University of Namur

DOCTOR OF SCIENCES

Formal Modeling and Verification of Access-Control Policies

Toussaint, Hubert

Award date:
2011

Awarding institution:
University of Namur

Link to publication

Facultés Universitaires Notre-Dame de la Paix, Namur

Faculté d'Informatique

Année Académique 2011-2012

Formal Modeling and Verification of Access-Control Policies

Hubert Toussaint

September 2011

Thèse présentée en vue de l'obtention du grade de Docteur en Sciences.

PhD Committee :

- **Prof. Dr. Jean-Marie Jacquet** (President)
  University of Namur, Belgium.
- **Prof. Dr. Ir. Pierre-Yves Schobbens** (Promotor)
  University of Namur, Belgium.
- **Prof. Dr. Jean-Noël Colin** (Internal Reviewer)
  University of Namur, Belgium.
- **Prof. Dr. Ir. Yves Le Traon** (External Reviewer)
  University of Luxembourg, Luxembourg.
- **Dr. Charles Morisset** (External Reviewer)
  CNR/IIT/ Information Security Group, Pisa, Italy.

Abstract

The construction of secure software is a notoriously difficult task. The abstract security requirements have to be turned into functional requirements and then implemented. However, only few techniques allow to verify that the implemented elements fulfill the originally expressed requirements. The potential gap between the specification and the implementation gets even wider with iterative development schemes where code (and sometimes specification) is updated numerous times.

In this document we propose a methodology aimed at facilitating the co-evolution of the security requirements and the implemented code. Focusing on the access-control perspective, we provide models and algorithms to specify the expected requirements and to extract the implemented access-control rules directly from the executable source code. Then we verify the conformance of the implemented features towards the specified requirements and, if inconsistencies are found, we provide potential corrective measures that can be applied directly into the source code.

keywords : access-control, security policy, verification, model extraction, model weaving

Acknowledgements

"To our parents who implemented us from genetic specifications and debugged us as best they could."

L. F. Johnson & R. H. Cooper
in File Techniques for Databases Organization in Cobol.

When this adventure started a few years ago, little did I expect that leading a doctoral research would be such a unique blend of sweet and sour paradoxes, with its faire share of upsides and downsides. As it has already been said elsewhere, on the one hand, being a PhD student is a lonely, laborious and worrisome experience. But on the other hand, it offers so many positive boons and rewards that I would rather only remember the latter. Indeed, during this timespan, I notably got the chance to inquire about many different and exciting topics and technologies, while meeting so many educated, interesting and caring people, always ready to provide their help and cheer you up.

Among these persons, I would first of all like to express my gratitude and appreciation to my supervisor, Prof. Pierre-Yves Schobbens, who notably gave me the opportunity to embrace the challenge of pursuing a PhD. Thank you for teaching me to become more autonomous and proactive in my research through your confidence and the latitude you granted me during these years. Thank you for your time, your precious advices and your insight on my work. Not forgetting the chance of being chosen as teaching assistant. Teaching and supervising students at the Faculty of Computer Science was indeed a fulfilling and enjoyable experience as well.

Next, I would like to thank the members of my jury, Prof. Jean-Marie Jacquet, Prof. Jean-Noël Colin, Prof. Yves Le Traon and Dr. Charles Morisset for accepting to participate in the evaluation of my work. It was an honour and a pleasure to discuss my research with you, especially given your encouragements and valuable feedback.

I would then like to thank my past and present colleagues of the Faculty of Computer Science. Thank you for your concern, your support, and all these good times that undoubtedly made us more than simple co-workers. You provided me with a stimulating and fun environment in which to learn and grow.

I would also like to thank our top-notch secretaries, Anne-Marie, Babette and Isabelle for making the organisation and practical progress of my work so much easier. Thank you also for your kindness and consideration, beyond the mere doctoral preoccupations.

More generally, I would like to thank all the other friends, colleagues and unsuspected strangers who rooted for me during my doctoral tenure. There are too many of them to mention them all, but I am truly grateful for all the consideration and support that I received. Thank you.

Last but not least, I wish to thank the members of my family for their unconditional love and support. In particular, my wife for her understanding and encouragements. Thank you for all these precious moments together, and for giving me the strength to carry on through enduring times. Thank you for being there no matter what, especially during this last year which has been incredibly rich in changes and challenges.

H. Toussaint,
September 2011

Contents

List of Figures

Introduction

Building secure software is a notoriously complex task: from the specification of requirements to the production of code, developers must make sure that any change they make (for instance for performance reason or to comply with late-customer requirement update) does not break the desired properties. Multiplying code updates, iterative software development further increases this risk. The costs, both in terms of time and money, of those updates increase as the delivery-time approaches. Due to those rising costs, late modifications are often applied directly to the source code without checking thoroughly their impact on the security model.

Most of the time, the "validation" of the new/updated feature is only done through "clever" programmer affirmation and/or some limited testing. Such a process makes it very difficult to detect every possible model corruption occurring due to the change. Several approaches have been developed to address this problem: automating the testing process like in JUnit [JUn10], maximizing test-coverage [LTMPB08] [MLTB09] [MFBLT08], automated changes specification/propagation like in Aspect-Oriented Programming [KLM$^+$97] or UMLSec [Jur05] or complete formal approach like the B-methodology [Abr96] or abstract interpretation [CDH$^+$00], [HP98]. The problem with such methodologies is that developers updating the code, albeit mastering the chosen development language, usually have little to no knowledge of models and automated proof. So they have to choose between understandable incomplete results given through limited testing or cryptic model-checking results abstracted from their code. Neither of those is satisfactory in terms of completeness and usability.

In our opinion, both the development process and the resulting software quality and security can be improved with the introduction of helper tools and methodologies based on formal methods. Our contribution consists in the proposal and precise definition of a methodology and an environment supporting such an helper tool. We will evaluate to what extent an automated security property extraction directly from the source code, its verification and, if applicable, its correction is feasible. And if positive, we will investigate the level of developer interaction required for this process.

The models and algorithms needed to assist the developer to find any side-effects following updates in the code will be presented. For simplicity, we will focus on the aspect-control perspective and offer the developers a methodology able to automatically make explicit the impacts of changes made to the source code on the desired security model and properties.

The two first chapters of this document delimit the field in which this reflexion takes place. Chapter 1 introduces the various aspects of software security and clarifies the notions of identification, authentication and authorization. Then a special focus is done on the access-control perspective and its particularities. Chapter 2 presents an overview of the current set of techniques / methodologies aimed at avoiding or limiting the impact of vulnerabilities on the resulting software system. The various approaches are detailed and confronted to the usual pitfalls of a real-world software development process.

Chapter 3 introduces the motivating problem from which our research takes source. It describes the scope of our approach and the set of problems frequently encountered by software developers we try to alleviate. It also introduces the running example that will be used every time applicable throughout this document to illustrate the methodology and the algorithms.

Chapters 4 to 6 detail our proposed methodology. The models and algorithms related to the extraction of the access-control model directly from the source code and the modeling of the considered software system are provided in chapter 4. Chapter 5 focuses on the verification of these models and properties. It also adresses the adaptation and reporting of the verification results back to the users, both in the model and directly into the source code. Then chapter 6 details the weaving of the resulting models back into the executable code

Finally, chapter 7 summarizes the achievements and setbacks of the presented methodology. The elements learned from the application of the methodology to the case study are discussed, followed by a general evaluation of the method and paths for potential future improvements. Chapter 8 then presents a quick overview of the prototype tool developed to support the presented methodology.

1

Software Security

Contents

This chapter introduces software systems and the related aspects of security. A quick introduction to the notions of identification, authentication and authorization will be also presented, followed by some notions of access-control.

1.1 Software System

The term “software system” covers a much broader scope than a simple assembly of pieces of code; it usually consists of a set of programs developed for a specific hardware in order to fulfill some requirements. It can be described as “*a system based on software forming part of a computer system (a combination of hardware and software). The term "software system" is often used as a synonym of computer program or software; is related to the application of systems theory approaches in software engineering context and are used to study large and complex software, because it focuses on the major components of software and their interactions; and also related to the field of software architecture.*”[1]

With such a definition, a software system can be anything from a simple phone-embedded controller to complex operating systems like Unix or Windows running on distributed hardware configurations. Subsystems of a software system can also be considered as a software system: database software, specialized librairies or a graphical interface are examples of autonomous software systems.

More formally, we’ll define a software system as any piece of software used by one or more users (be it human or another software system) in order to perform one task.

1.2 Software Security

With the proliferation of software systems in our environment, more and more aspects of our every day life are managed through software: paiements are done via electronic credit-cards, communications use computer networks, our lives and identities are stored in multiple databases... The need for security becomes ever more important. More or less recent events showed that insecure software can affect people’s life in many ways.

- the loss of the details of three million candidates for the driving theory test in the UK, jeopardizing some aspects of their private life. [2]
- Ariane 5 Flight 501[3], which resulted in the crash and total loss of the multi-million rocket.
- stolen CD with the names of about 1500 German citizens hiding cash away in Switzerland, bought by the German finance ministry. [4]

[1] feb. 2011, https://secure.wikimedia.org/wikipedia/en/wiki/Software_system

[2] dec. 2007 http://news.bbc.co.uk/2/hi/uk_news/politics/7147715.stm

[3] jun. 1996 http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=2

[4] feb. 2010 http://www.dw-world.de/dw/article/0,,5296146,00.html

- exposed JPMorgan Chase customers information after a breach at a marketing sub-contractant, once again exposing private clients information. [5]
- and many more...

Software security is a subjective notion varying with time and environment: depending on the point of view, an application can be considered secure if no unwanted behavior (with respect to the security policy) did ever happen, or will ever happen, or will ever happen as long as the application is properly maintained. Explicitly introducing the attacker in the equation makes it even more complex: the resources of the attacker can make the difference between a secure application and an insecure one. For example, online shopping websites usually protect their clients banking information with cryptography (AES, RC4, ...) to prevent this information from being intercepted and possibly used to buy unwanted goods. While providing sufficient security for lambda people, these cypherings are far from unbreakable when attacked via brute-force attacks. In the same way as no lock is unbreakable, no software can be ultimately secure. Instead, security is often the result of a cost-benefits trade-off : the level of security is raised high enough to, hopefully, prevent any attacker from gaining access to sensitive elements without spending a disproportionate amount of resources in the process.

Software security encompasses measures taken throughout the application's life-cycle to prevent unwanted behavior of the application and of the underlying system with respect to the security policy. These unwanted behaviors can be caused by flaws in the design, development, deployment, upgrade, or maintenance of the application.

But, where no architect could possibly have the idea to start building a bridge before proving his plans correct according to laws of physics and current domain best-practices, many software developers still start building software from scratch with little to no formal specification. This behavior makes it almost impossible to fully guarantee that the resulting software system is secure according to the client's requirements. Most of the time, the security of the software system relies, on one hand, on the confidence of the client in the provided tests on sample cases that the system has satisfied and, on the other hand, on an act of trust on the developer abilities to produce secure systems. Of course, such a "security by trust" may not always be satisfactory:

- the client might not want to engage his own name or liability in a software product that may leak his own business private information.

- the developer might not be able to bear the liability of a security leak in his client facility. For instance, the developer might not be able to insure himself at a reasonable price for potential liabilities

[5] apr. 2011 : `http://www.reuters.com/article/2011/04/01/epsilon-idUSL3E7F13F420110401`

caused by his software in case of malfunction.

To avoid this problem, it is usually considered a best practice in computer science to specify as precisely as possible the “key” features of the expected software system. This allows to check more in-depth the software system features according to the required behavior, thus reducing as much as possible the trust needed on the developer.

1.2.1 Identification, Authentication and Authorization

It is important, at this point, to clarify the notions of identification, authentification and authorization:

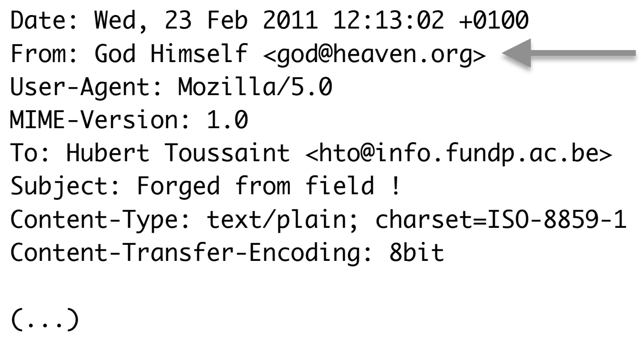
Identification is the capability to name without ambiguity a specific element. In a software context, the identification of an object or a user is the mapping between this concept and its own logical representation. For instance, inserting a smartcard into a card reader or giving a username to the system are identification methods; they allow the system to determine who is trying to access it.

Authentication is the process of verifying that the given identification is correct. It can take many forms: challenging the smartcard, asking the user to give a password known only by the user and the system, or any suitable method.

Authorization is the process of verifying that an identified and authenticated subject has then permission to perform a given operation or access a specific resource on the system. Authorization often involves checking the subject credentials against some defined rules.

In a security perspective, these three terms are strongly tied: authentication is useless without proper prior identification and no authorization decision can be made if the subject is not properly identified and authenticated first (figure 1.2).

It must be noted that, in many situations in computer science, identification is used without proper authentication, resulting on systems based on mutual trust and/or providing very limited guarantees. For instance, the SMTP protocol [Pos82], responsible for the delivery of emails over the internet, lets the sender freely specify what the source adress is (the “From:” field of the SMTP header, see figure 1.1). The sender of the mail can identify himself as whoever he wants without being asked for any authentication. SMTP servers usually identify and authenticate users on their IP adresses : allowing mails from users located in their dedicated subnetwork and refusing all others. If the recipient of a mail wants to verify that his newly received email really originates from the declared sender, he must use external authentication methods like phoning the expected sender for oral confirmation or asking the sender to cryptographically sign his mail. This lack of authentication allows spam-bots to forge emails in an attempt to lure people into thinking the email originates from a known friend or family member. Receivers of such mails are often less cautious on the mail content and may be more induced in following links to phishing or unwanted ads sites.



```
Date: Wed, 23 Feb 2011 12:13:02 +0100
From: God Himself <god@heaven.org>
User-Agent: Mozilla/5.0
MIME-Version: 1.0
To: Hubert Toussaint <hto@info.fundp.ac.be>
Subject: Forged from field !
Content-Type: text/plain; charset=ISO-8859-1
Content-Transfer-Encoding: 8bit

(...)
```

Figure 1.1: *Mail header with forged "From:" field.*

The mail format, and the underlying SMTP protocol, makes no verification at all on the validity of the information given in the header.

The rest of this document focuses on the authorization perspective. The problematics of the identification and the authentication of subjects, already broadly covered by the literature, will be left aside. The interested reader can find more informations on those topics in [US 85] and [And01].

1.3 Access Control

The next logical step after identifying and authenticating a user is to define precisely what he should be authorized to do, i.e. what he should be able to access and what he should not. The set of operations required for this process are usually grouped under the term "access-control". This section introduces the most commonly used models in the specification of ressource access.

The three key properties expected from an access-control policy are :

- confidentiality: private information must stay secret and private.
- integrity: information must be protected against unwanted modifications and/or modifications caused by unauthorized user.
- availability: information must be accessible from where it is required.

By restricting the set of users able to access the information, access-control, if well configured, can guarantee confidentiality. The same holds for integrity: if only allowed users access the information, it is very unlikely that they will temper it. Availability is a little different: while it is obvious that a user allowed to access the information has more ease to cause the system to malfunction than an external attacker, access-control alone cannot guarantee availability. Distributed denial of services (DDOS) are an example of that limitation: even without accessing the protected information on the server, a sufficiently large amount of forged queries can take almost any server down, preventing legitimate users to access their required resource.

For a complete story on evolution of access-control, the reader is invited to consult [FKC03], [Tou06] and [SSS94].

1.3.1 Basic elements

Before going any further in access-control policies, let's clarify the notions of user, subject, permission, operation and object and their relations in the context of access-control (figure 1.2):

User: a user refers to the person manipulating the system. Most of the time a single user can have several virtual identities (for instance, multiple logins) and may want to access several of those at the same time. Accurately matching users and subjects is the role of the identification/authentication routines.

Subject: a subject is the software process acting on behalf of the user. It represents the "virtual arm" of the user, the only way it can access the software system.

Operation: an operation is a software process invoked by a subject on an object in the system.

Object: an object represents any software or hardware resource on the considered system. It may be databases, peripherals, or memory areas for instance. Objects are typically viewed as passive entities, giving or receiving information. Some models support more complex objects like programs or permissions.

Permission: a permission (also sometimes called a privilege) is an authorization for a specific subject to perform a specific action on a specific object of the system. It consist in the combination of an object, a subject and an operation. A permission can also depend on the context in which it is asked : some operations may be allowed on certain hours, or only if certain conditions are met.

Aside from permissions authorizing (or not) some users to perform some actions on some objects, access-control systems often try to achieve higher level properties like least-privilege or separation of duty.

- The least-privilege property states that a user should only be granted access to objects strictly required to perform his duty, no more. It is often used to prevent unwanted information leaks. For instance, in a banking environment, a cashier may be granted access on his clients accounts to perform their operations when asked, but not on all the other client's accounts and not outside office hours.

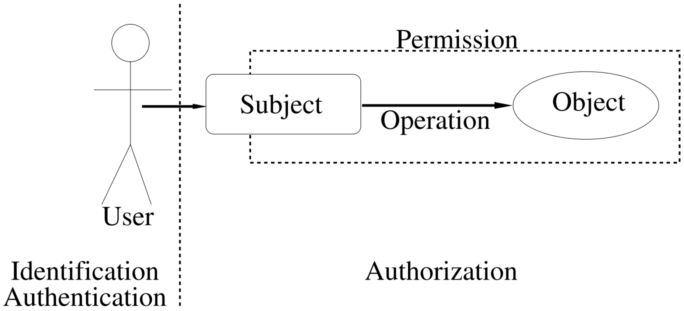


Figure 1.2: *Simple permission model*

- Separation of duty on the other hand tries to enforce that some operations cannot be conducted together by a single subject. For instance, once again in a banking environment, a good practice could be to allow a clerc to make paiements to customer and to approve (or not) what other clercs do. But he should never be able to approve his own work.

1.3.2 Access Control Policy

An access-control policy is a set of rules defining which resources are accessible to which user, possibly depending on the context of the action. It regroups all the individual permissions granted to the different users of the system. The basic question it answers is “Who can do what and when”.

Independently of the constraints imposed by the chosen specification model and its implementation, the definition of an access-control policy is a political choice before being a technological one. Considering the access to a particular resource, several options are possible: ranging from only allowing access to users requiring it to allowing access to any but some “forbidden” ones. Which one to choose depends on the political and strategical choices of the organisation wherein they take place. The result also often depends on a cost-benefits analysis: the more detailed the policy is, the more time it takes to write it and (most of the time) the more complex it is to implement and manage.

Hereunder are presented the most prevalent access-control models found in current software systems: DAC, MAC and RBAC. Readers interested by a more in-depth analysis of these models can consult [Tou06], [Mor07], [HFK06] and [FSG$^+$01].

1.3.2.1 DAC

Discretionary Access Control, or DAC, defined by the US Department of Defense in the Trusted Computer System Evaluation Criteria [US 85] "*as a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject which they belong*"' is an access-control model where subjects owning a particular permission can, under certains circumstances, grant other subjects this permission (hence the name discretionary). The same applies to the revocation of permissions.

A DAC access-control policy can be modelled as a permission-matrix (figure 1.3): assigning to each user the set of permissions granted on a each object.

Subject	Object 1	Object 2	Object 3	Object 4
Marc	R,W,X	R	W	
Paul	X			R,X
Pierre				R,W,X

Figure 1.3: *Permission matrix*

Each user is granted a combination of read(R), write(W) and execute(X) permissions on each object. The owner of an object can decide, at any time, to grant/revoke any permission he wish to any user.

Such a system is easily customizable to reflect almost every possible policy choices. But, as soon as the number of users and/or objects grows, the matrix becomes unmanageable. Any new object or new user on the system requires the administrator to properly set his initial rights. Relying entirely on the object owners to set permissions makes DAC systems very vulnerable to users's mistakes and trojan attacks. Furthermore, in this model, it is very complex to express properties like interdiction, role, task or context because the permission matrix represents a statical view of the permission granted on the system.

To simplify a little bit the management of such a huge permission matrix, several enhancements to the original model have been introduced. The first one is the notion of group: a group represents the set of users belonging to that group. So instead of assigning to every user its permissions on every object, the administrator has the choice to group users requiring similar privileges in groups and then to assign permissions directly to those groups (figure 1.4). Depending on the implementation, groups of groups may be available or not. The most frequent implementation of DAC, as found on many UNIX systems, only allows the specification of the permissions of the owner of the object, his group and to "others", representing all the users of the system but the owner and the member of his group (figure 1.5).

Groups composition :

user:marc, pierre, fred, root

compta: marc, pierre

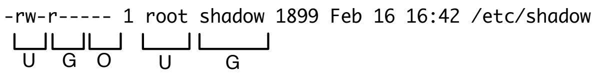
admin:root

Ressources

Group	Object 1	Object 2	Object 3	Object 4
user	R,W,X	R	W	
compta	R,W,X			R,X
admin	R,W,X	R,W,X	R,W,X	R,W,X

Figure 1.4: *Permission matrix based on groups*

Subject 'marc' is granted read(R) access on objects 1, 2 and 4. 1 and 2 as a member of group "user", 4 as a member of group "compta"'



```
-rw-r----- 1 root shadow 1899 Feb 16 16:42 /etc/shadow
```

Figure 1.5: *UNIX permission specification*

The permissions on the file are expressed in terms of user permission (U), here granting read(r) and write(w) access to user 'root', group permission (G), granting read(r) access to all members of the 'shadow' group and others (O) granting no permissions.

The concept of groups eases the specification of the policy, but makes it very complex to trace the origin of one specific permission, which may be very important in the case where a change occurs in an user authorization level. Each object belonging to this user and any group he is member of have to be checked and eventually have their permissions updated to reflect the new situation.

Another enhancement of the classical DAC model are the Access-Control Lists (ACL). In an ACL policy, permissions can be granted to groups and to specific users (figure 1.6 shows an ACL records taken from a

Linux system). A user then receives all permissions granted to the groups he belongs to, plus the specific permissions he has been personally granted.

```
# file: acldemo
# owner: root
# group: root
user::rw-
user:gsc:r-x
user:hto:rwx
user:pys:--x
group::---
group:staff:r-x
mask::rwx
other::---
```

Figure 1.6: *Access-Control List (ACL)*

The resource "acldemo" is accessible by :

- *his owner, with read and write privilege.*
- *to the gsc, hto and pys users with their respective rights as specified.*
- *to all members of the "staff" group, with read and execute privilege.*
- *nobody else.*

Despite being more flexible than flat groups specification, ACL tend to suffer from the same weaknesses: adding new resources or users to the system or modifying a user authorization level still require a check of all the resources he may have been granted access on which is often impractical and error-prone on systems with a large number of objects.

1.3.2.2 MAC

At the opposite of the DAC model lies the Mandatory Access Control, or MAC, also defined in the Trusted Computer System Evaluation Criteria [US 85] as "*a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity*". MAC is an access-control policy model where the system restricts the ability of a user to perform an action on an object depending on their respective "clearance" levels.

In a MAC system, user are assigned a clearance level. The objects they are allowed to access and the way they are allowed to access them depends on it. Summarized as a “no read up, no write down” policy, a system implementing MAC will forbid users to read objects with a clearance level above their own and to write information into objects with a clearance level strictly underneath their own. It should be noted that the clearance levels of an user and an object mays not always be comparable : clearance levels can be organized as lattices and/or only support a partial ordering. Figure 1.7 illustrates these constraints: subject s_1 can read o_1, o_2 and o_3, but can’t write in any of them because their clearance levels are strictly below his own, whereas subject s_2 can read o_3, write o_1 and read/write o_1.

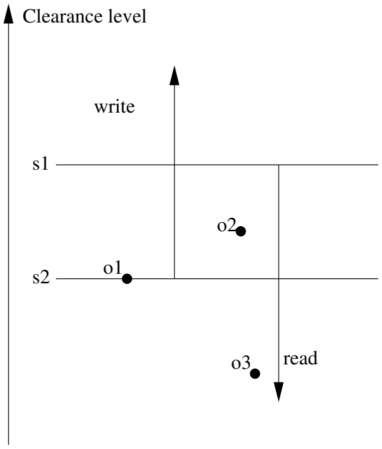


Figure 1.7: *Illustration of MAC policy*

The clearance level assigned to each subject and objects limits what operations can be applied.

It is important to note that, in opposition to what DAC systems do, a MAC system does not allow a user to modify the permissions attached to the objects he manipulates. In a MAC system, objects are created with a specific clearance level which cannot be changed; otherwise the security of the whole system may be compromised.

For a long time, MAC systems have only be used in domains where security is critical, like military subsystems or nuclear power plants. More recently, new implementations have been incorporated in mainstream operating systems like Linux (with the incorporation of SELinux in kernel 2.6.x) and Windows (with the Mandatory Integrity Control incorporated into Windows Vista and newer).

While being able to guarantee a high protection of the security of the data through the system enforcement of the authorization, the MAC model suffers from some drawbacks. The major one is his stiffness: MAC systems do not allow clearance levels of objects to be modified, making it very difficult to support evolution of contexts. In the same way, adding a new user or a new object to the system impose to determine very precisely what its clearance should be, which may not be easy on large systems or dynamic organisations.

1.3.2.3 RBAC

Role-Based Access Control (RBAC), based on the work of David Ferraiolo and Richard Kuhn [FK92], is an access-control model that breaks the direct mapping between a subject and the object contained in the permission.

To overcome the problems occurring when trying to manage large DAC permission sets and the stiffness of the MAC model, the RBAC model introduces the concept of role inside the permission relation (see figure 1.8, to be compared with figure 1.2 illustrating the permission). A user gains access to an object, not by some permission he's been personally granted, but only through the roles he has at the time the access is made. This allow to easily update the set of users and/or the set of objects without the need to verify the full set of possible permissions. Only the (smaller) set of roles has to be checked and eventually adapted.

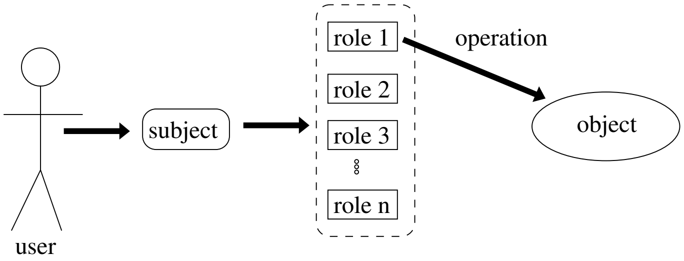


Figure 1.8: *Illustration of RBAC policy*

The direct link between a subject, an operation and an object (the permission) is now broken with the introduction of the notion of role.

A formal definition of the RBAC model was given by Ferraiolo and Kuhn in their original paper [FK92] and its followers. That definition wasn't provided to allow formal reasoning but to clarify some of the ambiguities found in different implementations of their model (see original citation in figure 1.9).

This formal model does not clarify all the ambiguities found on different implementations: RBAC is born from a conjunction of industrial implementations and had to be unified before being called a standard. But

> *"(...) This article is a first attempt to develop an authoritative definition of well-accepted RBAC features for use in authorization management systems. Although RBAC continues to be an evolving technology, the RBAC features that were chosen to be included within this proposed standard represent a stable and well-accepted set of features, and are known to be included within a breadth of commercial products and reference implementations."*

Figure 1.9: *Motivation for the formal definition of RBAC [FK92]*

the formal model had to include all of the commercial implementations available at that time, otherwise it wouldn't have been able to federate all of the RBAC users. Furthermore, RBAC still evolves: for instance with the addition of new concepts like inheritance or separation of roles. Or with the removal of some criticized features like the notion of session (see [LBB05] for a complete story on the removal of this feature).

Presented in one of the last version of the RBAC specification [FKC03], the formal specification of the RBAC model is based on four main concepts : users, roles, operations and objects (figure 1.10):

- $USERS$, $ROLES$, OPS and OBS are, respectively, the considered set of users, roles, operations and objects.

- $UA \subseteq USERS \times ROLES$, is a many to many mapping between the users and the roles. It represents the assignment of roles to the different users.

- $assigned\_users : (r : ROLES) \rightarrow 2^{USERS}$; $assigned\_users(r) = \{u \in USERS | (u, r) \in UA\}$: is the mapping of a role to the set of users acting it.

- $PRMS \subseteq OPS \times OBS$, is the set of permissions.

- $PA \subseteq PRMS \times ROLES$, represents the set of permissions granted to each role.

- $assigned\_permissions(r : ROLES) \rightarrow 2^{PRMS}; assigned\_permissions(r) = \{p \in PRMS | (p, r) \in PA\}$, is the mapping of a role r to the set of permission it grants.

