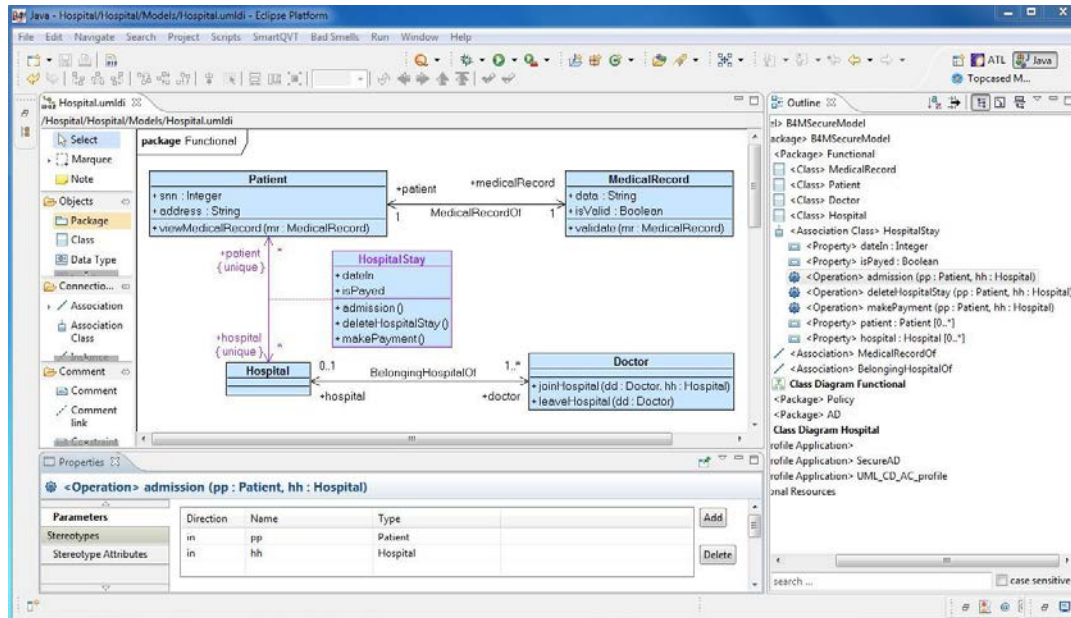


Figure 5.2: Editing a class diagram under the B4MSECURE platform

by storing his/her arrival date. During his/her stay, the patient undergoes medical examinations whose results are reported in his/her medical record. Such stored data should be validated by a doctor before the patient discharges. This arises the following security rules:

- Rule 1** Only Secretaries are authorized to make the admission (resp. make payment) of a patient (operations *admission/makePayment*),
- Rule 2** The admission and payment processes of each hospital stay should be made by the same secretary,
- Rule 3** Only Doctors are authorized to validate medical records of patients (operation *validate*),
- Rule 4** To validate a medical record, the doctor should be present, in the hospital, since the arrival time of the patient.

Analyzing the above rules gives:

- (i) Rules 1 and 3 specify which roles the user must play to perform the corresponding actions. Such security rules are static since they do not depend on the past or/and the future execution of the system.

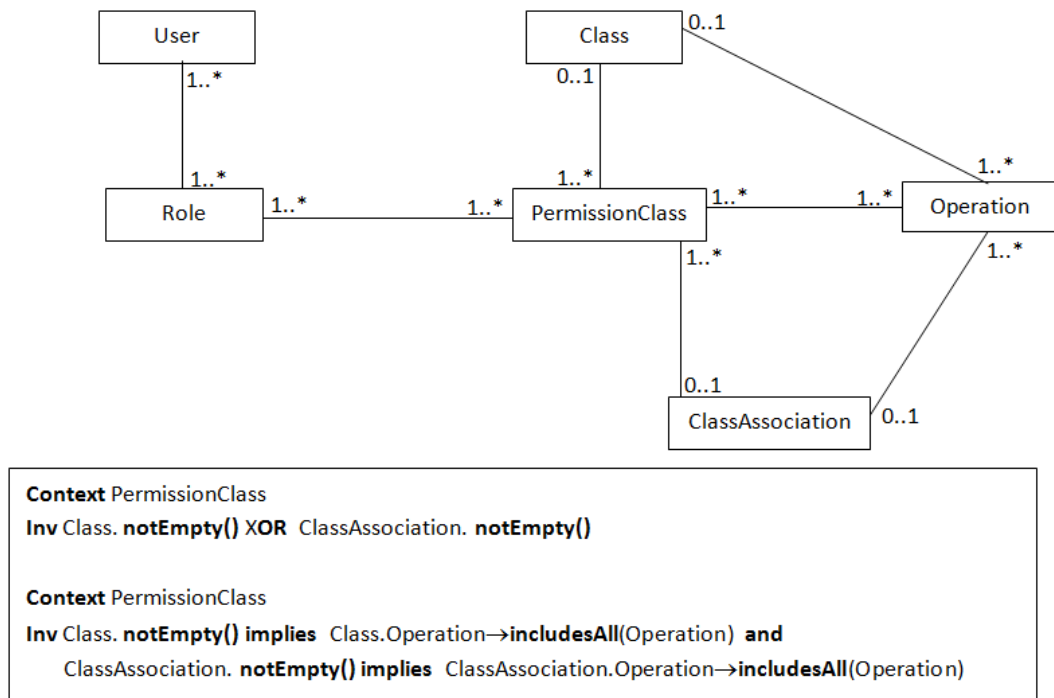


Figure 5.3: The SecureUML metamodel (adapted from [17])

- (ii) Rules 2 and 4 are dynamic rules since the permission of executing an action depends on an action executed in the past.

To give a visual representation of these rules, we describe static security rules with a SecureUML diagram whereas dynamic ones are modeled with an adapted version of UML activity diagrams. Roughly speaking, SecureUML allows to extend a UML functional model with concepts of role-based access control models (RBAC [82]) in order to specify the different roles that a user can play and their associated permissions on the resources of the system.

Figure 5.3 shows the metamodel used by the tool for the SecureUML models. Basically, a SecureUML model consists of a set of roles. Each role may be played by several users; each user can play several roles. Each role is associated with one to several permission classes. Each permission class is related to either a class or a class association: expressed by the UML multiplicities and the first OCL constraint. Also, each permission class concerns one to several operations of the related class (resp. class association): expressed by the UML multiplicities and the second OCL constraint.

According to this metamodel, Figure 5.4 depicts the SecureUML model of the

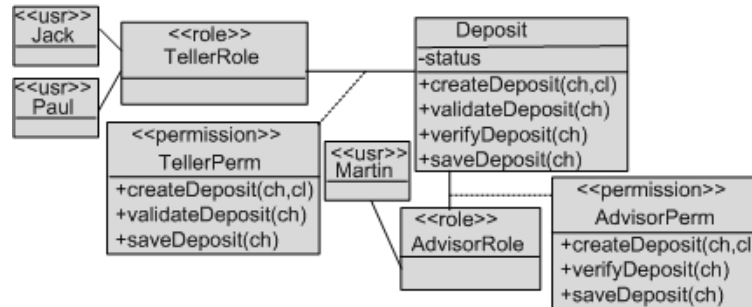


Figure 5.4: Editing a SecureUML diagram under the B4MSECURE platform

case study. Two roles *DoctorRole* and *SecretaryRole* are defined with their respective permissions: a user with the role *DoctorRole* (resp. *SecretaryRole*) has the permission to validate a medical record (resp. to make admissions/delete-HospitalStay/payments). Finally, specific users are attached to roles to state that they can play it: for instance, the users *Mary* and *Jack* (resp. *Paul* and *Bob*) can play the role *SecretaryRole* (resp. *DoctorRole*); thus they can execute all the actions permitted to the related role.

As stated before, dynamic security rules are modeled using specialized UML activity diagrams. Recall that UML activity diagrams are used to describe dynamic aspects of systems by depicting the control flow from one activity to another as sequencing, branching, parallel flow, swimlanes, etc. To use them to model dynamic security rules, we make some assumptions on these diagrams. Indeed, we consider that activities, appearing in a secure activity diagram, refer to the operations of classes/associations, whereas swimlanes denote the actors executing them. Moreover, a dynamic security rule is modeled by a partial activity diagram to show, on the one hand, the execution order of the operations and, on the other hand, the constraints on the users executing them. To do that, the tool is based on the secure UML activity metamodel depicted in Figure 5.5. A secure activity diagram is composed of a set of operations that may be designed in swimlanes to state the user that should execute them. An operation may be preceded/followed by another operation. As for SECUREUML diagrams, each operation must refer to either an operation of a class or a class association. Finally, the last OCL constraint imposes a single secured operation for each secure activity diagram.

Basically, Figures 5.6 and 5.7 are interpreted as follows:

1. each of these diagrams has a unique operation to which a stereotype `<<Secure>>` is attached. This means that any execution constraints will be put on this operation: Operations *makePayment* and *validate* are to

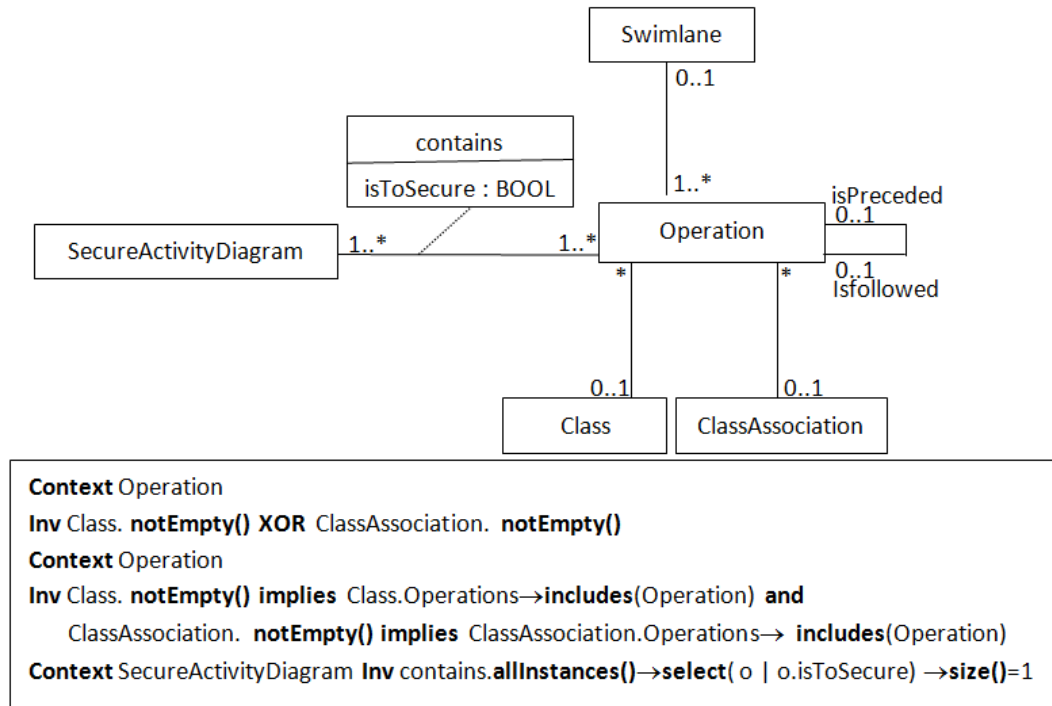


Figure 5.5: The secure UML activity metamodel

be secure, that is additional conditions should be verified before executing them,

2. all the operations occurring before the operation to secure have to be executed before it: the operation *makePayment* cannot be executed before the operation *admission*. Also, to validate a medical record, the doctor must have joined the hospital before the patient's arrival,
3. the operation that should not happen before the operation to secure is placed after it using a control flow: a doctor cannot validate a medical record after leaving the hospital.
4. if the users that perform the operations Op_i and Op_k should be different (resp. the same), then we put both operations in different (resp. the same) swimlanes: the operations *admission* and *makePayment* should be executed by the same secretary.

Under the tool, it is not possible to specify expressions as input parameters of an activity. Indeed in Figure 5.7, we would like to specify that the input

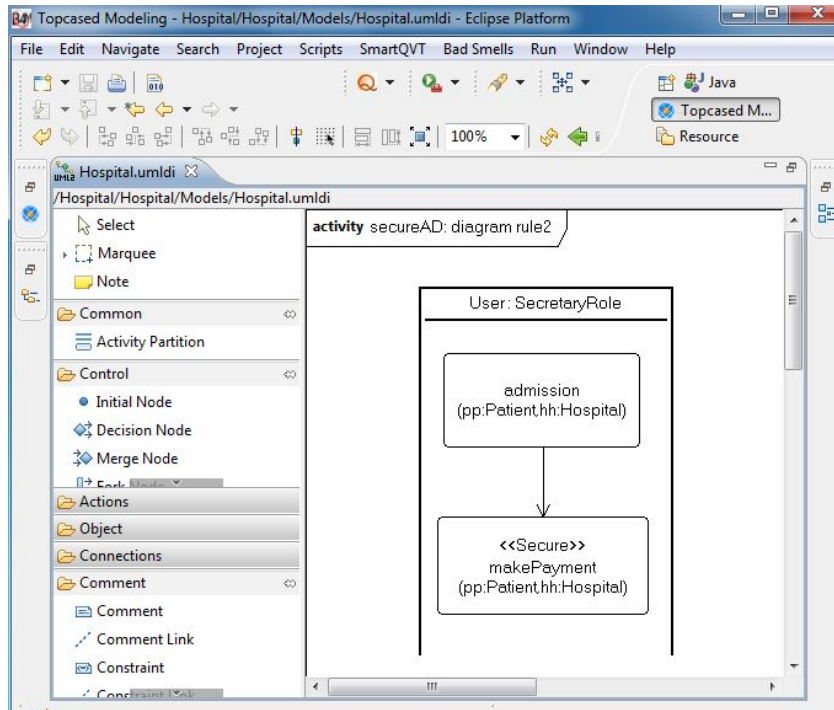


Figure 5.6: Example of dynamic security rules modeled by activity diagrams: Rule 2

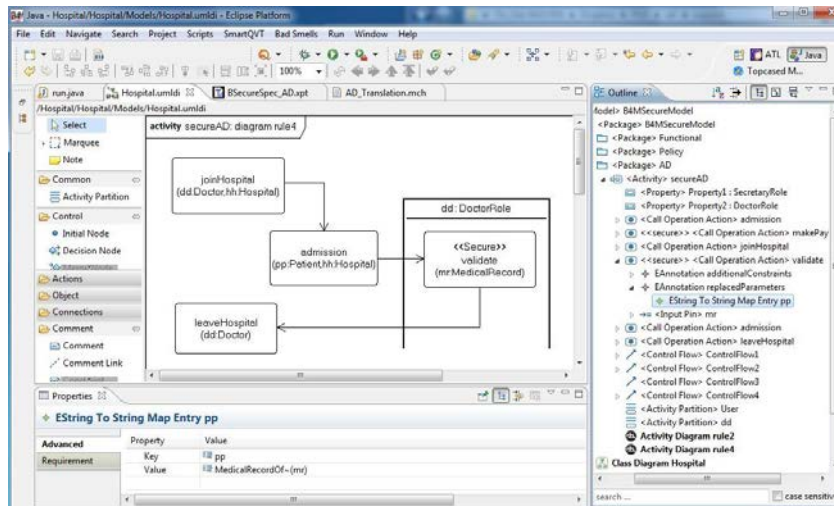


Figure 5.7: Example of dynamic security rules modeled by activity diagrams: Rule 4

parameter mr denotes the medical record of the patient pp . To do that, we use the annotation facility to introduce such information in the diagram. As illustrated by Figure 5.7, the common parameters appearing in the different activities denote the same object that may be a value of a given type, a user of a given role or/and a object of a given class.

5.5 From graphical diagrams to B formal notations

In this section, we describe the different steps followed by our tool to map the previous graphical diagrams into a B formal specification. The translation proceeds in three steps corresponding to the different diagrams. A detailed description of the translation rules can be found in [103].

5.5.1 Translation of the class diagram

To translate a class diagram into a B specification, the tool follows mainly the rules defined in [90, 91]. For each class C , we define a given set S_C and a variable V_C to respectively represent the set of all possible instances and the existing ones at each moment. Each attribute is mapped into a function from V_C to its type, the function modeling the key of the class is an injection. An association class involving two classes C_1 and C_2 is mapped into a B variable defined as a relation between V_{C_1} and V_{C_2} . According to its multiplicities, this association becomes a partial function (\rightarrow), a total function (\rightarrow), an injection (\rightarrow), etc. Initially, the tool does not adequately support the B translation of associations with attributes (called association class): it is translated like a class and its attributes are completely ignored. So we have extended it with the previous rules by considering the attributes of an association class like those of a classical class. To ease the integration of the security rules, this B specification is structured as follows: all the derived given sets are included into a machine called *Context* that can be seen (Clause **SEES**) by any machine to have read access to them. Variables and operations are defined in a separate machine that sees the machine *Context*. Figure 5.8 shows a part of the B specification generated for the case study.

