Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria

LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

ISAAC – Integrated System Application and Authorization Co-design

Thesis by

Diego Martinoia

Student ID

783081

Advisor: **Prof. Matteo Pradella** (Politecnico di Milano)

Co-advisor: **Prof. Riccardo Sisto** (Politecnico di Torino)

Co-advisor: **Prof. Lenore D. Zuck** (University of Illinois at Chicago)

A.A. 2012 – 2013

At the end of an era, it is hard to look back and identify all the peoples who helped along the way. My family, my friends, my colleagues and my companions of venture: their effects on me were often so interlaced that it is almost impossible to single out the individual sources of inspiration and motivation. And it is not just the peoples: my luggage is full of memories of places, tastes, sights and ideas[1] that shaped me up to this day.

Everything and everyone I have met on my path, and also a few I have never met, left something to me. I am really bad at remembering names, so I will not list anyone: if you reading this are one of my muses, you will know. And if you think you are not, believe me, you still are.

DM

"Homo sum,
humani nihil
a me alienum puto"

[1] Smells are missing from the list on purpose, as my nose is really bad at perceiving them.

Abstract

The use of formal methods for system design and system property verification is a topic of great interest in those application area where strong security and reliability guarantees are required. Their goal is to proof mathematically that if the system effectively reflects the model used, then the system will possess specific properties of interest.

These guarantees are often based on the assumption that, in general, the inputs offered by the users of the system are reliable and legitimate. In other words, during the design phase of the application part of the system, the risk related to malevolent behaviors of the users is often ignored. For these reason, the application part of the system is usually backed up by an authorization part in charge of the access control aspects, i.e. determine and regulate who is allowed to do what within the system.

Unfortunately, the commonly followed practice is to overlook the aspects related to authorization security until the end of the design of the application part of the system. This introduces the risk that integrating the two subsystems may be hard, inefficient or even dangerous.

Another aspect of the topic is the fact that, often, the formal design of systems is developed in a top-down fashion: starting from a very abstract model of the system, the model gets refined step-by-step, adding an increasing number of details, until a sufficient degree of concreteness is reached.

Albeit the literature offers a number of approaches to both the problems of the management of access control and the verification of system properties during the refinement steps, no method was found capable of offering an efficient solution to both problems simultaneously.

This thesis introduces ISAAC, a theoretical framework for the integrated co-design of the application and authorization parts of a system. ISAAC allows to model a system at any level of abstraction, designing simultaneously the components related to the application and authorization parts, to help verify the absence of circular dependencies among the system interfaces, and to formally verify that the refinements of the model of the system do not introduce inconsistency.

Additionally, a case study modeling the functioning of an hospital as a system is presented. The case study focuses on the subsystem related to the management of personal and medical data of the patients, and verifies that the introduction of the possibility for inter-ward consulting is implementable in a correct way, i.e. without invalidating the expected behavior of the system.

In conclusion, the original contributions of this work are:

- Offering a theoretical methodology for system design capable to include the authorization aspects of access control since the beginning of the design phase of the application part of the system.
- Offering a theoretical methodology for the formal verification of the consistency of the refinement steps of a model a system.
- Testing ISAAC in a real-world case study, related to the modeling of a subsystem of an hospital system.

Ampio estratto

L'utilizzo di metodi formali per la progettazione di sistemi e verifica delle loro proprietà è un tema di grande interesse in quegli ambiti applicativi dove sono richieste forti garanzie di sicurezza ed affidabilità. Il loro scopo è quello di dimostrare matematicamente che, qualora il sistema rispecchi il modello utilizzato, esso possiederà determinate proprietà di interesse.

Purtroppo però queste garanzie sono legate al funzionamento del sistema supponendo che, in generale, gli ingressi ricevuti dai suoi utenti siano tutti legittimi ed affidabili. In altre parole, solitamente, non viene considerato, durante la progettazione della parte applicativa del sistema, il rischio che gli utenti abbiano comportamenti illegittimi. Per questo motivo, alla parte applicativa di un sistema viene solitamente affiancata una parte di controllo degli accessi, il cui scopo è di determinare e regolare chi può fare cosa all'interno del sistema.

Sfortunamente, la pratica spesso seguita è di trascurare gli aspetti di sicurezza, intesa riguardo al comportamento degli utenti e non alle garanzie contro gli incidenti imprevisti, fino a valle della progettazione della parte applicativa del sistema, correndo così il rischio che l'inserimento del sottosistema dedicato al controllo degli accessi sia difficile, inefficace o addirittura dannoso.

In aggiunta a questo, spesso la progettazione formale dei sistemi avviene per gradi e raffinamenti successivi: partendo da un modello molto astratto del sistema, si procede via via aggiungendo un numero sempre maggiore di dettagli, fino ad arrivare a un livello di concretezza sufficiente per lo scopo prefissato.