- $SUBJECT$, is the set of subjects.

- $subject\_user(s : SUBJECT) \rightarrow USERS$, is the mapping between a subject and the corresponding user.

- $subject\_roles(s : SUBJECT) \rightarrow 2^{ROLES}$, is the mapping between a subject and the set of his roles (more formally : $subject\_roles(s) \subseteq \{r \in ROLES | (subject\_user(s), r) \in UA\}$).

From there, the authors define the concept of authorization as :

$$\begin{aligned}
& s : SUBJECT, op : OPS, o : OBS \\
& access(s, op, o) \Longrightarrow \exists r : ROLES, p : PERMS : && r \in subject\_roles(s) \\
& && \wedge \quad p \in assigned\_permissions(r) \\
& && \wedge \quad (op, o) \in p
\end{aligned}$$

Specifying that a subject s is granted access on an object o to perform an operation op if there exists a role r, currently owned by s, which opens the right to this permission.

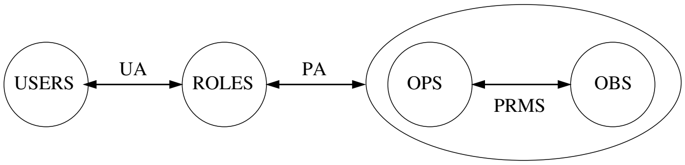


Figure 1.10: *Formal model of the core RBAC features (taken from [FKC03])*

The way a user can be granted a new role and the conditions under which this can happen are unspecified in this model: the reason is that the behavior of implemented systems differ on this particular point. To prevent one or more systems from being excluded from the standard, this element was removed from the specification. The same holds for the notions of inheritance, management of the decision points, ...

More informations on RBAC and all its evolutions can be found on the dedicated NIST website [NIS06]. A detailed description of all the proposed evolutions of the RBAC standard is available in [SFK00].

1.3.2.4 ODRL

DAC, MAC, and RBAC models, while being used extensively in the industry for decades for the specification and implementation of access-control systems, often show their weaknesses when they are confronted with a distributed and dynamic environment. The implementations behind the different models often use vendor-specific adaptations and optimization, causing interoperability problems. Furthermore, the fast evolution of some access-control usages, like the finer usage-control introduced with Digital Rights Management (DRM), showed the limitations of these classical models. With DRM's, each user can be granted specific permissions, possibly depending on the context where the usage of the resource takes place. For instance, it is not uncommon to see permissions like " *Alice is allowed to listen to audio file bob.mp3, on her own mp3-player, up to 5 times during the first 7 days after the date the file was purchased* " .

While classical access-control models attempt to solve the “Who can do what” question, DRM and usage-control approaches require a more fine-grained answer taking into account the context and the potential consequences of the operation. As such, usage-control attempts to solve the “Who can do what in wich context and with which consequences” question.

The Open Digital Rights Language Initiative (ODRL) [ini11] attempts to provide a solution to this issue. Defining themselves as *“an international effort aimed at developing and promoting an open standard for policy expressions. ODRL provides flexible and interoperable mechanisms to support transparent and innovative use of digital content in publishing, distribution and consumption of digital media across all sectors and communities.”* , the ODRL initiative introduced a permission model explicitly taking into account the different conditions in which the access is granted or forbidden.

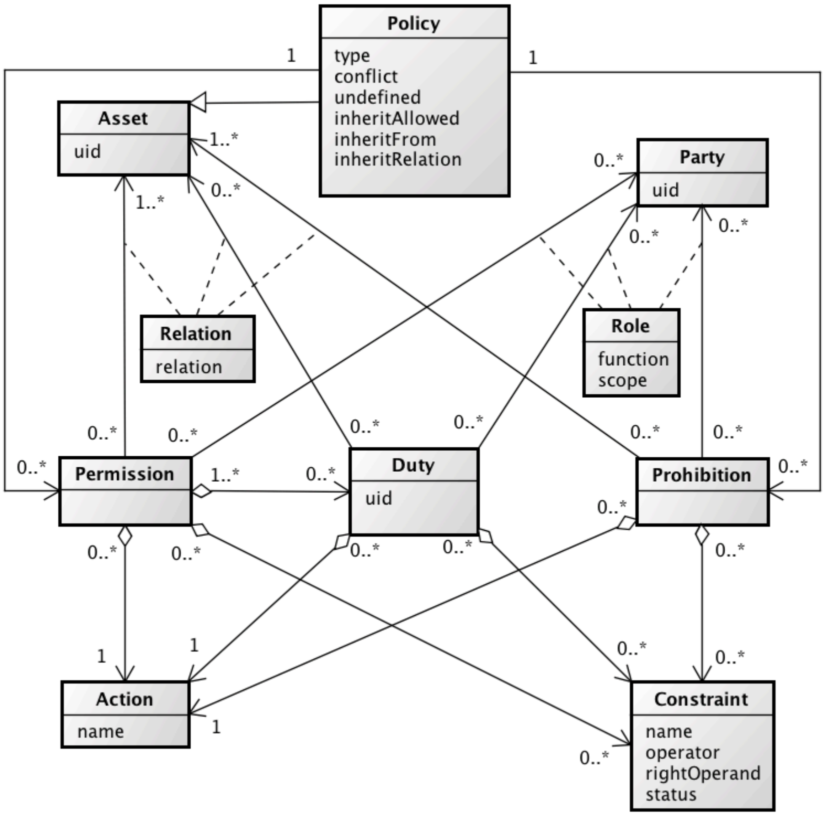


Figure 1.11: *ODRL Core Model Version 2.0 [ini11]*

Figure 1.11 shows the graphical specification of the core of the ODRL v2.0 model[6]. In the ODRL model,

[6] Version 2.0 of the ODRL standard is still a draft specification at the time of writing. The current status of the draft is available on

a policy is composed of a set of permissions and prohibitions (negative permissions), each of them representing the ability to perform an action on an asset in a specific context. Comparing with the classical DAC permission model from figure 1.2 or with the RBAC permission model from figure 1.10, the main addition are the notions of constraints, duty and party: the ODRL model explicitly expresses the set of constraints that must be satisfied for the permission (or prohibition) to take place. A permission can also be conditioned by the realisation of some duty (for example *"pay 1 euro for each viewing of the movie"*). The model also specifies, via the "party" element, who imposes the expressed constraints and duty and who grants the permission and prohibition.

1.3.3 Access Control Policy Verification

Once the access-control policy has been defined, whatever the formalism used, arise the questions of whether or not the declared policy :

1. precisely reflects the intended security policy. Said otherwise, do the access-control expressions written in the produce model properly reflect the author's intent ?

2. is complete, i.e. is the access-control policy precise enough to decide in a non ambiguous way, in any situation, whether or not an access should be granted.

3. and is coherent with itself, i.e. in no circumstances can the policy provide two opposite decisions in the same context.

While the first question can only be through the experience and knowledge of the policy creator, the two latter questions can benefit from helper processes. The question of whether policies are complete and coherent has been quite extensively studied both formally (see for instance [GRS04] for the formal model-checking of access-control policies) and in more practical ways (see [TKS05] and [YTB07])

The answers to these questions, and the confidence level associated with these answers, defines the level of security to be expected from the application implementing the considered policy. In the same way as it is almost impossible to build strong and steady buildings on swamps, only a strong and well-defined security policy can lead to the production of secure software.

1.3.4 Access Control Policy Enforcement

The specification of the access-control policy does not, in itself, make the application secure. Only a strict conformance of the implementation to the specified rules can guarantee that the resulting application really

the ODRL website : [ini11].

fulfills its requirements.

Some standards, like XACML [OAS05], explicitly specify which components of a possibly distributed system should decide wether an access request is granted or not. Unfortunately, the level of conformance of a software system to the original policy may depend on many factors :

- the capacity and understanding of the application developers to implement precisely what was specified.
- the chosen implementation language which may not support all the required security primitives, or may mis-implement some of them.
- physical constraints induced by the hardware platform designed to run the software system.
- time and cost mobilized for the verification of the conformance of the system.
- and many more...

The effect of some of these factors can be controlled to be maintained as low as possible, for instance using certified developers, verified implementation languages, extensive conformance testing. But it is usually very difficult to prove that, on the resulting software system, the implemented access-control model is exactly the same as the specified one, no more no less. Testing the system in sample cases can help, but only provides limited confidence on the conformance to the model, not certitudes.

1.4 Application Security

Building a secure application with its companion security model requires the precise identification of the elements it should be protected from. That's where the application security analysis comes into the play. The notion of "application security" regroups the capacity for an application to deal with abnormal situations. It is often expressed in terms of assets, threats, vulnerabilities, attacks and countermeasures.

Asset: A resource of value such as the data in a database or on the file system, or a system resource.

Threat: A negative effect.

Vulnerability: A weakness that makes a threat possible.

Attack: An action taken to harm an asset.

Countermeasure: A safeguard that addresses a threat and mitigates risk.

As for the access-control model specification, defining a secure application is a complex task. Aside from some obvious security vulnerabilities that will affect any user of the software system (for instance, an unprotected database access), most of the security vulnerabilities depend on the context in which the system is operated. The same vulnerability may be critical for one user while being unimportant to another. These two users will build very different definitions of what a secure application should be in their specific case.

Even worse: specifying the "perfect" (if it exists) security policy during the requirements analysis stage does not guarantee that the resulting application will be secure at all: developers may mis-implement some parts of it or the environment of the application can evolve in ways that breaks some of the hypothesis on which the security policy is built.

Furthermore, the software development process itself is far from a simple and unequivocal task: developers can introduce bugs, unwanted side-effects, or more simply misunderstanding of the security policy or of the effect of the access-control primitives. Benign code modifications apparently unrelated to access-control may create side-effects introducing flaws in the implemented security policy that may not be trivial to detect.

1.4.1 Norms and best practices

As it is already the case in many engineering fields, common knowledge, previous experiences and best practices have been compiled into knowledge bases in order to assist as much as possible the developer in the delicate task of producing secure software. Be they internal to the development team or publicly available documents produced by national or internationals bodies, all of these documents attempt to standardize the field they apply to. This allows to pinpoint the critical aspects of the considered development process and to choose the most suitable methods to reach them.

In the software engineering field, the Common Criteria [CCR11] are a compilation of definitions and methodologies aiming at the production of secure software. They define both the development framework and the reference documents to be produced. The Evaluation Assurance Level (EAL1 through EAL7) of an IT product or system is a numerical grade assigned following the completion of a Common Criteria security evaluation. The increasing assurance levels reflect added assurance requirements that must be met to achieve certification. The intent of the higher levels is to provide higher confidence that the system's principal security features are reliably implemented. The 7 EAL levels are commonly labelled as :

- EAL1: Functionally Tested
- EAL2: Structurally Tested
- EAL3: Methodically Tested and Checked
- EAL4: Methodically Designed, Tested, and Reviewed
- EAL5: Semiformally Designed and Tested
- EAL6: Semiformally Verified Design and Tested

- EAL7: Formally Verified Design and Tested

To achieve a particular EAL, the computer system must meet specific assurance requirements. Most of these requirements involve design documentation, design analysis, functional testing, or penetration testing. The higher EALs involve more detailed documentation, analysis, and testing than the lower ones.

- EAL1 to EAL3 basicaly consist in the production of adapted documentation together with the code. They usually do not require in-depth development usage changes.

- EAL4 lies inbetween the lower levels, mainly dedicated to the documentation of the project and the upper level, where more formal verifications will be required. This level requires to employ more rigorous techniques than the previous ones, but does not yet enforce the usage of security-dedicated verification tools. As an illustration of this category, most commercial user-level operating systems lie around EAL4 level.

- Higher EAL levels, labelled EAL5 to EAL7, stregthen the requirements of formal verification of some / all aspects of the code, progressively replacing the trust in the good design by proofs of good design. For instance, the definition of the highest EAL7 level, states :

  > "*EAL7 provides assurance by a full security target and an analysis of the security functional requirements in that security target, using a functional and complete interface specification, guidance documentation, the design of the target of evaluation, and a structured presentation of the implementation to understand the security behavior. Assurance is additionally gained through a formal model of select target of evaluation security policies and a semiformal presentation of the functional specification and target of evaluation design. A modular, layered and simple target of evaluation security function design is also required.*" (adapted from [CCR11])

The highest security levels (in the Common Criteria), EAL-5 to EAL-7, require formal verification (or at least validation) of the specification documents and of some of the software constructs. An example of the application of the Common Criteria's recommandations to a development project can be consulted in [KS06].

It should be noted that the EAL level does not measure the security of the system itself, it simply states at what level the system was tested or verified. Furthermore, the confidence to be given to the EAL obtained by the considered software system should always be relative to the confidence in the authority providing the certification. EALs should not be viewed as pure theoretical evaluations, but as a mix of trust and quantifiable elements.

Other norms and best practices compilation frequently encountered in the software development field includes the work of the International Organization for Standardization[7], for instance with the ISO 27000-

[7] http://www.iso.org/

family norms applying to “the security techniques applicable in the information technology field in order to manage the security of information management system”.

The next chapter will focus on the software development process and on methods used to maintain the implementation as close as possible to the specified access-control policy.

2
Secure Software Development

Contents

This chapter presents the current set of techniques / methodologies aimed at avoiding or limiting the impact of vulnerabilities on the resulting software system. Identifying the possible vulnerabilities is the very first step towards a secure software system.

2.1 Vulnerabilities of secure software development

The first step toward secure software development is to identify what are the main vulnerabilities threatening software systems. This section covers the main source of vulnerabilities affecting the development phase of the software development cycle. These vulnerabilities usually occur in three contexts: incorrect or incomplete specification, forgotten features and unwanted extra-features.

An incorrect / incomplete specification is a specification where some elements either conflict (incorrect specification) or are missing (incomplete specification). A specification can be made incorrect when a newly added feature conflicts with a previous one. When detected, such conflicts can usually be dismissed by adapting the set of rules to remove the conflit. For example (figure 2.1), the specification of a medical record management system could state that *"no one should be able to read someone else's record"* (A) and, at the same time, that *"a practitioner should be able to read/write his patient's records"* (B). Depending on the understanding of the developers, implementing these two rules will lead to a system where at least one of them will be broken as soon as a practitioner will try to consult one of his patients' records. Allowing the access would break rule A while refusing it would break rule B. Once detected, the incompatibility between these two rules can be resolved by updating the requirements on this specific point according to the user's choice. In this example, we could give priority to the seconde rule (the B one), thus choosing the upper result on the figure and allowing practitioners to consult their patient's records.

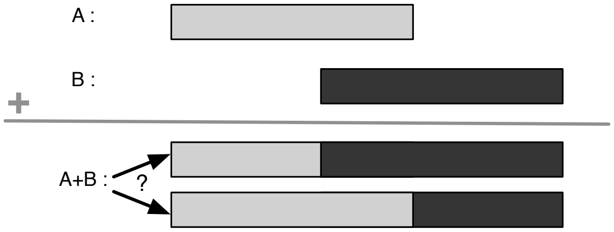


Figure 2.1: *Incorrect / incomplete specification*

The overlapping between the two conflicting rules must be resolved before starting implementation, otherwise the behavior of the system may be unpredictable on this point.

However, detecting conflicting rules is not always that simple: conflicts can occur when updating the rule set, or when performing apparently harmless operations, like reformatting the source code for pretty printing. Even worse, such conflicts may not exhibit harmful effects during the testing stage, leading to the delivery of a flawed system.

A forgotten feature is a feature that, although being specified, is not implemented in the application. There can be many causes to this absence: the feature may have been temporarily disabled during the development cycle because its implementation conflicts with some other on-development feature, or because the feature was skipped by the developers. The consequence is that the feature is not present in the final product and if its behavior is not properly verified or tested, the absence may not be detected before it shows effects. For instance, developers might be tempted to disable the firewall protecting the access to the servers during the development stage : at that point the systems are running in a protected environment where firewall may not be useful. Removing it will limit the possible interferences with the currently on-development software, but it will expose the system if it is not properly re-enabled before the testing and deployment stages.

Unwanted extra features are the exact opposite of forgotten features: they are features introduced during the development cycle that are not present in the specification but still appear in the final application. They may be introduced deliberately by the developers (for instance, a debug interface that was added to help diagnose some unwanted behavior and that has not been removed after use), result in a misunderstanding of the specification or be a consequence of the usage of a software library. Libraries often provide a set of features to an application, some of which may be superfluous. Being out of the specification, these extra-features are quite hard to detect: validation steps may not expose them because such verifications are based on the behaviors specified, not on what is really implemented. This may result in breaches in the security of the application as these extra features are neither checked nor monitored.

A special case of these vulnerabilities is the bug. A bug is an error in the software system, usually unwillingly inserted by a developer. Depending on his exact impact, a bug can be seen as any of the last two categories given above: a forgotten feature when the bug causes some part of the implementation to malfunction / not function at all and a extra feature when it causes unwanted behavior of the software system.

All of these vulnerabilities can cause the software system to exhibit behaviors that are out of the specification, possibly resulting in security flaws. To prevent such problems from happening, or to limit their impact when they are hardly avoidable, multiple methods have been proposed. The following sections will introduce the concept of secure software development as well as some of the main techniques used to detect and limit the impacts of existing vulnerabilities.

2.2 Software development cycle

The software development cycle refers to the set of stages required to obtain a software system. It ranges from the collection and analysis of the user's requirements to the delivery and maintenance of the software system. Refining a little further, the software development cycle can be divided in 5 interdependent stages :

requirement analysis : the collection and analysis of the user's requirements.

specification : the formalisation of the user's requirements and the selection and design of the according software system's feature.

program implementation : the traduction of the specification into the chosen implementation language.

system testing : the evaluation of the conformance of the software system against the specification.

maintenance : the deployment, the correction of vulnerabilities and the adaptation of the software system to his changing environment and/or to the new/modified user's requirement.

The granularity of this decomposition is arbitrary : depending on the focus, each of these stages can be refined into smaller task-oriented elements (think of the "system testing" stage, which could be decomposed into "unit testing", "integration testing", "system integration" and "full system testing"). For clarity purposes, the rest of this chapter will be limited to the 5-stages decomposition presented above.

Although these stages represent a logical progression from the user's requirements toward a running and maintained software system, their sequence is usually more complex: at each stage of the development process, developers might be tempted to revoke some of the options taken before, either because this would simplify their current work, or because the elements justifying these choices have been modified. If uncontrolled, this can lead to a cascade of changes, turning the development process in a never-ending story (figure 2.2).

The following sections will introduce the most prevalent models decomposing the software development cycle. The reader interested in a more in-depth analysis of the software development cycle and its implications should consult [Pfl01].

2.2.1 Waterfall model

In the waterfall model [Roy70], the set of stages required to obtain a software system are ordered into a strict sequence, leading to a stair-like model (figure 2.3). In this model, derived from the physical good manufacturing process, each stage is due to provide a deliverable on completion.

As the figure illustrates, each development stage must be completed before the next one begins. This model is very simple to follow and to explain to a user : the current activity is easily viewable, as well as the job already done and the upcoming activities. As such, the waterfall model was the basis for software development deliverables in the US. Dept. of Defense for many years [US 88].

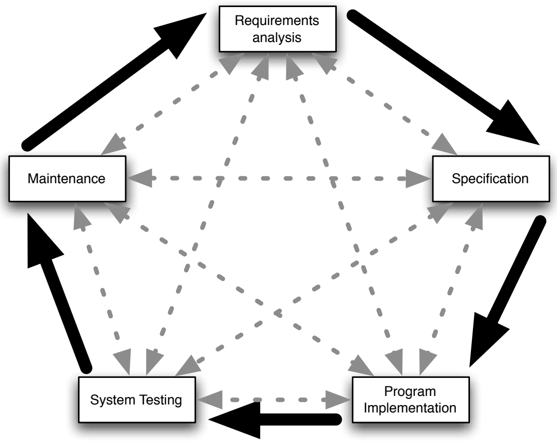


Figure 2.2: *Uncontrolled development*

An uncontrolled re-evaluation of the options taken can lead to an infinite loop in the development cycle.

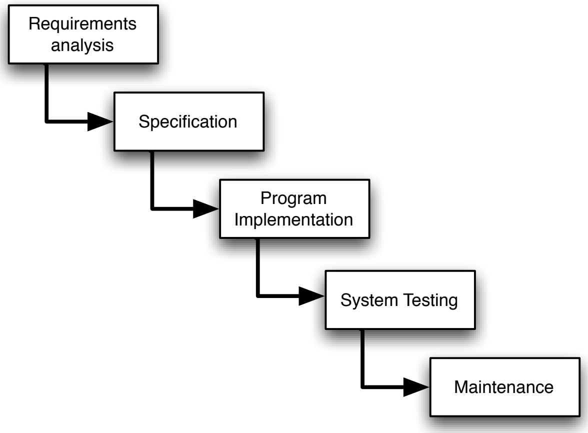


Figure 2.3: *Waterfall model*

However, such a strict ordering of the different stages has one major drawback : it leaves no way, in stage N, to modify, correct or update what has been done in stage $N - 1$ or above. This makes it almost impossible to reflect late changes in user's requirements or environment modification. It leaves no place for errors or evolution: in software development, the requirements usually evolve during the development phase, reflecting the knowledge of the problem gained and the already evaluated alternatives.

To allow a limited degree of modification to the deliverables of the previous stages while keeping the development process under control, upgrades to the waterfall models appeared, like de V-shaped model (German Ministry of Defense, 1992). In the V-shaped model (figure 2.4), testing activities are explicitly related to the analysis and design stages, allowing a limited level of correction of the previous stages while limiting as much as possible the risks of infinite iterations inside the model.

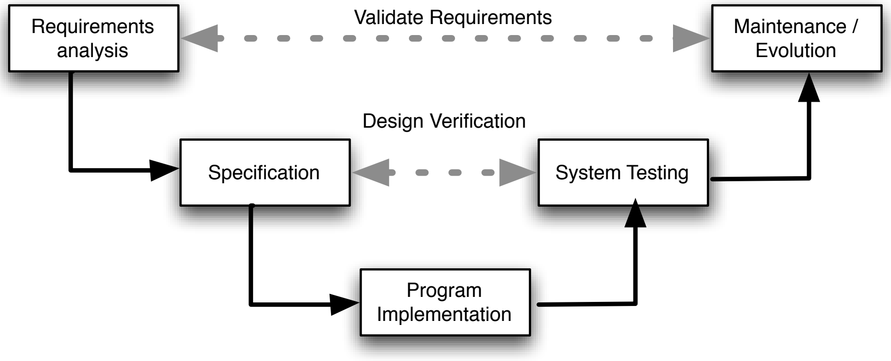


Figure 2.4: *V-shaped model*

Waterfall-like models are usually used in contexts where requirements are very stable, removing the need to update them during the development process, or in contexts where time and/or cost-constraints forbid such backward steps. For instance for systems required to be operational as soon as possible, even if the full features will only be reached after a strong maintenance phase.

2.2.2 Prototyping model

In order to limit the possibly infinite recursion of the uncontrolled development model (figure 2.2) without restricting too much the ability to revoke any options taken in the previous stages, the prototyping model allows all or part of a system to be constructed quickly to understand or clarify issued [Pfl01]. In a similar way to physical goods engineering, each produced prototype is evaluated and the feed-back is integrated back into the next generation of prototypes (figure 2.5).

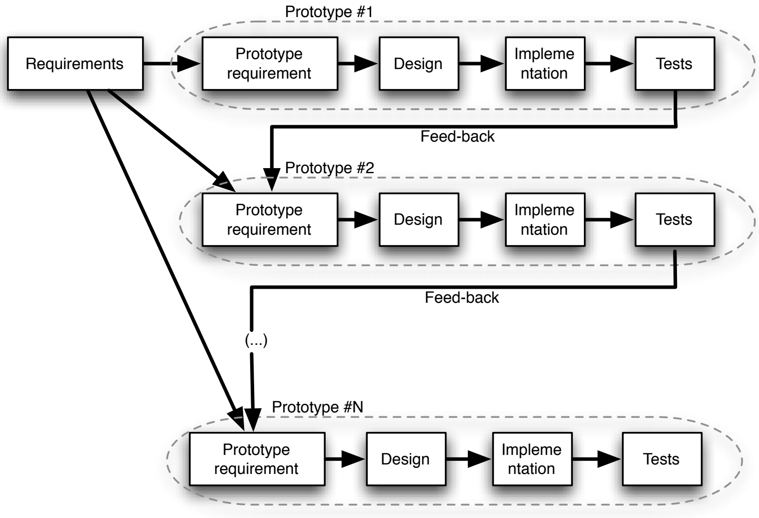


Figure 2.5: *Prototype model*

Each prototype implements a subset of the requirements, augmented by any useful feed-back gathered on the previous prototypes.

Prototypes can be built on a specific subset of the requirements, in order to test a possible implementation and/or to gather more efficient feed-back on them, or created incrementally: each prototype implementing a few more requirements than its predecessor. In both cases, prototyping allows developers to better understand the challenges involved and to obtain an easier feed-back from the users. Users are usually more comfortable to clarify their requirements when confronted to a partial version of the software system than when presented a formal specification.

The main drawback of the prototyping approach lies in the time and resources taken to develop the successive prototypes, which may sometimes exceed the time and resources required for a more "straight to the point" approach of the same project.

When working over a growing set of requirements instead of a specific one, the prototyping approach closely compares to iterative and incremental development (figure 2.6) where features are either added one at a time (incremental development) or progressively implemented in the software system (iterative development).

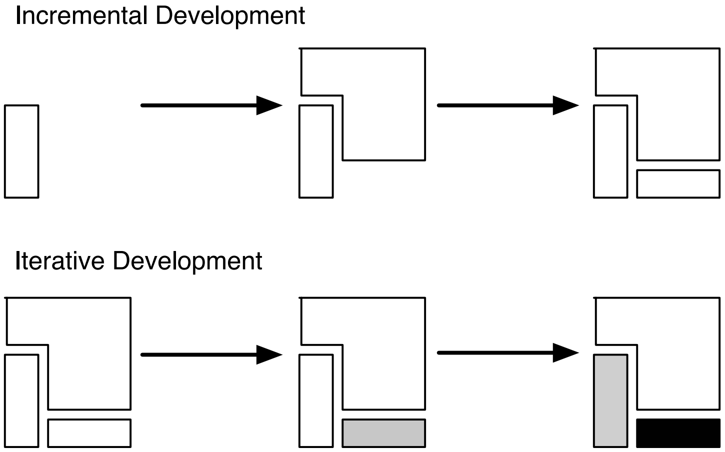


Figure 2.6: *Incremental Vs Iterative development.*

The methods based on prototyping proved very popular due to the rapid production of deliverables, allowing a quick feed-back from the user, thus limiting the impact of misunderstandings on his requirements. The same applies to conformance testing : each prototype can be tested on the restricted set of requirements it should implement, showing directly the elements to be corrected/improved instead of having to wait for the system to be fully developed and tested near the end of the development phase.

2.2.3 Practical software development cycle

The practical software development cycle usually differs from the classical approaches presented in the above sections : customers do change their mind between the requirement definition stage and the delivery of the software system, developers do make errors while implementing the requirements, environnement can change, time and cost constraints restrain the validations that can be made... All of these causes the sequence of stages required to build the system to change to an iterative-like process. Figure 2.7 presents these potential modifications in the case of an adapted waterfall model, but all the other software development models suffer from the same problems. A software system is not built in an isolated environment but must take into account changes occurring around it, so should the software development process.

This iterative process has great consequences on the requirements of the software system: each of the modifications adopted at every stage must be reflected to the original requirements document, which would otherwise become obsolete. However, experience shows that, when confronted to such requirements modifications, developers tend to dedicate their time and effort to the production of the code designed to implement them rather than to the updating of the original specification. Unless in critical contexts where security

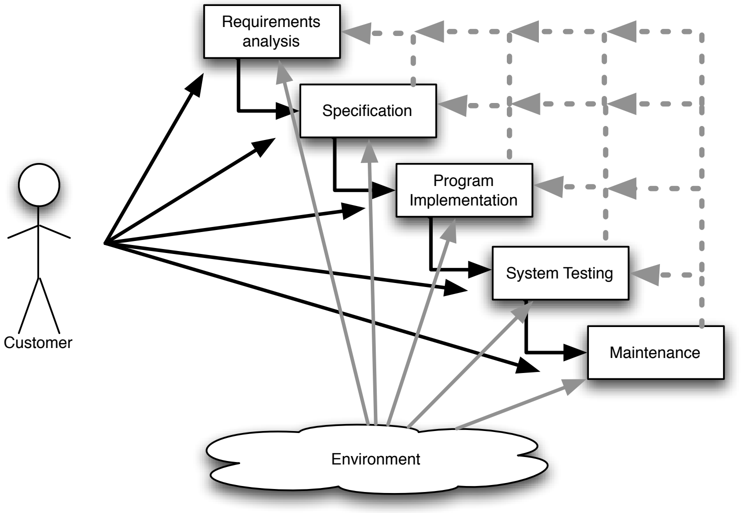


Figure 2.7: *Practical software development cycle.*

Each stage is influenced by the changing customer's requirements and the environment, possibly causing the need to revoke some of the decisions taken on the previous stages.

flaws cannot be tolerated, customers are often more willing to pay for systems covering their needs, even if they may contain potential security problems, that for secure systems completely useless in their business because some of the key features sought are not yet implemented.