The machine *Functional_Requirement* specifies a set of operations of the classes/associations. Such operations permit, for instance, to add/delete an instance and also to update the attribute values. Contrary to the translation rules defined in [90, 91], a defensive strategy is adopted in the present chapter for the

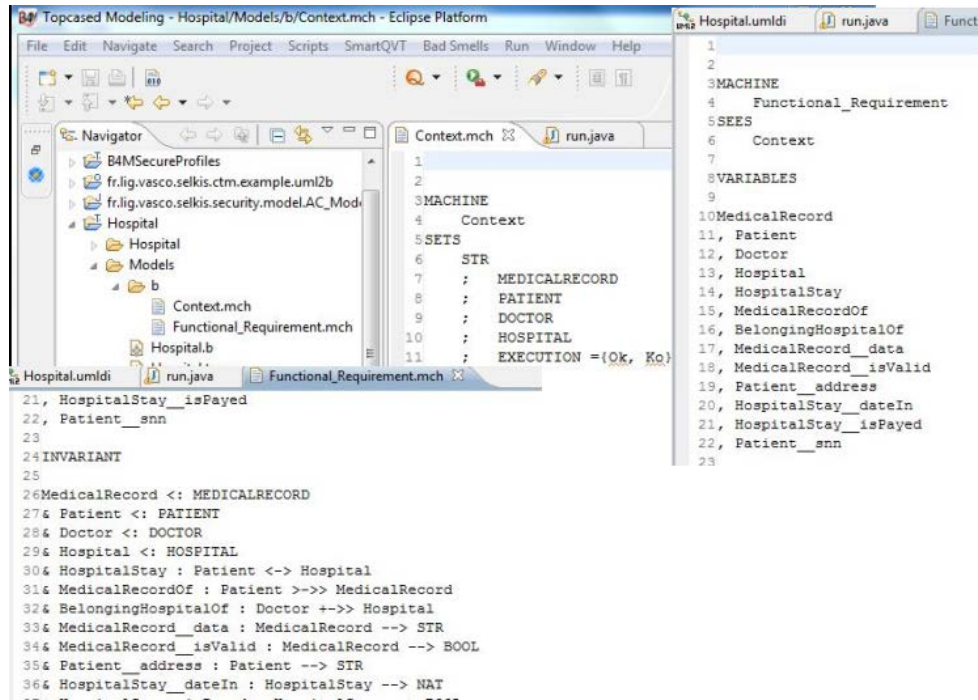
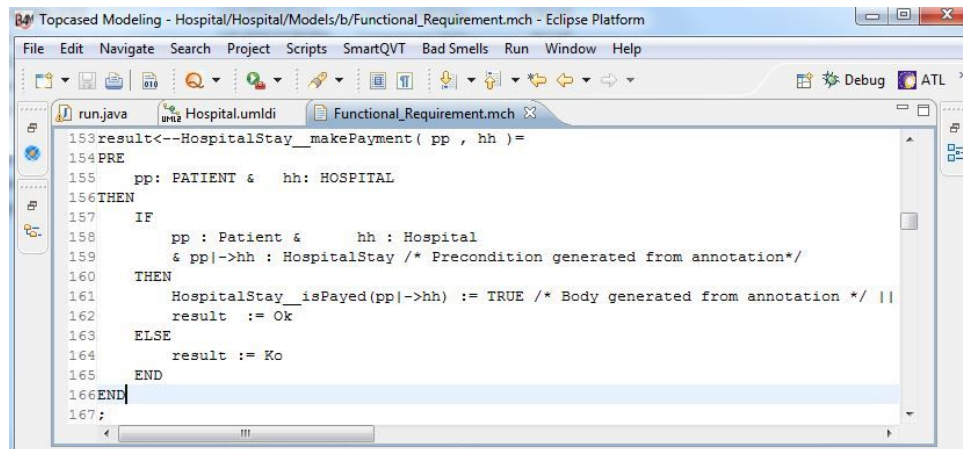


Figure 5.8: Translation of the class diagram

Figure 5.9: Generation of the B operation *makePayment* of the class *HospitalStay*

generation of these operations since we take all the possible parameter values into account. Basically, only the type of the input parameters is assumed in the precondition of an operation, all the other conditions required for its correct execution are checked in its body using an **IF** substitution: if the conditions are fulfilled then the state of the system evolves otherwise it remains unchangeable. In both cases, a value is returned to inform the user about the result of the execution of the operation. Figure 5.9 depicts the B specification of the operation *makePayment* defined on the association class *HospitalStay*. This operation assumes the type of its input parameter ($pp \in PATIENT \wedge hh \in HOSPITAL$), then it verifies if the related hospital stay exists: if so, the status *isPayed* is updated to *TRUE* and the user is informed that the state of the system has been updated ($result := Ok$). Otherwise, nothing is done and the value *Ko* is returned. Of course, the substitution that updates the variables is included as annotations for the operation in the class diagram.

5.5.2 Translation of the SECUREUML diagram

The B4MSECURE tool implements a translation of a SECUREUML diagram into B. Basically, each operation *Op* of the class diagram to which a permission is granted to a role *R*, an operation *SecureOp* is defined. Compared with *Op*, this operation has an additional parameter that denotes the user executing it. The body of the operation consists in executing *Op* under the precondition that the related user has the role *R*. However, this is not faithful to the semantics of the SECUREUML diagram that only states which role can execute a given operation but does not mean the actual execution of the operation. Indeed, the actual execution of the operation can require other conditions like dynamic security rules. Moreover, since the permission of the role to execute the operation *Op* is checked in the precondition, the user still can call the operation even if his/her role is not allowed to do that! This is why we have implemented our own rules as follows.

First, we define an additional context to model the different roles, the permissions granted to them but also the users associated to each role (See Figure 5.10). The context being defined, this latter is seen by a new machine that translates each operation *SecureOp* as follows: the operation has a single parameter denoting the user who wants to execute *Op*. The precondition of *SecureOp* simply assumes the right type of the parameter, then checks if the current role of the user permits to execute *Op*. If so, the operation returns *granted* otherwise *denied* is returned. Figure 5.11 depicts the B specification of the operation *SecuremakePayment*. Of course, we have specified another operation that per-

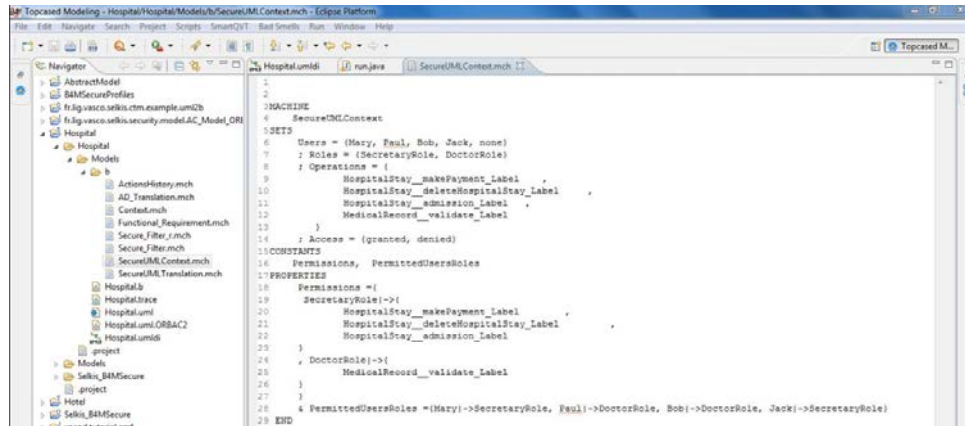


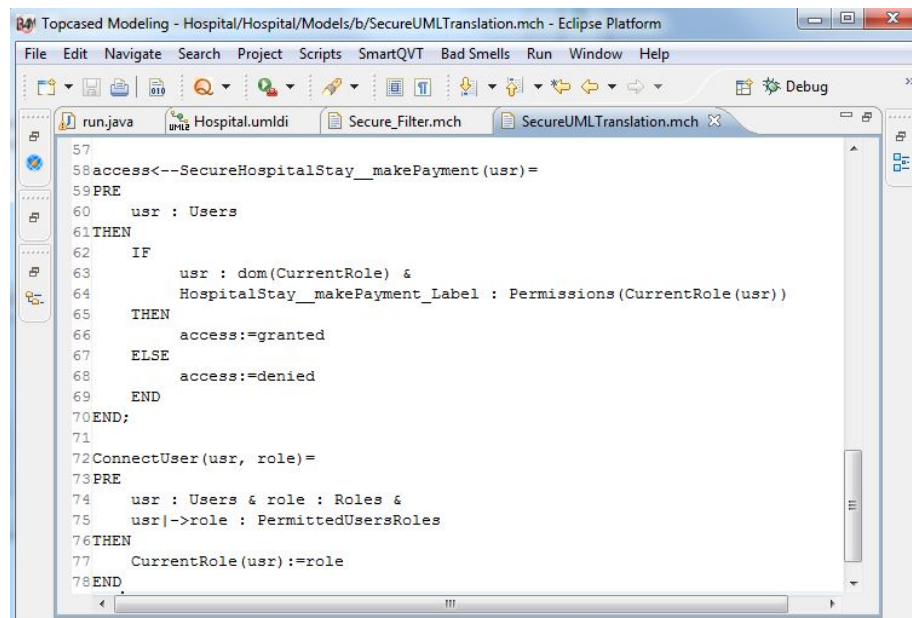
Figure 5.10: The B context machine of the SecureUML diagram of Figure 5.4

mits for a user to play a given role.

5.5.3 Translation of the secure UML activity diagram

As illustrated through the case study, a secure UML activity diagram states constraints on the execution order of some operations. This is why we need to store the moment when each operation occurs. Thus for each operation $op(p_1, \dots, p_n)$, we define two additional variables $historyOp$ and $orderExecutionOp$ that store its corresponding execution occurrences together with the user that executes it and the execution time. Also, we define a new operation $ADOp$ that has the same parameters as Op plus a new parameter usr ; it checks the same functional constraints (using the **IF** substitution). If the functional constraints required for the execution of Op are fulfilled, it calls the operation Op and updates the variables $historyOp$ and $orderExecutionOp$ using the value of the clock of the system $currentOrder$. This B specification is put in a new machine $ActionsHistory$ that includes (**INCLUDES** of B) the machine $Functional_Requirement$ in order to make the operation calls possible. Figure 5.12 depicts such a specification for the operation $makePayment$.

The translation of each secure UML activity diagram produces a single B operation $ADOp$ that corresponds to the operation to secure Op . This operation has as parameters all the parameters appearing in the different operations of the activity diagram together with the possible actors appearing in the swimlanes to state who executes the operations. In the precondition, we check the typing of the parameters, the body of the operation consists in returning the execution

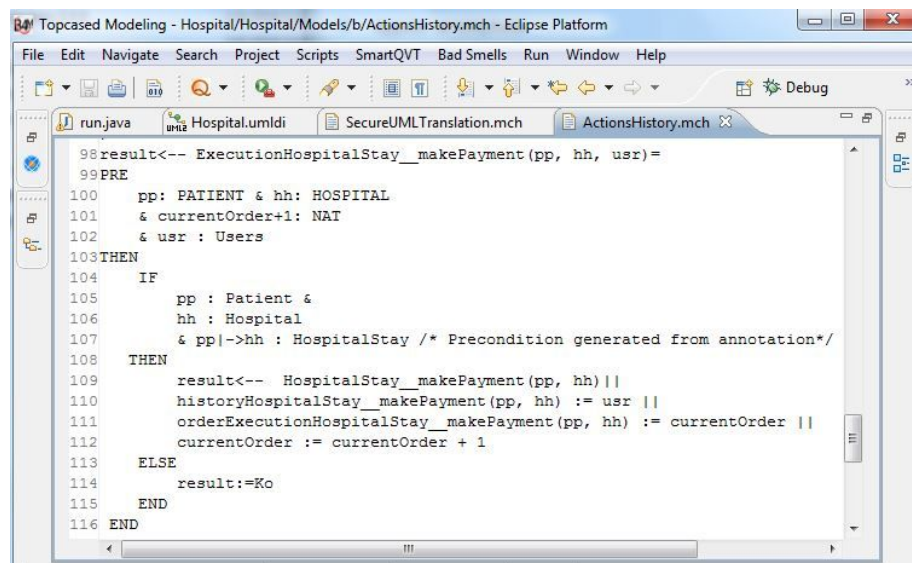


```

57
58 access<--SecureHospitalStay__makePayment(usr)=
59 PRE
60   usr : Users
61 THEN
62   IF
63     usr : dom(CurrentRole) &
64     HospitalStay__makePayment_Label : Permissions(CurrentRole(usr))
65   THEN
66     access:=granted
67   ELSE
68     access:=denied
69   END
70 END;
71
72 ConnectUser(usr, role)=
73 PRE
74   usr : Users & role : Roles &
75   usr|->role : PermittedUsersRoles
76 THEN
77   CurrentRole(usr) :=role
78 END

```

Figure 5.11: The B modeling of the static secure operation corresponding to the operation *makePayment* of Figure 5.4



```

98 result<-- ExecutionHospitalStay__makePayment(pp, hh, usr)=
99 PRE
100   pp: PATIENT & hh: HOSPITAL
101   & currentOrder+1: NAT
102   & usr : Users
103 THEN
104   IF
105     pp : Patient &
106     hh : Hospital
107     & pp|->hh : HospitalStay /* Precondition generated from annotation*/
108   THEN
109     result<-- HospitalStay__makePayment(pp, hh)||
110     historyHospitalStay__makePayment(pp, hh) := usr ||
111     orderExecutionHospitalStay__makePayment(pp, hh) := currentOrder ||
112     currentOrder := currentOrder + 1
113   ELSE
114     result:=Ko
115   END
116 END

```

Figure 5.12: The B specification of the history execution of the operation *makePayment*

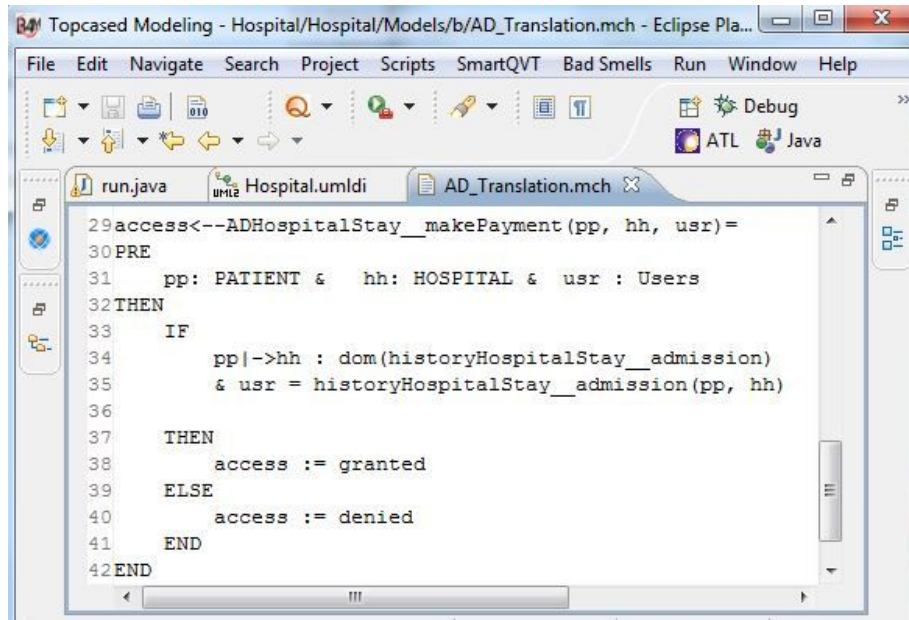


Figure 5.13: The B translation of the activity diagram of Figure 5.6

result of the corresponding operation Op if the other operations of the diagram are executed in the right order. Otherwise, Ko is returned. Figure 5.13 depicts the operation generated for the secure operation *makePayment* of the activity diagram of Figure 5.6.

5.6 The B specification of a secure filter

As we can remark, the B specification we have built makes a separation between functional and static/dynamic security modeling. This is why an additional step is needed to put all these parts together in order to take a decision about the permission/prohibition of an access to a specific user. Indeed among the operations we have derived right now, none of them takes all the constraints into account. Thus, we have to specify a new operation that permits/denies a user to execute an operation by taking all the security/functional constraints into account. We call such an operation a *filter* which we develop using the B refinement.