Sebbene esistano, in letteratura, molti approcci sia al problema della gestione del controllo degli accessi che alla verifica delle proprietà del sistema attraverso i suoi raffinamenti modellistici, non è stata trovata

nessuna metodologia in grado di offrire simultaneamente una soluzione efficace ad entrambi gli aspetti.

In questa tesi viene presentata ISAAC, una metodologia teorica di approccio alla progettazione integrata delle parti di applicazione e controllo degli accessi di un sistema. ISAAC consente di modellare rigorosamente un sistema ad un qualsiasi livello di astrazione, progettando congiuntamente sia le componenti legate agli scopi applicativi che quelle legate alla sicurezza, verificare l'assenza di dipendenze circolari tra le interfacce del sistema, e verificare formalmente che eventuali raffinamenti del modello del sistema siano tali da non introdurre inconsistenze.

Viene inoltre presentato un caso di studio che modelizza il funzionamento di una parte di una struttura ospedaliera, in particolare quella legata alla gestione dei dati personali e delle terapie dei pazienti, e che verifica che l'introduzione di consulenze mediche tra i vari reparti è realizzabile in modo corretto, cioè senza invalidare il comportamento previsto dal modello di base che non prevede consulenze.

In sintesi, i contributi originali sono:

- Fornire una metodologia teorica per la progettazione di sistemi che includa gli aspetti di sicurezza del controllo degli accessi fin dall'origine della progettazione della parte applicativa del sistema.
- Fornire una metodologia teorica per la verifica formale della consistenza dei passi di raffinamento del modello del sistema.
- Testare ISAAC in un caso d'uso del mondo reale, in particolare nella modellizzazione di un sotto-sistema di una struttura ospedaliera.

Dopo l'introduzione, nel Capitolo 2 vegonono illustrati il contesto dei metodi formali, le idee che hanno portato alla realizzazione di ISAAC e lo stato dell'arte per quanto riguarda gli approcci formali alla progettazione ed al raffinamento dei modelli di sistema.

Nel Capitolo 3, viene introdotta la struttura di un modello di sistema all'interno di ISAAC, descrivendone in dettaglio i moduli che lo compongono. Viene successivamente mostrato come sia possibile verificare, in certi casi, l'assenza di dipendenze circolari tra le interfacce del sistema. Il tutto viene illustrato utilizzando un esempio di sistema semplificato di gestione di file di sistema ispirato al modello Unix.

Il Capitolo 4 è dedicato alla specifica di un passo di raffinamento all'interno di ISAAC, alla definizione di correttezza del raffinamento e alla descrizione di come verificarla. Viene ripreso, come esempio esplicativo, il sistema di gestione di file di sistema introdotto nel capitolo precedente.

Il caso d'uso è esposto nel Capitolo 5. Viene mostrato come sia possibile usare ISAAC in scenari reali per verificare, con relativa semplicità, la correttezza di un raffinamento di sistema, nel caso specifico inteso come un'aggiunta di funzionalità.

Il tutto viene infine seguito dal Capitolo 6, contenente le conclusioni e le possibili direzioni di sviluppo futuro.

Contents

Introduction 1

Formal methods for system design have been around for a long time. Formal methods are mathematical techniques for the specification, design and verification of systems. Their main goal is to analyze the design of a system, not necessarily a computer system, and to verify in a mathematically rigorous way whether the proposed solution is correct, before starting its implementation. It is commonly recognized that the cost to fix an error in a system increases by orders of magnitude according to how late the bug is detected in the system life-cycle, but a thorough verification of a design is a long and expensive activity, which is often overlooked to give precedence to other factors, such as time-to-market. The usual compromise adopted in the industry is to use testing techniques, such as model checking or unit testing, instead of formal verification, to detect flaws in the design. The main advantage of these techniques is that they can be performed automatically by the machine, and therefore executed in a reasonably fast time. Unfortunately, *"program testing can be used to show the presence of bugs, but never to show their absence!"*[1] i.e. no matter how vast the coverage of a testing system is, it cannot guaran-

[1] E. W. Dijkstra, 1970

tee the correctness of the solution in exam. Additionally, when talking about computer systems, the digital nature of the technology makes it so that the interpolation of results, commonly accepted for analog systems, is meaningless: if a bridge can withstand 100 Mg of load, it is reasonable to assume it can also withstand lighter loads; if an input parser performs correctly on a 100 character long string, there is no guarantee it will work also with a shorter one, or even with an empty or "null" one.

In general, formal verification is used almost only in the design of critical systems, either where human lives or huge financial capitals are at stake during the implementation and execution phases, such as in the aerospace sector, in chip design or in the automotive and transportation sector. In most of the application cases, it is considered important to verify that the system is behaving correctly *when operated correctly from its users*: while it is normally the case that safety constraints consider also incorrect or dangerous inputs, a stronger emphasis is put on the correct elaboration of the inputs, rather than on whether the users are doing the right thing.