2.3 Secure Software Engineering

In an attempt to meet these two apparently opposite focuses, secure software engineering techniques have been developed. These techniques aim at the production of secure code and its verification. Some of these techniques can guarantee that the resulting code is fully vulnerability-free while others only focus on some specific aspects of the program or return partial results.

There are basically two ways to produce code that is proved secure: either build code through methods guaranteeing its security (the generation approach), or build code and prove it secure (the verification approach). The rest of this section will introduce examples of these two approaches as well as their main advantages and drawbacks.

2.3.1 Secure code generation

Secure code generation aims at the production of secure code by deriving the source code more or less directly from a higher-level model, directly derived from the specification. The actions possible for the developer are strictly supervised and limited in order to guarantee that any code produced through this kind of method satisfy the properties specified in the model.

We'll present here two examples of secure code generation methodologies : the B method [Abr96], which helps to formally refine a specification down to executable code, and Aspect Oriented Programming (AOP) [KLM$^+$97] where transversal features can almost automatically be inserted into the executable code, removing unwanted developer interaction. Although both techniques aim at the production of secure code, they differ in the way they convert the expressed requirements into executable code and in the level of freedom of choices they leave to the developers .

2.3.1.1 B method

The B method, defined by J-R. Abrial in [Abr96], is a tool-supported formal method where the specification of a software system is progressively refined into executable code. The refinement process is strictly supervised so the properties expressed on a stage cannot be violated in subsequents ones. The objectives of the B-method, as stated on [Cle11], are :

- To create correct software by construction
- To model systems in their environment
- To formalize specifications
- To simplify programming

To prevent the occurrence of the vulnerabilities presented in section 1, the B-method imposes a strict development process (figure 2.8). First, the specification is expressed in a formal model called the abstract machine. This formal model specifies the high level goals the software system should meet. Then refinements of this abstract machine are produced. Each of the refinements clarifies one of the goals or provides a more concrete implementation of the required properties. It must then be proved to be coherent and to include all the properties of the Abstract Machine and of the previous refinements. At the end of the process, the refinement is fine and deterministic enough to be automatically translated into an executable language. The B-method as been used in some safety-critical systems like the Ariane 5 rocket and the underground line 14 in Paris [Cle11].

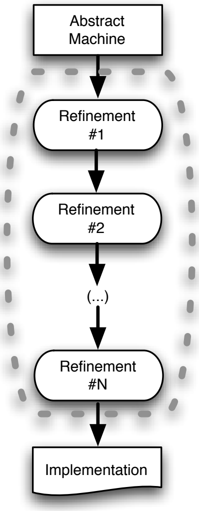


Figure 2.8: *B-method.*

The initial requirements, expressed as an abstract machine, are progressively refined into executable code while conserving the initial properties.

To illustrate the B-method, let's build a simple integer multiplication application. The abstract machine shown in figure 2.9 formalizes the simple requirements. It describes a simple model, containing only one operation : $mult(\cdots)$. This operation requires two parameters (a and b) and is only defined if both of them are naturals (the PRE-condition clause). The result of the $mult(a,b)$ operation is simply the product of a and b. A possible implementation of this abstract machine is presented in figure 2.10: the body of the $mult(a,b)$ operation has been refined into an algorithmic version of the multiplication operator. As the second machine refines the first one according to the B-framework, the methodology guarantees that any property required on the abstract machine still holds in the refined version.

```
MODEL
   Mult
OPERATIONS
  res <-- mult(a,b) =
        PRE a : NAT & b : NAT
        THEN res := a*b
        END
END
```

Figure 2.9: *B Abstract machine for the multiplication application*

```
IMPLEMENTATION
   MultImpl
OPERATIONS
  res <-- mult(a,b) =
        r:=a;
        q:=0;
        WHILE r>=b
          DO q:=q+1; r:=r-b
          INVARIANT r+(q*b) = a
          VARIANT r
        END
END
```

Figure 2.10: *B implementation of the multiplication application*

However, the B-method is far from being the universal panacea to prevent security issues from happening in software development. Using the B-method requires the developers to completely change the way they work: a B developer never writes code, he merely writes specifications (abstract machines) and refines them. This change of habits and the time dedicated to the experiencing of the methodology and its formalisms cause a huge time and cost overhead before any executable code can be written. The other huge drawback lies in the stiffness of the approach : once the abstract machine has been written, no change to the specified requirements is possible without losing all the already produced refinements. Late customers requirements and/or environnement changes are very hard to include. For these reasons, the B-method is usually reserved for projects with very stable requirements and where security is a critical factor and for companies able to cope with the high cost of entry to this methodology.

Further informations on the B-method can be found in [Abr96] and [Cle11]. A concrete application of the B methodology into the field of access-control is available in [Had05] and [DHM09] (in French).

2.3.1.2 Aspect Oriented Programming

Aspect-Oriented Programming (AOP) [KLM$^+$97] is a methodology designed to increase the modularity of the code. The main idea is to separate cross-cutting concerns from the rest of the code. The interest of AOP in the context of the secure code generation lies in the fact that AOP allows the developer to implement key security elements in one unique place and then automatically replicate/adapt that code in every code portions that requires it. It should be noted that AOP in itself does not guarantee to produce secure code at all, it only helps to reduce errors and makes the security measures implementations more uniform. It imposes little to no verification to what the implemented aspect do. A popular implementation of the AOP methodology is the AspectJ framework [Asp], developed for the Java programming language.

The places in the code where the cross-cutting concerns are to be inserted are called pointcuts, they can describe almost any constructs of the Java language : method calls, method executions, objects initialisation, constructor calls, exceptions being thrown, access to attributes,...

An illustration of the AOP methodology can be found in figures 2.11 and 2.12 (example adapted from [Mon09]). Figure 2.11 presents the source code of a Java class representing the $Account$ of a bank client. This class only contains business logic. Adding a logging feature to this class (and to all the other composing the software system) would require to edit each and every single class to add the corresponding method call. This error-prone process can be replaced by the specification of the logging component as an aspect (i.e. a transversal concern). Figure 2.12 shows a possible definition of such an aspect. Lines 4 and 5 define the join points where this aspect will apply; here on the creation of an $Account$ object and on any method call outside of the $Logging$ component. Lines 7 to 15 then define when and how the different aspect items implementation should be inserted into the code. For instance, applying the aspect specified on line 7 to the class source code would produce the result show in figure 2.13.

```
 1  public class Account {
 2          private int number;
 3          private int balance;
 4
 5          public Account(int number) { this.number = number;}
 6
 7          public void credit(int amount) { balance = balance + amount;}
 8
 9          public boolean debit(int amount) {
10                  if (amount > balance) return false;
11                  balance = balance - amount;
12                  return true;
13          }
14
15          public int getBalance() { return balance;}
16  }
```

Figure 2.11: *Source code for the Account class.*

AOP allows for easy and quick insertion of transversal features into the code, removing the risk of mis-implementation of the invocations of these features. It allows developers to focus on business logic inside the code and concentrate cross-cutting concerns like logging, encryption or access-control in small and manageable components instead of scattering them throughout the source code. From a validation point of view, AOP in itself does not guarantee any improvement in the security of the code produced. It relies on the correctness of the implementation of the aspects to be inserted. However, as the aspects are disjoints from the code, they allow for easy insertion/update/correction of their content without the need to check the whole source code.

In order to ease the transition from standard code to an AOP-style one, recent works attempt to convert the most frequent constructs into aspects. See for instance [TNTN11] for an application of to the access-control of Java applications.

```
1   public aspect Logging {
2           Logger logger = Logger.getLogger("simple_logger");
3
4           pointcut accountCreation() : execution (Account.new(..));
5           pointcut methodCall() : execution (* *.*(..)) && !whithin (Logger);
6
7           after() : accountCreation() { logger.info("New Account");}
8           before() : methodCall() {
9                   Signature sig = thisJoinPointStaticPart.getSignature();
10                  logger.trace("Entering "+sig.getName());
11          }
12          after() : methodCall()  {
13                  Signature sig = thisJoinPointStaticPart.getSignature();
14                  logger.trace("Leaving "+sig.getName());
15          }
16  }
```

Figure 2.12: *AOP : logging aspect specification*

```
   public Account(int number) { this.number = number;}
+
   after() : accountCreation() { logger.info("New Account");}
=
   public Account(int number) {
        logger.info("New Account");
        this.number = number;
   }
```

Figure 2.13: *Aspect insertion*

Effect of the application of rule 7 from figure 2.12 onto the Account object defined in figure 2.11.

2.3.1.3 Common drawbacks

While being very different, the two presented approaches share most of the pitfalls of the code generation methods. They both require developers to express the requirements of the system in a formalism quite different from the source code languages they are used to. Thus requiring developers to learn and experience new formalisms and tools; a process that takes time and consumes some of the resources of the company for “non immediately productive” work.

Another significative drawback is that the generated code is not meant to be easily human-readable and understandable. More than that, the generated code should never be manually edited : otherwise the properties granted by the code generation approach may be compromised. So the developer loses control over his code. Although he is still the one responsible for the possibly erroneous behaviors his code might exhibit, the developer has no direct way to verify that the generated code does not insert unwanted features inside his code. Only the confidence on the tool author and/or on the widely deployed methodology can help him here.

2.3.2 Secure code verification

Taking the opposite way of the secure code generation techniques, code verification methods are developed to check that source/executable code, produced by any means, does satisfy the expressed requirements.

The verification approach is built around two techniques : formal verification and code testing. The formal verification techniques build a model from the interpretation of the source code, often with the help of dedicated annotations, and checks the conformance of this model against some desired properties. On the other hand, the code testing techniques derive test cases from the requirements and check if the expected and observed behaviors of the system match while executing those test cases.

2.3.2.1 Verification

Code verification techniques aim at proving that a semantic model built from the considered source code satisfies the expressed requirements. They are dedicated to the identification of possible inconsistencies between the specification and the implemented code. They range from frameworks dedicated to specific inconsistencies detection (like the Valgrind tool [Sew11] dedicated to the discovery of memory errors) to more general methods adaptable to many situations (see for instance the Verifast tool suite [JSP$^+$11a], [JSP11b] or the Java PathFinder initiative [HP98], [NAS]).

These verification techniques can usually be grouped in 3 categories [DKW08] : abstract interpretation, model-checking and bounded model-checking. Static analysis encompasses a family of techniques for automatically computing information about the behavior of a program without executing it. On the other hand, model-checking techniques attempt to determine if a correctness property holds by exhaustively exploring the reachable states of a program and, if not, generate a counterexample or an execution trace leading to a state in which the property is violated. As the state-space of software programs is typically too large to be analyzed directly, model-checking algorithms are usually applied either on abstract versions of the software system (symbolic model checking) or on the full state-space but with a depth-bounded approach (bounded model-checking).

Abstract interpretation techniques quite rapidly show their limitation when the verification of large existing software systems is concerned : they are known to be computationally hard [Ric53] and are practically impossible to do in most modern languages. In the following, we will focus on model-checking techniques, both symbolic and bounded ones.

All of these model-checking methods share the same process (figure 2.14): the code is abstracted into a formal specification model which is then confronted to the desired properties. Abstracting the source code allows to get rid of irrelevant details about the semantics and to somehow simplify the verification step. Depending on the properties to be proven and on the chosen source code language, the abstraction process can be fully automated or require a more or less intensive developer intervention (for instance, via the annotation of some of the code constructs). The formal specification obtained from the code is then checked against the desired properties producing results that need to be reinterpreted in terms of the original source code.

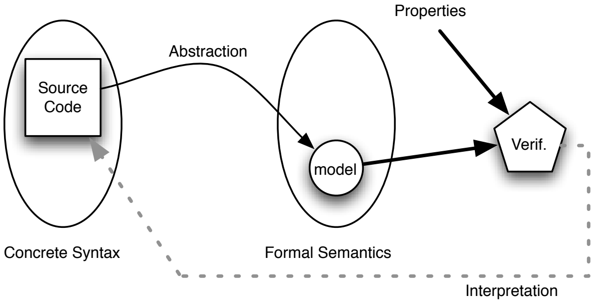


Figure 2.14: *Verification process.*

Abstracting the model from the code is far from an easy task: the quality of the extraction process affects the results of the verification. The extracted model must be at the same time precise and abstract enough. Precise so that any proof obtained on the model level still holds on the source code and abstract so that irrelevant elements don't pollute the formal reasoning process. Furthermore, a loss of precision is sometimes necessary to make the semantics decidable. There is usually a trade-off to be made between the precision of the analysis and its decidability.

To illustrate the verification approach, a small example (adapted from [JSP11b]) demonstrating the usage of the Verifast tool is presented in figure 2.15. It consist in a very naive Java implementation of a stack. Verifast requires the source code to be annotated with preconditions and postconditions written in a specific formalism. These properties to be verified as well as some interpretation-helper annotations have to be manually specified by the developer. In this example, the size of the annotations almost overweights the size of the original source code.

The main drawbacks of such annotation-driven techniques lie in the weight of these annotations : requiring developers to annotate all the critical sections of the code with a tool-specific formalism causes the same set of problems as those explained in the section about secure code generation techniques. It takes time to learn and master such languages; and during that time, developer's efforts are diverted from his main objective : the production of the application's code.

Furthermore, the results of those techniques are only as precise as the abstraction of the code is : the extra load induced by the manually assisted abstraction usually limit their use to critical sections of the software. This prevents the tools to verify the whole source code, possibly ignoring requirements violations in apparently harmless code portions.

More details on verification techniques as well as numerous examples can be found in [Cou08], [CVC99] and [Liv06].

2.3.2.2 Testing

Testing, on the other hand, derives a set of test-cases from the specification and checks whether the behavior of the system on these specific cases matches the expected behavior or not (figure 2.16). It is a very intuitive process, very close to the one used in physical goods manufacturing or empiric sciences.

The results of the tests heavily depend on the quality of the testing set: the considered test-cases must cover as much as possible of the possible situations the system might encounter. As the set of potential test-cases is often infinite, several approaches have been introduced to select the most representative candidates to test. The reader interested in detailed explanations of the existing methods (both heuristic and formal) of test-case selection can consult : [Pet], [JUn10] or [LTMPB08].

Testing has the defaults of its simplicity : proving that the system always satisfies its specification requires exhaustive testing. This process is very time consuming and impossible if the potential test-case set is in-

```
class Stack {
    Node head;
    //@ predicate valid(int count) = head |-> ?h &*& nodes(h, count);

    Stack()
        //@ requires true;
        //@ ensures valid(0);
    {
        //@ close valid(0);
    }

    void push(int element)
        //@ requires valid(?count);
        //@ ensures valid(count + 1);
    {
        //@ open valid(count);
        Node n = new Node();
        n.value = element;
        n.next = head;
        head = n;
        //@ close nodes(head, count + 1);
        //@ close valid(count + 1);
    }

    int pop()
        //@ requires valid(?count) &*& 0 < count;
        //@ ensures valid(count - 1);
    {
        //@ open valid(count);
        //@ open nodes(_, _);
        int result = head.value;
        head = head.next;
        //@ close valid(count - 1);
        return result;
    }
}
```

Figure 2.15: *Java Stack class, with Verifast annotations.*

Exemple adapted from [JSP11b].

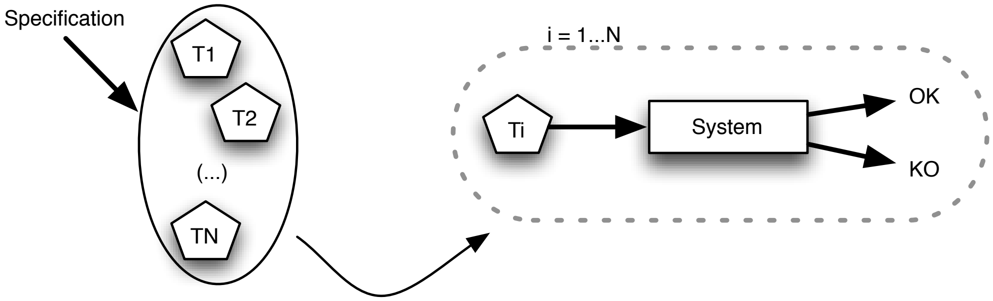


Figure 2.16: *The testing process.*

finite, as it is often the case. On the other hand, testing is very useful to quickly detect property violation: the test-case shows the violated property and the sequence of actions leading to that violation. This allows the developer to easily pinpoint the faulty elements and correct them.

As such, testing is used in almost every software project, at least on a limited scale, to detect the most obvious problems and to demonstrate the partial correction of the system to clients.

However, apart from a few very specific cases, testing a system in some selected cases is not sufficient to prove it correct. Tests can quickly demonstrate that a system does not conform to its specification by the production of a counter-example, but only formal verification or exhaustive testing can prove conformance.

2.4 Practical Software Engineering

As it is often the case real-world software engineering is a compromise between theory and practice. Even though all the actors of the software development process (clients, analysts or developers) agree on the importance of the respect of the specification, most of the software systems are deployed without being thoroughly validated conformant to their specification.

The main reason behind this fact lies in the conflicting market constraints imposed on the software development process : it should be as quick and as cheap as possible while producing secure and feature-rich software. A compromise has then to be made, sacrificing some aspects in favor of others; for instance giving up full system verification and/or testing to keep up with a previously announced release date. Even if this means deploying an incomplete systems until the corrective patches are available. The choice between the two approaches then becomes an political and economical one, where the costs and benefits of the different options have to be taken into account, be they quantifiable ones, like the price of potential defects, or more fuzzy ones like customer satisfaction. An extended description of these potential factors can be found in [Hol01]. A more complete description of the caveats of practical software engineering is available in [Pfl01] and in [Mea].

2.4.1 UMLsec

UMLsec [Jü02] [Jur05] is an extension of the widely used unified modeling language (UML) [OMG09] that allows one to define security properties and to formally check a model against those properties. The set of UMLsec properties is not limited : anyone is welcome to define new properties to address new security concerns. UMLsec uses the standard UML extension mechanism, stereotypes and tagged values, to describe security properties (figure 2.17).

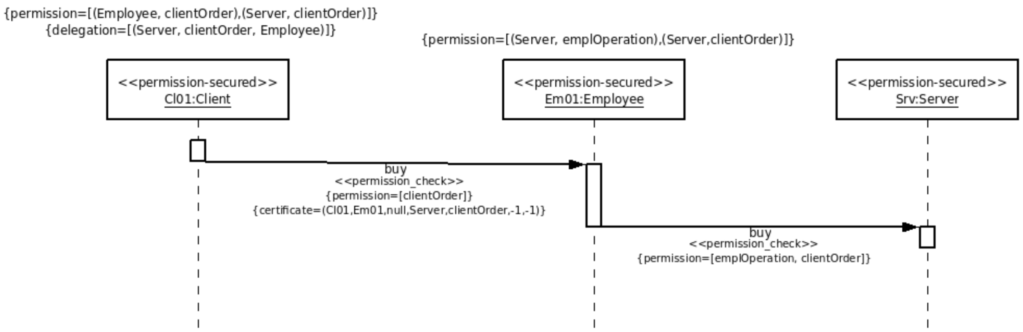

{permission=[(Employee, clientOrder),(Server, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
{permission=[(Server, emplOperation),(Server,clientOrder)]}
<<permission-secured>>
Cl01:Client
<<permission-secured>>
Em01:Employee
<<permission-secured>>
Srv:Server
buy
<<permission_check>>
{permission=[clientOrder]}
{certificate=(Cl01,Em01,null,Server,clientOrder,-1,-1)}
buy
<<permission_check>>
{permission=[emplOperation, clientOrder]}

Figure 2.17: *UMLsec : Sequence diagram with the "permission" property. (reproduced from [Mon09])*

The main idea behind UMLsec is to extend a model widely used by developers (the UML model) with the constructs required to express precisely the security properties expected from the modeled elements. These properties can then be used to automatically verify that the produced source code effectively exhibit the expected behavior. Recent works [MJH$^+$10] [Mon09] took the process one step further, producing the security-related code directly from the model instead of only verifying it, bypassing the risk of developer mis-implementation.

UMLsec, and the tools built on it, are an attempt to bring the best of the code production and code verification worlds together. However, to express their full potential, they require the methodology to be adopted from the early sketches of the software development process. The UMLsec methodology is hardly applicable to existing software development projects conducted without any prior UML (or UML-like) specification.

Furthermore, the UMLsec methodology is often viewed as too rigid and too flexible at the same time. The existing UMLsec extensions often do not exactly match the needs of the users. And, even though the open extension mechanism built into UMLsec allows one to define almost any possible extension, these extensions will not be understood and properly processed by existing tools. The developers are then left with a choice between constructs only partially representing his needs and custom constructs forbidding him to benefit from automated tools due to lack of support. Large teams may have the resources required to adapt the existing tools to better fit their needs, but smaller teams certainly have not.

3

Motivations and Objectives

Contents

This chapter introduces the motivating problem from which our research takes source. We present the scope of this work and the set of problems frequently encountered by software developers we try to alleviate. The running example that will be used in the following chapters to illustrate the models and the algorithms is also detailled.

3.1 Motivating example

The root of our reflexion comes from the observation of the implementation of access-control in a commercial software system. Developed and maintained by a small team, the application consisted in an online shared calendar augmented of some cooperative functionalities. It was suffering from an evolution-related problem: its code kept being updated with new/adapted features while its specification stayed mostly static and did not reflect all the changes operated.

The shared calendar application consisted of a set of programs designed to allow people to create, edit and share their time schedules. Each user originally has full control over his own personal events and limited access to any event shared with him, depending on the access-control properties specified by the object owner. As such, the original permission model behind it was very close to DAC (see section 1.3.2.1, page 9). In addition to this discrete permission model, some general rules where hard-coded directly into the application code to allow easy administration of the system as well as preventing data losses. For examples, rules like *"administrators always have access to any event"* and *"a user cannot give up his rights on an object if he is the last one able to access it"* were hard-coded to allow easy administration and to prevent events from being definitively unreachable.

Development was carried with a prototype-like approach : each revision of the system implemented a growing set of features in order to cover as many as possible of the requirements expressed by the different users. Prototyping allowed developers to obtain a quick feed-back on the functionalities of the system, allowing for a rapid evolution of the feature set.

From an access-control perspective, things became to get out of control when the application continued to evolve. New feature insertion, emerging customer queries or team modifications caused unwanted side-effects :

- more and more rules where hard-coded, some of them overlapping / shadowing previous ones, thus modifying the behavior of the system in a way not always easy to find (figure 3.1).
- user-managed groups were added to the system, quickly introducing the question of conflicts in existing permissions as well as the need for more global, hardcoded, rules dedicated to the management of these groups (for instance rules designed to prevent the deletion of the last user in a group still containing events).
- new developers joined the project, others changed affectation, taking away their full understanding of the rules they wrote.

As a result, more and more unexpected and incoherent behaviors were reported by the users and each "bug" correction made the access-control code a little bit more complex. Soon the developers began to realize that none of them still mastered the whole access-control aspect of their system: event though they understood the effect of each rule separately, the cumulative effect of the whole rule set was more than obscure to them. Any new modification was introducing unwanted side-effects, which in turn required new modifications, initiating an almost infinite loop of patches close to the never-ending development problem presented in

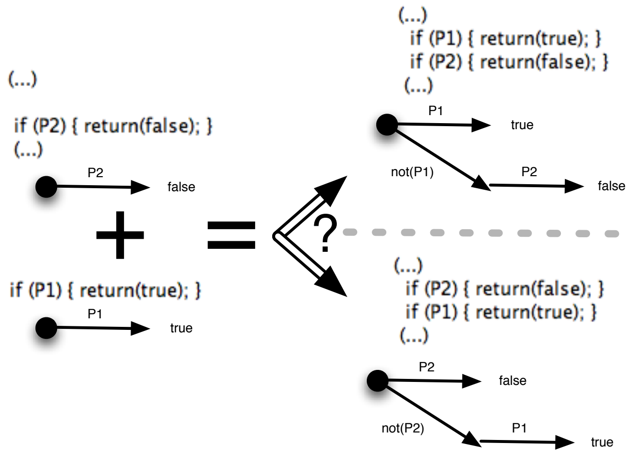

(...)
if (P2) { return(false); }
(...)
P2
false
+
=
if (P1) { return(true); }
P1
true
?
(...)
if (P1) { return(true); }
if (P2) { return(false); }
(...)
P1
true
not(P1)
P2
false
(...)
if (P2) { return(false); }
if (P1) { return(true); }
(...)
P2
false
not(P2)
P1
true

Figure 3.1: *Rule insertion : uncertain side-effects*

Inserting a new rule (here on P1) may cause side-effects on previously added rules that are not trivial to detect

chapter 2 (page 27).

To solve this problem, a complete rewrite of the access-control code seemed the proper solution, but it quickly turned out that without anyone able to understand what was exactly doing the current code, writing a new one would inevitably impact current users in the way they were used to the system. More than that, this would only be a temporary measure, as the new code would also be subject to the same evolution-related problem as the old one, resulting in the emergence of the same problems in the near future.

After a quick search, two possible solutions were identified: either should the rewritten access-control code be frozen in its initial state, preventing any modification thus preserving its integrity but preventing some possible clients to use their software because some of their requirement might not be met. Or should they find a way to know exactly what is the impact of every modification in their code, in order to detect potential conflicts between rules before they adversely affect the behavior of the system.

However the company didn't have the resources required for such a formal modeling of their requirements and code : time and costs involved in this formal stage would cause the whole project to get out of schedule and budget. It would also require the developers team to be trained to formal modeling and verification, moving them away from their primary objective - producing and maintaining code - to an unproductive state: learning formal methods. This would cause a unacceptable hiatus in system releases possibly leading customers to try other vendor's solutions to match up with their new specific requirements. Furthermore, such training is an investment for better and more secure software in the future, but it is an investment almost impossible to pay for small structures.

3.2 Analysis

The source of their problems lied in the gap between what a software development process should be and what it usually is: the development of a software system usually takes places in several consecutive steps (figure 3.2) : the developer and the users settle on a set of requirements for the upcoming application, the developers translate this set of requirements into a more or less formal specification aimed at resolving any existing ambiguities, then implements it. If the developer doesn't make any implementation errors, any properties verified by the specification still holds in the final application, providing the desired security properties to the users' system.

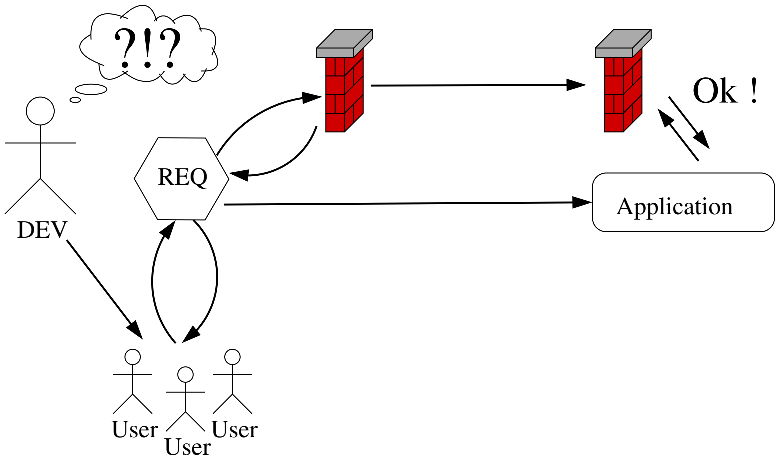


Figure 3.2: *Straight development.*

Close to the waterfall model presented page 27, the ideal software development process is a forward-going process where choices never need to be reconsidered. In absence of programming bugs, all properties specified during the requirements stage still hold in the final application.

However, this canonical scheme is far from reflecting the reality of most software development projects. During the development of the software system, the client can change his mind on some features. For example modifying the specified behavior of some element or adding/removing some feature. External constraints may cause developers not to reflect those changes on the specification, or reflect them but not check again that this new specification still meets the original requirements. The modifications are then implemented into the code by the developers and delivered to the customer as a prototype to be tested (figure 3.3). The question then arises : *“does the delivered application still meet the original specification?”* . Most of the time, the answer is false, or at best unknown, as the original specification does not contain all the updates introduced along the development cycle. So the delivered application cannot be guaranteed to exhibit the behaviors specified on the early stages of development. This is problematic as these behaviors specified in the early stage of development are the basic features sought by the client on his application, and the ones he’s willing to pay for.