The filter is defined in a new machine *Secure_Filter* that sees both *Context* and *SecureUMLContext*. It also includes *ActionsHistory*. At the abstract level, the filter defines an operation *FilterOp* for each operation Op defined in a class

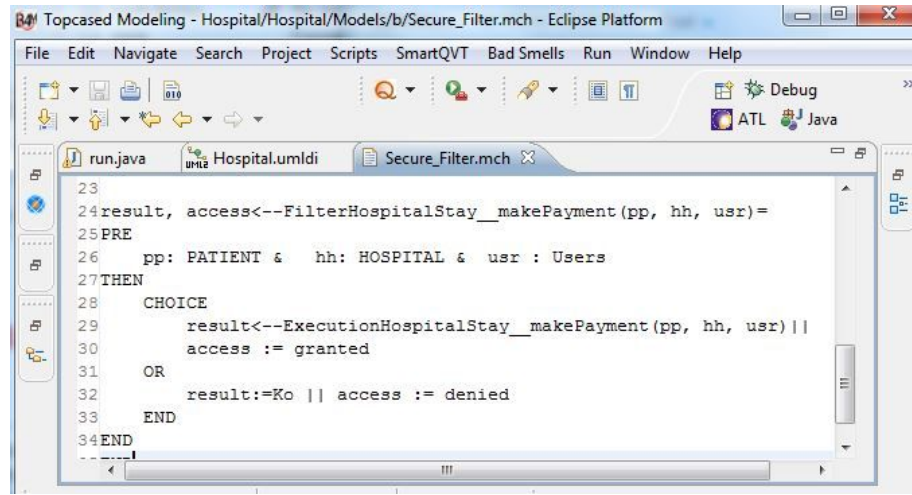


Figure 5.14: The B specification of the filter for the operation *makePayment*

or an association class. Compared with the operation *Op*, this operation has an additional parameter *usr* to denote the user who is invoking *Op*. It returns two values *access* and *result*: (*access = granted*) means that all the static/dynamic security rules are fulfilled, otherwise *denied* is returned; *result = Ok* means that the operation *Op* has been successfully executed, otherwise *Ko* is returned (See Figure 5.14).

At the abstract level, the filter states that when Operation *makePayment* is invoked by a user, two options are possible: either the access is denied and the operation is not called at all (*result = Ko*) or the access is granted and the result is equal to what the operation call returns. More precisely, the result depends on the functional requirements. At this level, we do not specify how the access rights are defined for a user. It is the purpose of the refinement that details the steps to follow to do that.

To give access to the resources of the system, the filter has to check both static and dynamic security rules. If both rules are satisfied, the access is granted and the user can safely invoke the operation, otherwise the access is denied and nothing is done (See Figure 5.15).

5.7 Conclusion

This chapter describes the tool, we have developed, to support the formal generation of a secure access control filter for information systems. It implements

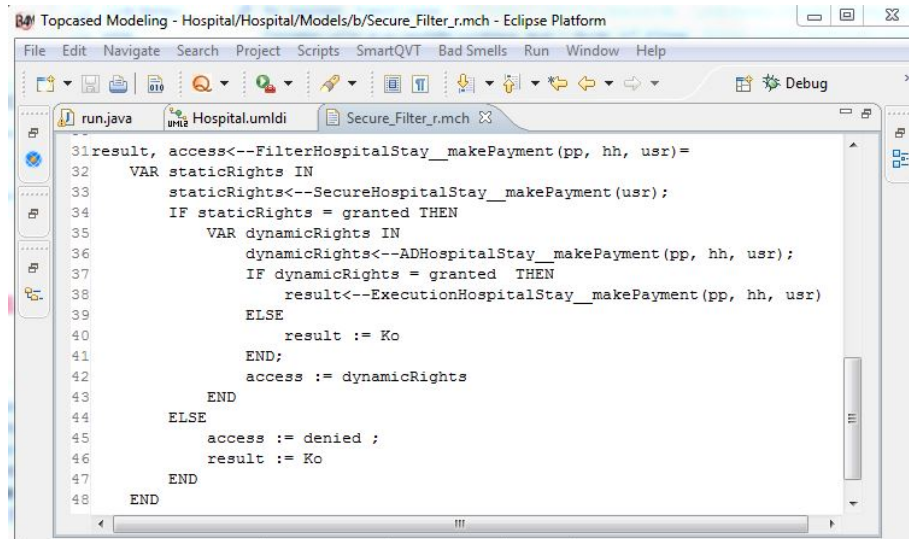


Figure 5.15: The concrete specification of the filter of the operation *makePayment*

translation rules that produce a B formal specification from the UML modeling of the functionalities and the security rules of a system. Basically, data are represented with a class diagram whereas the static and dynamic security rules are modeled by SECUREUML and specialized UML activity diagrams. The tool is developed in the JAVA language and uses the TOPCASED framework for the construction of the different UML diagrams. We did not build the tool from scratch but we have extended/adapted the B4MSECURE tool built by a team from the French LIG laboratory. In fact, we have augmented the tool by translation rules to take class associations into account. We have also adapted the translation rules related to the SECUREUML diagrams by adopting a defensive style mapping into a B specification. Finally, the modeling and the translation to B of secure activity diagrams have been introduced to support secure dynamic rules. The development of the tool took us five months. This time includes the initiation to the tool whereas no documentation is available. Indeed, the extension of the tool required a careful study of the code in order to extend it. But this is compensated by the significant time-saving for users when automatically generating several secure filters. Indeed, the generated B specification is correctly checked using AtelierB [80] without any additional modification. Moreover, most proofs are automatically discharged: the abstract and the refinement specification of the filter are automatically proved.

As a next step, we plan to extend the tool to support more UML concepts like inheritance and aggregation/composition. We would like also to define refinement rules to generate an implementation in a relational database environment.

A Formal Approach to Derive an AOP-Based Implementation of a Secure Access Control Filter

Contents

6.1	Introduction	132
6.2	The case study: a purchase order system	134
6.3	A formal B specification of a secure filter	136
6.3.1	Translation of the class diagram into a B specification	136
6.3.2	Translation of the SecureUML diagram into a B specification	137
6.3.3	Translation of the secure UML activity diagrams into a B specification	139
6.3.4	Designing the secure filter	140
6.4	From an abstract B specification to a relational-like B implementation	141
6.4.1	Data refinement	141
6.4.2	Behavioral refinement	143
6.5	The AspectJ implementation of the application	146
6.5.1	Transformation rules of B into JAVA/SQL	148
6.5.2	Deployment of the class diagram	151
6.5.2.1	Definition of the tables and the associated JAVA methods and stored procedures	153
6.5.2.2	Translation of the operations of the class diagram	154
6.5.3	Deployment of the SecureUML diagram	155

6.5.4	Deployment of the secure activity diagrams	158
6.5.4.1	Definition of the log tables and the associated JAVA classes	158
6.5.4.2	Translation of the secure operations of the secure activity diagrams	160
6.5.4.3	Translation of the log operations	161
6.5.5	Deployment of the filter	161
6.6	Tool support	163
6.6.1	Translating the B specification of the class diagram	163
6.6.2	Translating the B specification of the SecureUML diagram	164
6.6.3	Translating the B specification of the secure activity diagram	166
6.6.4	Translation of the access control filter	167
6.7	Conclusion	168

This chapter presents a formal approach for the development of a secure filter that regulates the access to sensitive resources of information systems. This approach consists of three complementary steps. Designers start by modeling the functionalities of the system and its security requirements using dedicated UML diagrams. These diagrams are then automatically translated into a B specification suitable not only for reasoning about data integrity checking but also for the derivation of a trustworthy implementation. In a last step, the B implementation is translated into JAVA/SQL code following the aspect oriented programming paradigm, which allows a separation of concerns by making a clear distinction between functional and security aspects. This chapter focuses more on the two last steps by describing the refinement process that permits to obtain a relational-like B implementation and presenting a set of rules to translate it into an AspectJ implementation connected to the SQL Server (release 2014) relational database system.

6.1 Introduction

Ensuring the security of the data part of an information system becomes a necessity, especially because systems are nowadays at the heart of critical applications not necessarily for human risks but at least for economic interests. Indeed, consequences of some security breaches might be irreversible and even

lead to lawsuits. Such a risk increases significantly with the advent of the web through which most organizations offer accesses to their information systems in order to make things easier for their users.

In the previous chapter (Chapter 5), we started to provide an answer for the data security problem by proposing a formal approach for the development of a filter that controls accesses to data of a system according to a set of security rules that specify, for an authenticated user, which actions are allowed/forbidden according to his/her current role and context. The security rules that have been considered range from static rules to dynamic ones. Static rules refer to a single moment of the system whereas dynamic ones require to take the execution history of the system into account, that is the actions already performed in the system in general or by a given user in particular. If we consider the case of a hospital, a static rule will be, for instance, *only a person with the role Doctor can make a diagnosis*, whereas a dynamic rule will be, for example, *the person who performs a laboratory test cannot validate it*. In addition to these kinds of rules, we have also to consider the usual functional constraints like, for instance, the maximum number of patients each doctor can treat. Basically, the approach consists in designing data together with functional requirements using a UML class diagram, the static and dynamic security rules using a **SecureUML** [77] diagram and dedicated UML activity diagrams respectively. These diagrams are then mapped into a formal specification for verification purpose. We have chosen the B method [78] because of its powerful support tools (provers, animators, etc.). As some constraints on an operation execution may be specified in different diagrams, a filter is built to coordinate all these constraints and to make possible the execution of the operation only if they are all fulfilled.

Even if the obtained B specification permits to verify some properties and also to simulate the system to build, it can neither be executed nor straightforwardly translated into an executable code. This is why a refinement step is required to bridge the gap between the B abstract level and the chosen target implementation. We have already defined such a process for the development of trustworthy database applications that satisfy integrity constraints [104]. The approach presented here extends this previous work with the following contributions:

- a set of mapping rules that translates the B specification corresponding to the translation of the **SecureUML** diagram into a set of SQL orders. The targeted database management system is SQL Server, release 2014.
- a set of mapping rules that generates an **AspectJ** implementation from the refined B specification: this includes the definition of the SQL orders

to create the database, the roles, the users, and the execution rights but also the JAVA program corresponding to the operations of the system to build. We choose an aspect programming technique (**AspectJ**) in order to separate the security concerns from the functionalities of the application in such a way that security rules could be added or removed without altering the whole application.

- a tool support that automatically generates an AspectJ implementation from a B specification. This tool extends the tool presented in the previous chapter.

The rest of the chapter is organized as follows. Section 6.2 presents the case study used throughout the chapter to illustrate the proposed approach. Section 6.3 shows how the B specification of a secure filter is built from its UML-based description. Section 6.4 illustrates the refinement of the obtained B specification into relational-like B implementation. Such an implementation is directly translated into an **AspectJ**-based program as described in Section 6.5. The tool that supports this translation is presented in Section 6.6. The last section concludes and presents some future work.

6.2 The case study: a purchase order system

The proposed approach is illustrated through a simplified purchase order system whose class diagram is depicted in Figure 6.1 (the white part). This case study involves two classes *PurchaseOrder* and *Supplier*, which are linked by an association *BelongTo*. Each class is described by a set of attributes and defines some operations to create new instances or delete existing ones. We make the assumption that the first attribute of each class denotes its key. As specified by the multiplicities, each purchase order belongs to one supplier (multiplicity 1), whereas each supplier may have zero to several purchase orders (multiplicity *).

To ensure the security of this process, a set of security rules have been identified. Such security rules may be *static* referring to a given single moment of the system (*i.e.*, the values of the data are taken at the same moment) or *dynamic* requiring to take the execution history of the system into account, that is the actions already performed in the system in general or by a given user in particular. Examples of such rules are as follows:

- Rule 1.** Only **Managers** are authorized to approve a purchase order (operation *approve*)

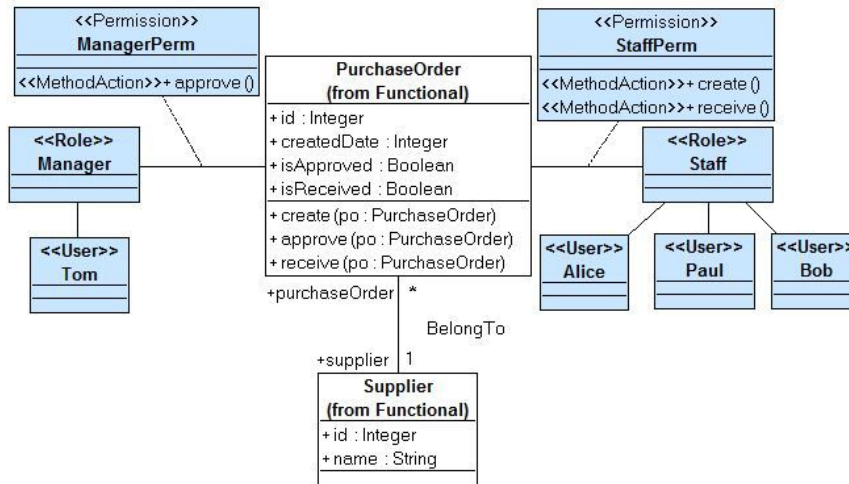


Figure 6.1: The class diagram of a simplified purchase order

Rule 2. Only **Staffs** are permitted to make the creation and the reception of a purchase order (operations *create* and *receive*)

Rule 3. The creation and the reception of a purchase order should be executed by two different persons.

Rules 1 and 2 specify which roles the user must play to perform the corresponding actions. Whereas, Rule 3 denotes a dynamic rule that implies to check that the user, with the role *Staff*, who wants to perform a receive operation on a given purchase order is different from the user, with the role *Staff*, who executed the create operation on the same purchase order. This requires to register this last action and then to consider the execution history of the system. In [76], we have discussed in detail the modeling of such rules using **SecureUML** to deal with static rules and activity diagrams [79] for dynamic ones. Formal rules to translate such diagrams into a B formal specification are also provided.

Figures 6.1 (the grey part) denotes the **SecureUML** diagram associated with the rules 1 and 2. This diagram depicts two classes, stereotyped by `<<Roles>>`, to denote two roles (*Manager* and *Staff*) which can be played by users {Tom} and {Alice, Paul, Bob} respectively. Moreover, a stereotype `<<Permission>>` is attached to an association class linking a class, from the class diagram, to a class representing a role. This association class specifies which operations of the class can be executed by the users playing the related role.

The rule 3 is depicted in Figure 6.2. The activity diagram states that the operations *create* and *receive* should be executed by different users by represent-

ing them in two different swimlanes *usr1* and *usr2*. Also, the activity diagram permits to describe the order in which these different operations are executed.

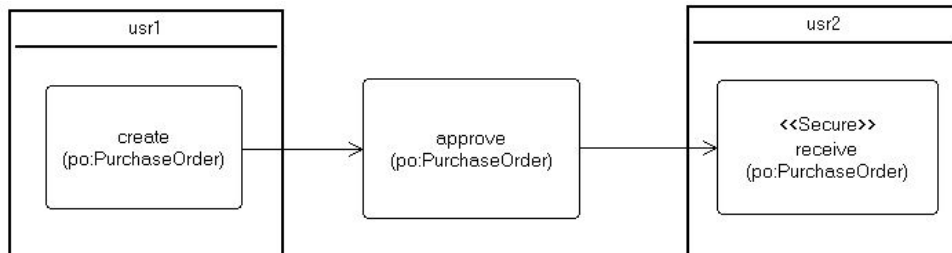


Figure 6.2: A secure activity diagram modeling a dynamic security rule

The next section describes the translation of these diagrams into a B formal specification.

6.3 A formal B specification of a secure filter

The goal of this section is to characterize the B formal specification of a filter that controls access to the information system of the previous case study. This filter is generated from the different previous diagrams and allows to give access to the operations of the system while respecting all the security rules. The generation of the filter is achieved into three steps described in detail in [76].

6.3.1 Translation of the class diagram into a B specification

According to the rules defined in [90, 104], we define an abstract set S_C for each class C to denote the set of all its possible instances, and an enumerated set $Execution = \{OK, KO\}$ to state whether an operation has been executed or not. For each class C , we define a variable V_C to represent the set of the existing instances at each moment. Each attribute is mapped into a function from V_C to its type, the function modeling the key of the class is an injection. An association involving two classes C_1 and C_2 is mapped into a B variable defined as a relation between V_{C_1} and V_{C_2} . According to its multiplicities, this association becomes partial function (\rightarrow), total function (\rightarrow), an injection (\rightarrow), etc. In addition, a set of operations are derived to read/update the data of the system. Such operations check the type of the parameters in the precondition clause, then update the variables and return OK if the parameters satisfy the

functional requirements, otherwise the state of the system remains unchanged and *KO* is returned (See Figure 6.3).

```

SETS
  PURCHASE_ORDER;
  SUPPLIER;
  EXECUTION = {OK, KO}
VARIABLES
  Purchase_Order, id, createdDate, isApproved,
  isReceived, ...
INVARIANT
  Purchase_Order ⊆ PURCHASE_ORDER ∧
  id ∈ Purchase_Order ↦ NAT ∧
  createdDate ∈ Purchase_Order → NAT ∧
  isApproved ∈ Purchase_Order → BOOL ∧
  isReceived ∈ Purchase_Order ↦ BOOL ∧
  ...
  result ← receive(po) ≐
PRE po ∈ PURCHASE_ORDER THEN
  IF po ∈ Purchase_Order ∧ isReceived(po) = FALSE
  THEN
    isReceived(po) := TRUE ||
    result := OK
  ELSE
    result := KO
  END
END

```

Figure 6.3: The B specification of the class diagram

6.3.2 Translation of the SecureUML diagram into a B specification

From the SecureUML diagram, we generate a set of abstract sets to describe the static part of the diagram, that are, the roles and its associated users, the operations and the permissions of the roles to execute the operations. In addition, a variable, called *Session*, is added to memorize the current role of each connected user. Two operations are specified: *ConnectUser* to permit a user to connect to the system and a single generic operation, *checkUserPermission*, that

takes a user and the name of an operation as parameters and returns *granted* if the user has the right to execute the operation and *denied* otherwise (See Figure 6.4)

```

SETS
  Users = {Alice, Bob, Paul, Tom};
  Roles = {Staff, Manager};
  Operations = {create, approve, receive}
CONSTANTS Permissions, UsersRoles
PROPERTIES
  Permissions = {Staff ↦ create, Staff ↦ receive,
    Manager ↦ approve} ∧
  UsersRoles = {Alice ↦ Staff, Bob ↦ Staff,
    Paul ↦ Staff, Tom ↦ Manager}
VARIABLE Session
INVARIANT Session ∈ Users ↔ Roles ∧
  Session ⊆ UsersRoles
OPERATIONS
  connectUser (usr, rol) ≐
  PRE   usr ∈ Users ∧ rol ∈ Roles ∧
    usr ↦ rol ∈ UsersRoles ∧ usr ∉ dom(Session)
  THEN
    Session(usr) := rol
  END;
  access ← checkUserPermission (usr, op) ≐
  PRE   usr ∈ Users ∧ op ∈ Operations THEN
    IF   usr ∈ dom(Session) THEN
      IF   (Session(usr) ↦ op) ∈ Permission THEN
        access := granted
      ELSE   access := denied
      END
    ELSE   access := denied
    END
  END

```

Figure 6.4: The B translation of the SecureUML diagram

6.3.3 Translation of the secure UML activity diagrams into a B specification

As stated, a secure activity diagram is used to specify order-based and separation of duty constraints related to a set of operations, in which there is an operation to secure. For each operation, we define two variables to store the user whose executes the operation together with the execution instant and an operation to update such variables. Then, the secure operation of a secure activity diagram is mapped into a B operation that returns *granted* if the execution orders of the operations and the separation of duty constraints are satisfied, *denied* otherwise.

For example, the variables *userExecutedReceive* and *orderExecutionReceive* are defined to memorize the information related to the execution of the operation *receive* as follows:

$$\begin{aligned} userExecutedReceive &\in PURCHASE_ORDER \rightarrow Users \\ orderExecutionReceive &\in PURCHASE_ORDER \rightarrow NAT \end{aligned}$$

The operations *LogReceive* updating these variables and *ADReceive* granting/denying the execution of the secure operation *receive* (See Figure 6.2) are defined in Figure 6.5 (*currentOrder* denotes a natural number initialized to 0):

```

LogReceive (po, usr)  $\hat{=}$ 
PRE po  $\in$  PURCHASE_ORDER  $\wedge$  usr  $\in$  Users THEN
  userExecutedReceive(po) := usr ||
  orderExecutionReceive(po) := currentOrder ||
  currentOrder := currentOrder + 1
END
access  $\leftarrow$  ADReceive(po, usr)  $\hat{=}$ 
PRE po  $\in$  PURCHASE_ORDER  $\wedge$  usr  $\in$  Users THEN
  IF po  $\in$  dom(userExecutedApprove)  $\wedge$ 
    orderExecutionCreate(po)
       $\leq$  orderExecutionApprove(po)  $\wedge$ 
      usr  $\neq$  userExecutedCreate(po)
  THEN access := granted
  ELSE access := denied
END
END

```

Figure 6.5: The B translation of the activity diagram in Figure 6.2

6.3.4 Designing the secure filter

As one can remark, the operation *receive*, of Figure 6.3, does not take into account the static constraints specified in Figure 6.4: according to Figure 6.3, the operation can be executed by any user even if he/she is not playing the role *Staff*. This is why we need to specify a filter that allows the execution of such an operation only for a user playing the adequate role. To this aim, we add a filter operation for each operation to secure. This operation returns two values *access* and *result*: (*access = granted*) means that all the static/dynamic security rules are fulfilled, otherwise *denied* is returned; *result = OK* means that the operation *Op* has been successfully executed, otherwise *KO* is returned (See Figure 6.6) .

```

result,access ← FilterReceive(po, usr) ≡
  VAR staticRight IN
    staticRight ← checkUserPermission(receive, usr);
  IF staticRight = granted THEN
    VAR dynamicRight IN
      dynamicRight ← ADReceive(po, usr);
      IF dynamicRight = granted THEN
        result ← receive(po);
        IF result = OK THEN
          LogReceive(po, usr)
        END
      END;
      access := dynamicRight
    END
  ELSE
    access := denied; result := KO
  END
END

```

Figure 6.6: The B specification of the filter of the operation **Receive**

6.4 From an abstract B specification to a relational-like B implementation

In this section, we present the refinement process to generate a relational-like B implementation from the previous abstract specification. To do so, we mainly reuse the B refinement process defined in [104]. It consists of two main steps: *Data refinement* and *Behavioral refinement*. The goal of the data refinement process is to transform the variables of the B specification in order to be close to the structure of the tables used in relational databases, whereas the behavioral refinement aims at replacing the parallel substitutions with sequential ones and eliminating preconditions since they are supported neither in JAVA nor in SQL languages. Hereafter, we sum up the result of this process on the case study; more details about the process itself can be found in [104].

6.4.1 Data refinement

We target a relational implementation whose data structures are tables or records that permit to gather the different information related to entities of the system. This is why we define a table structure T_V for each B variable V which is source of several functions (V_1, \dots, V_n) that become its attributes. One of these variables should be an injective function to represent the key of the created table otherwise a new injective variable is introduced. Indeed, the B language is object oriented whereas the relational language is value oriented. Consequently, the transition from the B language to the relational language consists in replacing each object by its key value. In other words, each action on a B object, whose the value of its key V_1 is equal to val , is replaced by an adequate action on the record whose the value of its field V_1 is equal to val too. For instance, we define the table structure $T_Purchase_Order$ for the variable $Purchase_Order$ with attributes $purchaseOrderId$, $createdDate$, $isApproved$ and $isReceived$. We also define some operations to add (resp. delete) tuples and to update the values of the different attributes (See Figure 6.7).

As the variable $T_Purchase_Order$ will replace all the variables related to $Purchase_Order$, we have to specify some additional operations that return the values of the different predicates/expressions that are used in the initial specification and depend on these omitted variables. For instance, we define the operation $NotReceive$ that returns the truth value of the predicate $(po \in Purchase_Order \wedge isReceived(po) = FALSE)$ used by the operation $receive$ (See Figure 6.8).

⁰ $x'y$: denotes the value of the field y of the record x .

```

VARIABLES
  T_Purchase_Order
INVARIANT
  T_Purchase_Order  $\subseteq$  struct(purchaseOrderId:NAT,
    createdAt:NAT, isApproved:BOOL, isReceived:BOOL)
INITIALISATION
  T_Purchase_Order :=  $\emptyset$ 
OPERATIONS
  T_receive(po)  $\hat{=}$ 
  PRE po  $\in$  NAT THEN
    ANY pos, date, isApproved WHERE
      pos  $\in$  T_Purchase_Order  $\wedge$ 
      pos'purchaseOrderId = po  $\wedge$ 
      date = pos'createdAt  $\wedge$ 
      isApproved = pos'isApproved
    THEN
      T_Purchase_Order := T_Purchase_Order - {pos}
       $\cup$  {rec(po, date, isApproved, TRUE)}
    END
  END

```

Figure 6.7: The B specification of a relational table

```

res  $\leftarrow$  NoReceive(po)  $\hat{=}$ 
PRE po  $\in$  NAT THEN
  res := bool(  $\exists$  pos.(pos  $\in$  T_Purchase_Order  $\wedge$ 
    pos'purchaseOrderId = po  $\wedge$ 
    pos'isReceived = FALSE ))
END

```

Figure 6.8: The B evaluation of a predicate

Let us recall that we use UML activity diagrams to express dynamic security rules, which often relate to the execution history of the different system's actions. Hence, it is important to store the execution information of each operation $op(p_1, \dots, p_n)$, such as its corresponding execution occurrences together with the user that executes it and the execution time. To do so, we define the two following variables $userExecutedOp$ and $orderExecutionOp$, where $Type_{p_i}$ is the type of the parameter p_i of the operation op .

$$userExecutedOp \in Type_{p_1} \times \dots \times Type_{p_n} \rightarrow Users$$

$$\text{orderExecutionOp} \in \text{Type}_{p_1} \times \dots \times \text{Type}_{p_n} \rightarrow \text{NAT}$$

Since we target a relational implementation, we create a table structure T_Op_log for each pair $userExecutedOp$ and $orderExecutionOp$. Such a variable T_Op_log defines each parameter's type Type_{p_i} as an attribute and two additional attributes $userExecutedOp$ denoting the person who executed op and $orderExecutionOp$ representing its executed moment. The set of the attributes $i=1..n \text{Type}_{p_i}$ represents the key of the created table. Similar to the transition of the previous object table ($T_Purchase_Order$ for instance), this transition also consists in replacing an object by its key value. For example, the table structure $T_Receive_log$ gathers an attribute $purchaseOrderId$ to represent the key of each purchase order along with two variables $userExecutedReceive$ and $orderExecutionReceive$.

To query about the execution history of the operation op , we define some operations on the table T_Op_log . We create an operation $AddOp_log$ to add a new tuple to the table T_Op_log . The operation $NoOp_log$ is created to check whether or not the operation op is performed in the past: it returns *false* if op is already executed, *true* otherwise. In addition, the operations Op_user and Op_order are defined to return respectively the user who executed the operation and the executed instant. For example, Figure 6.9 denotes the table structure $T_Receive_log$ and its associated operations $AddReceive_log$, $NoReceive_log$, $Receive_user$, and $Receive_order$.

Notice that the B specifications of the defined tables, that are Figures 6.7, 6.8, and 6.9 are no longer refined because they can be straightforwardly translated in JAVA/ SQL (See Section 6.5.2).

6.4.2 Behavioral refinement

The variables of the first level have been replaced by new ones to represent the relational tables. This is why we have to replace each substitution/predicate/expression of the initial variables with call to the operations acting on the new variables. In addition, all the preconditions can be eliminated since they are just typing constraints and then are already assumed as true. Moreover, the sequential operator is introduced to replace the parallel one. When the parallel substitutions modify distinct variables then the parallel operator (\parallel) is simply replaced by the sequential one ($;$) by keeping the same substitutions. For instance, the operation $receive(po)$ is rewritten as in Figure 6.10.

The operation $LogReceive$ is refined by calling, on the one hand, the operation $getIdPurchaseOrder$ to get the key of the object po and, on the other hand, the operation $AddReceive_log$ to update the imported variable $Receive_log$ that


```

VARIABLES  $T\_Receive\_log$ 
INVARIANT  $T\_Receive\_log \subseteq \mathbf{struct}(purchaseOrderId: \mathbf{NAT},$ 
   $userExecutedReceive : Users, orderExecutionReceive : \mathbf{NAT})$ 
INITIALISATION  $T\_Receive\_log := \emptyset$ 
OPERATIONS
  //check if the operation receive is already executed on a given object:
  //returns false if it is executed, true otherwise.
   $res \leftarrow \mathbf{NoReceive\_log}(po) \hat{=}$ 
  PRE  $po \in \mathbf{NAT}$  THEN
     $res := \mathbf{bool}(\neg (\exists log.(log \in T\_Receive\_log \wedge log'purchaseOrderId = po)))$ 
  END;
  //get the user who executed the operation receive on a given object
   $res \leftarrow \mathbf{Receive\_user}(po) \hat{=}$ 
  PRE  $po \in \mathbf{NAT}$  THEN
    ANY  $log$  WHERE
       $log \in T\_Receive\_log \wedge log'purchaseOrderId = po$ 
    THEN
       $res := log'userExecutedReceive$ 
    END
  END;
  //get the executed time of the operation receive on a given object
   $res \leftarrow \mathbf{Receive\_order}(po) \hat{=}$ 
  PRE  $po \in \mathbf{NAT}$  THEN
    ANY  $log$  WHERE
       $log \in T\_Receive\_log \wedge log'purchaseOrderId = po$ 
    THEN
       $res := log'orderExecutionReceive$ 
    END
  END;
  //adds a new record of the operation receive execution on a given object
  AddReceive_log( $po, usr, time$ )  $\hat{=}$ 
  PRE
     $po \in \mathbf{NAT} \wedge usr \in Users \wedge time \in \mathbf{NAT}$ 
  THEN
     $T\_Receive\_log := T\_Receive\_log \cup \{\mathbf{rec}(po, usr, time)\}$ 
  END

```

Figure 6.9: The B specification of a log table

```

result ← receive(po) ≐
  VAR id, notyet_receive IN
    id ← getIdPurchaseOrder(po);
    notyet_receive ← NoReceive(id);
    IF notyet_receive = TRUE THEN
      T_receive(id);
      result := OK
    ELSE
      result := KO
    END
  END

```

Figure 6.10: The B implementation of the operation *receive*

replaces the variables *userExecutedReceive* and *orderExecutionReceive* (See Figure 6.11):

```

LogReceive(po, usr) ≐
  VAR id IN
    id ← getIdPurchaseOrder(po);
    AddReceive_log(id, usr, currentOrder);
    currentOrder := currentOrder + 1
  END

```

Figure 6.11: The B implementation of a log operation *LogReceive*

Similarly, the operation *ADReceive* is refined by calling a number of operations: the operation *getIdPurchaseOrder* that gets the key of the purchase order *po*; the operation *NoApprove_log(po)* that makes sure that *po* is already approved; the operations *Approve_order(po)* and *Create_order(po)* that check the executed instants of the operations *approve* and *create*; and the operation *Create_user(po)* that ensures the user who created the object *po* is different from the user who is trying to perform the operation *receive*. Figure 6.12 denotes the implementation of *ADReceive*.

The next section describes the translation of the refined B specification into an executable AspectJ program.

```

access ← ADReceive(po, usr) $\hat{=}$ 
VAR id, notyet_approve, notyet_create,
     create_user, create_order, approve_order
IN
  id ← getIdPurchaseOrder(po);
  notyet_approve ← NoApprove_log(id);
  notyet_create ← NoCreate_log(id);
  create_user ← Create_user(id);
  create_order ← Create_order(id);
  approve_order ← Approve_order(id);
  IF notyet_approve = FALSE  $\wedge$ 
     notyet_create = FALSE  $\wedge$ 
     usr  $\neq$  create_user  $\wedge$ 
     create_order < approve_order
  THEN
    access := granted
  ELSE
    access := denied
  END
END

```

Figure 6.12: The B implementation of the operation *ADReceive*

6.5 The AspectJ implementation of the application

In this section, we describe the translation of the refined B specification into an AspectJ implementation that uses a relational DBMS, in our case SQL Server release 2014. The use of an aspect programming technique, like AspectJ, permits to enforce the security as crosscutting concerns by separating it from the functional part of the application in order to avoid the scattering of the security code over the whole system. Figure 6.13 shows how each JAVA/SQL and AspectJ element is derived from the B implementation level. Multiple colors are used to show the correspondence between the B specification and the AspectJ implementation elements. The AspectJ-based implementation generated from the previous refined B specification is composed of the following JAVA classes:

1. a set of JAVA classes to implement the tables described as B structures (**struct**) together with their associated operations declared as JAVA meth-

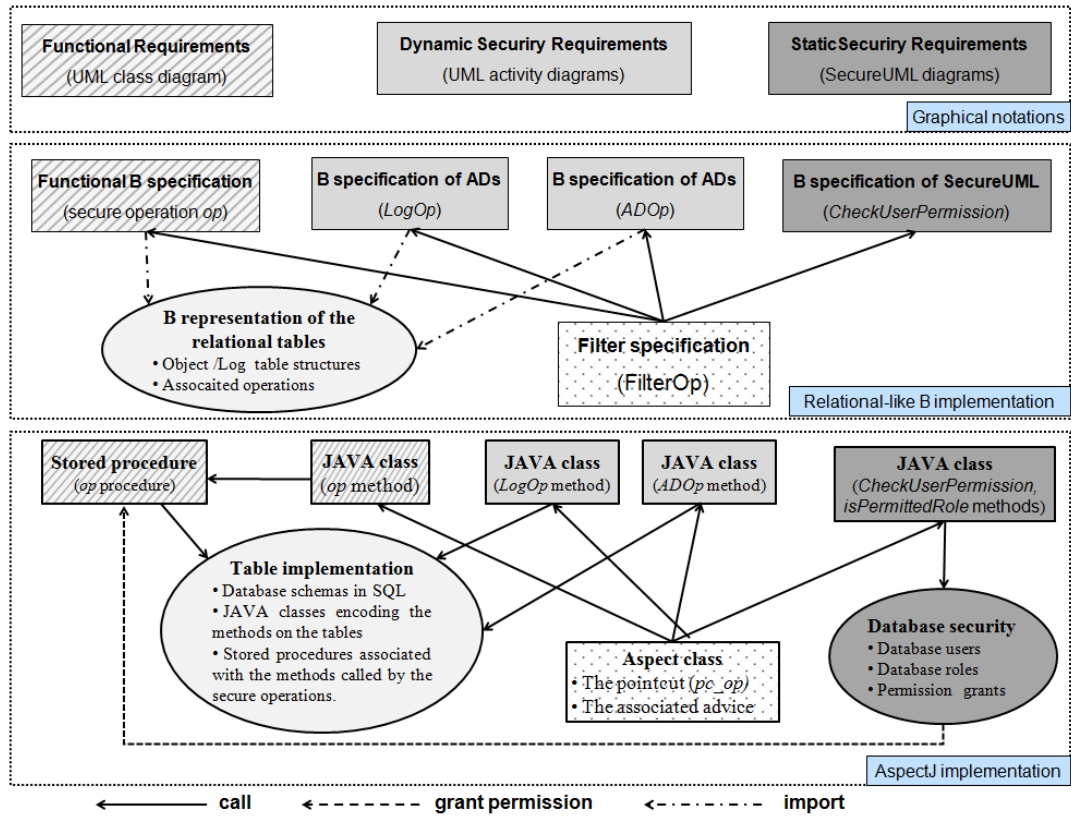


Figure 6.13: Derivation of the AspectJ implementation

- ods,
2. a JAVA class to implement, as JAVA methods, the operations derived from the class diagram. Such methods call their associated stored procedures, which have the same signatures as them,
 3. a JAVA class *SecureUMLJAVATrans* to implement the B operations generated from the **SecureUML** diagram (operations *checkUserPermission*, *isPermittedRole*, and *connectUser*),
 4. a JAVA class *ADJAVATrans* to implement the B operations generated from the secure activity diagrams (the operation *ADOp*),
 5. a JAVA class *ActionsHistoryJAVATrans* to implement the log operations *LogOp*,
 6. an *AspectJ* class for the operations for which a filter is specified.