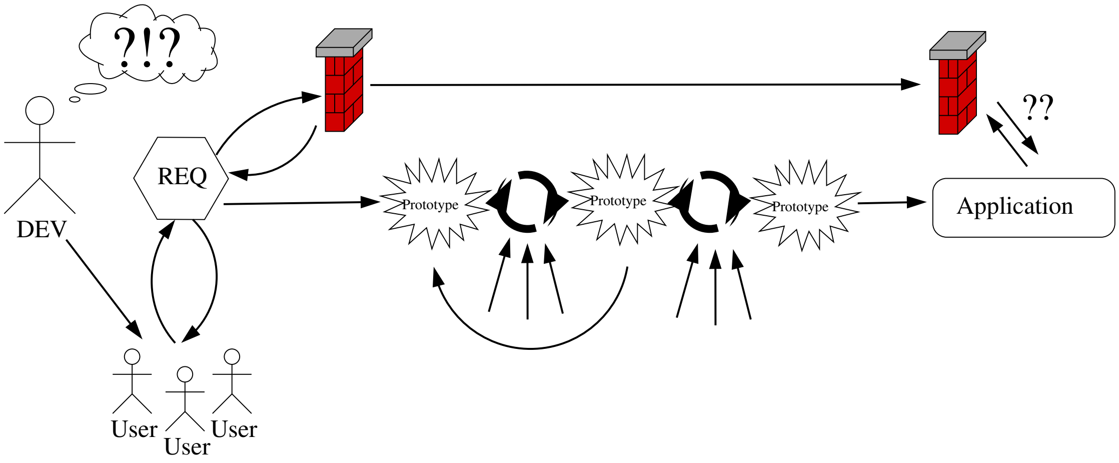


Figure 3.3: *Prototype development.*

The relation between the final application and the original model is broken because of the code-changes introduced by new user requirements are not fully properly reflected on the specification.

We propose a methodology to help the developer to update the specification of the application at each step of the development cycle. Attempting to guarantee that any updates made to the code (or to the specification) is propagated to the specification (resp. the code) (figure 3.4). During this co-evolution process, we try to be as un-intrusive as possible : the developer should not be burdened with constant formal model specification or verification and cryptic messages, leaving him as focused as possible on his main task : producing code implementing the given requirements.

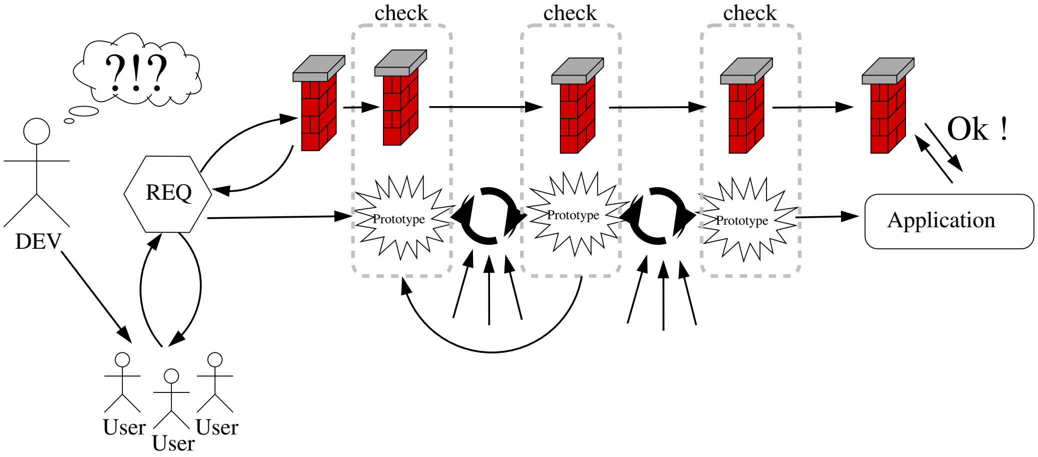


Figure 3.4: *Assisted prototype development.*

The relation between the final application and the original model is maintained through assisted propagation of the modifications.

3.2.1 Objectives

Our methodology should be simple enough to be used in limited development environments: many software company do not have enough resources (both financially and in terms of people) to maintain a team dedicated to the verification of their software systems, be it through testing or using more formal methods. In small teams, analysts, developers, testers and validators usually share the same seats; taking on one role or another depending on the planning of the day.

More precisely: we aim at small teams (usually a few people) composed of developers having a good knowledge of the chosen development language and little to no knowledge of formal verification techniques. This absence of knowledge of formal techniques is often caused by a lack of time / money dedicated to "unproductive" formation. Formal methods do not make developers code faster, they can only assist them to produce better code.

We try to help the developer to make the code as secure as possible without changing their habits beyond tolerance. Most developers are willing to be helped by tools in their development practice if (and only if) it does not force them to completely change the way they work. Small adaptations/improvements can be tolerated, but not complete revolutions (at least not in one step). Producing more secure code is a step further on the long path to completely secure applications.

The need for an helper methodology and tool is present as long as it does not have a deep impact on everyday work. Ideally, developers should be able to submit their source code directly into the helper tool, along with their requirements, and receive an human-readable report of inconsistencies eventually found, accompanied with proposed corrective measures (figure 3.5). Used formalism should be easily understandable by someone with a good programming background but little to no formal method knowledge. The same requirement of simplicity holds for the display and propagation of results (in terms of proposed modification in the code and/or in the requirements); they should be presented in a way allowing the developers to keep complete control over his own code. At the end of the day, the developer still is the one responsible for the correction of his code, so must he at all times be able to understand and validate any change proposed, for instance in order to explain them to his management.

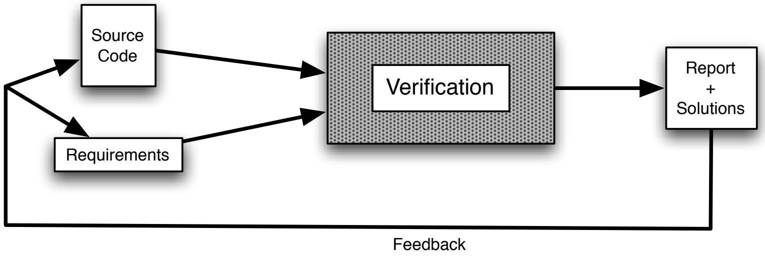


Figure 3.5: *Blackbox verification.*

Depending on the hypothesis that can be made on the source code, the precision of the results (both in terms of proved requirements and proposed modifications) may be affected. Our methodology will attempt to provide a best-effort approach, producing results as precise as possible without requiring too deep changes in developers habits.

3.3 Running example

Every time possible in the next chapters, a single example will be used to illustrate the methodology and the different algorithms used. It is built from a huge simplification of the case study presented above. We will consider a very simplified calendar system containing events, private events and group events. In the following, the notion of "owner" of an object will refer to the user (or group) to which it belongs. The original owner of an object is the user who initiated it's creation, while the current owner is the user to which the object was given by the previous owner.

- an event represents the usual notion of event in a calendar. It can be created by users and accessed by those allowed to do so by the event owner.
- a private event is an event only accessible by his owner.
- a group event is an event belonging to a group of users instead of a single user. It is managed by the group owner and it can be accessed by all the members of the group he belongs to.

Sample access-control properties desired to be applied on such a system by the clients could be :

- an administrator should always be able to access any event, regardless of its type. $(\star)$
- the owner of an event should be able to share it with some selected others.
- the owner of an event should be able to stop sharing it with others.
- private events should only be accessible by their owner. $(\star)$
- group events should be readable by all group members.

Some of the rules expressed here clearly conflict : for instance, the two rules marked with a star $(\star)$ are incompatible. Implementing one of the two would inevitably break the other. If detected during the development stage, this incompatibility must be sorted out by refining the requirements. If not, only thorough testing and/or verification can prevent the system to exhibit potentially erroneous behaviors if one of their conflicting scenario ever happen.

From a development point of view, implementing this system in an object-oriented language like Java could end up in a class hierarchy like the one illustrated in figure 3.6. The access-control is managed through dedicated methods named $isAuthorized(requestor, op)$. These methods return true if the subject identified as $requestor$ is allowed to perform operation op on the current object, false otherwise. Each object is required to implement this method and to call it before executing any operation on itself.

Oversimplified code snippets of the access-control part of the different objects are presented in figures 3.7 to 3.10 :

- the class $CObject$ (figure 3.7) represents the common elements of all the objects used in the calendar system. It contains functional methods dedicated to object manipulation like cloning, deletion, creation, ... Its access-control part is quite simple, consisting of only two rules : one explicitly allowing administrators to perform any action on the object, and the other forbidding specific users from accessing it. The $isForbidden(requestor, op)$ element appearing inside the code refers to an explicit

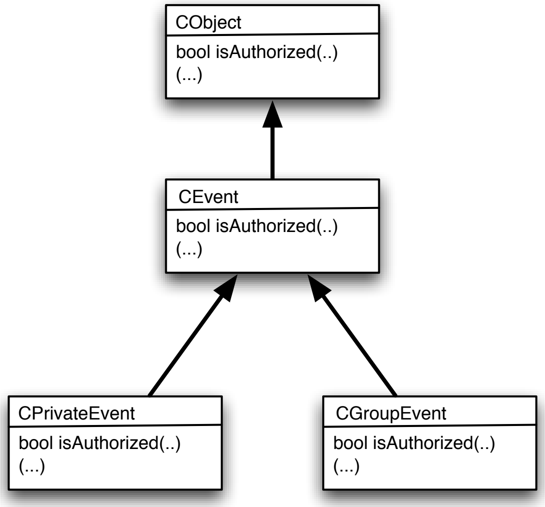


Figure 3.6: *Running example object Hierarchy.*

prohibition list given inside the source code. It's role is to deny the specified $requestor$ to execute the operation op on the current object [1].

- the class $CEvent$ (figure 3.8) represents a calendar event. It stores all the information associated with this event (for instance its name, date, note, ...) and provides methods to access and update this information. From an access-control point of view, an event is accessible under the same conditions as a $CObject$ plus in the case where the requestor is the owner of the object or if the requestor has been explicitly authorized to access it by the owner.

- the class $CPrivateEvent$ (figure 3.9) represents a private event, i.e. an event that can only be viewed and edited by his owner. It cannot be shared among users.

- the class $CGroupEvent$ (figure 3.10) represents an event owner by a group, not by a single user like in the $CEvent$ class. As such, the access-control section contains group-specific rules in addition to the rules inherited from the $CEvent$ class. These specific rules allow the group owner to access any object owned by the group as well as any group member to have read access to the group's objects.

[1] From a general point of view, nothing forces the prohibitions defined by the $isForbidden(\cdots)$ to be disjunct from the positive permissions granted by the $isAuthorized(\cdots)$, meaning that the order of appearance of the instructions influences the results of the method.

```
public class CObject  {
(...)
        public boolean isAuthorized(CSubject requestor, COperation op ) {

                if (requestor.isAdministrator()) {return true;}
                if (isForbidden(requestor, op)) {return false;}

                return false;
        }
(...)
}
```

Figure 3.7: *CObject access-control code*

```
public class CEvent extends CObject {
(...)
        public boolean isAuthorized(CSubject requestor, COperation op ) {
                if (super.isAuthorized(requestor, op)) return true;
                else return requestor.equals(this.owner)
                             || isAllowed(requestor, op);
        }
(...)
}
```

Figure 3.8: *CEvent access-control code*

```
public class CPrivateEvent extends CEvent {
(...)
        public boolean isAuthorized(CSubject requestor, COperation op ) {
                return requestor.equals(this.owner);
        }
(...)
}
```

Figure 3.9: *CPrivateEvent access-control code*

```
public class CGroupEvent extends CEvent {
(...)
   public boolean isAuthorized(CSubject requestor, COperation op ) {
     if (super.isAuthorized(requestor, op)) return true;

     if (owner.isGroup())
        if (requestor.equals(owner.getOwner())) return true;
        else if (owner.isMember(requestor) && op == COperation.READ)
                return true;

     return false;
    }
(...)
}
```

Figure 3.10: *CGroupEvent access-control code*

4

Model Extraction

Contents

In this chapter, we present the models and algorithms designed to help developers to detect potential mismatches between the specification and the implementation of a software system. When applicable, corrective measures can be offered to the developers. For simplicity, we focus only on the access-control perspective.

4.1 Overview

The main idea behind our approach is to provide methods and algorithms to help the developer make his executable source code and the underlying access-control model co-evolve. We provide him a way to propagate modifications made on one of them to the other.

The proposed approach (figure 4.1) consists in the extraction and formalization of the access-control elements implemented into the code and their verification against the desired properties specified by the developer. If inconsistencies are found, an execution trace leading to the error as well as potentially corrective measures are proposed :

- either by updating the requirements to reflect the desired behavior.
- either by updating the access-control model and propagating the updates directly into the executable code.
- or both.

The automation of the extraction and weaving of the access-control model into the code allows the developer to focus only on the requirements and on his code, hiding away all the formal “details”.

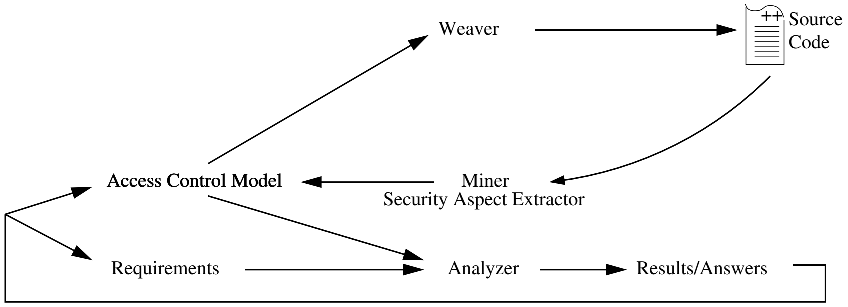


Figure 4.1: *Overview.*

The following sections and chapters detail each of the steps involved with the model extraction / weaving and the verification of the results. The remaining of this chapter focuses on the extraction of the access-control model from the source code, the specification of the requirements, the specification of the environment. The verification of the conformance of the extracted model to the requirements and the generation of the proposed corrective measures will be presented in the next chapter. Chapter 6 then addresses the weaving of the (possibly modified) access-control model back into the code and introduces some possible optimizations.

4.2 AC Model extraction

The first step toward proving the correctness of the produced implementation with respect to the specified requirements is to parse the source code and extract the implemented access-control model (figure 4.2). This step is obviously very dependent on the source code language chosen by the considered developers, their coding habits and followed conventions as well as on the structures found into it.

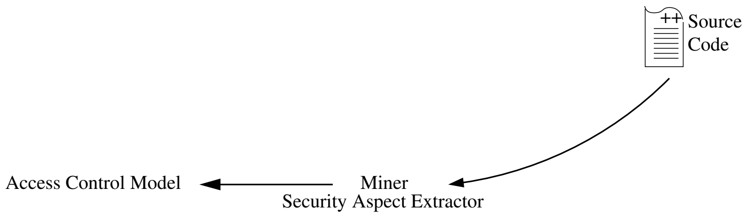


Figure 4.2: *AC Model extraction.*

This kind of exact static extraction is known to be computationally hard (see [Ric53] and the halting problem for more details) and is practically impossible to do in most modern language like Java or C without taking into account simplification hypothesis on the code to be analyzed.

4.2.1 Taking advantage of coding conventions

To alleviate the computational cost and complexity of the extraction as well as to sharpen the extraction result, the extraction algorithm uses the coding conventions followed by the developers. For ease of development and maintenance, developers usually produce code following some standards, be they explicit or implicit. For instance, they may follow variable naming schemes, standard code structures, use standardized API, ... The exact set of conventions used in the considered application is usually company-specific : each company uses the set most suitable to their needs and to the capacities of their developers.

The code snippet presented in figure 4.3, adapted from our case study, exhibits some of the possible conventions that may be encountered :

- regroup all the implemented access-control rules in a/some specific location(s) in the source code. This can be a be a dedicated function (as illustrated in the figure 4.3) or more distributed behaviors, like Dijkstra guarded commands (see [Dij76] for details on Dijkstra guarded commands).
- structure the code following a common framework. Here, the access-control rules are implemented under the form of simple "conditions implies decision" rules.
- voluntarily limit the source-language features used in the expression of those rules, in order to provide easy and quick readability as well as to avoid potential understanding problems between various team members.
- ...

This kind of coding conventions are not restricted to the object-oriented languages; similar or equivalent conventions can be found in almost all coding paradigms. In fact, many best-practices and coding standards encourage developers to apply such conventions to more readable and manageable code. See for instance [Fou11] , [EM09] or [US 88].

4.2.2 AC Model

The extracted access-control model will be expressed in a simple "condition ⇒ action" form using the simple syntax shown in figure 4.4. Inspired from logic languages, this language is designed to support monolithic access-control models as well as those structures through inheritance.

The model basically consists of a list of *classModel* items. Each of these *classModel* is identified by a name and contains a non-empty list of rules. A rule is formed of a precondition, expressed as a logical combination of predicates, and an action, here a permission (possibly a negative one, a prohibition) for a subject S to perform operation Op on object Ob. The set of rules applicable to an access-control class C is then formed by the combination of the rules of every classes it inherits from (H_1, H_2, $\ldots$, H_n) and all the rules specified in its body.

Any operation can be considered here, be the classic ones like read, write and execute or more advanced ones like grant read, grant write, grant execute, revoke read, revoke write, revoke execute, destroy and give. The only exception concerns the permission to create objects: this model is centered on per-object permission and, as, such, specifying the permission to create an object on a not yet created object would be odd. The constraints weighting on object creation will be specified later on in section 4.5.

```
public abstract class CustomObject  {
(...)
  public boolean isAuthorized(Person requestor, short nOps) {
    if (requestor.isAdministrator()) {return true;}

    if (isForbidden(requestor, nOps)) {return false;}

    PersonOrGroup owner = getOwner();
    if (requestor.equals(owner)) {return true;}

    if (owner.equals(Somebody.getInstance())) {return true;}

    if (owner.isGroup()) {
      if (requestor.equals(owner.getOwner())) {return true;}
      else if (((Group) owner).isMember(requestor)) {
         if (nOps == OperationType.READ) {return true;}
         }
      } (...)
     }
     return false;
  } (...)
}
```

Figure 4.3: *AC Model extraction, sample source code.*

In this sample Java class file, developers have grouped all the access-control elements in a specific method. Its content is structured in simple "condition => action" elements to maintain readability.

model	::	*classModel* *
classModel	::	**class** C [**extends** $H_1, H_2, \ldots, H_n$] *rule* +
rule	::	*pre* **:** [**not**] **allowed(S,Op,Ob)**
pre	::	predicate($\cdots$) | pre $\wedge$ pre | pre $\vee$ pre | $\neg$ pre | (pre)

Figure 4.4: *AC model syntax*

4.2.2.1 Expressivity

The syntax of this access-control rule specification language is voluntarily kept as simple as possible to allow developers to understand it without too much effort. It is however expressive enough to cover most of the access-control primitives found in standards like XACML v2 [OAS05].

In the same way, both DAC and MAC permission models can be expressed in this proposed language (figure 4.5). A DAC system can be modelled through the creation of a $class$ element for each system object, assigning the proper permission on the considered object to each couple user, operation. On the other hand, a MAC permission system can be modelled using a single permission class, deciding which access to grant (or to forbid) based solely on the clearance level of the considered user and object.

```
class DAC_Object_X
  : allowed(John, read, x)
  : allowed(Mary, write, x)
  : not allowed(S, Op, x)
```

```
class MAC
  S.level >= Ob.level : allowed(S, read, Ob)
  S.level <= Ob.level : allowed(S, write, Ob)
  : not allowed(S,Op,Ob)
```

Figure 4.5: *Expressing DAC and MAC properties.*

4.2.2.2 Semantics

The semantics of this simple access-control model language is almost straightforward. The expressed models are mapped to ordered sets of rules with respect to the following equations.

A rule represents a permission under a simple implication form :

$$[\![\text{pre* } \textbf{: allowed(S,Op,Ob)}\ ]\!] = [\![\text{pre*}]\!] \rightarrow can(S, Op, Ob) \tag{4.1}$$

$$[\![\text{pre* } \textbf{: not allowed(S,Op,Ob)}\ ]\!] = [\![\text{pre*}]\!] \rightarrow cannot(S, Op, Ob) \tag{4.2}$$

Where the $can(\cdots, \cdots, \cdots)$ predicate (resp. $cannot(\cdots, \cdots, \cdots)$) refers to the presence in the underlying access-control matrix of the corresponding positive (resp. negative) permission. The semantics of the $[\![$pre* $]\!]$ element is derived directly from the semantics of first-order logic.

Note : The order in which the rules appear in the model is important, so semantics of rules will be encompassed with a order number, allowing to manage the couples (*rule id* , *rule*) as set elements without losing ordering information. For readability reasons and to limit the size of the definitions, the *rule* element will sometimes be given in extended form $c \rightarrow perm(S, Op, Ob)$ where c and and $perm(S, Op, Ob)$ denote, respectively, the condition and the permission associated with the considered *rule* element.

Definition 1. *The rules* $(a, c_a \rightarrow perm_a(S_a, Op_a, Ob_a))$ *and* $(b, c_b \rightarrow perm_b(S_b, Op_b, Ob_b))$ **overlap** *if*

$$\exists\sigma : \left\{ \begin{array}{cc} & \sigma S_a = \sigma S_b \\ \wedge & \sigma Op_a = \sigma Op_b \\ \wedge & \sigma Ob_a = \sigma Ob_b \end{array} \right\} \wedge (\sigma c_a \cap \sigma c_b) \neq \emptyset$$

with $\{perm_a, perm_b\} \subset \{can, cannot\}$.

Two rules overlap if they can apply to the same set of permissions and their preconditions overlap; i.e. if there exists a substitution σ such that the permissions granted by the two considered rules apply on the same object and the rules preconditions overlap.

Definition 2. *The rules* $(a, c_a \rightarrow perm_a(S_a, Op_a, Ob_a))$ *and* $(b, c_b \rightarrow perm_b(S_b, Op_b, Ob_b))$ *are* **independent** *if they do not overlap.*

Two independent rules never apply on the same permission or have disjunct preconditions.

Definition 3. *The rules* $(a, c_a \rightarrow perm_a(S_a, Op_a, Ob_a))$ *and* $(b, c_b \rightarrow perm_b(S_b, Op_b, Ob_b))$ **conflict** *if they overlap and provide opposite permissions.*

$$\Longleftrightarrow \exists \sigma : \left\{ \begin{array}{ll} & \sigma S_a = \sigma S_b \\ \wedge & \sigma Op_a = \sigma Op_b \\ \wedge & \sigma Ob_a = \sigma Ob_b \end{array} \right\} \wedge (\sigma c_a \cap \sigma c_b) \neq \emptyset$$

with $perm_a = can$ *(resp.* $cannot$*) and* $perm_b = cannot$ *(resp.* can*).*

The simple ruleset shown in figure 4.6 illustrates these properties. Rules 1 and 2 overlap, because both of them can grant permissions to user $jean$ to $read$ object $system$. Rules 1 and 3 are independent : whatever the substitution, they will never apply on the same permission. Finally, rules 2 and 4 conflicts : there are situations where they overlap and provide opposite permissions to user $jean$ on object $system$.

```
1   (1, p(X) -> can(X, read, Z))
2   (2, true -> can(jean, Op, system))
3   (3, q(X) -> can(X, write, board))
4   (4, p(X) -> cannot(X, Op, system))
```

Figure 4.6: *Sample ruleset.*

Numerals and lower-case elements are constants, upper-case elements are variables.

In order to capture the full semantics of the set of rules contained in a $classModel$ element, resolving potential overlapping and/or shadowing between rules, we define a specialized union operator $\overset{M}{\bigcup}$:

Definition 4. *The* **rule combination operator** $\overset{M}{\bigcup}$ *computes the combination of two rules, removing any existing overlapping between them.*

$$(i, r_i) \overset{M}{\bigcup} (j, r_j) = \{(1, r_i), (2, u)\}$$

where :

- $r_i = p : perm(S, Ob, Op)$
- $fresh(r_j, r_i) = q : perm'(S', Op', Ob')$
- $u = \neg(p \wedge S = S' \wedge Op = Op' \wedge Ob = Ob') \wedge q : perm'(S', Op', Ob')$

With $fresh(a, b)$ providing a fresh renaming of all free variables occurring in a and previously appearing in b.

Applying the $\overset{M}{\bigcup}$ operator on the two first rules of the ruleset presented in figure 4.6 gives :

$$\left\{ \begin{array}{l} (1, p(X)- > can(X, read, Z)) \\ (2, not(p(X)\&X = jean\&Op = read\&Z = system)\&true- > can(X, Op, system)) \end{array} \right\}$$

where the second element can be simplified to obtain the more readable permission set

$$\left\{ \begin{array}{l} (1, p(X)- > can(X, read, Z)) \\ (2, not(p(jean)\&Op = read)- > can(jean, Op, system)) \end{array} \right\}$$

stating that anyone satisfying the precondition $p(X)$ can read any object and that user $jean$ can execute any operation on object $system$ if satisfying the given condition. The precondition appearing in the second rule only states that this particular rule should not grant the permission to user $jean$ to read object $system$ when satisfying p because this permission is already managed by the first rule.

In case of overlapping / conflict between the two rules, as it is the case in the example above, the $\overset{M}{\bigcup}$ operator gives priority to the left-hand rule (figure 4.7). A direct consequence of this definition is the behavior with respect to the empty set :

$$\emptyset \overset{M}{\bigcup} (i, r) = \{(1, r)\} = (i, r) \overset{M}{\bigcup} \emptyset \tag{4.3}$$

This adapted union operator is then easily extended to combine set of rules :

$$\{(i, r_i)\}_{i=1,\cdots,n} \overset{M}{\bigcup} \{(j, s_j)\}_{j=1,\cdots,m} = (1, r_1) \overset{M}{\bigcup} (2, r_2) \overset{M}{\bigcup} \cdots \overset{M}{\bigcup} (n, r_n) \overset{M}{\bigcup} (1, s_1) \overset{M}{\bigcup} (2, s_2) \overset{M}{\bigcup} \cdots \overset{M}{\bigcup} (m, s_m) \tag{4.4}$$

With the "shortcut" notations :

$$\overset{M}{\bigcup}{}_{a \leq i \leq b} (i, r_i) = \left\{ (1, r_a) \overset{M}{\bigcup} \left(\overset{M}{\bigcup}{}_{a+1 \leq l \leq b} (l, r_l) \right) \right\} \tag{4.5}$$

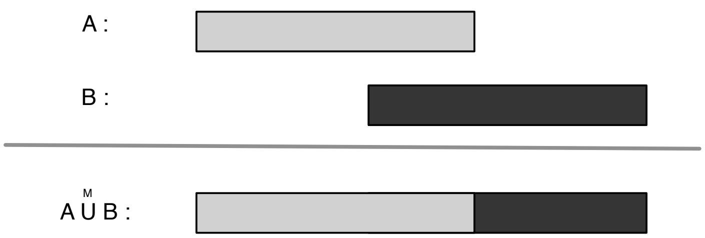


Figure 4.7: $\overset{M}{\bigcup}$ *operator.*

In case of conflicting rules, the rule combination operator gives priority to the left-hand member.

From there, the semantics of a *classModel* element can be obtained by the combination of all the relevant rules, both inherited and explicit :

$$[\![\textbf{class} \ \mathrm{C} \ [\ \textbf{extends} \ H_1, H_2, \ldots, H_n \ ] \ \textit{rule} \, +]\!] = \left(C, \left(\overset{M}{\bigcup} {}_{1 \leq j \leq n} [\![H_j]\!] \right) \overset{M}{\bigcup} \left(\overset{M}{\bigcup_{1 \leq i \leq | \ \textit{rule} \, +|}} (i, \{[\![\ \textit{rule} \ [i]]\!]\}) \right) \right) \tag{4.6}$$

Decomposing equation 4.6 in more manageable parts, the semantics of the *classModel* element is given by the combination of all the rules inherited into the class (left-hand part of the head rule combination operator) and class-specified rules (right-hand part).

Finally, the semantics of the *model* element is simply given by the set of the semantics of the *classModels* it contains :

$$[\![\ \textit{classModel} \, *]\!] = \bigcup_{i=1,\cdots,n} \{[\![classModel_i]\!]\} \tag{4.7}$$

4.2.3 Extraction Process

The above-presented access-control model imposes constraints on the extraction process. Representing the access-control rules as a sequence of logical implications and replicating some of them through several rules requires that the basic blocks of these implications behave as state-less predicates and functions.

Definition 5. *A* **state-less predicate** *is a predicate $p(a_1, \cdots, a_n)$ whose truth value depends solely on the values of its arguments and does not modify the execution environment.*

Definition 6. *A* **state-less function** *is a function $f(a_1, \cdots, a_n)$ whose return value depends solely on the values of its arguments and does not modify the execution environment.*

Manipulating state-less functions and predicates allows the adapted union operator $\overset{M}{\bigcup}$ to replicate calls into the rules during its disambiguation process without modifying the semantics of the rule set.

The choice of state-less functions and predicates has an impact on the code constructs that can be accepted: it prohibits the use of random and/or environment-driven functions inside the access-control methods. For instance, methods asking the user to allow or forbid an access through a dialog box or allowing the five first access queries cannot be represented in this model.

The access-control model extraction process is driven by the chosen implementation language and developer's coding conventions. The remaining of this section will present the extraction techniques adapted from our case study and illustrate their behavior on the running example from section 3.3.

4.2.3.1 Specific hypothesis

Apart from the constraints presented above, building an automated AC-extraction algorithm requires identifying precisely what coding conventions can be used to ease the process. These are obviously specific to the considered project, but the same kind of strategy can apply in most of the applications.

Considering the running example introduced in section 3.3 (page 51), the access-control related elements of the code are clearly identifiable. They are contained in $isAuthorized(requestor, op)$ methods and their inner structure is mainly based on conditional statements. A call to this method is used to determine if the specified $requestor$ is allowed or not to perform operation op on the current instance of the object. An execution branch ending with a $return(true)$ instruction permit the queried access while branches ending with $return(false)$ refuse it. The access-control methods of the different object classes from the example are linked together through the utilisation of the $super.isAuthorized(\cdots)$, referring to the access-control properties of the supertype of the currently considered object instance (following Java class-hierarchy).

In order to comply with the state-less requirement applying to functions and predicates manipulated by the adapted union operator and to keep the extraction algorithm as readable as possible, all function calls encountered into the access-control methods will be considered state-less. There is no easy way to check this rather strong hypothesis but to rely on developer's understanding of his code.

4.2.3.2 Extraction Algorithm

The algorithms in this section will be presented in a pseudo-code format. Some details will be moved from the algorithm's code to the accompanying descriptions for clarity purposes.

The basic idea is to process each source class and parse the corresponding access-control method. Then, for each execution path ending with a $return(\cdots)$ statement, produce the access-control rules representing the semantics of the execution paths leading to that $return(\cdots)$ statement.

Algorithm 1 SECURITY ASPECT EXTRACTION

```
 1: for all class C do
 2:   $m \leftarrow$ AC_Method_Name
 3:   for each return statement R in m do
 4:     $P \leftarrow$ paths(m,R)
 5:     for each $p \in P$ do
 6:       for all (Q,Action) $\in$ outcomes(R, $p$,requestor, Op, Ob) do
 7:         Model $\leftarrow^{+}$ $(class(C, Ob) \wedge Q \rightarrow$ Action)
 8:       end for
 9:     end for
10:   end for
11: end for
```

Algorithm 1 details this process :

[**l. 1-2**] for each class C in the considered source code, the AC-method is extracted (m)[1]

[**l. 3**] then for each $return(\cdots)$ statement R found in m:

- [**l. 4**] all the execution paths inside m leading to R are collected into P (see algorithm 3 for details on this process) .
- [**l. 5-7**] an access-control rule is added to the model for every potential outcome of each of the considered execution path (see algorithm 2 for details on these potential outcomes generation)

The Ob appearing in the rules refers to the current instance of the object of class C, i.e. the object on which the access is requested.

The $outcomes(R, P, requestor, Op, Ob)$ algorithm (algorithm 2) converts the selected return statement (R) into a guarded permission, making the implicit elements inherited from the access-control function prototype explicit. It returns a set of couples containing a sequence of instruction and their associated permission.

Algorithm 2 OUTCOMES($\cdots$)

```
if (R == return(true)) then
    returns( { (P, allowed(requestor, Op, Ob) } )
else
    if (R == return(false)) then
        returns( { (P, not allowed(requestor, Op, Ob) } )
    else
        { R == return(X) }
        return ( { (P $\wedge$ X , allowed(requestor, Op, Ob)) , ((P $\wedge$ $\neg$ X , not allowed(requestor, Op, Ob)) } )
    end if
end if
```

The $paths(m, R)$ algorithm (algorithm 3) returns all execution paths in m leading to R. It works by reverse induction over the structure of the abstract syntax tree representing the considered method source code (figure 4.8).

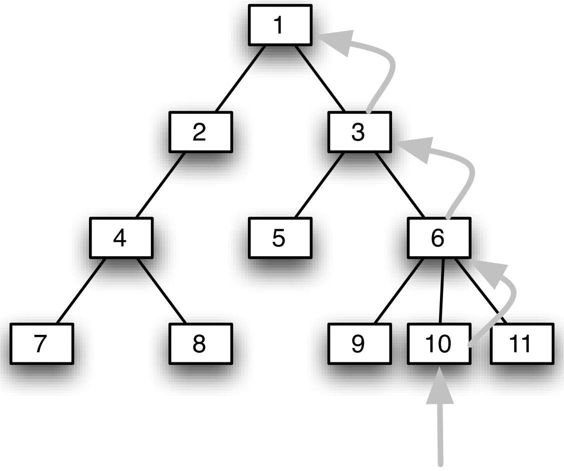


Figure 4.8: $paths(\cdots)$ *: reverse induction over abstract syntax tree.*

Algorithm 3 PATHS(M, R)

```
S ⟵ R
A ⟵ ∅
while ($m \neq S$) do
    ($\cdots$) {rev-induction(S,A)}
    S ← S.father();
end while
return A
```

With the reverse induction on structure (rev-induction(S,A)) mapping a syntactic structure from the abstract syntax tree to a set of sequences of instructions according to the rules presented in equations 4.8 to 4.11:

$$\texttt{return(X)} \quad \longrightarrow \quad \{\ [\mathrm{return(X)}]\ \} \tag{4.8}$$

A return instruction basically marks the end of the instruction sequence leading to itself. As such, it is represented by the sequence of instructions containing only itself.

$$\texttt{if (E) then } S_1 \texttt{ else } S_2 \quad \longrightarrow \quad \left\{ \begin{cases} [\,(\mathrm{E} == v)\ \mid \mathrm{A}] & \text{if } A \subseteq (S_1 \bigcup S_2) \\ [\,(\mathrm{E{=}{=}true})\ ;\ S_1\ \mid \mathrm{A}]\,,\,[\,(\mathrm{E{=}{=}false});\ S_2\ \mid \mathrm{A}] & \text{otherwise} \end{cases} \right\} \tag{4.9}$$

A conditional instruction can be encountered in two distinct contexts : the execution path can either come out of one of the two branches of the alternatives or it can come across the whole instruction. The two parts of the equation 4.9 adress these two possibilities.

- The upper part of the equation manages the case where the execution path is coming out of the “then” or of the “else” branch of the conditional. The resulting execution path is composed of the instructions declared in the condition expression (E) and the expected truth value (v) followed by the instructions contained in the alternative we are coming from. The expected truth value (v) is based on the branch containing the considered $return(\cdots)$ statement : $v = true$ for S_1 and $v = false$ for S_2. The accumulator A contains the sequence of instructions required to go from the return statement at the leaf of the syntax tree up to the conditional we are considering.
- The lower part adresses the case where the whole conditional instruction has to be included in the execution path. It then produces two (possibly sets of) execution paths : one considering the execution of the “then” part of the alterative, the other considering the “else” one. As a simplification, if any of the sequences generated from these branches contains a $return(\cdots)$ statement which is always executed, they can be omitted.

The notation $[\ \mathrm{A}\ \mid \mathrm{B}]$ denotes the set of sequences formed by the element A followed by every sequence in B.

$$\texttt{x=y} \quad \longrightarrow \quad \{\ [\text{z=y} \mid (\text{A} \ \{\ \text{x / z}\ \})]\ \} \tag{4.10}$$

An affectation instruction consists in the evaluation of the affectation and the propagation of the modified value in the remainder of the sequence. The z variable appearing in equation 4.10 is a fresh variable containing the effects of the affectation. The notation $A\,\{$ x / z $\}$ represents the substitution, in A, of all the free occurrences of x by z.

$$\texttt{f}(a_1, \cdots, a_n) \quad \longrightarrow \quad \{\ [\text{f}(a_1, \cdots, a_n) \mid \text{A}]\ \} \tag{4.11}$$

Much like the $return(\cdots)$ statement, a function call simply inserts itself in the sequence of instructions.

Some instructions have been voluntarily excluded from this induction process : the loop instructions ($while$ and for for the Java language). Finite loops can be unrolled as sequences of instructions (see figure 4.9 for an example of such loop unrolling) while allowing potentially infinite ones would make the code parsing and the attached semantics much more complex without adding much to the expressivity of the access-control code.

```
i = 0;
while (i < 5) do
    foo(i);
    i = i+1;
done;
```

```
foo(0);
foo(1);
foo(2);
foo(3);
foo(4);

```

Figure 4.9: *Unrolling finite loops.*

Some loops can be unrolled and converted to a sequence of instruction while preserving their semantics.

4.2.3.3 Example

As an illustration of the behavior of these algorithms, they will be applied to the running example introduced in section 3.3 (page 51).

The dedicated access-control method found in the $CObject$ class source code (presented in figure 3.7) can be represented as the abstract syntax tree show in figure 4.10. Graphically speaking, the extraction process will :

1. identify all the return statements present into the code (marked with a $\star$ in figure 4.10)

2. build the sequence of instructions running from the beginning of the access-control method to the considered $return(\cdots)$ statement. The three paths identified here are labelled P_1, P_2 and P_3.

3. create the access-control rules representing each of these paths.

The resulting access-control model can be found in figure 4.11. In this example, the rule for path $P2$ (lines 6 and 7) was simplified by removing the dead branches created by the presence of the $return(\cdots)$ statement in the first conditional instruction. The typing element $class(CObject, Ob)$ found on every line allows to trace any rule back to it's originating source code class, even after the access-control rules have been disambiguated.

As an illustration, the extracted ruleset from the $CObject$ class (figure 4.11) produces the following independent rules after the merging process :

$$\left\{ \begin{array}{l} S.isAdministrator() == true \longrightarrow allowed(S, Op, Ob) \\ S.isAdministrator() == false \wedge isForbidden(S, Op) == true \longrightarrow not\ allowed(S, Op, Ob) \\ S.isAdministrator() == false \wedge isForbidden(S, Op) == false \longrightarrow not\ allowed(S, Op, Ob) \end{array} \right\}$$

For readability purposes, the $class(\cdots)$ elements were omitted from the above formulas and some variables have been renamed.

The models obtained at this stage reflect the semantics of the access-control decisions taken into the considered source code, they both share the same strengths and weaknesses. For instance, the model obtained from the $CEvent$ class does not cover all of the possible execution paths[2]. This lack of a default case is a hint for a mis-expressed security property and should be reported back to the developer if this missing default case appears to be reachable the end of the verification process.

The same process leads to the production of the access-control models for the $CEvent$ (figure 4.12) , $CPrivateEvent$ (figure 4.13) and $CGroupEvent$ (figure 4.14) classes.

[2] This code takes no decision in the case where the $requestor$ element satisfies *not requestor.equals(owner) and not isAllowed(requestor, op)* .

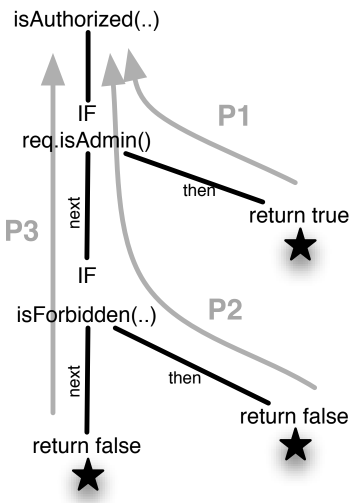


Figure 4.10: *Abstract syntax tree for the CObject class.*

```
class CObject

(class(CObject, Ob) & requestor.isAdministrator() == true)
                    : allowed(requestor, Op, Ob)

(class(CObject, Ob) & isForbidden(requestor, op) == true)
                    : not allowed(requestor, Op, Ob)

(class(CObject, Ob) : not allowed(requestor, Op, Ob)
```

Figure 4.11: *Access-control model extracted from the CObject class source code.*

```
class CEvent

(class(CEvent, Ob) & super.isAuthorized(requestor, op) == true)
                   : allowed(requestor, Op, Ob)

(class(CEvent, Ob) & super.isAuthorized(requestor, op) == false
                   & requestor.equals(this.owner))
                   : allowed(requestor, Op, Ob)

(class(CEvent, Ob) & super.isAuthorized(requestor, op) == false
                   & isAllowed(requestor, op))
                   : allowed(requestor, Op, Ob)
```

Figure 4.12: *Access-control model extracted from the $CEvent$ class source code.*

```
class CPrivateEvent

(class(CPrivateEvent, Ob) & requestor.equals(this.owner) == true)
                          : allowed(requestor, Op, Ob)


(class(CPrivateEvent, Ob) & requestor.equals(this.owner) == false)
                          : not allowed(requestor, Op, Ob)

```

Figure 4.13: *Access-control model extracted from the $CPrivateEvent$ class source code.*

```
class CGroupEvent

(class(CGroupEvent, Ob) & super.isAuthorized(requestor, op) == true)
                        : allowed(requestor, Op, Ob)

(class(CGroupEvent, Ob) & owner.isGroup()
                        & requestor.equals(owner.getOwner()) == true)
                        : allowed(requestor, Op, Ob)

(class(CGroupEvent, Ob) & owner.isGroup()
                        & requestor.equals(owner.getOwner()) == false
                        & owner.isMember(requestor) == true
                        & op == COperation.READ)
                        : allowed(requestor, Op, Ob)

(class(CGroupEvent, Ob) : not allowed(requestor, Op, Ob)
```

Figure 4.14: *Access-control model extracted from the $CGroupEvent$ class source code.*

4.3 Requirements specification

Aside from the production of the source code, the second major developer’s intervention in our proposed methodology lies in the specification of the requirements. No verification is possible without a good description of the properties expected from the implementation. The perso most suited to produce such a requirement specification being the developer, we attempt to provide him a simple yet expressive language for this task.

The simple requirements expression language detailled in this section was developed to allow the developer to define precisely what properties are to be expected from the implemented source code. It aims both at readability and expressivity and is adressed at a public a priori not introduced in formal languages and methods.

4.3.1 Language

The requirement specification language tries to conjugate simplicity and expressivity. Its main goal is to allow developers to express the access-control properties the system should meet. It tries to stay as natural as possible while providing enough expressivity to describe real-world situations in access-control.

In terms of expressivity, the chosen language has to be able to represent the two most frequent access-control requirements:

- The static query : *is “X” able/forbidden to do operation “Y” on object “Z" ?* (said otherwise : is (X, Y, Z) an active permission/prohibition)
- And the dynamic one : *is there a way “X” can be become able/prohibited to perform operation ”Y” on object “Z" ?* This question can be refined to take into account the possible allies and/or adversaries of X in this process. From an access-control point of view, a question like *“can my adversaries access my private data without my consent ?”* is much more interesting than a question like *“can my adversaries access my private data ?”* , which is usually true because nothing prevents me from granting them the access.

It is not only interesting to know if something can or will happen, but also if some agent(s) can control the evolution of the system in order to enforce a given property, whatever the other agents do.

4.3.1.1 Syntax

Partially inspired from the alternating-time temporal logic (ATL) [AHK97] the syntax chosen for the requirements specification language is shown in figure 4.15. This syntax attempts to be rich enough to express the most commonly encountered requirements while staying as simple as possible to be understandable without too much effort by the developers.

A requirement consists of a property (here labelled $agentProp$) to be satisfied under some hypothesis (the $domain$ element). The specified property can be a simple permission (or prohibition) or more complex structures like restricted branching time logic properties or agent-dependent properties.

- a $requirement$ represents a property (here labelled $prop$) to be satisfied under certain circumstances (labelled $domain$).
- a $domain$ represents the preconditions of a property. It can be a simple predicate or more complex structures like a conjunction of domains or the negation of a domain. For readability purposes, a domain can also be written using parenthesis.
- a $prop$ element represents a property. It can be either a direct permission (labelled $localProp$), a temporal property, an agent-related property or a conjunction/disjunction/negation of properties.
- a $localProp$ represents a simple permission ($allowed(\cdots)$) or prohibition ($not\ allowed(\cdots)$).
- a $timeProp$ represents a temporal property, i.e. a property applying to the present or to one (or many) future state(s) of the system.
- finally, an $agentProp$ is a property specifying the set of users to be considered for the satisfaction of the defined property. The syntactic element $alone(A, prop)$ specifies that the $prop$ property should be satisfied only by the actions of agent A, whatever other agents can do on the system. Similarly, $together(\{A+\}\,, prop)$ specifies that the set of agents $\{A+\}$ should be able to enforce $prop$ by their own actions.

The $timeProp$ element $now(\cdots)$ is introduced only for readability purposes and can be omitted. The same applies to the $agentProp$ element $alone(A, prop)$ when it is used with A referring to the default considered user. They have been introduced in the syntax only to make it as regular as possible, easing the learning curve of the language for developers.

This simple syntax allows us to define complex properties like liveness enforcement or context-dependent requirements. Let us illustrate these possibilities.

```
administrator(X) : alone(X, alwayswill( allowed(X, Op, Ob )))
```

This property expresses the fact that "an administrator should always have access to any object on the system". Any user satisfying the precondition (hence being an administrator) should have a strategy to enforce that he will always be able to access any object on the system, whatever other users on the system can do,

```
requirement :: domain : prop

domain :: predicate | domain & domain
        | ~ domain | (domain)

prop :: localProp | timeProp | agentProp
      | prop & prop | prop || prop | ~prop

localProp :: allowed(S, Op, Ob)
           | not allowed(S, Op, Ob)

timeProp :: now(prop) (*)
          | may(prop)
          | alwayswill(prop)

agentProp :: alone(A, prop) (*)
           | together({A+}, prop)
```

Figure 4.15: *Requirement specification language.*

Properties marked with a (*) are considered default properties and can be omitted.

possibly attempting to prevent him to access the object.

```
cashier(X)  & supervisor(Y, X) & work(X,Z) & special(Z)
      : may(alone(X, not allowed(Y,_,Z)))
```

This property states that, under some specific circumstances (expressed by the domain predicate `special(Z)`), a cashier may be allowed to punctually prevent his supervisor from accessing some of his work.

```
1. object(S) : alwayswill(may(allowed(X,_,S))
2. object(S) : alwayswill(alone(X,may(allowed(X,_,S)))
```

These requirements act as a liveness enforcement system on the object S for user X: they enforce that, at any time, someone will have the possibility to access the object. The difference between the two is that the second one guarantees that, at any time, X will have a strategy on his own to access S, while the first one may require actions from any other agents on the system.

4.3.1.2 Semantics

The semantics of this language is defined through the mapping of the different constructs to ATL logic formulas. In the following, the notation $[\![X]\!]_Y$ will refer to the semantics of the X element, considering the contextual set of agents Y. $[\![X]\!]_\_$ indicates that the contextual set of agents is irrelevant to the considered element.

- a *requirement* is a property to be satisfied under the specified hypothesis :

$$[\![domain : agentProp]\!]_\_ = [\![domain]\!]_\_ \wedge [\![agentProp]\!]_{Agt}$$

- a *domain* is a simple propositional logic formula :

$$\begin{aligned}
&[\![p]\!]_\_ = p\\
&[\![domain\&domain]\!]_\_ = [\![domain]\!]_\_ \wedge [\![domain]\!]_\_\\
&[\![\sim domain]\!]_\_ = \neg[\![domain]\!]_\_\\
&[\![(domain)]\!]_\_ = [\![domain]\!]_\_
\end{aligned}$$

- the boolean operators on the properties behave as usual :

$$[\![p_1 \,\&\, p_2]\!]_C = [\![p_1]\!]_C \wedge [\![p_2]\!]_C$$

$$[\![p_1 \,||\, p_2]\!]_C = [\![p_1]\!]_C \vee [\![p_2]\!]_C$$

$$[\![\sim p]\!]_C = \neg[\![p]\!]_C$$

- a $localProp$ is a permission or prohibition valid on the current state of the system, it is equivalent to the presence of the specific permission/prohibition in the permission matrix.

  $$[\![allowed(S, Op, Ob)]\!]_{\_} = can(S, Op, Ob)$$

  $$[\![not\ allowed(S, Op, Ob)]\!]_{\_} = cannot(S, Op, Ob)$$

  The $can(\cdots)$ and $cannot(\cdots)$ elements are the same as those presented in section 4.2.2.2.

- a $timeProp$ is a property related to the evolution of the access-control model of the system, it is mapped almost directly to the corresponding ATL constructs.

  $$[\![now(prop)]\!]_C = [\![prop]\!]_{\_}$$

  $$[\![may(prop)]\!]_C = \langle\langle C\rangle\rangle\Diamond[\![prop]\!]_{\_}$$

  $$[\![alwayswill(prop)]\!]_C = \langle\langle C\rangle\rangle\Box[\![prop]\!]_{\_}$$

- an $agentProp$ is a property specifying which set of users is to be considered as potential allies in the process of obtaining the desired property. A property specified through the $alone(A, prop)$ construct should be reached only via the actions of A, where a property specified through $together(\{A^+\}, prop)$ should be satisfied through a coordinated action from all the users specified in A^+.

  $$[\![alone(A, prop)]\!]_C = [\![prop]\!]_A$$

  $$[\![together(\{A+\}, prop)]\!]_C = [\![prop]\!]_{\{A+\}}$$

4.3.2 Expressivity/Limitations

Evaluating the expressiveness of a language is a complex task; from a formal point of view, the requirement language defined above is strictly less expressive than the full ATL logic (see [LMO07] for an analysis of the expressiveness of ATL). This difference is mainly due to the absence of the ATL “until” operator in our requirements language.

However, this language is aimed at developers with little to no knowledge of formal methods. Interviews among them showed a great deal of confusion with the possible semantics of an ATL-like $until$ operator. Furthermore, no obvious properties requiring such an operator appeared during the analysis stage preceding the development of this requirement specification language.

4.3.3 Example

Applying the requirements specification language to the running example defined in section 3.3 could bring the requirements shown in figure 4.16. It contains two very simple requirements. The first one states that any administrator can always access any object on the system, whatever the object or the operation he wishes to execute are. The second one states that the owner of an object of type $CPrivateEvent$ can always behave such that he can prevent any other user from accessing his object or, said otherwise, a user cannot access a private object without explicit approval from the object's owner.

```
# an administrator can always access any object
S.isAdministator() : allowed(S, Op, Ob)


# private events cannot be read without owner's will
class(CPrivateEvent, Ob), X=Ob.owner(), user(Y), X!=Y
           : alone(X, alwayswill(not allowed(Y, Op, Ob)))
```

Figure 4.16: *Requirements for the running example.*

Nothing at this stage prevents the developer from specifying conflicting properties. For instance, the two properties declared into this example are clearly incompatible : satisfying the first one will almost undoubtedly break the seconde one and vice versa. The detection and settling of these conflicts will be done during the verification stage presented in the next chapter.

4.4 Initial configuration

Specifying the initial state of the system is done in a very straightforward way : developers are required to provide the return values of all the predicates and functions appearing in the required properties and in the access-control model extracted from the code. To simplify as much as possible, only true predicates will be specified, any non-specified one will be considered as false.

This specification will consist in the set of the initial users and objects present on the system at initialization and a list of the predicates to be considered as true. Figure 4.17 shows the simple syntax : the list of initial users is specified in line 1, the list of objects in line 2 and the set of predicates to be considered as true follows.

The generation of such an initial configuration description can be greatly facilitated by a tool. The set of users and objects explicitly appearing into the requirements and/or the code can be extracted automatically requiring only limited confirmation from the developers. The same applies to the predicates : the set of

```
users = U*
objects = Ob*

pred*
```

Figure 4.17: *Initial configuration specification.*

predicates declared in the initial configuration is a subset of the predicates appearing in the models; listing them in the tool allows the developers to select only those relevant and give their truth value, easing the whole process.

While it may seem unpractical to express all the properties holding on the initial state of the system, it should be noted that most of the systems only define a very limited number of such static elements. Usually, only a few "important" elements are defined prior of the initialization of the system, for instance a general administrator account. All other resources and permissions are then defined at runtime through dedicated methods.

4.4.1 Example

As a quick illustration of this simple initial properties specification, figure 4.18 shows a possible initialization for the running example from section 3.3. Two users are hard coded at system initialization: a $root$ and a $nobody$ user. Only one object exists, called $sandbox$ and the only properties to be true at startup will be that user $root$ is an administrator, any other will be considered false.

```
users = root nobody
objects = sandbox

root.isAdministrator()
```

Figure 4.18: *Initial configuration specification for the running example.*

4.5 System Model

The access-control model defined in the above sections is based on per-object permissions. To model the full set of permissions governing the system, it is necessary to define how the set of objects and users populating the system can evolve. More precisely, the permissions to perform actions such as read, write, execute, grant read, grant write, grant execute, revoke read, revoke write, revoke execute and give on a specific object are already managed by the model defined in section 4.2.2, but operations like object creation or user creation / destruction are not[3] . The purpose of the simple syntax introduced in this section is twofold: first to specify the conditions under which those object-less operations can be executed and second, what are the effects, from an access-control point of view, of the execution of the permitted operations on the system model (figure 4.19).

```
1  class C:
2    [pre : create()]*
3    [op : effect]*
4
5  System.users :
6    [pre : create()]*
7    [pre : destroy(u)]*
```

Figure 4.19: *System model syntax.*

For each class, the developers are able to define what set of properties is required from the user to be able to create an object of type C (line 2) and what are the effects of the execution of a specific operation op on this object.These can be expressed in terms of modified predicates/system variables and/or modified permissions. The $effect$ element on line 3 expresses the set of system variables changed by the considered operation as a parallel affectation. For readability purposes, the values of the variables after the operation are suffixed with a prime symbol. True boolean values can also be omitted. As an illustration, an operation incrementing an internal counter (labelled V) by 1 and an operation swapping the value of 2 arguments could be specified with :

$$inc() : V' = V + 1$$
$$swap(A, B) : A', B' = B, A$$

It is important to note here that no preconditions are specified for the operations based on the object (i.e. all operations but the $create()$ one) in this model. The permissions on such operations are managed through

[3] As this section does not take into account operations executed on a specific object, the permission to destroy an object is not present here. It is defined in the class-specific access-control rules from section 4.2.2

the access-control model defined in section 4.2.2, via $allowed(\bullet, give, this)$ permissions.

The same applies to the $System.users$ pseudo-class which represents the set of users populating the system. Any existing user satisfying the requirements expressed in the pre statement can either create a new user on the system (line 6) or destroy an existing user (line 7). Once the considered objects / users have been created, they can be granted permissions on / to them through the standard access-control rules.

4.5.1 Example

As an illustration of such a system model in the case of our running example (see section 3.3 for an introduction to the running example), a possible system model could look like the one presented in figure 4.20.

```
class CObject:

class CEvent, CPrivateEvent, CGroupEvent :
  user(X) : create()
  give(X) : this.getOwner()' = X
  grant_read(X) : allowed(X, read, this)
  grant_write(X) : allowed(X, write, this)
  revoke_read(X) : not allowed(X, read, this)
  revoke_write(X) : not allowed(X, write, this)

System.users :
  X.isAdministrator() : create()
  X.isAdministrator() : destroy(u)
```

Figure 4.20: *System model for the running example.*

This model states that :

- no one is able to create $CObject$ objects (no creation rule appears)
- any user of the system can create a calendar event (i.e. an object of class $CEvent$, $CPrivateEvent$ or $CGroupEvent$) [4].
- the operations $grant\_read$, $grant\_write$, $revoke\_read$ and $revoke\_write$ have their usual signification.

[4] this rules was written using a few syntactic sugar, preventing the duplication of the same identical rule into the 3 separate classes.

- the creation and removal of users from the system is only allowed to users flagged as administrators.

Rules in lines 6 to 9 are expressed in compact format, equivalent to the extended syntax :

$$\begin{aligned}
&grant\_read(X) : allowed(X, read, this)' = true \\
&grant\_write(X) : allowed(X, write, this)' = true \\
&revoke\_read(X) : notallowed(X, read, this)' = true \\
&revoke\_write(X) : notallowed(X, write, this)' = true
\end{aligned}$$

5

Model Verification

Contents

This chapter presents model-checking and result reporting techniques applicable to the models and properties defined in the previous chapter. A quick introduction to the foundations of the logic system used is presented, followed by the model-checking process in itself. The reporting of the model-checking results back to the developer is then covered. All these methods are then illustrated on an excerpt from the running example.