The execution flow of the obtained implementation is depicted by Figure 6.14 where:

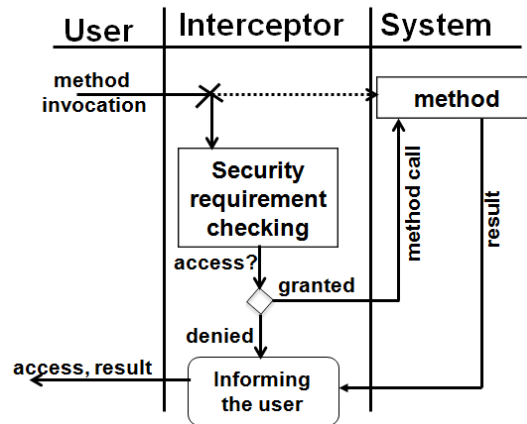


Figure 6.14: AspectJ implementation principle

1. A *pointcut* is associated with each operation to secure,
2. When a user invokes a method for which a *pointcut* is associated, the interceptor immediately intercepts the invocation. That means that the method is not actually called (represented by the dashed arrow) but instead the interceptor executes the related advice that consists in:
 - (a) Checking whether the security rules are fulfilled.
 - (b) Calling the method only if the security rules are verified.
 - (c) Logging the execution of the method if it is performed successfully.
 - (d) In both cases, informing the user about the result of the method execution: are the security rules verified? what is the result of the method call?

6.5.1 Transformation rules of B into JAVA/SQL

This section describes a set of mapping rules that allow us to translate the previous refined B specification into the JAVA/SQL language. These rules are used to deploy the components of the targeted AspectJ application.

Table 6.1: Type mappings table among B, JAVA, and SQL Server

B Type	SQL Server Type	JAVA Type
NAT	int	int
BOOL (TRUE/FALSE)	bit (1/0)	boolean (true/false)
Enum set (Users/Roles/...)	varchar	String
defined string	varchar	String

$res \leftarrow op(p_1, \dots, p_n) \hat{=}$ PRE $p_1 \in T_{p_1}, \dots, p_n \in T_{p_n}$ THEN $body$ END	CREATE PROCEDURE op $@p_1 T_{p_1}, \dots, @p_n T_{p_n},$ $@res T_{res}$ OUTPUT AS $body$ GO
--	--

Figure 6.15: Transformation of a B operation to a stored procedure

Type mappings

The definition of types in B is similar to the definition of types in JAVA/SQL. Yet, their syntax is different for each language. In Table 6.1, we map the B types into those supported in the JAVA language and the SQL Server (version 2014).

From a B operation to a stored procedure

The translation of a B operation into a stored procedure includes the mapping of its parameters and the predicates defined in its body. The signature of the stored procedure translated from the B operation defined in Figure 6.15 has:

- the same name op
- the same input parameters p_i . The syntax declaring an input parameter of the stored procedure is $@p_i T_{p_i}$.
- the same output parameter • res typed T_{res} . The syntax declaring an output parameter of the stored procedure is $@res T_{res}$ OUTPUT.

Of course the type of these parameters is translated as defined in Table 6.1.

Table 6.2: Transformation of B expressions to SQL Server

B expression	Stored procedure code
VAR var IN	DECLARE @var
IF... THEN... (ELSE THEN...) END	IF(...) ... ELSE...
var = TRUE/FALSE	@var = 1 /0
var = / > / < number	@var = / > / < number
operation call	stored procedure call
expression on tables [91]	SQL query

The body of the obtained procedure is derived by mapping the B predicates into the corresponding SQL code and replacing the operation call by the stored procedure call. Table 6.2 describes the translation of some B expressions into SQL Server. The expressions acting on tables are translated into SQL statements by applying the rules defined in [91].

In the application, a JAVA method is created to call the associated stored procedure. It informs the execution result of the stored procedure: if the SQL code is successfully executed, it returns *OK*, *KO* otherwise. The structure of such a call is as follows:

//The generated JAVA method associates with the stored procedure op.

```

public String op( $T_{p_1}$  p1, ...,  $T_{p_n}$  pn)
{
    String res = "KO";
    CallableStatement statem;
    try {
        statem = prepareCall("{ call op(?) }");
         $\bigwedge_{i=0..n}$  statem.setTpi(i, pi);
        int exe = statem.executeUpdate();
        if (exe > 0)
            res = "OK";
        else
            res = "KO";
        statem.close ();
    } catch (SQLException e) {...}
    return res;
}

```

$res \leftarrow op(p_1, \dots, p_n) \hat{=}$ PRE $p_1 \in T_{p_1}, \dots, p_n \in T_{p_n}$ THEN $body$ END	public T_{res} op (T_{p_1} p_1, \dots, T_{p_n} p_n) { T_{res} $res = initialisation;$ $body$ return $res;$ }
--	---

Figure 6.16: Transformation of B operation to JAVA method

From a B operation to a JAVA method

Mapping a B operation into a JAVA method is based on the translation of its signature (i.e. the input/output parameters) and the predicates encoding its body. The obtained JAVA method has the same name and the list of parameters as those of the B operation (See Figure 6.16), while the B expressions can be straightforwardly translated into JAVA. Indeed, the B control structures obtained at the implementation level are supported in JAVA. Hence, we just have to replace such B control structures by the corresponding JAVA control structures and the operation call by the method call. Figure 6.17 shows the translation of expressions used in our case study.

In the following sections, we describe the translation of each part of the targeted *AspectJ* implementation.

6.5.2 Deployment of the class diagram

The deployment of the B specification generated from the class diagram permits to define the database and a set of stored procedures and JAVA methods to translate the associated B operations. The creation of the database is obtained following the rules defined in [91] whereas we introduce a new manner to map the operations as stored procedures. Indeed, in SQL, grants can be defined on basic statements (**INSERT**, **UPDATE**, **DELETE**) or on stored procedures. Of course, another solution would be to encode an ad hoc roles/permissions management system using several tables. However such a solution may be very cumbersome, this is why we preferred to use stored procedures.

<p>1. VAR constructor</p> <p>VAR <i>var</i> IN <i>var</i> \leftarrow <i>someOp</i>(...) <i>(the returned type is T_{var})</i></p>	<p>1. local variable declaration</p> <p>T_{var} <i>var</i> = <i>someOp</i>(...);</p>
<p>2. IF constructor</p> <p>IF ... THEN ... (ELSE THEN...) END</p>	<p>2. if statement</p> <p>if (...) {...} (else {...})</p>
<p>3. bool expression</p> <p><i>cond</i> = <i>TRUE/FALSE</i></p>	<p>3. bool condition</p> <p><i>cond</i> = true/false</p>
<p>4. compare numbers</p> <p><i>nb1</i> = />/< <i>nb2</i></p>	<p>4. compare numbers</p> <p><i>nb1</i> = />/< <i>nb2</i></p>
<p>5. compare strings</p> <p><i>str1</i> = <i>str2</i>)/<i>str1</i> \neq <i>str2</i></p>	<p>5. compare strings</p> <p><i>str1</i>. <i>equals</i>(<i>str2</i>)/!<i>str1</i>. <i>equals</i>(<i>str2</i>)</p>
<p>6. and predicate</p> <p><i>expr1</i> \wedge <i>expr2</i></p>	<p>6. and condition</p> <p><i>expr1</i> && <i>expr2</i></p>
<p>7. operation call</p> <p><i>someOp</i>(...)</p>	<p>7. method call</p> <p><i>Class</i>_{<i>someOp</i>}-<i>instance</i>.<i>someOp</i>(...)</p>

Figure 6.17: Transformation of B expressions into JAVA

6.5.2.1 Definition of the tables and the associated JAVA methods and stored procedures

From the B specification defining variables as structures (**struct**) and its associated B operations (See Figure 6.7 and 6.8), a SQL table is created with a set of JAVA methods/stored procedures translating the related operations. An operation is mapped into a stored procedure plus a JAVA method that invokes it if it is called by an operation to secure (appears in the **secureUML** diagram), otherwise it is translated into a classical JAVA method. Indeed in SQL Server, a stored procedure can only call another stored procedure and not a JAVA method. Let us note that the stored procedure and the JAVA method must have the same name. From Figures 6.7 and 6.8 for instance, a SQL table *T_Purchase_Order* and the stored procedure and JAVA method are derived as follows:

```

CREATE TABLE T_Purchase_Order(
  purchaseOrderId INT PRIMARY KEY,
  createdDate INT NOT NULL,
  isApproved BIT NOT NULL,
  isReceived BIT NOT NULL)

-- checks whether a purchase order is received or not,
-- returns 1 (true) if it has not been received, 0 (false) otherwise.
CREATE PROCEDURE NoReceive @po int, @res bit OUTPUT
AS
  DECLARE @count int
  SELECT @count = COUNT(*) FROM dbo.PurchaseOrder
    WHERE purchaseOrderId = @po AND isReceived = 0
  IF(@count > 0)
    SET @res = 1
  ELSE
    SET @res = 0
GO

//calls the associated procedure
public boolean NotReceive(int po){
  CallableStatement statem;
  try{
    statem = prepareCall("{call NotReceive (?)}");
    statem.setInt(1, po);
  }
}

```

```

    //execution of the statement, returns true if the
    //result of the query equals to 1
    return (statem.executeQuery()==1);
}catch (SQLException e){};
}

```

The operation *NotReceive* has been translated into a stored procedure because it is called by the implementation of the operation *receive* that is itself translated into a stored procedure. Similarly, a stored procedure and a JAVA method are generated for the operation *T_receive*.

6.5.2.2 Translation of the operations of the class diagram

The translation of the operations generated from the class diagram depends if the operation is to secure or not, that is, if a user needs to play specific roles to execute the operation. An operation to secure appears in the SecureUML diagram. In this case, it is translated into a stored procedure plus a JAVA method that invokes it, otherwise a classical JAVA method is defined. In both cases, the JAVA method calls those on the different tables defined in Section 6.5.2.1. For instance, the operation *receive* gives the following stored procedure and JAVA method:

```

CREATE PROCEDURE receive(@po int)
AS
    DECLARE @res bit
    //executing the stored procedure NotReceive
    EXEC NotReceive @po, @res OUTPUT
    IF(@res = 1)
        BEGIN
            EXEC T_receive @po
        END
GO

```

```

public String receive(int po){
    String result = "KO";
    CallableStatement statem;
    try{
        //defining the CallableStatement JAVA variable statem to contain
        //the call to the stored procedure receive
        statem = prepareCall("call receive(?)");
    }
}

```

```
    statement.setInt(1, po);  
    //execution of the stored procedure,  
    //returns the number of the modified tuples  
    int res = statement.executeUpdate();  
    if (res > 0) result = "OK";  
    statement.close();  
} catch (SQLException e)  
return result;
```

6.5.3 Deployment of the SecureUML diagram

The deployment of the B specification generated from the SecureUML diagram permits to create the users with their roles and allowed permissions. For that purpose, we use the authentication mechanism provided by SQL Server (release 2014) database management system as follows:

- For each value $Users_i$ of the set $Users$, a SQL query is generated in order to create the user. User *Alice* for instance is created by the following SQL statements:

```
CREATE LOGIN Alice  
WITH PASSWORD = pwdAlice
```

```
CREATE USER Alice  
FOR LOGIN Alice
```

- For each value $Roles_i$ of the set $Roles$, a SQL query is generated in order to create the role. Role *Staff* for instance is created by the following SQL statement:

```
CREATE ROLE Staff;
```

- For each pair of $(Users_i, Roles_i)$ of the constant $UsersRoles$, a SQL query is generated in order to assign the role $Roles_i$ to the user $Users_i$. The following SQL statement is generated for the pair $(Alice \mapsto Staff)$:

```
ALTER ROLE Staff  
ADD MEMBER Alice;
```

- For each pair of $(Roles_i, Op_i)$ of the constant *Permissions*, a SQL query is generated in order to grant the execution of the stored procedure Op_i to the role $Roles_i$. The following SQL statement is generated for the pair $(Staff \mapsto receive)$:

GRANT EXECUTE ON *receive* **TO** *Staff*;

In addition to these SQL statements, a JAVA class *SecureUMLJAVATrans* is created. It defines two attributes: *connectingUser* and *currentRole* derived from the *Session* relation, which store respectively the user who is connecting to the system and his/her current role. Of course, the getter (*getConnectingUser/getCurrentRole*) and the setter (*setConnectingUser/setCurrentRole*) of these attributes are also created in order to use them externally. It also encodes the B operations *connectUser(usr, role)* and *checkUserPermission(usr, op)* as JAVA methods with the same signature.

When a user *usr* connects to the system with one of his/her roles *rol*, the method *connectUser* specifies that *usr* is the connecting user and *rol* is his/her current role.

```
public void connectUser(String usr, String rol){
    connectingUser = usr;
    currentRole = rol;
}
```

The *checkUserPermission* method verifies whether a user *usr* is permitted to perform an operation *op* by checking if his current role is granted to execute the associated stored procedure or not. To do so, we define an additional method *isPermittedRole* encoding a SQL query to check an execution permission of a given role *rol* on a given stored procedure *op*. Hence, *checkUserPermission* grants the execution of the operation *op* for the connecting user *usr* only if his current role is allowed to execute the stored procedure *op*. The JAVA methods *checkUserPermission* and *isPermittedRole* are defined as follows:

```
public String checkUserPermission(String usr, String op) throws
    SQLException {
    String access = "denied";
    //getting the connecting user,
    //making sure that the verified user is currently
    //connecting to the system.
    String conUser = getConnectingUser();
```

```

    if (usr.equals(conUser)) {
        //checking the permission based on the current role of the user,
        //the execution is granted if the method isPermittedRole()
        // returns true, denied otherwise.
        boolean isPermitted = isPermittedRole(currentRole, op);
        if (isPermitted)
            access = "granted";
        else
            access = "denied";
    }
    return access;
}
//checking the execution permission of a given role on a given method
public boolean isPermittedRole(String rol, String op) throws
    SQLException {
    boolean access = false;
    //defining a PreparedStatement JAVA variable to verify
    //permission of a given role on executing a given stored procedure
    PreparedStatement stm = prepareStatement("SELECT COUNT(*)"
        + "FROM sys.database_permissions "
        + "WHERE USER_NAME(grantee_principal_id) = (?) "
        + "AND OBJECT_NAME(major_id) = (?) "
        + "AND permission_name = 'EXECUTE'");
    try {
        stm.setString(1, rol);
        stm.setString(2, op);
        //the execution of the statement,
        //returns true if there exists at least one row in the database,
        //returns false otherwise.
        ResultSet resSet = stm.executeQuery();
        if (resSet.next() && resSet.getInt(1) > 0)
            access = true;
        resSet.close();
    } catch (SQLException e) {e.printStackTrace();}
    return access;
}

```

6.5.4 Deployment of the secure activity diagrams

Recall that the translation into a B specification of the secure activity diagrams gives a variable (T_Op_log) that stores the execution information of each operation and two operations ($LogOp$ and $ADOp$) that aim at updating the value of this variable and also to check whether the dynamic constraints specified by the secure activity diagram are fulfilled or not. The translation into JAVA of this B specification gives:

1. each variable T_Op_log is mapped into an SQL table with the same attributes (See Section 6.5.4.1).
2. a JAVA class T_Op_log is introduced to define a set of methods that update/read the different data stored in the previous table (See Section 6.5.4.1).
3. a JAVA class $ActionsHistoryJAVATrans$ is introduced as the deployment of the B specification of the log operations ($LogReceive$ for example) (See Section 6.5.4.3).
4. a JAVA class $ADJAVATrans$ is defined to encode the operation $ADOp$ ($ADReceive$ for instance) as a method that calls the methods defined in the class T_Op_log (See Section 6.5.4.2).

6.5.4.1 Definition of the log tables and the associated JAVA classes

For each variable T_Op_log defined as structures (**struct**) and its associated B operations, we create respectively a SQL table and a set of JAVA methods. The derivation of the JAVA/SQL code is obtained following the rules defined in [91]. For instance, the variable $T_Receive_log$ gives a SQL table with the attributes $PurchaseOrderId$, $userExecutedReceive$, and $orderExecutionReceive$.

```
CREATE TABLE  $T\_Receive\_log$ (
   $PurchaseOrderId$  INT PRIMARY KEY NOT NULL,
   $userExecutedReceive$  VARCHAR(25),
   $orderExecutionReceive$  DATETIME CURRENT _TIMESTAMP)
```

The JAVA class T_Op_log gives accesses to the table T_Op_log through a set of methods. Such methods encode SQL statements querying the table T_Op_log . For example, the B specification of the associated operations of the variable $T_Receive_log$ (See Figure 6.9) derives a class $T_Receive_log$ including the JAVA methods as follows:

```

public class T_Receive_log {
    private PreparedStatement stmt;
    private Connection dbcon;
    public T_Receive_log(Connection conn) throws SQLException {
        dbcon = conn;
        stmt = conn.prepareStatement("SELECT orderExecutionReceive,
            userExecutedReceive "
            + "FROM T_Receive_log "
            + "WHERE purchaseOrderId = ?; "); }
    public boolean NoReceive_log(int po) {
        try {
            stmt.setInt(1, po);
            //execution of the statement that checks if the method receive
            //is already performed on the purchase order po or not,
            //returns false if it is already executed, true otherwise.
            ResultSet res = stmt.executeQuery();
            if (res.next())
                return false;
            else
                return true;
        } catch (SQLException e) {} }
    public void AddReceive_log(int po, String usr){
        try {
            //defining a SQL statement to add a new record to the table
            Statement stm = dbcon.createStatement();
            String log = "INSERT INTO T_Receive_log"
                + "(purchaseOrderId, userExecutedReceive) "
                + "VALUES(" + po + ",'" + usr + "')";
            //executing the statement
            stm.executeUpdate(log);
            stm.close();
        } catch (SQLException e) {} }
    public String Receive_user(int po){
        try {
            //checking whether the method receive is already executed on po,
            //if it is the case (i.e. NoReceive_log(po) returns false),
            //executes the statement to return the user who executed it.
            if (!NoReceive_log(po)){
                stmt.setInt(1, po);

```



```

        ResultSet res = stmt.executeQuery();
        return res.getString("userExecutedReceive");
    }
} catch (SQLException e) {} }
public String Receive_order(int po){
    try {
        //checking whether the method receive is already executed on po,
        //if it is the case (i.e. NoReceive_log(po) returns false),
        //executes the statement to return the executed instant.
        if (!NoReceive_log(po)){
            stmt.setInt(1, po);
            ResultSet res = stmt.executeQuery();
            return res.getString("orderExecutionReceive");
        }
    } catch (SQLException e) {} }
}

```

6.5.4.2 Translation of the secure operations of the secure activity diagrams

To call the methods of a log class within the method *ADOp*, we create an instance of the log class as an attribute. For example, we define two attributes to use the methods of the classes *T_Approve_log* and *T_Create_log* for the deployment of the operation *ADReceive*.

```

public class ADJAVATrans {
    //parser the datetime type
    SimpleDateFormat sdf = new
        SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
    private T_Approve_log t_approve_log;
    private T_Create_log t_create_log;
    public String ADReceive(int po, String usr) {
        String access = "denied";
        boolean notyet_approve = t_approve_log.NoApprove_log(po);
        boolean notyet_create = t_create_log.NoCreate_log(po);
        String create_user = t_create_log.Create_user(po);
        String create_order = t_create_log.Create_order(po);
        String approve_order = t_approve_log.Approve_order(po);
        //verifying (1) the action approve() is already performed,
    }
}

```

```

    //(2) the action create() is also executed
    //and it should be executed before approve(),
    //(3) the current user is not the person who created the order.
    //the execution is granted if all these conditions are fulfilled ,
    //denied otherwise.
    if (notyet_approve == false && notyet_create == false &&
        sdf.parse(create_order).before(sdf.parse(approve_order) &&
            !usr.equals(create_user) )
        access = "granted";
    else
        access = "denied";
    return access;
}
}

```

6.5.4.3 Translation of the log operations

The execution of a specific method is saved through the class *ActionsHistory-JAVATrans*, which is the deployment of the B specification of the log operations (*LogReceive* for instance). To use the method *AddOp_log* within the method *LogOp*, we declare its class as an attribute. For instance, we define an attribute *T_Receive_log* to deploy the method *LogReceive*. Recall that the B variable *currentOrder* is used to calculate an executed order (an executed instant). In fact, we can omit it in the deployment of the operation *LogReceive* since the DBMS allows to automatically capture any moment in the system thanks to the property *CURRENT_TIMESTAMP*.

```

public class ActionsHistory.JAVATrans {
    private T_Receive_log t_receive_log;
    public void LogReceive(int po, String usr){
        t_receive_log.AddReceive_log(po, usr);
    }
}

```

6.5.5 Deployment of the filter

On the invocation of a given secure operation *op*, the filter aims at verifying the static and dynamic security rules before its actual execution. If these security rules are verified, the operation *op* is called. In all cases, the user is informed

about the result of its operation invocation. Thus, we deploy the filter as an advice associated with the pointcut defined on the operation *op*. This advice is defined with the *around* keyword since some actions are executed before and others after the invocation of the operation *op*. The signature of the advice is that of the filter omitting the user who is requesting to execute the operation *op*. The reason is that the method *proceed()* within the advice must have the same signature with both the operation *op* and the advice. Moreover, security rules are checked for the connecting user, which can be obtained by calling the method *getConnectingUser()* of the class *SecureUMLJAVATrans*. To call a security-checking method, we define an attribute for its class in the aspect *SecureFilter2AspectJ*.

```

public aspect SecureFilter2AspectJ {
  private SecureUMLJAVATrans staticC;
  private ADJAVATrans dynamicC;
  private ActionsHistoryJAVATrans logC;
  ...
  //the pointcut to a method named receive
  //regardless of its return type (denoted by wildcard *)
  pointcut pc_receive(int po): args(po) && call(* receive(int));
  //the advice associated with the previous pointcut
  String around(int po): pc_receive(po){
    String result = "KO";
    String usr = SecureUMLJAVATrans.getConnectingUser();
    try {
      // check static security rules
      String staticRight = staticC.checkUserPermission("receive", usr);
      if (staticRight == "granted") {
        // check dynamic security rules
        String dynamicRight = dynamicC.ADReceive(po, usr);
        if (dynamicRight == "granted") {
          // proceed the method execution
          result = proceed(po);
          if (result == "OK") logC.LogReceive(po, usr);
        }
      }
    }
    catch (SQLException e) {}
    return result;
  }
}

```

6.6 Tool support

In this section, we describe the extension of the tool presented in Chapter 5 to generate the AspectJ implementation of an application from its B formal specification. Towards this end, we define a grammar for the B specification in XText [105], which is an adaptation of the grammar of the ABTool [106] (written in the ANTLR syntax [107]). The translation process is performed in four steps corresponding to the different B components.

6.6.1 Translating the B specification of the class diagram

Once the B specification of the class diagram is refined into a relational-like implementation, the tool generates:

```

Table_Purchase_Order.mch
MACHINE
  Table_Purchase_Order
VARIABLES
  T_Purchase_Order
INVARIANT
  T_Purchase_Order <:
    struct(purchaseOrderId:NAT, createdAt:NAT,
           isApproved:BOOL, isReceived:BOOL)

T_Purchase_Order.sql
--SQL code of Table_Purchase_Order
/*generate the SQL table from the B variable T_Purchase_Order*/
CREATE TABLE T_Purchase_Order {
  /*the key is the first attribute*/
  purchaseOrderId INT /*convert type NAT in B to INT in SQL*/ PRIMARY KEY NOT NULL ,
  createdAt INT /*convert type NAT in B to INT in SQL*/ NOT NULL ,
  isApproved BIT /*convert type BOOL in B to BIT in SQL*/ NOT NULL ,
  isReceived BIT /*convert type BOOL in B to BIT in SQL*/ NOT NULL
}
  
```

Figure 6.18: Generation of a table

```

Table_Purchase_Order.mch
MACHINE
  Table_Purchase_Order
VARIABLES
  T_Purchase_Order
INVARIANT
  T_Purchase_Order <:
    struct(purchaseOrderId:NAT, createdAt:NAT,
           isApproved:BOOL, isReceived:BOOL)
INITIALISATION
  T_Purchase_Order := {}
OPERATIONS
  T_Purchase_Order := {}
  res<--NoReceive(po)=
  PRE po :NAT THEN
    res := bool(#pos.(pos:T_Purchase_Order &
                    pos.purchaseOrderId = po &
                    pos.isReceived = FALSE ))
END

T_Purchase_Order.sql
/**
 * generated stored procedure of the operation NoReceive
 */
CREATE PROCEDURE NoReceive
  @po INT /*convert type NAT in B to INT in SQL*/,
  @res BIT OUTPUT /*BOOL type in B is converted to BIT type in SQL*/
AS
  /*checks the existence of records*/
  DECLARE @exist int
  SELECT @exist = COUNT(*) FROM T_Purchase_Order
  /*generates conditions from the B predicates*/
  WHERE purchaseOrderId= @po AND isReceived= 0
  /*returns 1 (corresponds to TRUE in B) if there exists such an instance,
  returns 0 (corresponds to FALSE in B) otherwise*/
  IF(@exist > 0)
    SET @res = 1
  ELSE
    SET @res = 0
GO
  
```

Figure 6.19: Generation of a stored procedure *NotReceive*

- a SQL table for each B variable in the form of a structure (**struct**). For example, Figure 6.18 shows the derivation of a SQL table from the B structure *T_Purchase_Order*.

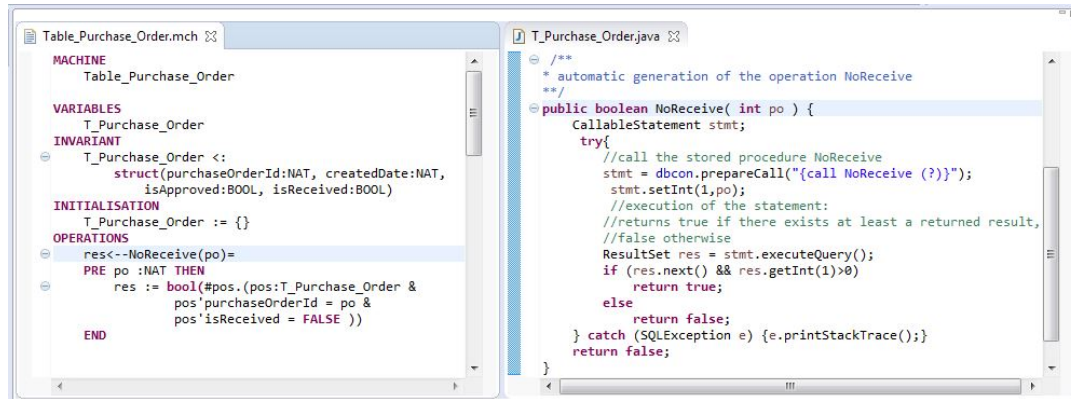


Figure 6.20: Generation of the associated JAVA method *NotReceive*

- a set of JAVA methods/stored procedures for the associated B operations of the previous variable, which are called by secure operations. For instance, Figure 6.19 and Figure 6.20 are the translation of the stored procedure and the JAVA method generated from the operation *NotReceive* on the table *T_Purchase_Order*, called by the secure operation *receive*.
- a stored procedure and a JAVA method that calls it for each operation to secure . These transformations are illustrated by the operation *receive* in Figure 6.21 and 6.22

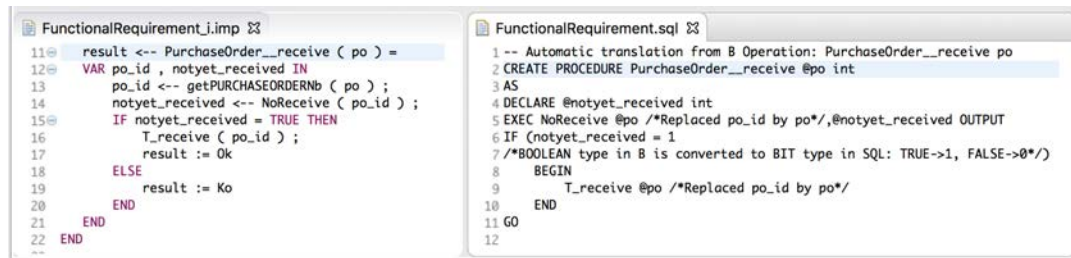


Figure 6.21: Generation of the stored procedure *receive*

6.6.2 Translating the B specification of the SecureUML diagram

The B specification of the SecureUML diagram encodes the security data, such as users, roles, user-role assignments, and permission-role assignments together

```

FunctionalRequirement_imp
15 OPERATIONS
16 result <- PurchaseOrder__receive ( po ) =
17 VAR po_id , notyet_received IN
18 po_id <- getPURCHASEORDERNb ( po ) ;
19 notyet_received <- NoReceive ( po_id ) ;
20 IF notyet_received = TRUE THEN
21 T_receive ( po_id ) ;
22 result := Ok
23 ELSE
24 result := Ko
25 END
26 END
27 END
28
29
30
31
32

FunctionalRequirement.java
7 /* Automatic generation of 8 Operation PurchaseOrder__receive*/
8 public String PurchaseOrder__receive( int po ) {
9 String result="KO";
10 CallableStatement stmt;
11 try{
12 // defining the CallableStatement JAVA variable stmt to contain
13 //the call to the stored procedure receive
14 stmt = dbcon.prepareCall("{call PurchaseOrder__receive (?)}");
15 stmt.setInt(1,po);
16 //execution of the store procedure
17 //returns the number of modified tuples
18 int res = stmt.executeUpdate();
19 if(res>0) result = "OK";
20 stmt.close();
21 }catch (SQLException e) { e.printStackTrace();}
22 return result;
23 }
24 }

```

Figure 6.22: Generation of the associated JAVA method *receive*

with security relevant operations. In Section 6.5.3, we described the deployment of such a specification in JAVA/SQL. Consequently, the tool generates the following components from this specification:

```

SecureUMLContext.mch
MACHINE
SecureUMLContext
SETS
Users = {Bob, Paul, Tom, Alice, noneuser};
Roles = {Staff, Manager, nonerole};
Operations = {
PurchaseOrder__approve,
PurchaseOrder__create,
PurchaseOrder__receive};
Access = {granted, denied}
CONSTANTS
Permissions, UsersRoles
PROPERTIES
Permissions ={
Manager|-> PurchaseOrder__approve,
Staff|-> PurchaseOrder__create,
Staff|-> PurchaseOrder__receive} &
UsersRoles ={
Bob|->Staff,
Paul|->Staff,
Tom|->Manager,
Alice|->Staff,
noneuser|->nonerole}
END

SecureUMLContext.sql
//Create database users
CREATE LOGIN Bob WITH PASSWORD = pwdBob
CREATE USER Bob FOR LOGIN Bob

CREATE LOGIN Paul WITH PASSWORD = pwdPaul
CREATE USER Paul FOR LOGIN Paul

CREATE LOGIN Tom WITH PASSWORD = pwdTom
CREATE USER Tom FOR LOGIN Tom

CREATE LOGIN Alice WITH PASSWORD = pwdAlice
CREATE USER Alice FOR LOGIN Alice

//Create database roles
CREATE ROLE Staff
CREATE ROLE Manager

//Grant permissions
GRANT EXECUTE ON PurchaseOrder__approve TO Manager
GRANT EXECUTE ON PurchaseOrder__create TO Staff
GRANT EXECUTE ON PurchaseOrder__receive TO Staff

// Assign roles to users
ALTER ROLE Staff ADD MEMBER Bob
ALTER ROLE Staff ADD MEMBER Paul
ALTER ROLE Manager ADD MEMBER Tom
ALTER ROLE Staff ADD MEMBER Alice