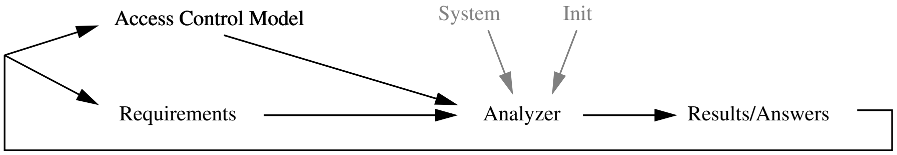


Figure 5.1: *Model-checking.*

The model-checking process can dispose of all the available models from the previous chapter : the access-control model, the requirements specification, the system description and the initial configuration specification (figure 5.1). More formally, the access-control model, noted Ac_M (p. 60), models the rules governing the access or refusal of access to the system's objects. It consists in a set of $classModels$ components, each composed of a set of independent access-control rules modelled as simple implications. The requirements R_M (p. 76) represents the properties expected to be verified by the system. They are modelled as ATL logic formulas. The initial system configuration I_M (p. 81) describes the set of objects, the set of users and the set of true propositions existing at the initialization of the system. Finally, the system model S_M (p. 83) is a set of class actions description, each modeling the effects of all the possible operations on this class type.

The model-checking and result reporting processes presented in the following sections will basically consist in two steps. First the requirements R_M will be verified on a transition system build from Ac_M , I_M and S_M , then the results of this verification will be reflected back in an appropriate way on the input models R_M , Ac_M , I_M and S_M .

5.1 Logic and Properties

This section introduces the syntax and semantics of ATL logic as well as some key properties that will be used later on in this chapter. The original definitions and model-checking procedures proposed by the authors in [AHK97] are reproduced here as they will be the foundation of the algorithms proposed in the following sections. Some useful properties taken from [WWZX09] are also presented.

Definition 7. *[AHK97] The temporal logic* **ATL (Alternating-time Temporal Logic)** *is defined with respect to a finite set Π of propositions and a finite set $\Sigma = \{1, \cdots, k\}$ of players. An ATL formula is one of the following:*

(S1) *p, for propositions $p \in \Pi$.*

(S2) *$\neg\phi$ or $\phi_1 \vee \phi_2$, where ϕ_1 and ϕ_2 are ATL formulas.*

(S3) *$\langle\langle A \rangle\rangle \bigcirc \phi$, $\langle\langle A \rangle\rangle \Box \phi$, or $\langle\langle A \rangle\rangle \phi_1 \,\mathcal{U}\, \phi_2$, where $A \subseteq \Sigma$ is a set of players, and ϕ, ϕ_1 and ϕ_2 are ATL formulas.*

The notation "$\langle\langle . \rangle\rangle$" is a path-quantifier and "$\bigcirc$" (next), "$\Box$" (always) and "$\mathcal{U}$" (until) are temporal operators. An additional notation "$\Diamond$" (eventually) is used as a shortcut : $\langle\langle A \rangle\rangle \Diamond \phi = \langle\langle A \rangle\rangle \; true \; \mathcal{U} \; \phi$.

Definition 8. *[AHK97] A* **concurrent game structure** *is a tuple $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ with :*

- *A natural number $k \geq 1$ of agents. Players will be identified with their number $1, \cdots, k$ and the set $\{1, \cdots, k\}$ of players will be noted Σ.*
- *Q is a finite set of states.*
- *Π is a finite set of atomic propositions.*
- *π is the labeling function, i.e. for each state $q \in Q$, $\pi(q) \subseteq \Pi$ contains the set of atomic propositions true at q.*
- *For each player $a \in \{1, \cdots, k\}$ and each state $q \in Q$, a natural number $d_a(q) \geq 1$ of moves available at state q to player a. We identify the moves of player a at state q with the numbers $1, \cdots, d_a(q)$. For each state $q \in Q$, a move vector at q is a tuple $\langle j_1, \cdots, j_k \rangle$ such that $1 \leq j_a \leq d_a(q)$ for each player a. Given a state $q \in Q$, we write $D(q)$ for the set $\{1, \cdots, d_1(q)\} \times \cdots \times \{1, \cdots, d_k(q)\}$ of move vectors. The function D is called move function.*
- *For each state $q \in Q$ and each move vector $\langle j_1, \cdots, j_k \rangle \in D(q)$, a state $\delta(q, j_1, \cdots, j_k) \in Q$ that results from state q if every player $a \in \{1, \cdots, k\}$ chooses move j_a. The function δ is called transition function.*

For two states q and q', q' is a **successor** of q if there is a move vector $\langle j_1, \cdots, j_k \rangle \in D(q)$ such that $q' = \delta(q, j_1, \cdots, j_k)$. Thus, q' is a successor of q if and only if whenever the game is in state q, the players can choose moves so that q' is the next state. A computation of S is an infinite sequence $\lambda = q_0, q_1, q_2, q_3, \cdots$ of states such that for all positions $i \geq 0$, the state q_{i+1} is a successor of the state q_i . A computation starting at state q is a q**-computation.** For a computation λ and a position $i \geq 0$, $\lambda[i]$, $\lambda[0, i]$ and $\lambda[i, \infty]$ denote the i-th state of λ, the finite prefix $q_0, q_1, \cdots, q_i$ of λ, and the infinite suffix $q_i, q_{i+1}, q_{i+2}, \cdots$ of λ respectively.

Considering a CGS $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ with $\Sigma = 1, \cdots, k$, a **strategy** for player $a \in \Sigma$ is a function f_a that maps every nonempty finite state sequence $\lambda \in Q^+$ to a natural number such that if the last state of λ is q, then $f_a(\lambda) \leq d_a(q)$. Given a state $q \in Q$, a set $A \subseteq \{1, \cdots, k\}$ of players and a set $F_a = \{f_a | a \in A\}$ of strategies, one for each player in A, the **outcomes** of F_A from q is the set $out(q, F_A)$ of q-computations that the players in A enforce when they follow the strategy. That is, a computation $\lambda = q_0, q_1, q_2, \cdots$ is in $out(q, F_A)$ if $q_0 = q$ and for all positions $i \geq 0$, there is a move vector $\langle j_1, \cdots, j_k \rangle \in D(q_i)$ such that (1) $f_a = f_a(\lambda\,[0, i])$ for all players $a \in A$ and (2) $\delta(q_i, j_1, \cdots, j_k) = q_{i+1}$.

The formal semantics of ATL can then be expressed. With $S, q \models \phi$ indicating that the state q satisfies the formula ϕ in the structure S, the satisfaction relation $\models$ is defined inductively, for all states q of S, as follows :

- $q \models p$, for propositions $p \in \Pi$, iff $p \in \pi(q)$.
- $q \models \neg\psi$ iff $q \not\models \psi$
- $q \models \psi_1 \vee \psi_2$ iff $q \models \psi_1$ or $q \models \psi_2$
- $q \models \langle\langle A \rangle\rangle \bigcirc \psi$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$, $\lambda\,[1] \models \psi$
- $q \models \langle\langle A \rangle\rangle \Box \psi$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$ and all positions $i \geq 0$, $\lambda\,[i] \models \psi$.
- $q \models \langle\langle A \rangle\rangle \psi_1 \,\mathcal{U}\, \psi_2$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$ there exists a position $i \geq 0$ such that $\lambda\,[i] \models \psi_2$ and for all positions $0 \geq j < i$, $\lambda\,[j] \models \psi_1$

The authors then provide a symbolic model-checking algorithm and show [AHK97, Thm. 5.2] that the model-checking problem for ATL is PTIME-complete, and can be solved in time $\mathcal{O}(m.l)$ for a game structure with m transitions and an ATL formula of length l. They also state that the problem is PTIME-hard even for a fixed formula, and even in the special case of turn-based synchronous game structures. However, the model-checking algorithm and the accompanying results require the CGS to be fully generated, with all the states and transitions created, which can in itself be a hard problem.

The reader interested in a more in-depth presentation and analysis of ATL can refer to [AHK97], [GvD06], [vdHLW06] and [LMO07] for an exhaustive analysis of ATL model-checking complexity.

5.2 Verification Algorithms

The model-checking and result reporting techniques presented in the following sections are based on the original symbolic model-checking algorithm for ATL from [AHK97]. As such, it requires us to build a explicit CGS $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ representing the considered systems. The construction of the various components of this CGS from the different models R_M , Ac_M , I_M and S_M extracted from the system is described in the following pages.

Before entering the CGS construction process in itself, it should be noted that building the explicit CGS requires a fixed (and finite) set of agents, actions and propositions. This does not represent a problem for the actions and the propositions, which are extracted from the models. On the contrary, the agents are dynamic entities that can be created and destroyed during the system evolution through the dedicated operations specified in the system model S_M . To allow the agent set to be treated as a fixed set, it was partitioned into a set of active agents, i.e. agents that have been introduced into the game and are allowed to perform some operations depending on the permissions they have received, and a set of inactive agents, who can only perform the no-operation action (NOP) until they are properly introduced into the game (figure 5.2). The same kind of technique is applied to the other dynamic set: the set of objects present in the system.

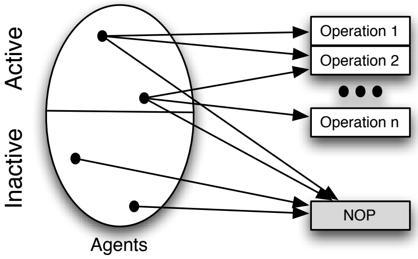


Figure 5.2: *Agent set is partitioned into active and inactive ones.*

The set Π of atomic propositions consists of the collection and instantiation of all the atomic propositions encountered in the various models. It contains all the atomic permissions (the $can(\bullet, \bullet, \bullet)$ and $cannot(\bullet, \bullet, \bullet)$ propositions) generated from the permissions specification as well as all the instances of the observable properties specified in the system model and the access-control model.

The set k of players is constructed from the list of users specified in the initial system model I_M , completed with a set of abstract "inactive" players dedicated to the semantics of the user creation methods found in the system model S_M . As explained above, inactive players are players whose only enabled operation on the system is NOP (no-operation) and have no permissions at all. They can be turned into active players, able to receive permissions and perform operations on objects, through the execution of the dedicated agent creation operation by an already active player.

The move function d is constructed from the conjunction of the access-control model Ac_M and the system model S_M : for a state $q \in Q$ and an agent a, $d_a(q)$ contains the list of actions permitted for agent a. This list is built from the evaluation of the access-control model in state q : every permission granted opens access to the operation semantics associated with the considered object in S_M .

The set of states Q, the transition function δ and the labeling function π are built inductively through a simple state-space exploration (algorithm 4). Starting from the initial state Q_0 built from the initial system state model I_M , states are processed one by one until state-space exhaustion. The algorithm works as follows: a state R is selected from the set of reached but not yet analyzed states then, for every possible move vector applicable to that state, the destination state is added to the res accumulator and its true propositions are set according to the semantics of the state operations defined in the system model S_M . The sets of processed and yet-to-analyze states are then updated appropriately. For simplicity of the algorithm, states are suposed to include their set of true propositions, giving the direct consequence $q_1 = q_2 \Rightarrow \pi(q_1) = \pi(q_2)$.

Algorithm 4 STATE SPACE GENERATION

```
O $\longleftarrow Q_0$
Q $\longleftarrow \emptyset$
$\pi(Q_0) \longleftarrow init$
while $O \neq \emptyset$ do
  $R \longleftarrow pick(O)$
  $res \longleftarrow \emptyset$
  for each $v \in D(q)$ do
    $res \longleftarrow res \bigcup \delta(Q, v)$
    $\pi(\delta(Q, v)) \longleftarrow$ update($\pi(Q)$,effects($Q, v$))
  end for
  $Q \longleftarrow Q \bigcup R$
  $O \longleftarrow (O \bigcup res) \backslash Q$
end while
```

The explicit CGS generated through algorithm 4 can be large. A system containing k agents, o objects, n actions and p different observable predicates can lead to the production of $np = k \times o \times n$ different permissions and $no = p \times k$ different observables, leading to a state space counting $\mathcal{O}(\mathcal{P}(np) \times \mathcal{P}(no))$ states. However, this high upper bound is rarely a problem in real systems models : only the subset of the state-space reachable from the initial state Q_0 is built and checked.

Furthermore, simulating the system does not always require to take into account large numbers of users, objects or properties. On the one hand, usually only a few of them are sufficient disprove the desired requirements. On the other hand, proving a property satisfied requires to analyse all the potential reachable states and can quickly turn impractical without properly bounding the problem. These bounds on the number of objects and subjects in the simulation are provided as parameters to the verification routines.

The main drawback of this choice is that, while negative results (violated properties) can be obtained quite fast, obtaining complete positive results (thus holding properties) can take quite a long time. Even worse, too strict bounds on objects and subjects set can lead to false positives. Inserting the right bounds values requires some experience with the algorithm and sometimes some trial.

5.3 Reporting Verification Results

The next step after the verification of the properties on the generated model is to report the results back to the originals models and, through the weaving process, back into the executable source code.

Depending on the verification results, two distinct situations occur : either none of the properties have been disproved or some of them have failed the verification. The first case is of little interest in the context of this work : if all the desired properties are satisfied, then the source code does not need any corrective measure, so end the reporting process. On the other hand, if some properties are disproved, corrective measures should be proposed to the developer. Simply forwarding the model-checking results directly back to the user does not satisfy our basic requirements stating that the target user should not need to be familiar with formal methods and verification techniques.

The verification results have to be reported back to the user in a easily understandable way. Our target audience being source code developers, we will attempt to provide corrective measure directly to the code (through the weaving process of the acces-control model) every time it is possible.

More formally, a property p from R_M which fails the verification process is a property such that there exists a q_0-computation $\lambda : \exists i \geq 0 : (\forall j < i : \lambda[j] \models p) \wedge (\lambda[i] \not\models p)$, i.e. a property that admits an execution trace running from the initial state (q_0) to a state violating the property ($\lambda[i]$) (figure 5.3). For each property, the set of offending states, and the execution traces leading to them, have to be analyzed and proper corrective measures proposed on the access-control model Ac_M . The remaining of this section will tackle each of these possible outcomes individually, plus another one closely related to the iterative development scheme. Then a practical combined approach will be presented.

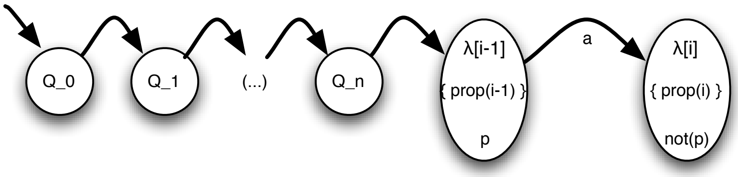


Figure 5.3: *Property violation : execution trace*

5.3.1 Requirements level

The most obvious, and probably the most human, way to circumvent the violation of a requirement p by a model m is to remove or adapt the conflicting property such that the violation disappears (figure 5.4).

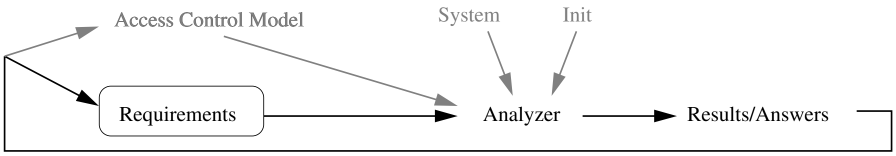


Figure 5.4: *Reporting verification results back to the requirements*

Removing the conflicting property from the set of requirements is a trivial solution to the violation and, in general, should not be admitted. This should never happen unless the considered property was misspecified or does not reflect any more the clients requirements. Removing a property means to miss some of the clients requirements, and thus producing code not conformant to what was intended. However, experience shows that this behavior happens in almost every software project : time and cost constraints often impose to make choices during the development stages. Many software projects deliver incomplete systems which will be patched “later on” to correct the missing or incomplete features.

A more fine-grained approach to this problem is to correct the property to remove the violation. Considering the offending q_0-computation leading to the violation, a straightforward solution is to modify the property p such that it does not apply anymore in offending state $\lambda\left[i\right]$. Based on the structure of the property p (see p. 76), two scenarios are possible : if p is a $localProp$ element (p. 80), thus a property without temporal or agent operator, preventing the violation in state $\lambda\left[i\right]$ can be done by strengthening the preconditions of the property such that it does not apply anymore in the offending state. Modifying p from '$p = domain : localProp$' to '$p' = domain \,\&\, \neg(\pi(\lambda\left[i\right])) : localProp$' will prevent the conflict.

On the other hand, if p is a temporal property (a $timeProp$ element) or an agent specification property (an $agentProp$ element) , there is no straightforward corrective measure to be proposed. Depending on what the signification of the violated property is and confronted to the violation trace, the developer can decide to adopt one or more of the following strategies :

- strengthen the preconditions of the property to prevent the appearance of the violation execution trace.
- enlarge or reduce the coalition of agents responsible of the enforcement of the property (in the case of an $agentProp$ element)
- modify the property to take into account the violating scenario (for instance, modifying a property $always(p)$ with the addition of an alternative to reflect some situation, like in $always(p \vee q)$)
- completely rewrite the property to avoid the violation.
- ...

As a special case, if the property has been violated since the first step of the trace (i.e. if $(\lambda\,[0] = q_0) \not\models p$), it means that the initial state, as defined in the initial system model I_M , conflicts with the considered property. The resolution process remains the same.

Excepted for the case of the simple local requirement, the process of choosing the most adapted alternative for a specific violation cannot be automated : only the author of the offended property can decide which alternatives are the most adapted to the situation. The verification routines can only provide him an execution trace leading to the violation of the property and show him what are the consequences of the changes he makes.

5.3.2 Access-control level

Reporting the results of the verification back to the access-control model (figure 5.5), and also back to the code thanks to the weaving process, requires to identify precisely which access-control rules from Ac_M have allowed the actions taken to put the system into the violating state.

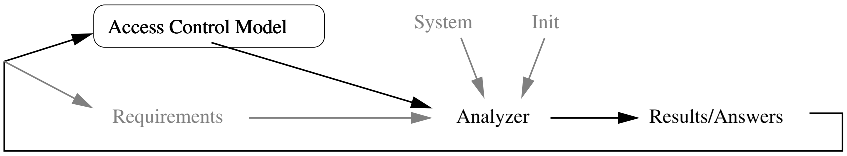


Figure 5.5: *Reporting verification results back to the access-control model.*

The execution trace violating property p, $\lambda : \exists i \geq 0 : (\forall j < i : \lambda\left[j\right] \models p) \wedge (\lambda\left[i\right] \not\models p)$ gives a hint on the operations which might have compromised the property. The set of actions (one for each agent) that triggered the transition from the last state satisfying the property p to the offending state (labelled $\lambda\left[i\right]$) is given by the transition function : $\lambda\left[i\right] = \delta(\lambda\left[i-1\right], j_1, \cdots, j_k)$. Identifying the access-control rules that have allowed the different agents to perform those actions is pretty straightforward :

1. we identify the set of propositions modified in the transition : the set P^+ contains the propositions introduced in the violating state, while the set P^- contains the propositions which have been removed.

$$\begin{cases} P^+ = \pi(\lambda\left[i\right]) \backslash \pi(\lambda\left[i-1\right]) \\ P^- = \pi(\lambda\left[i-1\right]) \backslash \pi(\lambda\left[i\right]) \end{cases}$$

2. we identify the sets of propositions, called V that cause the violation of the property p :

$$\forall s \in \mathcal{P}(P^+ \cup P^-) : \text{ if } (\pi(\lambda\left[i-1\right]) \cup s) \not\models p \text{ then } V \longleftarrow V \cup \{s\}$$

i.e. we search for subsets s of the set of modified propositions, such that the addition of s to the previous (satisfying p) state is sufficient to make it violate p.

3. we restrict these sets of propositions to their most significative elements (in Off):

$$Off = \{o \in V | \forall v \in (V \backslash o) : v \not\subset o\}$$

4. then for each of these sets of significative propositions :

   (a) we identify in the system model S_M the set of actions A from $(j_1, \cdots, j_k)$ that have caused these modifications. $A = \{(a_i, op_i, ob_i) | i = 1, \cdots, n\}$ with (a, op, ob) representing the action for agent a to perform operation op on object ob.

   (b) we identify the access-control rule responsible for the permission to perform each of the offending actions :

$$R = \left\{ r \in Ac_M \ : \llbracket r \rrbracket_{\lambda[i]} \Rightarrow can(a, op, ob) | (a, op, ob) \in A \right\}$$

   Note that the rules in Ac_M do not overlap (due to the adapted union operator $\overset{M}{\bigcup}$).

   (c) then we update each rule in $c : perm \in R$ to prevent it to grant the permission $can(a, op, ob)$ authorizing the action leading to the violation of the considered property.

   - either we prohibit agent a (or a generalization of it) to benefit from the permission granted by the rule r :

$$r = \begin{cases} c \wedge prop_a : \neg perm \\ c \wedge \neg prop_a : perm \end{cases}$$

   - either we prohibit the rule r to apply in state $\lambda\left[i-1\right]$

$$r = \begin{cases} c \wedge \pi(\lambda\left[i-1\right]) : \neg perm \\ c \wedge \neg\pi(\lambda\left[i-1\right]) : perm \end{cases}$$

- or a mix of both.

5. or we generate a fresh access-control rule preventing the violation of $p = (c : a)$:

$$F = (c \wedge \neg(P^{+} \cup P^{-}) : a)$$

The modifications obtained on the model can then be showed either directly to the developer, or be turned into executable code through the weaving process, leaving him with the choice between all proposed alternatives.

5.3.3 Status Quo

This option differs from the previous ones : in the status quo alternative, the developer acknowledges the property violation, but takes no action to solve it. This behavior usually appears in iterative development schemes, where all features of the system, although being specified in the requirements, might not be fully implemented yet when the source code is verified.

The execution trace leading to the violation is simply forwarded to the developer “as-it-is”, as a remainder for the unresolved violation to be taken care of later in the development stages.

5.3.4 Practical approach

In practical situations, the verification routines on the models cannot, by themselves, decide which corrective measure among those proposed best fit the requirements and wish of the developer.

The same holds for the level of abstraction to be used inside the rules. While some developers won’t mind if the number of access-control rules increases as long as each of them stays relatively small and understandable, others will attempt to generalize as much as possible each encountered rule modification in order to maintain the rule set as small as possible.

To fit both of these behaviors, and any in-between, we limit ourselves to the production of a set of possible corrective measures for each violated property. The developer is then free to choose which one to apply, or to implement his own fix (figure 5.6) . Furthermore, the developer is sometimes required to implement the corrective measure by himself, for instance when some properties conflict with the initial system configuration.

Keeping the developer inside the feed-back loops not only permit to decide which modification to apply, but also allows him to keep full control over his source code : the effects of the proposed changes are directly

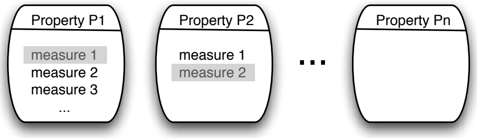


Figure 5.6: *Selection of the most appropriated options among the sets of proposed corrective measures.*

visible on the source code and/or the required properties.

5.3.5 Generalization

The techniques presented above tend to produce explicit rules (both for the requirement model and for the access-control model). These rules can often be simplified through the generalization of some of their elements. Consider for example the following access-control rules:

```
1.  p & q(a)  :  perm_a
2.  p & q(b)  :  perm_b
3.  p & q(c)  :  perm_c
```

It might be tempting to collapse them into a single rule, generalizing the argument appearing in the q predicate from the precondition, which would give the rule : `p & q(X) :` $perm_X$.

However, this process has two major drawbacks : first it enlarge the semantics of the rule to elements that were not part of it. For instance, an element d potentially failing $q(d)$ might receive a permission in the generalized rule while the sequential ones would have forbid it. Secondly, the access-control rules are closely tied to the developer's source code. He might have a very good reason for replicating the rules, for instance, each rule might be accompanied by a comment justifying the existence of the code block behind the rule.

The verification and the result reporting routines have no way to prevent these problems. For this reason, we limit ourselves to the production of generalized rules in addition to the explicit rules created above. They will appear as extra alternatives to choose from to the developer.

5.4 Example

This section provides an illustration of the proposed methodology and result reporting techniques detailed above on the running example introduced page 51. The related models can be found in figures 4.11 to 4.14 (page 71) for the access-control model of the various source class files, figure 4.16 (page 81) for the requirements specification, figure 4.20 (page 84) for the system model and figure 4.18 (page 82) for the initial system configuration.

A simple execution trace like :

```
root -> John = System.user.create()
John -> ob = CPrivateEvent.create()
```

where the administrator, defined in the system initialization model, creates a user named John, who then create a private calendar event (an $CPrivatEvent$ object), is sufficient to break the first of the two properties required on the system. The access-control rules defined on this kind of objects only allow the object's owner to access them, thus preventing the administrator from reaching them.

This behavior breaches the "`S.isAdministator() : allowed(S, Op, Ob)`" property defined in the requirements, stating that an administrator can always perform any operation on any object.

There are two options to solve this property violation : either modify the requirements and avoid the conflicting situation, or modify the access-control rules governing the access to the private object.

- on the requirements level, if we omit the trivial solution of removing the violated rule, the only option is to modify the rule such that it does not apply anymore in the offending situation. In the considered case, it means strengthening the rule preconditions with the generalized negation of one of the properties valid in the offending state :

  ```
  (...)
  root.isAdministrator()
  ob.owner = John
  ob.class = CPrivateEvent
  (...)
  ```

  This leads to the generation of the following modified access-control rules:

  ```
  R1.  S.isAdministator() & ¬ (S.isAdministrator()) :  allowed(S, Op, Ob)
  R2.  S.isAdministator() & ¬ (Ob.owner = John) :  allowed(S, Op, Ob)
  R3.  S.isAdministator() & ¬ (Ob.class = CPrivateEvent) :  allowed(S, Op, Ob)
  ```

  The first rule is clearly useless as its preconditions conflicts each other and can be ignored. On the other hand, rules 2 and 3 propose interesting requirement limitations : rule 2 states that an administrator can access any object but John's ones, while the third one states that the administrator can access any object on the system, but private ones.

- on the access-control level, the rules modification will fail to provide useful elements (the $create()$ operation considered here is a bit special : it is not restricted by a permission). On the other hand, the ad-hoc rule creation rule will produce the two following items :

  ```
  M1.  S.isAdministator() & (Ob.owner = John) :  allowed(S, Op, Ob)
  M2.  S.isAdministator() & (Ob.class = CPrivateEvent) :  allowed(S, Op, Ob)
  ```

  M_1 allows an administrator to access objects owned by John and M_2 allows an administrator to access private objects (i.e. objects from the $CPrivateObject$ class).

At the end of the process, the developer is confronted with the execution trace violating the required property and the 5 corrective options $R1, R2, R3, M1$ and $M2$. Rules $M1$ and $M2$ are proposed either under their access-control model form or directly weaved back into code fragments. As an illustration, figures 5.7 and 5.8 show such code fragments, respectively, for $M1$ and $M2$. The developer is then free to choose which proposition(s) should be accepted (into the requirements model and/or into the code), or to modify them / implement his own solution to resolve the conflict.

```
public class CPrivateEvent extends CEvent {
(...)
        public boolean isAuthorized(CSubject requestor, COperation op ) {
                // M1
                if (requestor.isAdministator())
                        if (this.owner == John)
                            return true;
                return requestor.equals(this.owner);
        }
(...)
}
```

Figure 5.7: *Proposition M1 weaved into the code.*

It is important to note that, as our proposed methodology considers the requirements one at a time, some of the potential corrective measures proposed at one stage can conflict with other previous or following requirements. For instance, in the above example, applying any of the proposed solution will break the second system requirements :

```
# private events cannot be read without owner’s will
class(CPrivateEvent, Ob), X=Ob.owner(), user(Y), X!=Y : alone(X, always(not allowed(Y, Op, Ob)))
```

```
public class CPrivateEvent extends CEvent {
(...)
        public boolean isAuthorized(CSubject requestor, COperation op ) {
                // M2
                if (requestor.isAdministator())
                        return true;
                return requestor.equals(this.owner);
        }
(...)
}
```

Figure 5.8: *Proposition M2 weaved into the code.*

The test "(Ob.class = CPrivateEvent)" was redundant and is omitted to preserve readability.

Which is quite expectable as the two requirements specified on our considered system clearly conflict with each other.

This question of potentially conflicting requirements in not explicitly adressed in this document. Conflicting requirements are more a specification problem than a problem related to an access-control policy implementation.

6
Model Weaving

Contents

This chapter presents the techniques used to weave back the extracted (and possibly modified) access-control model back into the executable code. A first naive but straightforward implementation of the weaving process will be presented and discussed followed by an improved version. Then a set of possible further optimisations and perspectives, improving the quality and readability of the produced code, will be introduced.

6.1 AC Model Weaving

By opposition to the extraction algorithms presented in section 4.2.3.2 (p. 68), which extract an access-control model from the source code, the weaving algorithms presented in this section take an access-control model as main input and produce executable code implementing the semantics of the model (figure 6.1).