```

Figure 6.23: Generation of the security data

- a set of SQL statements depending on SQL Server (release 2014) to create users and roles, to assign roles to users, and to grant permissions to roles (See Figure 6.23),

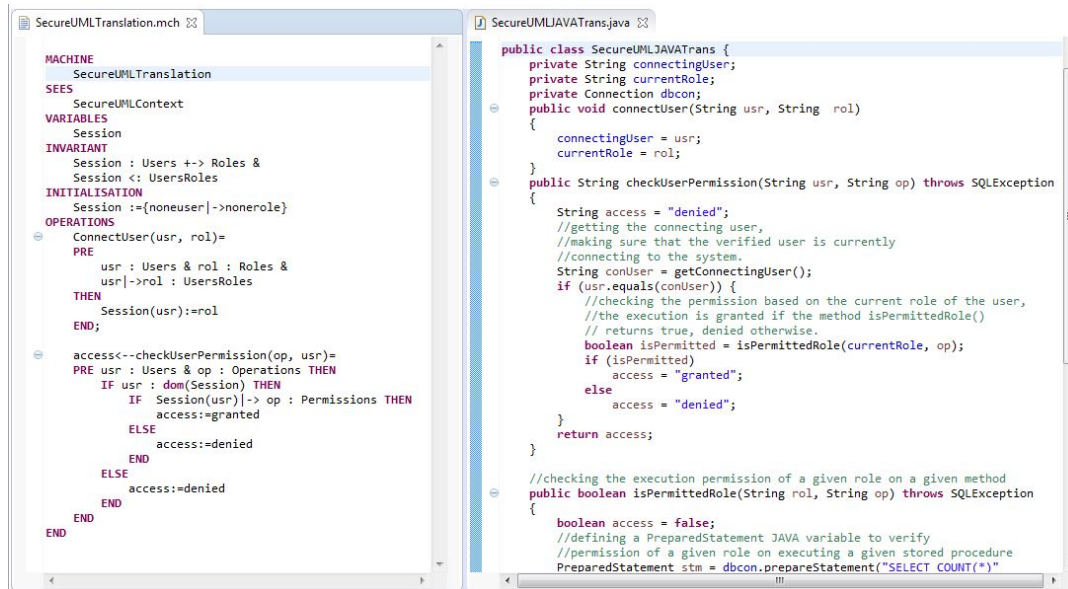


Figure 6.24: Generation of the class *SecureUMLJAVATrans*

- a JAVA class *SecureUMLJAVATrans* (See Figure 6.24) defining security checking methods (*connectUser*, *checkUserPermission*, and *isPermittedRole*)

6.6.3 Translating the B specification of the secure activity diagram

Taking the B specification of the secure activity diagram as an input, the tool gives:

- a log table and its associated JAVA class for each B table structure *T_Op_log*. For instance, Figure 6.26 and 6.25 depict the mappings of the variable *T_Receive_log* and its related operations into a SQL table and the JAVA methods on this table.
- a JAVA class *ActionsHistoryJAVATrans* encoding the log methods derived from the log operations. This translation is demonstrated through the operation *LogReceive* in Figure 6.27.
- a JAVA class *ADJAVATrans* deploying dynamic security-checking operations. For example, the operation *ADReceive* is mapped into a JAVA method as in Figure 6.28.


```

LogFile_receive.mch
VARIABLES
  T_Receive_log
INVARIANT
  T_Receive_log <: struct(purchaseOrderId: NAT,
  userExecutedReceive: Users, orderExecutionReceive: NAT)
INITIALISATION
  T_Receive_log := {}
OPERATIONS
  res<-NoReceive_log(po)=
  PRE po : NAT THEN
  res := bool(not(#log.(log : T_Receive_log &
  log/purchaseOrderId = po)))
  END;
  AddReceive_log(po, usr, time)=
  PRE po : NAT & usr : Users & time : NAT
  THEN
  T_Receive_log := T_Receive_log ∨
  {rec(po, usr, time)}
  END;
  res<-Receive_user(po)=
  PRE po : NAT THEN
  ANY log WHERE
  log : T_Receive_log &
  log/purchaseOrderId = po
  THEN
  res := log.userExecutedReceive
  END
  END;
  res<-Receive_order(po)=
  PRE po : NAT THEN
  ANY log WHERE
  T_Receive_log.java
public T_Receive_log(Connection conn) throws SQLException {
  dbcon = conn;
  stmt = conn.prepareStatement("SELECT *
  + "FROM T_Receive_log "
  + "WHERE purchaseOrderId=?;");
}
/**
 * automatic generation of the operation NoReceive_log
 */
public boolean NoReceive_log( int po ) {
  try {
    stmt.setInt(1,po);
    //execution of the statement that checks if the method
    //is already performed or not,
    //returns false if it is already executed, true otherwise.
    ResultSet res = stmt.executeQuery();
    if (res.next())
      return false;
    else
      return true;
  } catch (SQLException e) {e.printStackTrace();}
  return false;
}
/**
 * automatic generation of the operation AddReceive_log
 */
public void AddReceive_log(int po,String usr
  /*omit the datetime type that is automatically calculated in java/sql*/) {
  try {
    //defining a SQL statement to add a new record to the table T_Receive_log
    Statement stmt = dbcon.createStatement();
    String log = "INSERT INTO T_Receive_log"
    + "(purchaseOrderId, userExecutedReceive) "
    + "VALUES(* po +", "+ usr +")";
    //executing the statement
    stmt.executeUpdate(log);
  }
}

```

Figure 6.25: Generation of a log class

```

LogFile_receive.mch
VARIABLES
  T_Receive_log
INVARIANT
  T_Receive_log <:
  struct(purchaseOrderId: NAT,
  userExecutedReceive: Users,
  orderExecutionReceive: NAT)
INITIALISATION
  T_Receive_log := {}
T_Receive_log.sql
--SQL code of LogFile_receive
/*generate the SQL table from the B variable T_Receive_log*/
CREATE TABLE T_Receive_log {
  purchaseOrderId INT /*convert type NAT in B to INT in SQL*/ NOT NULL ,
  userExecutedReceive VARCHAR(25) NOT NULL ,
  orderExecutionReceive DATETIME DEFAULT CURRENT_TIMESTAMP
  /*the key is a set of attributes denoting the parameters of the method to log*/
  PRIMARY KEY(purchaseOrderId)
}

```

Figure 6.26: Generation of a log table

```

ActionsHistory_limp
14
15
16 OPERATIONS
17 LogPurchaseOrder_receive ( po , usr ) =
18 VAR po_id IN
19 po_id <- getPURCHASEORDERnb ( po ) ;
20 AddReceive_log ( po_id , usr , currentOrder ) ;
21 currentOrder := currentOrder + 1
22 END
23
ActionsHistory.JAVATrans.java
4 public class ActionsHistoryJAVATrans {
5   private T_Receive_log t_receive_log;
6   /**
7    * Automatic translation of B Operation: LogPurchaseOrder_receive( po, usr)
8    */
9   public String LogPurchaseOrder_receive(int po,String usr){
10    t_receive_log.AddReceive_log(po /*Replace po_id by po*/,usr)
11    /* Omit time variable currentOrder that is automatically calculated in java/sql */
12  }
13 }

```

Figure 6.27: Generation of the log method

6.6.4 Translation of the access control filter

The B specification of the filter results in an aspect class defining a pointcut and an associated advice for each filter operation. This transformation is illustrated by the filter operation *FilterPurchaseOrder_receive* in Figure 6.29.


```

ADTranslation_i_imp
8
9 OPERATIONS
10 access <- ADPurchaseOrder__receive ( po , usr )=
11 VAR po_id, notyet_approve, notyet_create,
12 create_user, create_order, approve_order
13 IN
14 po_id <- getPURCHASEORDERNb ( po ) ;
15 notyet_approve <- NoApprove_log ( po_id ) ;
16 notyet_create <- NoCreate_log ( po_id ) ;
17 create_user <- Create_user ( po_id ) ;
18 create_order <- Create_order ( po_id ) ;
19 approve_order <- Approve_order ( po_id ) ;
20 IF notyet_approve = FALSE &
21 notyet_create = FALSE &
22 usr /= create_user &
23 create_order < approve_order
24 THEN
25 access := granted
26 ELSE
27 access := denied
28 END
29 END
30 END

ADJAVATrans.java
12 /**
13 * Automatic translation of B Operation: ADPurchaseOrder__receive( po, usr)
14 */
15 public String ADPurchaseOrder__receive(int po,String usr){
16 String access = "denied";
17 boolean notyet_approve = t_approve.NoApprove_log(po /*Replace po_id by po*/);
18 boolean notyet_create = t_create.NoCreate_log(po /*Replace po_id by po*/);
19 String create_user = t_create.Create_user(po /*Replace po_id by po*/);
20 String create_order = t_create.Create_order(po /*Replace po_id by po*/);
21 String approve_order = t_approve.Approve_order(po /*Replace po_id by po*/);
22 if(notyet_approve == false &&
23 notyet_create == false &&
24 !usr.equals(create_user) &&
25 sdf.parse(create_order).before(sdf.parse(approve_order))
26 ){
27 access = "granted"
28 }else{
29 access = "denied"
30 }
31 return access;
32 }
33 }
34
    
```

Figure 6.28: Generation of the method *ADOP*

```

SecureFilter_i_imp
10
11 result,access<-FilterPurchaseOrder__receive(po, usr)=
12 VAR staticRight IN
13 staticRight
14 <-checkUserPermission(PurchaseOrder__receive_Label, usr);
15 IF staticRight = granted THEN
16 VAR dynamicRight IN
17 dynamicRight<-ADPurchaseOrder__receive(po, usr);
18 IF dynamicRight = granted THEN
19 result<-PurchaseOrder__receive(po);
20 IF result = OK THEN
21 LogPurchaseOrder__receive(po, usr)
22 END
23 END;
24 access := dynamicRight
25 END
26 ELSE
27 access := denied; result := KO
28 END
29 END
30 END

SecureFilter2AspectJ.aj
10
11 pointcut pc_PurchaseOrder__receive(int po):args(po)
12 &&call(* PurchaseOrder__receive(int));
13
14 String around(int po):PurchaseOrder__receive(po){
15 initAspectAFC();
16 String result = "KO";
17 String usr = SecureUMLJAVATrans.getConnectingUser();
18 try{
19 String staticRight=staticC.checkUserPermission("PurchaseOrder__receive",usr);
20 if (staticRight=="granted"){
21 String dynamicRight=dynamicC.ADPurchaseOrder__receive(po,usr);
22 if (dynamicRight=="granted"){
23 result = process(po);
24 if(result=="OK"){
25 logC.LogPurchaseOrder__receive(po,usr);
26 }
27 }
28 }
29 }
30 }catch (SQLException e) {}
31 return result;
32 }
33 }
34 }
35 }
36 }
    
```

Figure 6.29: Generation of the aspect class

6.7 Conclusion

In this chapter, we presented a systematic approach for developing a secure access control filter. The approach consists in designing the functionalities of the system to build using a UML class diagram and the static and the dynamic security concerns with SecureUML and activity diagrams respectively. These diagrams are then translated into a B formal specification following a set of well defined rules in order to be verified by establishing some invariant properties. To generate an executable implementation, the verified B specification is refined until obtaining a relational-like B implementation that is straightforwardly mapped into an AspectJ-based program.

Some approaches have been developed in the area of the security enforcement. In [14], transformation rules are proposed to generate an aspect for access control implementation from a SecureUML model. The validation of the transformation is made through the evaluation of some OCL constraints that

are evaluated on both the source and the target models. Even if the **SecureUML** model is implemented as a JAVA program that verifies the different user's rights which is different from our implementation, this work is close but the approach does not describe how the generated code can be mixed with the functional part of the application. An other approach introduced in [69, 70] consists in annotating the elements to protect in a JAVA program (i.e. methods, interfaces, and classes) with the roles that a user has to play to call them. The problem with such an approach is to know where annotations should be put without overloading the program. Moreover, this approach produces a scattered and tangled code. This is why an aspect oriented programming-based approach is chosen in this chapter in order to overcome these drawbacks by making a clear separation between the functional and security concerns as described in existing approaches with JAVA [15] for user-based access control and with C languages [31] for buffer overflow protection, log data, etc.

A next step in the presented work concerns the correctness of the derived **AspectJ** code. Our derivation process being done outside any formal environment, one cannot be sure of its correctness. In general, the correctness of a code generation algorithm is a very complex task. This problem has been addressed in [108] (for the C language) where the used structures of the target language are very closed to those of the B implementation. We think that the correctness of our translation process would require more in-depth analysis.

Conclusions and Future Work

Contents

7.1 Contributions	172
7.2 Future Work	173

Security is a central issue in information systems since any security bleaches may cause serious consequences for the organizations that use them. However, dealing with security properties of such systems is a difficult and tedious task. Developers tend to postpone the security concerns until the final phases of software development. In consequence, errors are often discovered late making them difficult and even impossible to fix. Therefore, it seems important to take into account security requirements since the early phases of system development.

This thesis aims at developing information systems and their access control mechanisms using formal techniques. We first specify the considered system and its access control requirements using UML-based languages. Even if graphical models give an intuitive view, but their semantics often cause the ambiguity of interpretation. Hence, we formalize these models in B formal language. The formal specification offers an unambiguous and a precise representation. Furthermore, it can be rigorously validated and verified, enabling to early errors detection. Finally, we refine the abstract specification until obtaining a concrete one, which can be easily translated into a trustworthy relational-based implementation. Such an implementation is based on an AOP paradigm in order to promote a separation of concerns.

7.1 Contributions

In this work, we started by reviewing the existing approaches related to our thesis subject. Our study of the state of the art was divided into four parts related respectively to specification techniques for access control policies, support tools, implementation methods for an access control specification, and security enforcement approaches. We then recapitulated the benefits and the drawbacks of the surveyed works. Consequently, we proposed three major contributions in order to provide a comprehensive MDE approach to build secure information systems. In this thesis, we are particularly interested in two types of security rules: static and dynamic. Static rules refer to a given single moment of the system without keeping the history of actions. Whereas dynamic ones require to take the execution history of the system into account, that is the actions already executed in the system in general or by a given user in particular. For example, in the case of an inventory, a static rule could be: *Only users playing the Manager role is permitted to approve an order*, and a dynamic rule could be: *The person who is trying to receive the goods of an order should not be the person who created that order*.

In the first contribution, we visualized security policies and functional requirements of a system using UML-based languages: SecureUML diagrams are used to represent the static aspect of access control, dedicated UML activity diagrams, called secure activity diagrams, are used to model dynamic security rules, and the functional requirements are introduced using a class diagram. Also at the platform independent level, we defined mapping rules of these graphical models into B. As such, the system specification is precise enough to be validated and verified using AtelierB [26] and ProB [27]. We also defined an access control filter where the functional and different security specifications are combined.

The second contribution uses the refinement technique of the B method. We defined transformation rules of the abstract specification targeting an AOP-based implementation. The obtained implementation follows the separation of concerns principle. Basically, the functional component is transformed into a relational-based application, while the security component is mapped into an AspectJ code. Thanks to the AspectJ weaver, the security check can be dynamically injected into the functional program.

Finally, we developed a tool to support the approach proposed in this thesis. In fact, we have extended the B4MSECURE tool [13] built by a team from the French LIG laboratory by translation rules to take class associations into account. We have also adapted the translation rules related to the SECUREUML

diagrams by adopting a defensive style mapping into a B specification. The modeling and the translation into B of secure activity diagrams have been introduced to support dynamic security rules. The tool is an Eclipse platform integrated the Topcased modeling environment. No modification is required for the validation and the verification of the generated B specifications. The implementation of these initial B specifications derives a trustworthy database application based on Java/SQL and AspectJ. The transformation of the formal implementation into an executable code is also performed automatically using our tool. The generated SQL code is correct with respect to the SQL Server syntax. The Java/SQL program is derived from the functional specification. Security enforcement policies are separately implemented in an aspect, making the code easier to track and maintain.

7.2 Future Work

This section suggests a number of possible directions for extending the research carried out in this dissertation. Limitations of our work open future perspectives to follow in short and long terms.

In this thesis, we proposed a transformation approach from the *Platform Independent Model* level to the *Platform Specific Model* level (i.e. from the B abstract specification to an AspectJ-like application connected to a relational database management system) based on the refinement technique of the B formal language. Yet, our refinement is currently performed manually. Although this hand-operated refinement is systematic, it requires a good background about mathematic notations and B refinement techniques. As a short term work, we plan to automate the refinement process.