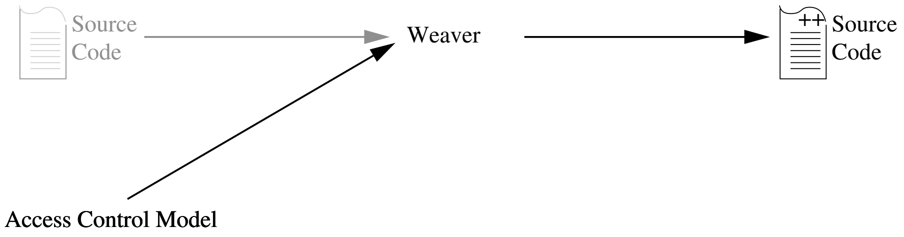


Figure 6.1: *AC Model weaving.*

The term 'weaving' comes from the Aspect Oriented Programming paradigm [KLM+97] and is used here with the same semantics. It denotes the conversion of the access-control model into executable code clauses and their introduction at an appropriate location into the source code. The conversion from the model to executable code and the insertion of this code into the existing application tries to modify the existing code as little as possible, produce human-readable statements and attempt no to modify unnecessary elements.

6.1.1 Naive approach

A naive implementation of the weaving process is, for each $rule$ appearing in each $classModel$ in the model, to produce a code statement implementing the permission (see algorithm 5). Then the generated statements overwrite the possibly existing content of the authorization method in the executable source code.

Algorithm 5 WEAVING(M)

```
for all class $C \in m$ do
  for each rule $r \in C$ do
    if r $\equiv$ (pre : allowed(S,Op,Ob)) then
      RV $\longleftarrow$ true
    else
      {r $\equiv$ pre : not allowed(S,Op,Ob)}
      RV $\longleftarrow$ false
    end if
    produce( “if (pre) then return RV;”)
  end for
end for
```

This algorithm produces executable code implementing the access-control model, however, it makes the code completely unreadable and unmaintainable. Much like what is done in most popular AOP implementations, the resulting source code is only meant to be compiled and executed, not manually edited. This goes against the primary objectives targeted by our approach : the model extraction, its verification and the weaving of the model back into the code should leave the code as human-readable as possible such that the developer keep total control and full understanding over it.

6.1.2 Improved approach

A first obvious improvement to the algorithm shown above is to better use the structures of the rules found in the access-control model. Algorithm 5 considers the rules as independent items, which is most of the time false, especially if those rules are the result of the extraction algorithms presented in section 4.2.3.2 (p. 68). The weaving process being almost the inverse of the extraction process, adapting a reversed version of algorithm 1 gives the more efficient weaving algorithm 6.

[**l. 1-3**] It extracts the access-control rule set R found in each class present in the model m. Z is an accumulator which will store the sequence of all the elements to be weaved into the final source code.

[**l. 4-5**] The rules found in the rule set are analyzed one by one to reverse the effect of algorithm 2 :

- [**l. 6-8**] If the current rule (and it’s successor in the table R) are the result of the disjunctive clause they are merged back together and we skip the next rule.
- [**l. 10-14**] Otherwise we simply reverse the effects of algorithm.

[**l. 17**] The executable code corresponding to the rules found is produced (see below for details).

The $genCode(C, Z)$ generates access-control code for the C class from the rules contained in Z. It groups rules based on common prefixes while preserving their ordering, this prevent meaningless code replication and keeps the output code as readable as possible. Figure 6.2 illustrates this process with a source $Z = (A, B, C : T), (A, B, D : F), (A, E; T), (A, -E, F : F)$, showing, on the left hand part, a rule per rule code generation and, on the right hand side, a common prefix factoring.

Algorithm 6 WEAVING(M) - V2

```
for all class $C \in m$ do
    R $\longleftarrow$ rules(C)
    Z $\longleftarrow \emptyset$
    for $i = 1$ to $\#R$ do
        {R[i] $\equiv$ (class(C, Ob) $\wedge$ Q) $\longrightarrow$ Action}
        if $(i < \#R) \wedge ($ R[i+1] $\equiv$ (class(C, Ob) $\wedge$ Q') $\longrightarrow$ not Action $) \wedge (Q = A \wedge X) \wedge (Q' = A \wedge \neg X))$ then
            Z $\longleftarrow^{+}$ (A, return(X) )
            $i \longleftarrow i + 1$
        else
            if (Action $\equiv$ allowed(requestor, Op, Ob) then
                Z $\longleftarrow^{+}$ (Q, return(true))
            else
                Z $\longleftarrow^{+}$ (Q, return(false))
            end if
        end if
    end for
    genCode(C, Z)
end for
```

```
if (A & B & C) return(true);
if (A & B & D) return(false);
if (A & E) return(true);
if (A & not E & F) return(false);
```

```
if (A)
  if (B)
    if (C) return(true);
    if (D) return(false);
  if (E) return(true);
  else if (F) return(true);
```

Figure 6.2: *generate(C,Z) : unstructured Vs structured output.*

6.1.3 Effects on existing code

Even though the method presented here tries to have as little impact as possible on the existing code, nothing guarantees that cycling through the extraction and weaving stages does not alter presentation of the source code (figure 6.3). The implemented semantics stays constant but the appearance of the code as well as the constructs used can vary, which may greatly disturb the developer. This gives him the feeling that he is losing control over his own code, not recognizing anymore some of the constructs he had built into it.

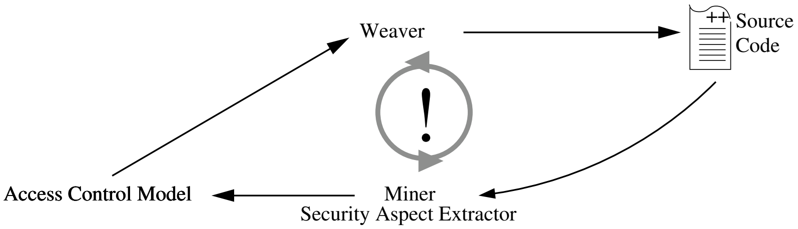


Figure 6.3: *Iterating the extraction and weaving processes.*

Iterating the extraction and weaving processes produces equivalent code instead of leaving the code unchanged.

These unintended variations in the code presentation and the different possible ways to prevent them are the source of the optimization techniques presented in the next section.

6.1.4 Example

As an illustration, weaving back into the code the access-control model extracted from the $CGroupEvent$ class (figure 4.14) gives the source code shown in figure 6.4.

The generated code differs slightly from the original source code (see figure 3.10 for the original source code), but implements the same semantics. The difference lies in the lines 8 and 9 : in the original source code, the two tests are combined inside one single if construct, where our weaving process has split them on two consecutive ones.

While such slight differences will probably not disturb too much the developer when he will try to update his code, their accumulation might slowly obfuscate the code. The next section will present some techniques aimed at removing these unwanted side-effects.

```
public class CGroupEvent extends CEvent {
(...)
   public boolean isAuthorized(CSubject requestor, COperation op ) {
      if (super.isAuthorized(requestor, op)) return(true);

      if (owner.isGroup())
         if (requestor.equals(owner.getOwner())) return true;
         else if (owner.isMember(requestor))
           if (op == COperation.READ) return(true);

         return false;
    }
(...)
}
```

Figure 6.4: *CGroupEvent class : model weaved back to code.*

6.2 Possible optimisations

This section provide two possible further optimizations techniques applicable to the extraction and weaving algorithms presented in the above sections in order to improve their preciseness and limit as much as possible unintended code structure alteration.

6.2.1 Pragmas

The first way to optimize the extraction and weaving processes is to allow them to better cooperate in their respective tasks. Most of the time, the model to be weaved into the code is very close, at least for some parts of it, to the model that was extracted from the code. Letting the extractor efficiently communicate to the weaver what code portions resulted in what model rules allows the weaver to simply recopy the code fragments whose model has not been modified (figure 6.5). This allows to preserve the structure and the formatting of those code elements, reducing the risk of loss of control of the developer over his code.

These pragmas can either be automatically inserted into the code and the model or be manually edited by the developer. For instance, automatic pragma could be used to delimit the code fragment responsible for a specific rule found in the model, or specify how to interpret some complex code statement. Although the latter would break the minimal developer intervention policy our approach try to enforce. This would allow

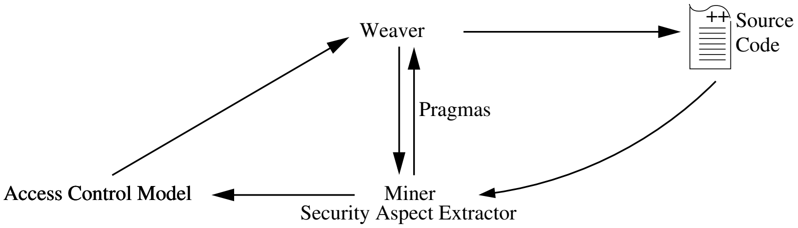


Figure 6.5: *Communication through pragma.*

a fine tuning of the extraction and weaving processes according to the developer desires.

6.2.2 Assisted extraction/weaving

A second optimization technique is to shift from an automatic code extraction and weaving processes to manually assisted ones. When confronted to complex code structures or unexpected behaviors, the tools could interact with the developer to select the correct action to be taken.

These interventions could consist in the replacement of unsupported instructions, like loops, the rewriting of complex nested code structures, ... Not only would this ease the automatic manipulation of the code for the extraction and weaving processes, but it would also help the developer to clarify some potentially obscure parts of his sources and make them conformant to his own coding conventions.

This assisted behavior also plays an important role in the specification of the initial configuration model and of the system model. The tool can collect all the static references done to users and objects into the access-control rules and insert them in the ad-hoc models. The same goes for the permissions and the predicates appearing in mode rules.

6.3 Weaving to another model

A nice potential extension of the weaving process lies in the possibility to weave the access-control model, not directly into the code, but into external and dedicated components and replace the contents of the AC-checking methods in the source code by calls to these external components (figure 6.6).

This would go beyond one of our initial objectives, namely the intent to leave the code as untouched as possible, allowing the developer to keep full control over it at all times. But it would allow the developer to quite easily delegate the AC of his application to dedicated components and technologies. The extracted AC model could then be checked/monitored using other verification methods, for example the JAAS framework [Ora11].

In a way, such a behavior would come close to the aspectization approach found in [TNTN11]. It would extract a particular functional aspect of the source code (here the access-control one) into an aspect in the sense of AOP.

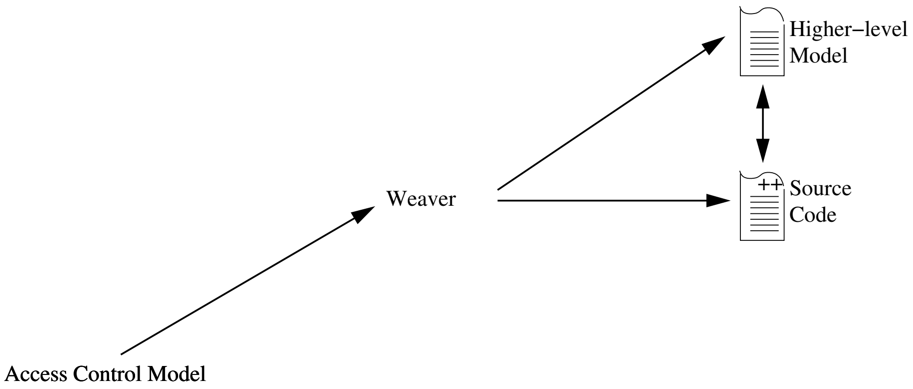


Figure 6.6: *Extraction of the access-control model.*

Such an extension would be a interesting feature and would allow the access-control elements not only to be extracted, but also to be refactored and optimized. It would represent a nice and easy first step towards the use of formal methods to improve the code structure and behavior.

7

Evaluation and Perspectives

Contents

This chapter summarizes and evaluates the achievements of the previous chapters against the objectives defined in chapter 3. It presents the knowledge and feed-back obtained during the application of the proposed methodology to the case study, followed by some general reviews on the presented approach. Paths for potential future improvements are then introduced.

7.1 Case study feed-back

The application of the proposed access-control modeling and verification techniques on a limited "commercial" subsystem of our source case study exacerbated both the pro's and con's our approach showed on the presented running example. This section details the most notable feed-backs received.

7.1.1 Pro's

The presence of a formal methodology requiring a clear and precise specification of the system to be written encouraged the developers to maintain it as up-to-date as possible. Turning the specification from a document "*redacted before implementation and then left aside*" to a close companion of the code gives an opportunity to encourage continuous reflection on the implemented methods. It also limits the risk of appearance of missing or extra features into the application.

Furthermore, the ability to extract an access-control model directly from the code, modulo some approximations caused by an approximate coding convention enforcement, allowed to easily manage and discuss the evolutions of the access-control related sections. As such, it facilitated the co-evolution of the system's specification and the implemented features.

Then, the production of execution traces illustrating the detected violations allowed the developers to (almost) easily spot the problematic elements. Potential corrections directly provided on the source code level were seen as an interesting feature but, unfortunately, were not employed very much. Maintenance teams are trained to detect bugs and fix them directly into the code. When given the execution traces, their first action was often to fix the code directly, without looking at first to the proposed potential corrections. However, for more complex situations or when their direct fixes were not as correct as they first thought, they were inspired from the proposed corrections for their code updates.

Finally, the iterative process of extraction - verification - weaving of the code allowed developers to easily detect unwanted side-effects of some of the code updates. Early detection of inconsistencies between source code at revision n and revision $n + 1$ was possible through the comparison of the access-control models extracted from both revisions. By contrast to code-revision managements systems like CVS or SVN which can only provide the syntactic differences between two versions of the source code, the difference between the two extracted access-control models help pinpoint the semantic differences.

7.1.2 Con's

The first drawback of the approach lies in the time (and effort) required to provide the first interesting results. By opposition to testing, which produces almost immediately tangible scenarios applicable to the system, verification requires the developers to write several models of their system prior to any results. Even though some elements were automatically extracted from the source code, this step seemed overly redundant to them: in their minds, a system able to extract an access-control model from the code should also be able to extract the system model and any other required element. Redacting and maintaining those additional models appeared to be an extra "unproductive" cost counter-balancing the potential advantages of the verification.

Another problematic elements are the format code modifications induced as side-effects by the extraction-weaving loop. These modification tend to obfuscate the code. Even though the code produced is functionally equivalent to the original one, it was very frustrating for some developer to see their code appearance changed. It reduces the readability of some of the code segments and makes their manual correction more complex, even for the original source code author.

Furthermore, the need to strictly respect the coding conventions to allow the access-control model to be extracted was sometimes seen as a huge burden. Although these coding conventions were the ones used by the developers themselves before the introduction of our methodology (with some minor modifications), their habit was to follow the coding conventions as much as possible, not strictly. The respect of the conventions was encouraged everywhere posible, but not sanctioned if small violations were done in limited and justified contexts.

Then, the verification is only as good as the requirements are: specified properties can be checked and, if applicable the code can be corrected to fit them. But a quick bug-tracking analysis showed that most of the problems related to access-control were not caused by a breach in the main properties. Instead they were caused by small "obvious" features that no-one ever took the time to specify. This problem is common to all of the formal methods approaches : it is hard to fully specify all the expected requirement, even more when the authors of the specification are not keen on formal models.

Finally, applying the methodology requires quite a lot of manpower, specially during the first stages where the different models have to be prepared. Applying the verification results to the system was most of the time done without too much problem (albeit the elements cited above), excepted the fact that some of the alternatives proposed by the result reporting stage are not always easily understandable. This is in particular true when the alternatives produced are very close from each other; developers sometimes have difficulties figuring out what was the exact effect of the different propositions. This can cause hesitations on which alternative to choose.

7.1.3 Wrap-up

+	-
Execution traces	Time consuming
Usage for the specification	Code obfuscation
Automated AC extraction	Strict coding conventions
Code version comparison	Limited to requirements
	Choice between corrections

Figure 7.1: *Summary of main pro's and con's obtained on the case study.*

Figure 7.1 summarizes the feed-back received from the application of our proposed methodology to the considered case study. From an usage point of view, the introduction of a semi-automated code verification process inside the development stages allowed the developers to detect some inconsistencies between their implementation and the underlying requirements. However, the need to build, aside from the source code, models describing the system appeared to be a harsh and time-consuming task, specially for developers with no knowledge of formal methods.

Furthermore, due to their background, trainings and past habits, developers often had more confidence in testing than in results obtained through verification methods. At some points, they even checked manually the proposed corrections against the violating execution trace to make sure that what was proposed actually met their requirements.

As a conclusion to the case study, it should be noted that the case study evaluation could not be run till completion. Various reasons, both internal and external, forced the considered application source code to evolve to new technologies in a time-scale incompatible with the one of a research project. Participating developers had to be reaffected to new projects and support inside the organisation was slowly diluted, halting the case study evaluation process.

7.2 General thoughts

From a more general point of view, the proposed approach did reach some of its objectives : the extraction of an access-control model from the code, its verification and the weaving of the results back into the code is indeed possible and useful in practical situations. However, applying the methodology to other software projects still requires custom adaptation. Furthermore, the methodology requires a deep developer implication and the learning of some specification formalisms. The rest of this section details these elements.

The automatic extraction of the access-control model directly from the source code allows the developer to express the access-control rules in a well-mastered syntax : the one of the chosen implementation language. As such, it limits the potential mis-understandings that might appear during the rule writing because of an "exotic" or un-sufficiently mastered formalism. A drawback of this choice lies in the obligation to follow some coding conventions aimed at facilitating the parsing and interpretation of the source code. They limit the code constructs that can be used, and force the developer to structure his code following canvas. This does not cause much trouble when considering a software project from the start, but can be a huge problem if the methodology is to be applied to an existing project : coding conventions might need to be adapted or more strictly enforced, leading to potential modifications in the whole considered source code.

Event though the approach tries to hide away as many of the formal methods aspects as possible, it remains complex and requires the developer to write some of the models by himself. As such, it misses one of the objectives defined in the beginning of the projet : be transparent to the developer. While some of the models can probably be automatically extracted from the source code like the access-control one (for instance the system model), it would require the enforcement of strict coding conventions on a ever larger part of the source code. This would bring the convention-conformant source code ever closer to a formal model specification using the syntax of the considered programming language, which is not the objective. In this aspect, formal methods will always be less intuitive than testing. Testing derives real tangible execution traces based on some kind of specification or code parsing, these are almost physical concepts. On the other hand, verification requires an abstraction of the considered system to be done before producing any useful results.

The same holds for the interpretation of the proposed corrections.The updates to the access-control model can be presented in a developer-friendly way directly on the source code, but the potential property updates still require to learn and understand the selected requirement language. This might not be an easy task, specially when considering temporal properties. There is usually a huge gap in developers minds between the perception of the semantics of temporal operators and their effective semantics. Local or agent properties refer to tangible elements found on the system, where temporal operators refer to the behavior of potential futur executions.

Another limitation of the proposed methodology is that it only verifies that the implemented access-control methods inside the source code are compliant with the expressed requirements. It does not check that these access-control methods are effectively called with the proper arguments before the access are granted or forbidden by the application. It relies entirely on the developer capacity on this point.

The methodology presented in this document differs from the other approaches existing in the literature. Existing approaches tend to focus either on the extraction of an access-control model from the source code then the verification of some properties on the obtained model (see for instance [GSL07]), or they focus on the verification of some properties on a previously given model then its insertion into the code (for instance in Aspect Oriented Programming). Combining both approaches in a single loop allows to focus more on the developer capacities and expectations. Furthermore, keeping the developer inside the decision loop allows us to benefit from his knowledge to choose the right decision when updating rules and/or requirements. This allows to produce results more fit to the developer's needs.

7.3 Adaptability

The work presented in this document is closely tied to the structure and coding conventions encountered in the case study. In particular, the extraction and weaving components are built to take advantage of the code structure and coding conventions available on the considered source code.

However, the other components of the methodology were built to be easily adaptable to new systems and conventions sets. The various models and verification and reporting processes are almost independent from the source code aspect. They only consider access-control rules granting or forbidding accesses, which should be able to model almost all of the access-control situations.

As such, the methodology should be quite easily adaptable to new software systems and/or new source code languages. Only the extraction and weaving components should be adapted to manage the chosen development language, the code structure and the selected coding conventions, all other components can be reused with minor adaptation (see the components inside the grey dotted line in figure 7.2).

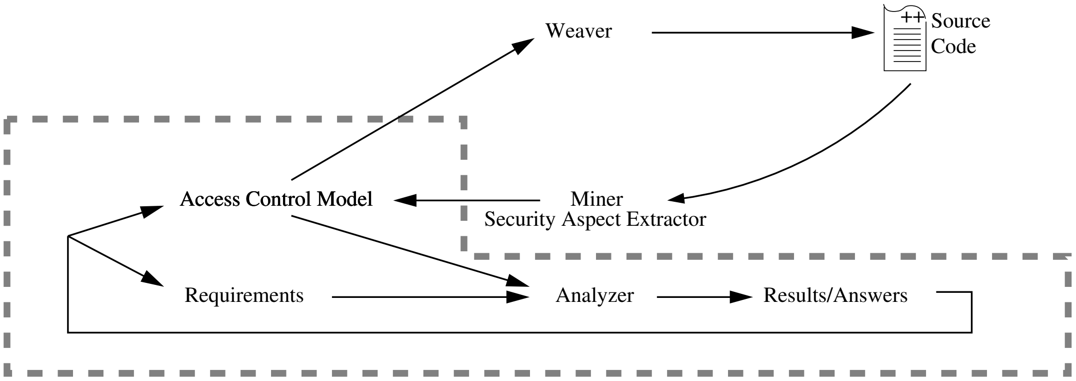


Figure 7.2: *Reusable components.*

7.4 Perspectives

Based on the knowledge accumulated both on the case study and on the later generalization of the approach, some perspectives for enhancement appeared. They can be grouped in two interdependent contexts : enhancements related to the user-model interface and enhancements on internal features.

The interface between the developer and the formal models and core could be greatly improved through the "humanization" of the models. Models used in the proposed approach are closely related to the inputs expected by the model-checking subroutines. Making them more developer-oriented, or more user-friendly, would further incitate the developers to use them. For instance, this could be done through the automatic extraction of the system model (or parts of it) in a very similar way to what is currently done with the access-control model.

Another area of improvements concerns the requirement language : some operators are very disturbing for developers not aware of formal methods. In particular, the temporal operators introduce semantics variations in the property requirements that are not always easy to understand. For instance, think of the subtle difference between these two expressions :

```
true :  alone(John, alwayswill(allowed(John, read, Z))
true :  alwayswill(alone(John, allowed(John, read, Z))
```

The first one states that John has a strategy such that he will always be able to read object "Z", where the second one states that John will always have a strategy to be able to read "Z", i.e. that John will always be able to read "Z". While path or agent operators behave closely to what the intuition expects, understanding the potential consequences of the introduction of the temporal operators requires to understand their inner semantics. Sadly, this process is out of reach for most of our target public. This causes lots of errors and incertitudes when adding temporal properties into the requirements, which might impact the results of the verification process. A possible solution for this problem could be to provide the users with a scenario-based property editor : a tool which would provide them some instantiated trace examples for each property written. This would allow the developer specifying the property to visually verify if the property he intended to specify is equivalent to the property he just wrote.

On the potential internal enhancements side, the very first one should be to tackle the potential code scrambling problem. Cycling through the extraction/verification/weaving processes tend to modify the whole source code structure even though little to no modifications were made on the model. This has the nasty side effect of making the code harder to understand to the developer. Furthermore, he may not understand why a minor modification on the model should completely change the appearance of his code.

The same applies to the proposed correction generalization techniques : the set of currently proposed techniques usually generates only very specific corrective measures. Some of them are extended to very generic ones, but this process often misses its objective of finding the “proper” generalization. A finer-grained analysis should be able to decide which generalization to apply and to what extent. This could be further improved if the method could consider the set of requirements as a whole instead of as a sequence of independent requirements. Considering all the rules together would allow the generalization of the results obtained in several rules, thus making the process easier.

The collection and externalizing the access-control related elements into dedicated components could also be a direct way to improve code quality and security. Limiting the number of places where the security elements are scattered into the code allows the developer to manage more easily the potential changes and reduces the possibility of mistakes.

Finally, it should be possible to relax some of the coding conventions imposed on the source code without adding too much complexity to the extraction, verification and weaving steps. This would allow developers to use more code constructs, removing the sometimes present feeling that only ridiculously simple source code fragments are allowed.

As a final thought on these future perspectives, automatic model extraction from the source code is definitely a way to allow many developers to benefit from the advantages given by the formal methods without requiring them to write all the models themselves. It reduces the entry-cost to verification and limits the number of errors that can be made during the redaction of the models. As such, it allows developers to produce better and more secure code, and code that can provably satisfy the requirements. It can help to show the benefits of a formal method approach without having to bear the full price of conventional formal methodologies. It also helps the developers think twice about the features they have to implement : once for the specification and once for the implementation.

7.4.1 Certification

Even if this was not the primary objective of this work, the question of wether or not our proposed approach can meet the quality standards of a widely accepted industry standard has to be tackled. Convincing the developers of the quality of the produced results is the main objective, but being also able to convince external users is a nice extension. This would allow the considered software to publicly claim its verified quality level.

Attempting to confront this approach with the various evaluation assurance levels requirements defined in the Common Criteria (see section 1.4.1, page 21) is not an easy task. The lower EAL levels are trivially satisfied by the production of a proper and verified specification (in our case an access-control model) representing the code and its verification against the requirements. However, the higher requirements (from EAL5 up to EAL7) require both 1) a formal verification of the conformity of the implementation towards the specification and 2) the proper confidence in the certification authority issuing the verification.

In the context of our methodology, the first criteria is met through the automatic extraction of the model from the code, so could our proposed methodology compete for the highest EAL level. However, the second criteria will almost be impossible to satisfy : as the verification (and more importantly, the correction of the detected problems) is a user-assisted process, the confidence in the produced results will always be limited by the confidence in the craft of the considered user and will hardly be recognized outside of the development team.

It is also hardly possible to hire a well-recognized external reviewer to apply the methodology to the code. Our methodology requires the usage of the coding conventions found into the code as well as a certain user craft into the management of the verification routines (for instance through the proper setting of the upper bounds on objects sets). Both of these are not easily reachable for an external reviewer.

7.4.2 Longer-term perspectives

If we push the reflexion one step further, the place of the developer inside the application development scheme has to be re-thinked. How can we expect a fully secure, flaw-less, application to be created by mistake-prone elements like human beings ? Errors, approximations and misunderstandings are inherent parts of human beings, as is creativity, and cannot be avoided in human enterprises.

Limiting ever more the acting space of developers to prevent the occurrence of problems may not be the proper way to make software engineering evolve on the long-term. In the last century, the physical goods engineering processes have been extensively adapted to allow the execution of tasks by entities designed to avoid errors : robots. This has improved the quality and speed of production of many products, but it removed any proactivity from the process. Robots only do what they were programmed to do; no more, no less. As such they cannot anticipate the future problems and needs or provide innovative solutions.

When applied wisely, automatization eases the development process. For instance, only few people still produce languages parsers by hand nowadays, because automatic tools exist to produce them almost automatically. But we must keep in mind that a software system is not only an assembly of pieces of code; it is much more than the simple concatenation its components. It is an vision of the mind. It is designed and implemented to perform a task and react (more or less) appropriately to any unexpected situation that could occur.

In our opinion, the creativity of developer can be supervised, for instance by guidelines, standards or helper tools, but should never be muted. Firstly because turning developers into mere executants will cause a great deal of resistance among them, opposing any changes. Secondly because the human factor is and should remain an integral part of the software development process. The intervention of the developer is required to translate the wishes of the client into a proper specification; preventing the developer from producing code will simply move the problem from code-verification to specification-verification. Finally, software systems ultimately are tools, used by human beings, for human beings. Even though it sometimes might be desirable, they cannot be much more perfects than their creators.

8

Prototype Description

Contents

This chapters presents an overview of the internals of the prototype designed to support the methodology presented in the previous chapters. A general definition of the implementation objectives will be given. Then some insights on the code and model parsing and weaving techniques used will be shown (items ① and ② in figure 8.1), followed by details on the verification and reporting routines (items marked ③ and ④).

Remarks :

The development of this prototype supporting the presented methodology was done in the context of the motivating example presented in section 3.1 (page 45). As such, it focussed on the operations expected by the audience developers and on step-by-step interaction. The complete automation of the process was not

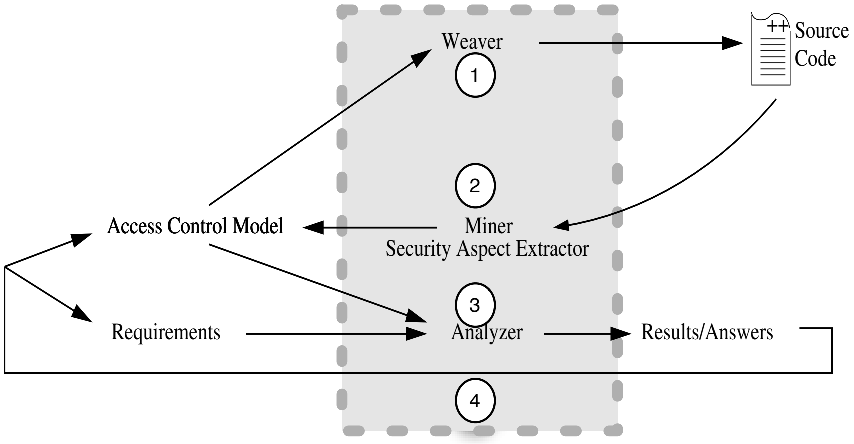


Figure 8.1: *Scope of the implementation.*

the primary objective of this work.