As a long term future work, we aim to go further in the diversity of security concerns. In our approach, we solved various access control requirements, namely, essential access control policies, dynamic separation of duty constraints (history-based and ordering-based rules). It would be interesting to consider other constraints, such as delegation, prerequisite, and so on. We are aware that adding new kinds of security requirements will bring new challenges. But we are confident that the benefits of our approach will help us deal with it. Indeed, our approach allows the modularity of different aspects throughout the development life-cycle: from the design phase to the implementation phase. As a result, additional security constraints can enhance the safety and security of a system without having a remarkable impact on the existing components. To take into account new types of security constraints, other UML-based diagrams may be necessary, for example, sequence diagrams and state diagrams. A set of

translation rules is also required for the transformations from graphical models into B and from B specifications into Java/SQL.

Finally, we did not evaluate the scalability of our tool. We only used it for small systems with acceptable response times. It is important to experiment the tool on real-size systems and, if needed, optimize its performance and efficiency.

Bibliography

- [1] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.
- [2] David Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. Automated analysis of security-design models. *Information and Software Technology*, 51(5):815–831, 2009.
- [3] Michael E Shin and Gail-Joon Ahn. UML-based representation of role-based access control. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000.(WET ICE 2000). Proceedings. IEEE 9th International Workshops on*, pages 195–200. IEEE, 2000.
- [4] Gail-Joon Ahn and Michael E Shin. Role-based authorization constraints specification using object constraint language. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001. WET ICE 2001. Proceedings. Tenth IEEE International Workshops on*, pages 157–162. IEEE, 2001.
- [5] Indrakshi Ray, Na Li, Robert France, and Dae-Kyoo Kim. Using UML to visualize role-based access control constraints. In *Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 115–124. ACM, 2004.
- [6] Dae-Kyoo Kim, Indrakshi Ray, Robert France, and Na Li. Modeling role-based access control using parameterized UML models. In *Fundamental Approaches to Software Engineering*, pages 180–193. Springer, 2004.
- [7] Ruth Breu, Gerhard Popp, and Muhammad Alam. Model based development of access policies. *International Journal on Software Tools for Technology Transfer*, 9(5-6):457–470, 2007.
- [8] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):39–91, 2006.

-
- [9] Hongxin Hu and GailJoon Ahn. Enabling verification and conformance testing for access control model. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 195–204. ACM, 2008.
- [10] Manachai Toahchoodee, Indrakshi Ray, Kyriakos Anastasakis, Geri Georg, and Behzad Bordbar. Ensuring spatio-temporal access control for real-world applications. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 13–22. ACM, 2009.
- [11] Jérémy Milhau, Akram Idani, Régine Laleau, Mohamed-Amine Labiadh, Yves Ledru, and Marc Frappier. Combining UML, ASTD and B for the formal specification of an access control filter. *Innovations in Systems and Software Engineering*, 7(4):303–313, 2011.
- [12] J Milhau, M Frappier, F Gervais, and R Laleau. Systematic translation of EB3 and ASTD specifications in B and EventB. *Université de Sherbrooke, Rapport technique*, 30:v3, 2010.
- [13] A Idani and Y Ledru. B for modeling secure information systems-the B4MSecure platform. ICFEM, 2015.
- [14] Christiano Braga. A transformation contract to generate aspects from access control policies. *Software & Systems Modeling*, 10(3):395–409, 2011.
- [15] Bart De Win, Bart Vanhaute, and Bart De Decker. Security through aspect-oriented programming. In *Advances in Network and Distributed Systems Security*, pages 125–138. Springer, 2002.
- [16] Minhuan Huang, Chunlei Wang, and Lufeng Zhang. Toward a reusable and generic security aspect library. *AOSD: AOSDSEC*, 4, 2004.
- [17] D. Basin, J. Doser, and T. Lodderstedt. Model Driven Security: From UML Models to Access Control Infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1), 2006.
- [18] Ravi S Sandhu, Edward J Coynek, Hal L Feinstein, and Charles E Youmank. Role-based access control models yz. *IEEE computer*, 29(2):38–47, 1996.
- [19] Oracle. <https://www.oracle.com/database/index.html>.

-
- [20] Microsoft sql server 2014 express. <https://www.microsoft.com/en-us/download/details.aspx?id=42299>.
- [21] Jos B Warmer and Anneke G Kleppe. The object constraint language: Precise modeling with UML (addison-wesley object technology series). 1998.
- [22] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
- [23] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.
- [24] JR Abrial. The B book. 1996.
- [25] Nafees Qamar, Yves Ledru, and Akram Idani. Validation of security-design models using Z. In *International Conference on Formal Engineering Methods*, pages 259–274. Springer, 2011.
- [26] Atelier B. The industrial tool to efficiently deploy the B method. URL: <http://www.atelierb.eu/index-en.php> (access date 22.03. 2015), 2008.
- [27] Michael Leuschel and Michael Butler. ProB: A model checker for B. In *International Symposium of Formal Methods Europe*, pages 855–874. Springer, 2003.
- [28] Regine Laleau and Amel Mammar. An overview of a method and its support tool for generating B specifications from UML notations. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 269–272. IEEE, 2000.
- [29] Hung Ledang. Automatic translation from UML specifications to B. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, page 436. IEEE, 2001.
- [30] Colin Snook and Michael Butler. U2B-A tool for translating UML-B models into B. 2004.
- [31] John Viega, JT Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, 2001.

-
- [32] Bart De Win, Wouter Joosen, and Frank Piessens. Developing secure applications through aspect-oriented programming. *Aspect-Oriented Software Development*, pages 633–650, 2005.
- [33] Jaime Pavlich-Mariscal, Laurent Michel, and Steven Demurjian. A formal enforcement framework for role-based access control using aspect-oriented programming. In *Model Driven Engineering Languages and Systems*, pages 537–552. Springer, 2005.
- [34] Carliss Young Baldwin and Kim B Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.
- [35] John C Knight. Safety critical systems: challenges and directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 547–550. IEEE, 2002.
- [36] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor: A successful application of B in a large project. In *FM'99—Formal Methods*, pages 369–387. Springer, 1999.
- [37] Frédéric Badeau and Arnaud Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In *International Conference of B and Z Users*, pages 334–354. Springer, 2005.
- [38] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.
- [39] Richard Soley et al. Model driven architecture. *OMG white paper*, 308(308):5, 2000.
- [40] David Ferraiolo, D Richard Kuhn, and Ramaswamy Chandramouli. *Role-based access control*. Artech House, 2003.
- [41] David F Ferraiolo, John F Barkley, and D Richard Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):34–64, 1999.
- [42] Richard T Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 183–194. IEEE, 1997.

-
- [43] Torsten Lodderstedt, David Basin, and Jürgen Doser. SecureUML: A UML-based modeling language for model-driven security. In *UML 2002—The Unified Modeling Language*, pages 426–441. Springer, 2002.
- [44] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.
- [45] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.
- [46] Jos B Warmer and Anneke G Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.
- [47] Gail-Joon Ahn and Hongxin Hu. Towards realizing a formal RBAC model in real systems. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 215–224. ACM, 2007.
- [48] Bruno T Messmer and Horst Bunke. *Subgraph isomorphism in polynomial time*. Citeseer, 1995.
- [49] Karsten Sohr, Gail-Joon Ahn, Martin Gogolla, and Lars Migge. Specification and validation of authorisation constraints using UML and OCL. In *European Symposium on Research in Computer Security*, pages 64–79. Springer, 2005.
- [50] Mark Richters et al. *A precise approach to validating UML models and OCL constraints*. Citeseer, 2002.
- [51] Paul Ziemann and Martin Gogolla. An OCL extension for formulating temporal constraints. *Rapport technique, Universitat Bremen*, page 87, 2003.
- [52] Till Mossakowski, Michael Drouineaud, and Karsten Sohr. A temporal-logic extension of role-based access control covering dynamic separation of duties. In *Temporal Representation and Reasoning, 2003 and Fourth International Conference on Temporal Logic. Proceedings. 10th International Symposium on*, pages 83–90. IEEE, 2003.

-
- [53] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security for process-oriented systems. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 100–109. ACM, 2003.
- [54] Andreas Schaad and Jonathan D Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 13–22. ACM, 2002.
- [55] John Zao, Hoetech Wee, Jonathan Chu, and Daniel Jackson. RBAC schema verification using lightweight formal model and constraint analysis. *Submitted to SACMAT*, 2003.
- [56] Behzad Bordbar and Kyriakos Anastasakis. UML2ALLOY: A tool for lightweight modelling of discrete event systems. In *IADIS AC*, pages 209–216, 2005.
- [57] Anthony Boswell. Specification and validation of a security policy model. *IEEE Transactions on Software Engineering*, 21(2):63, 1995.
- [58] Shoichi Morimoto, Shinjiro Shigematsu, Yuichi Goto, and Jingde Cheng. Formal verification of security specifications with common criteria. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1506–1512. ACM, 2007.
- [59] Ali E Abdallah and Etienne J Khayat. Formal z specifications of several flat role-based access control models. In *2006 30th Annual IEEE/NASA Software Engineering Workshop*, pages 282–292. IEEE, 2006.
- [60] Chunyang Yuan, Yeping He, Jianbo He, and Zhouyi Zhou. A verifiable formal specification for RBAC model with constraints of separation of duty. In *International Conference on Information Security and Cryptology*, pages 196–210. Springer, 2006.
- [61] Sophie Dupuy, Yves Ledru, and Monique Chabre-Peccoud. An overview of roz: a tool for integrating UML and Z specifications. In *International Conference on Advanced Information Systems Engineering*, pages 417–430. Springer, 2000.
- [62] M Utting. Jaza: Just another z animator, 2005.

-
- [63] Marc Frappier, Frédéric Gervais, Régine Laleau, Benoit Fraikin, and Richard St-Denis. Extending statecharts with process algebra operators. *Innovations in Systems and Software Engineering*, 4(3):285–292, 2008.
- [64] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.
- [65] Marc Frappier and Richard St-Denis. EB3: an entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2(2):134–149, 2003.
- [66] Kevin Lano, David Clark, and Kelly Androutsopoulos. UML to B: Formal verification of object-oriented models. In *International Conference on Integrated Formal Methods*, pages 187–206. Springer, 2004.
- [67] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 730–733. IEEE, 2000.
- [68] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User authentication and authorization in the Java platform. In *Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual*, pages 285–290. IEEE, 1999.
- [69] Jeff Zarnett, Mahesh Tripunitara, and Patrick Lam. Role-based access control (RBAC) in Java via proxy objects using annotations. In *Proceedings of the 15th ACM symposium on Access control models and technologies*, pages 79–88. ACM, 2010.
- [70] J. Fischer, D. Marino, R. Majumdar, and T. Millstein. Fine-Grained Access Control with Object-Sensitive Roles. In *ECOOP–Object-Oriented Programming*. Springer, 2009.
- [71] Shu Gao, Yi Deng, Huiqun Yu, Xudong He, Konstantin Beznosov, and Kendra Cooper. Applying aspect-orientation in designing security systems: A case study. In *SEKE*, pages 360–365, 2004.
- [72] Indrakshi Ray, Robert France, Na Li, and Geri Georg. An aspect-based approach to modeling access control concerns. *Information and Software Technology*, 46(9):575–587, 2004.

- [73] Djedjiga Mouheb, Chamseddine Talhi, Mariam Nouh, Vitor Lima, Mourad Debbabi, Lingyu Wang, and Makan Pourzandi. Aspect-oriented modeling for representing and integrating security concerns in UML. In *Software Engineering Research, Management and Applications 2010*, pages 197–213. Springer, 2010.
- [74] Bart De Win, Bart Vanhaute, and Bart De Decker. How aspect-oriented programming can help to build secure software. *Informatica*, 26(2):141–149, 2002.
- [75] Jonathan Knudsen. *Java cryptography*. " O'Reilly Media, Inc.", 1998.
- [76] A. Mammar, T-M. Nguyen, and R. Laleau. Formal Development of a Secure Access Control Filter. In *17th IEEE International Symposium on High Assurance Systems Engineering, (HASE)*, pages 173–180, 2016.
- [77] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the 5th International Conference on The Unified Modeling Language, UML '02*. Springer-Verlag, 2002.
- [78] J-R. Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge University Press, 2005.
- [79] UML2. Unified modeling language: Superstructure. Object Management Group, version 2.4, 2011.
- [80] Clearsy. Atelier De Génie Logiciel Permettant De Développer Des Logiciels Prouvés sans Défaut. <http://www.atelierb.eu/>.
- [81] P. Konopacki. *Une Approche Evenementielle Pour La Description De Politiques De Controle D'Acces*. PhD thesis, 2012.
- [82] R-S. Sandhu, E-J. Coyne, H-L. Feinstein, and C-E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2), 1996.
- [83] J. Milhau, A. Idani, R. Laleau, M-A. Labiadh, Y. Ledru, and M. Frappier. Combining UML, ASTD and B for the formal specification of an access control filter. *ISSE*, 7(4), 2011.
- [84] M. Frappier, F. Gervais, R. Laleau, B. Fraikin, and R. St.-Denis. Extending statecharts with process algebra operators. *ISSE*, 4(3):285–292, 2008.

-
- [85] S. Dupuy, Y. Ledru, and Mo. Chabre-Peccoud. An Overview of RoZ: A Tool for Integrating UML and Z Specifications. In *Advanced Information Systems Engineering, 12th International Conference (CAiSE 2000)*, 2000.
- [86] H. Ledang and J. Souquières. Contributions for Modelling UML State-Charts in B. In *Integrated Formal Methods, Third International Conference, IFM, 2002*.
- [87] H. Treharne. Supplementing a UML Development Process with B. In *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, 2002*.
- [88] Y. Chen and H. Miao. From an Abstract Object-Z Specification to UML Diagram. *Journal of Information & Computational Science*, 1(2), 2004.
- [89] A. Idani, Y. Ledru, and D. Bert. Derivation of UML Class Diagrams as Static Views of Formal B Developments. In *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods (ICFEM)*, 2005.
- [90] R. Laleau and A. Mammar. An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations. In *ASE*, 2000.
- [91] A. Mammar and R. Laleau. A formal approach based on UML and B for the specification and development of database applications. *Automated Software Engineering*, 13(4), 2006.
- [92] S. Preda, N. Cuppens-Boulahia, F. Cuppens, J. García-Alfaro, and L. Toutain. Model-Driven Security Policy Deployment: Property Oriented Approach. In *Engineering Secure Software and Systems, Second International Symposium, (ESSoS)*, 2010.
- [93] M. Toahchoodee, I. Ray, K. Anastasakis, G. Georg, and B. Bordbar. Ensuring spatio-temporal access control for real-world applications. In Barbara Carminati and James Joshi, editors, *14th ACM Symposium on Access Control Models and Technologies, (SACMAT) Proceedings*. ACM, 2009.
- [94] N. Qamar, Y. Ledru, and A. Idani. Validation of Security-Design Models Using Z. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM*, volume 6991 of *Lecture Notes in Computer Science*. Springer, 2011.

- [95] A. Idani, Y. Ledru, and A. Radhouani. Modélisation graphique et validation formelle de politiques RBAC en systèmes d'information. plateforme B4MSecure. *Ingénierie des Systèmes d'Information*, 19(6):33–61, 2014.
- [96] M. Leuschel and M-J. Butler. ProB: A Model Checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods, International Symposium of Formal Methods Proceedings*, volume 2805 of *Lecture Notes in Computer Science*. Springer, 2003.
- [97] T-M. Nguyen, A. Mammam, R. Laleau, and S. Hameg. A Tool for the Generation of a Secure Access Control Filter. In *10th IEEE International Conference on Research Challenges in Information Science (RCIS)*, 2016.
- [98] H. Ledang and Jeanine S. Contributions for modelling UML state-charts in B. In *Integrated Formal Methods*. Springer-Verlag, 2002.
- [99] H. Treharne. Supplementing a UML Development Process with B. In *FME 2002: Formal Methods—Getting IT Right*. Springer, 2002.
- [100] A. Idani, Y. Ledru, and D. Bert. Derivation of UML Class Diagrams as Static Views of Formal B Developments. In *Formal Methods and Software Engineering*. Springer, 2005.
- [101] K-C. Mander and F. Polack. Rigorous Specification using Structured Systems Analysis and Z. *Information and Software Technology*, 37(5), 1995.
- [102] <http://b4msecure.forge.imag.fr/>.
- [103] A. Mammam, T-M. Nguyen, and R. Laleau. Formal Development of a Secure Access Control Filter. In *17th International IEEE Symposium on High-Assurance Systems Engineering, HASE*. IEEE Computer Society, 2016.
- [104] A. Mammam and R. Laleau. From a B formal specification to an executable code: application to the relational database domain. *Information & Software Technology*, 48(4):253–279, 2006.
- [105] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.
- [106] J-L Boulanger. Abtools: Another B tool. In *Application of Concurrency to System Design, 2003. Proceedings. Third International Conference on*, pages 231–232. IEEE, 2003.

-
- [107] TJ Parr and RW Quong. Antlr: A predicated. *Software—Practice and Experience*, 25(7):789–810, 1995.
- [108] D. Bert, S. Boulmé, M-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, 2003.