A direct consequence of this is that many routines and screens do produce lots of “debug-like” outputs aimed at allowing a quick and deep understanding of the processes being run. These outputs could be removed from future versions but were a big help to convince partner developers of the interest of the tool and give intuitive insights of what was done. More importantly, these outputs allowed them to understand how and why the given results where produced.

8.1 Usage Scenario

The main use-case governing the development of this prototype can be found in figure 8.2[1]. It contains three different users profiles :

- the “lone” developer, or integrator, labelled *Dev-A*. He’s the one in charge of the verification of the whole considered project and is only interested in the end results of the verification, not on detailled considerations.
- the specification editor, labelled *Spec*. This user is in charge of the redaction of the requirements; his role is to express the properties expected from the software system in the given requirements language.
- the debugger developer, labelled *Dev-B*, is a refinement of the *Dev-A* one. This user profile differs from its predecessor in the interest he has for the deep understanding of the internal flows leading to

[1] In order to keep the figure as readable as possible, only the most important relations between the tasks have been drawn.

the produced results. As such, he may attempt to directly access intermediary datas and execute the inner routines of the model.

These three roles can be assumed by the same user. In small development teams, developers, specification writers and integrators are often the same physical being, changing roles with the evolution of the needs and with the time available for the validation stage.

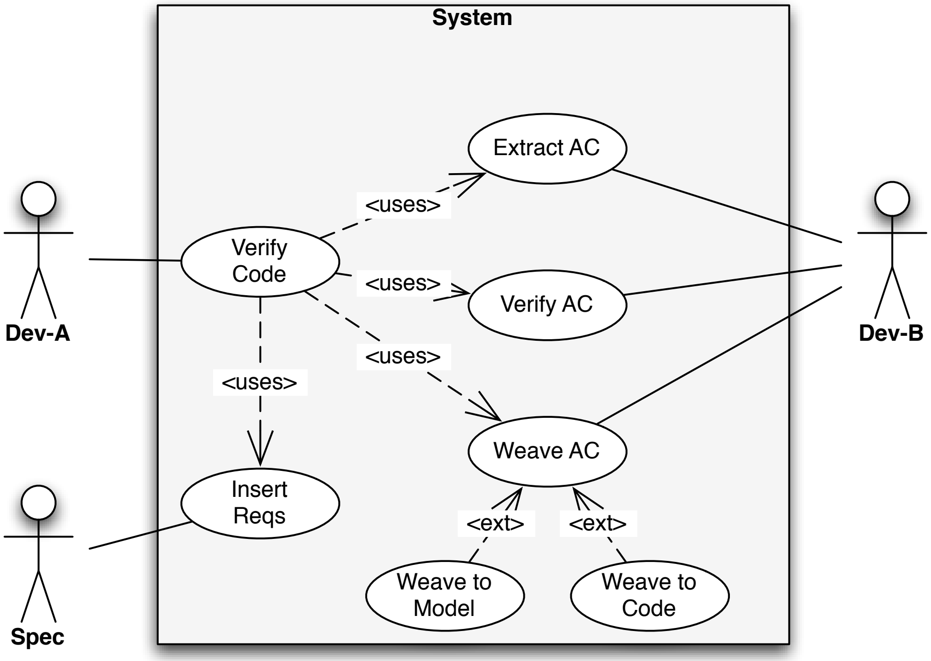


Figure 8.2: *Main use-cases*

The various tasks that these users can perform are almost directly derived from their naming convention :

- **Extract AC** : extracts the access-control related elements from the considered source code. Often takes extensive usage of the coding conventions to ease the process.

- **Verify AC** : confronts the obtained access-control model with the expressed requirements. Produces execution traces in case of detected violation.

- **Weave AC** : translates the (possibly modified) access-control model back into the desired format.

  – **Weave to Model** : produces a access-control component representing the access-control model into the chosen model. If needed, also produce the code stubs interfaces to this component.

  - **Weave to Code** : produces executable code (here Java code) implementing the rules appearing into the access-control model.

- **Insert Reqs** : express the requirements expected from the considered software system into the requirements formalism.

- **Verify Code** : combines the other tasks into a single workflow and proposes an easy to use frontend to the various contained routines, hiding away as much inner details as possible.

Modulo some implementation optimisations, all the "atomic" tasks map almost directly to the structures of the previous chapters. They respectively implement the procedures and produce the deliverables defined in the specified section :

Task	see
Extract AC	section 4.2, p. 59
Verify AC	section 5.2, p. 91
Weave AC	section 6.1, p. 104
Insert Reqs	section 4.3, p. 76

In the following sections, we will focus on the interactions and expectations of these 3 specific profiles in the process of performing the desired tasks.

8.2 Architecture

To optimize the adaptability of the implementation to new source code structures and/or to new formalisms, the general architecture of the prototype was designed to insulate verification and result production routines from the inputs management.

The general architecture of this prototype is detailed in figure 8.3. The core element is interfaced with the source code, requirements and access-control model via adapters components. The role of these adapters is to convert the external formalism (for instance the Java source code language) into an internal general tree-like format. It allows to encompass in these components all the complexity of the parsing and code interpretation routines. For instance, considering the case of the source adapter in the context of our running example, the source code parsing (including the usage of the coding conventions) is completely encapsulated into a single component. Furthermore, an evolution of the external format only requires us to adapt the interface component, keeping the core routines untouched.

All these components can communicate through the core component. Shared-interest elements can be stored in a shared storage area managed by the core routines and can be retrieved every time necessary.

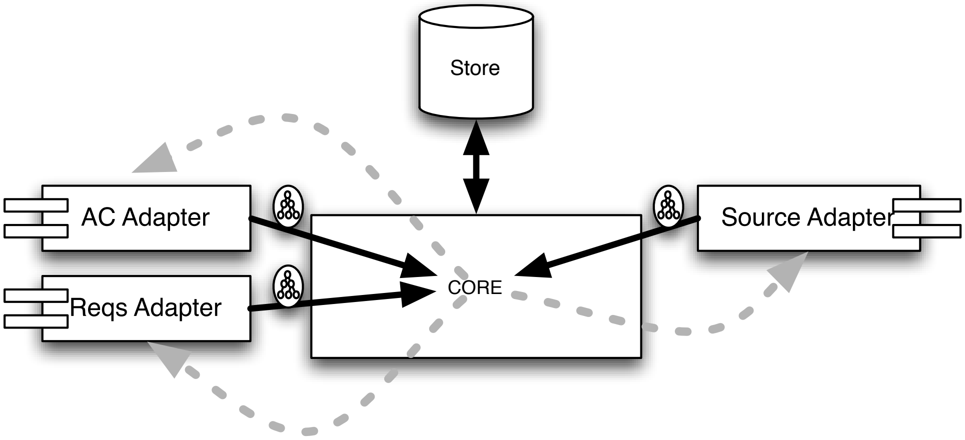


Figure 8.3: *General architecture*

The adapter components are responsible for the management of all the interaction of the core routines and the considered languages, both for the inputs and the outputs. This allows to keep coherent coding conventions between the extraction and the weaving of the elements, while, at the same time, allowing to read from one model and output to another one if required.

The remaining of this section focusses on these components and details their inner structures and design choices.

8.2.1 Adapters

The adapter components are responsible for the management of the interface between an external formalism (e.g. Java source code, requirements language from chapter 4, ...) and the corresponding internal representation. They act both as input and as output filter.

For readability purposes, the input and output processes will be separated in the remaining of this section. However, their implementations are often merged into one single component to be able to benefit from the choices and findings made on previous runs. Think for instance of the source code adapter : when producing code in the same formalism as it was read from, the code weaving process can greatly benefit from the structures and conventions used in the reading stage.

This leads to the adapter class structure found in figure 8.4 : the obtained adapter, here labelled *JavaAdapterC1*, inherits from the selected parser and weaver components to become a bi-directional interface. In this illus-

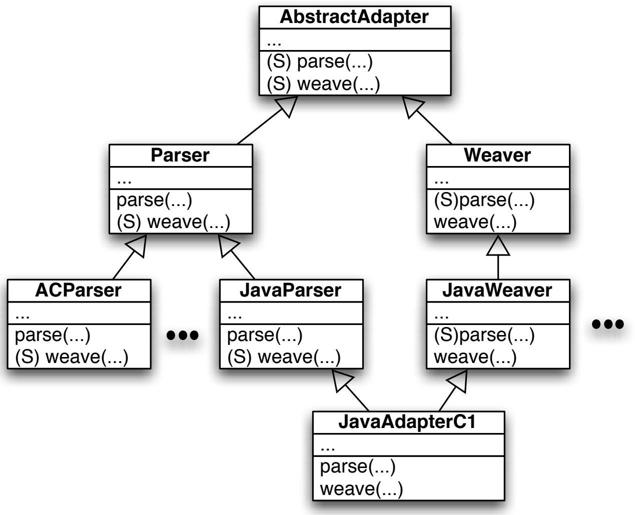


Figure 8.4: *Adapters hierarchy.*

Operations marked with an "(S)" are stubs operations, they do not perform any action. parse() is the input operation and weave() the output one.

tration, the resulting adapter component is dedicated to the parsing and the weaving of java source code with the help of the specific coding convention set labelled $C1$.

8.2.1.1 Parsers

Parsers, or input adapters, are components dedicated to he reading of an external formalism and to its translation into the adapted internal representation. Two kinds of parsers can be distinguished here : "dumb" parser and "extended" parser.

"Dumb" parser, who only translate their input into the proper structure (for instance the world model parser or the requirements parser) are produced almost automatically from a grammar representing the expected language. Parser generation tools are then used to produce the executable source code implementing the adapter component. In our case, the ATNLR parser generation tool [Par11] was used.

On the other hand, "extended" parser, are parsers which need to translate and interpret/analyze their input before producing all their results. An example of this second category is the source code parser: in addition to the standard abstract syntax tree representing its input, it is also required to check that some elements of the input (in our example the access-control related methods) can be properly extracted and converted in the access-control model. These parsers heavily depend on the coding conventions to produce useful results.

The exact set of coding conventions that can be used by the parser greatly vary depending on the considered source code project. In the context of this study, the parser is expected to be able to 1) precisely identify which sections of the code are related to access-control and 2) extract the sequence of operations representing each of the execution paths found in these sections. The upper and lower bounds on the coding conventions set are trivial : with no coding conventions at all the interpretation of the source code is hardly computable and with super-restrictive conventions forbidding every code constructs, the parsing process is trivial.

As a consequence, the allowed conventions set usually is a trade-off between the freedom to write any code constructs into the code and the need to be able to precisely identify what is access-control related and what is not. In addition to that, real software engineering processes also have to take into account the conventions used by the software developers themselves : as members of the same developer team, they usually share (at least partially) the same philosophy about “good” and “bad” code constructs.

8.2.1.2 Weavers

Weavers, or output adapters, are the exact opposite of parsers : they translate the internal representation of an element into the chosen external formalism. They, however, work pretty in the same way, with the addition that their source formalism is completely non-ambiguous. The weaving process usually consist in the translation of the source formalism to the target formalism and in the insertion of the produced elements at the right place in target code.

When weaving to a model different from the initial source code (see section 6.3, p. 109 for details), the weaver can also generate the method stubs to redirect the source code calls to this external model. In this way, the code functionality is maintained.

8.2.1.3 Joint effort

Both kinds of adapters have the ability to store some parsing state informations designed to help any future parsing or weaving to the considered formalism. This local storage can be done locally to the adapter (for instance, to help weaving back the extracted code) or globally in the code storage area (for informations that could be useful for other components).

The most obvious example of such information transfert between an input and an output adapter lies in the Java source code adapter developer in the context of the case study presented in chapter 3. In order to limit as much as possible the code structure changes between the successive iterations of the approach, the access-control rules appearing into the model are linked with the code structures from which they were created (i.e. when they are parsed). In this way, the weaver can reproduce the exact same code blocs each time it encounters an access-control rules untouched by the verification and correction routines, limiting the appearance of the code obfuscation side-effect.

8.2.2 Core Routines

The core component englobes two main elements : shared utilitarian routines and components (including the GUI and user interactions management routines) and the verification / result reporting subsystem. In the following, we will only adress the verification/result reporting aspect.

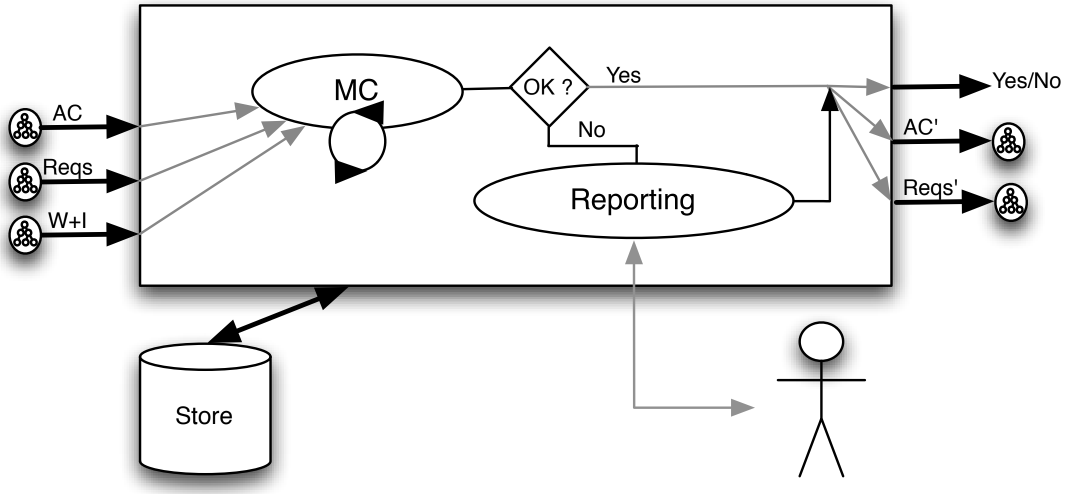


Figure 8.5: *Core Internals.*

The general workflow of the verification stage is presented in figure 8.5. It basically consists in a human-assisted process taking the various models of the considered system as inputs and, if applicable, producing possibly modified versions of the access-control and requirements models as outputs.

It should be noted that both of these processes are human-assisted ones. The interaction of the developer is required to fix the upper bounds on the model-checking state space generation and, more importantly, to guide the reporting process and select the most adapted correction propositions.

8.2.2.1 Verification

The verification stage, labelled MC (model-checking) in the figure 8.5, verifies whether or not the access-control model and world/system models given as input satisfy the expressed requirements.

Requirements are verified one at a time, in a iterative approach. The model-checking algorithm used is based on a bounded model-checking approach : upper bounds are placed on objects and subjects sets cardinality, then the reachable state-space is built and checked using an adapted version of the verification algorithm found in [AHK97].

This approach is far from being optimal and produces huge state spaces, but it allows to almost intuitively explicit the behavior of the verification back to the developers. Thereby counterbalancing its weaknesses.

8.2.2.2 Reporting

As described in section 5.3 (p. 93), the result reporting process only produce correction proposals. When a requirement violation is detected in the model-checking stage, the error trace leading to the violating state and the potential property/requirement corrections are presented to the developer. His role is then to choose the most appropriate one from the list or to implement his own one.

Selecting the most suitable solution quickly becomes a problem when the number of alternatives rises : the set of potential solutions becomes hardly manageable by the developer. Furthermore, as the differences between the proposed corrections tightens, the helping developer becomes rapidly confused between the alternatives. Attempts to limit this problem, for instance by sorting the list based on a similarity factor with the previously selected solutions, have been made, but none was satisfactory.

Up to now, no satisfying human-computer interface have been found to support this process : the proper visualisation of the violating trace and the modeling of the exact consequences of a proposed rule/requirement update still are open questions, specially when the target audience is not familiar with formal methods.

9
Conclusion

In this document, we addressed the problematic of the co-evolution of the requirements specification and the source code implementing them. Focussing on the access-control aspect of a software system, our objective was to provide the developers with an almost automatic way to propagate any modification made on the source code (or the access-control specification) to the access-control model (resp. the source code). We aimed at making the iterative code development scheme more secure by automatizing the detection and resolution of the violations of the requirements introduced through code features evolution.

In particular, we approached this topic in the context of small development teams, with little to no understanding of standard formal verification techniques. We attempted to provide them with a methodology that does not disturb their current development practices but provides them useful information on the potential inconsistencies between their requirements specification and their implemented features.

We designed a language to model the access-control rules implemented into the executable code. This language is able to model almost any access-control primitives, including DAC, MAC, RBAC and XACML and was built with the idea of being usable directly by the developer if needed. As such it only uses simple syntax constructs and its semantics was kept as intuitive as possible.

An automatic access-control model extraction procedure was implemented to extract, from the executable source code, the elements required to model the access-control behavior of the considered application. This procedure takes advantage of the coding conventions used by the developer inside his source code as well as some simplification hypothesis. It produces an access-control model that is at the same time human-readable and understandable and complete enough to faithfully model the implemented decision rules.

Then a language adapted to specify any access-control related requirements was designed. It takes into account both simple permissions and complex decision rules involving coalition of users (both allies and adversaries) and temporal operators. Again, this language was built to be editable by the developer; it was kept as simple as possible without losing too much expressivity.

A procedure to transform the access-control model into executable code able to fit inside the considered application was given. This procedure tries to be as un-damaging as possible, it attempts not to modify any items in the source code that are not affected by the model updates. However, this is not always possible and, sometimes, the produced code, although being functionally equivalent to the original code, is generated using a different structure. Some potential optimizations of the proposed procedure aiming at limiting this problem, like the pragma, have also been discussed.

Then, using standard model-checking algorithms, we verified the extracted access-control model against the expressed requirements. If inconsistencies are found, we propose an execution trace leading to the property violation along with corrective measures if applicable. These measures can apply either on the access-control model, or on the requirement specification. They can be either ad-hoc corrections, forbidding only the violating scenario, or generalized corrections, attempting to "guess" what missing property is causing the inconsistency. Modification on the access-control model can be proposed either on the access-control model language or directly into the executable code. This process is human-assisted : corrective measures are proposed to the developer, his role is to choose which one to apply. This allows to keep the developer in total control over his code : he is the one to decide which correction to implement and how to implement it.

After that, the main strengths and weaknesses of the approach have been discussed and general possibilities for improvements have been proposed. In particular, some lessons learned from the application of the methodology to a real software system, with the intervention of the original developers showed the limitations of the approach, mostly the change of mind required to move from a "code then test" to a "code then verify" philosophy. Some habits are very hard to go against and it may take time before the benefits of formal methods overcome the inherent costs induced on the software project.

Some questions about the verification result reporting have been left open. In particular, the representation of the semantics of the temporal logic operator still cause trouble in the mind of developers not used to formal methods. Building textual or graphic representations of time-dependent elements is a challenge with a broader impact than the field of computer science.

Another open research topic lies in the tight dependency between the quality of the access-control extraction from the source code and the coding conventions required on the source code. Lots of improvements in the code parsing and understanding are possible. Going along with this topic is the problem of the code structure delimiting and the code scrambling prevention. Better code semantics extraction algorithm could allow to prevent the unwanted side-effects of the weaving process obfuscating correct code.

Finally, this approach still heavily depends on the quality of the developer's code. We attempt to verify that the specified requirements do match with the access-control model implemented into the code, but we do not check if the access-control sections of the source code are properly called throughout the application. Enlarging the scope of the extraction and analysis could permit to verify these calls, but this remains an open question in this document.

As a conclusion to this work, building secure code is and will remain a difficult and open question. Currently, the only way to guarantee a 100% secure code is to generate it from scratch using adapted methods (think for instance of the B method) or to use sometimes very complex formal proofs. These approaches go against the general trend in current software development which is based on rapid prototyping and more or less extensive software testing. The presented method, as every method based on formal methods, provides models and tools to assist the developers in their harsh task of building good software. But by themselves, tools are useless; only when used by the proper artisan can they show their benefits and produce masterpieces.

As secure code is the founding stone of secure software, any step toward more secure software is a step in the right direction; as no method can satisfy all needs, this methodology only is an attempt to bring interested developers closer to secure software.

References

[Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Pres, 1996.

[AHK97] Rajeev Alur, Tomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. In IEEE Computer Society Press, editor, *Proceedings of the 38th Annual Symposium on Foundation of Computer Science (FOCS)*, pages 100–109, 1997.

[And01] Ross Anderson. *Security Engineering*. Wiley, first edition edition, 2001.

[Asp] AspectJ. Aspectj. http://www.eclipse.org/aspectj/.

[CCR11] CCRA. Common criteria for information technology security evaluation, version 3.1, 2011.

[CDH^{+}00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. *Proceedings of the 22nd international conference on Software engineering*, 1:439–448, 2000.

[Cle11] Clearsy System Engineering. B-method. http://www.bmethod.com/, 2011.

[Cou08] P. Cousot. Abstract interpretation. http://www.di.ens.fr/ cousot/AI/, 2008.

[CVC99] Sergio Vale Aguiar Campos, Srgio Vale, and Aguiar Campos. Symbolic model checking in practice, 1999.

[DHM09] Frédéric Dadeau, Amal Haddad, and Thierry Moutet. Test fonctionnel de conformité vis-à-vis d'une politique de contôle d'accès. *Technique et Science Informatiques (TSI)*, 28:533–563, 04 2009.

[Dij76] Edsger W. Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, N.J. :, 1976.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165 –1178, july 2008.

[EM09] EU and Prof. Fabio Massacci. Security engineering for lifelong evolvable systems, 2009.

[FK92] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[FKC03] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, Inc, 2003. ISBN:1-58053-370-1.

[Fou11] FSF Free Software Foundation. Gnu coding standards, August 2011.

[FSG$^+$01] David Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, August 2001.

[GRS04] Dimitar P. Guelev, Mark Ryan, and Pierre Yves Schobbens. Model-checking access control policies. *Proceedings of FCS'04*, 1:23–40, 2004.

[GSL07] R. Groz, M. Shahbaz, and K. Li. Une approche incrémentale de test par extraction de modèles. In *AFADL'07 (Approches Formelles dans l'Assistance au Développement de Logiciels, 10ème anniversaire)*, Namur, June 2007.

[GvD06] Valentin Goranko and Govert van Drimmelen. Complete axiomatization and decidability of alternating-time temporal logic. *Theoretical Computer Science*, 353(1-3):93 – 117, 2006.

[Had05] Amal Haddad. Modélisation et vérification de politiques de sécurité. Rapport de master 2 recherche (système d'information), Université Joseph Fourier, 2005.

[HFK06] Vincent C. Hu, David F. Ferraiolo, and D. Rick Kuhn. Assessment of access control systems. Interagency report 7316, NIST, September 2006.

[Hol01] Gerard J. Holzmann. Economics of software verification. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 80–89, New York, NY, USA, 2001. ACM.

[HP98] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder, 1998.

[ini11] ODRL initiative. Open digital rights language. http://odrl.net/, 2011.

[JSP$^+$11a] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, pages 41–55, 2011.

[JSP11b] Bart Jacobs, Jan Smans, and Frank Piessens. Verifast website. http://people.cs.kuleuven.be/ bart.jacobs/verifast/, 2011.

[JUn10] JUnit.org. Junit. http://www.junit.org/, 2010.

[Jur05] Jan Jurjens. *Secure Systems Development with UML.* Springer Academic Publishers, 2005.

[Jü02] Jan Jürjens. Umlsec: Extending uml for secure systems development. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *«UML» 2002 — The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin / Heidelberg, 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwini. Aspect-oriented programming. In Springer-Verlag, editor, *proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland*, volume LNCS 1241, June 1997.

[KS06] Feisal Keblawi and Dick Sullivan. Applying the common criteria in systems engineering. *IEEE Security and Privacy*, 4:50–55, 2006.

[LBB05] Ninghui Li, Ji-Won Byun, and Elisa Bertino. A critique of the ansi standard on role-based access control. empty, 2005.

[Liv06] Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, 2006.

[LMO07] Francois Laroussinie, Nicolas Markey, and Ghassan Oreiby. On the expressiveness and complexity of ATL. In *Proceedings of the 10th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'07), Lecture Notes in Computer Science*. Springer, 2007.

[LTMPB08] Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry. Test-driven assessment of access control in legacy applications. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 238 –247, Washington, DC, USA, april 2008. IEEE Computer Society.

[Mea] Ben Meadowcroft. Why systems fail. http://www.benmeadowcroft.com/reports/systemfailure/.

[MFBLT08] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. A model-based framework for security policy specification, deployment and testing. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Volter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 537–552. Springer Berlin / Heidelberg, 2008.

[MJH+10] Lionel Montrieux, Jan Jürjens, Charles B. Haley, Yijun Yu, Pierre-Yves Schobbens, and Hubert Toussaint. Tool support for code generation from a umlsec property. In *ASE 2010, 25th*

IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, pages 357–358, 2010.

[MLTB09] T. Mouelhi, Y. Le Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing. *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pages 171 –180, april 2009.

[Mon09] Lionel Montrieux. Implementation of access control using aspect-oriented programming. Master's thesis, FUNDP, 2009.

[Mor07] C. Morisset. *Sémantique des systèmes de contrôle d'accès*. PhD thesis, Université Pierre et Marie Curie - Paris 6, 2007.

[NAS] NASA. Java pathfinder. http://babelfish.arc.nasa.gov/trac/jpf.

[NIS06] NIST. Role based access control, 2006. National Institute of Standard and Technology.

[OAS05] OASIS. extensible access control markup language (xacml) version 2.0, February 2005.

[OMG09] OMG. Unified modeling language superstructure version 2.2., February 2009. The Object Management Group.

[Ora11] Oracle. Java authentication and authorization service, 2011.

[Par11] Terence Parr. ANTLR : Another tool for language recognition, 2011. University of San Francisco.

[Pet] Bret Pettichord. Resources for professional software testers. http://www.io.com/ wazmo/qa/.

[Pfl01] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[Pos82] J. Postel. RFC 821: Simple mail transfer protocol, August 1982.

[Ric53] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[Roy70] Walker W. Royce. Managing the development of large software systems: concepts and techniques. *Proc. IEEE WESTCON, Los Angeles*, pages 1–9, August 1970. Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, March 1987, pp. 328–338.

[Sew11] Julian Seward. Valgrind. http://valgrind.org/, 2011.

[SFK00] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The nist model for role-based access control: towards a unified standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, RBAC '00, pages 47–63, New York, NY, USA, 2000. ACM.

[SSS94] Ravi Sandhu, Ravi S. S, and Pierangela Samarati. Access control: Principles and practice. *IEEE Communications*, 32(9):40–48, 1994.

[TKS05] Artem Tishkov, Igor Kotenko, and Ekaterina Sidelnikova. Security checker architecture for policy-based security management. In Vladimir Gorodetsky, Igor Kotenko, and Victor Skormin, editors, *Computer Network Security*, volume 3685 of *Lecture Notes in Computer Science*, pages 460–465. Springer Berlin / Heidelberg, 2005.

[TNTN11] R. Toledo, A. Nunez, E. Tanter, and J. Noye. Aspectizing java access control. *Software Engineering, IEEE Transactions on*, PP(99):1, 2011.

[Tou06] Hubert Toussaint. Formalisation des politiques de contrôles d'accès. DEA thesis, University of Namur, 2006.

[US 85] US Department of Defense. Department of defense trusted computer system evaluation criteria, december 1985. DOD 5200.28-STD.

[US 88] US Department of Defense. Dod standard 2167-a : Defense systems software development, February 29 1988. On December 5th, 1994 it was superseded by MIL-STD-498, which merged DOD-STD-2167A, DOD-STD-7935A, and DOD-STD-2168 into a single document,.

[vdHLW06] Wiebe van der Hoek, Alessio Lomuscio, and Michael Wooldridge. On the complexity of practical atl model checking. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 201–208, New York, NY, USA, 2006. ACM.

[WWZX09] Jun Wu, Chongjun Wang, Lei Zhang, and Junyuan Xie. Coalitional planning in game-like domains via atl model checking. In *Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '09, pages 645–652, Washington, DC, USA, 2009. IEEE Computer Society.

[YTB07] Le Traon Yves, Mouelhi Tejeddine, and Baudry Benoit. Testing security policies : going beyond functional testing. In *ISSRE'07 : The 18th IEEE International Symposium on Software Reliability Engineering*, 2007. Trollhätan, Sweden.