Nonetheless, one of the critical aspects of any system is access control, i.e. who is authorized to do what. While it is common to associate access control with information management systems, access control is actually a core issue in any kind of system, also non computer-based ones. Over the course of the years, a number of access control systems have been formally described and tested, such as Access Matrices, Role-Based Access Control or Bell-La Padula Access Control and their variants. These examples are application agnostic, in the sense that they can easily be adapted to fit in any kind of application that requires some sort of access control management. While their structure has been proved to be correct and safe when left by themselves, it is easy to imagine that plugging one access control systems in an existing application "in

the wrong way" could lead to bugs, information leaks or security vulnerabilities.

The goal of this work is to try and propose a novel formal method framework for the co-design of applications and their access control systems. This work focuses on both the specification of a joint model of the whole system, composed by the application and access control elements, at a given level of abstraction, and the verification of the correctness of a single refinement step from a more abstract to a more concrete model. At the single level of abstraction, the framework describes the system as a combination of mutable and immutable elements, usually represented by mathematical entities such as sets and higher order functions, while the correctness of the refinement step is defined as the combination of how the interfaces of the system are translated and how the system states are mapped from the higher level of abstraction to the more concrete one.

In summary, the main advantages of the proposed framework are:

- The framework is general purpose: it can be applied to any kind of system, not just computer-based ones.
- The framework assists, at the single level of abstraction, with the verification of deadlocks between interfaces calls using parametric graphs.
- The framework offers the possibility to specify the system for any general state or for partially or totally specified states. This is useful when knowledge about the state is necessary to prove system properties, or to limit the verification to specific states.
- The framework is designed to work during refinement steps of the system model, but can also be used to verify the correctness of the mapping of two different systems. This is useful, for example, when a system has to be replaced and it is necessary to prove that the new one will offer the same behavior of the old one.

- The framework is designed to be easy to implement in automated verification systems, to let the machine help with the tedious details of the proofs.

Background and related works 2

This chapter illustrates briefly the background in the research related to formal methods applied to system design and access control systems.

In the first section, the reader is given background knowledge about what formal methods are, what access control systems are and how these two topics are related in the literature.
The second section explains the ideas that led to the development of ISAAC, starting from considerations related to the current research trends.
The third section offers a quick overview of related works in the state of the art on the topics of access control and system model refinement.

2.1 Background

Giving a precise definition of Formal Methods (FM's) is hard: they come in many flavors and are used in many application fields, and their research and evolution are in continuous change. Borrowing the definition from [1], one can imagine formal methods as "*all notations having a precise mathematical definition of both their syntax and semantics [...] that allow for describing and reasoning about the behavior of systems in a formal manner*". Formal methods can roughly be divided in two main categories: notation techniques and analysis techniques. The former are those methods used to describe a system, such as Petri nets [2], temporal logic [3] or more industry-oriented tools such as Z [4] or B [5]. The latter, on the other hand, are used to reason about and, possibly, prove properties of a system. Examples in this category are model checking techniques, such as SPIN [6], and automated theorem provers, such as PVS [7].

One of the false myths about formal methods is that no one uses them: formal methods are nowadays used extensively in all those areas that require strong security and correctness guarantees, both in the computer science field and outside of it. In particular, formal methods are becoming a major topic in the field of system security, with a strong focus on the sub-topic of access control. Access control refers to "*any method or mechanism by which the access to and by entities of the system is regulated*" [8]. In particular, system administrators usually define a *policy* which states who can do what in the system, and every request made by the users is submitted to a *policy decision point* (PDP), which interprets the current policy against the request and the user who made it, and decides whether to grant the request or deny it. The expressiveness of the language used to define the policy affects not only the possibilities available to the system administrator to combine different rights, but also the possibility for policies to be verified automatically, both from the PDP at the time of the requests and from the system designer during the design of the system.

As usual, the more expressive a policy language is, the harder it is to verify it automatically and the higher is the risk to introduce inconsistency. The system consisting of the authorization policy, the PDP and the interfaces to query and modify the current policy is usually referred to as an *access control system* (ACS).

A number of works in the field of access control systems have been dedicated to quantifying, in absolute or relative terms, the expressive power of ACS's [9, 10, 11]. While it is important to know whether an ACS is expressive enough to implement a given abstract policy, there is no guarantee that raw expressive power alone will make it easy for the ACS to be integrated in the rest of the system it is guarding, nor that it will fully preserve the host system functionality. In other words, the fact that an ACS, when analyzed by itself, has been proven strong in terms of expressiveness and safety, means nothing once you actually have to plug it in the application system which it has to guard. Too often the security of the system is taken in consideration only at the end of the implementation, and the risk of purchasing commercial-grade, off-the-shelf security tools and just plugging them into an existing system hoping it works is a common scenario, which usually leads to more or less serious problems.

2.2 The idea behind ISAAC

Starting from these consideration, a different trend of research has stem: comparing access control systems not by their raw expressive power, but rather by their fit towards the specific application they need to be implemented into. From this perspective, being expressive enough is a necessary condition, but not the only metric taken in consideration. Other common metrics include, for example, ease of integration, ease of property verification, scalability, and safety. One of the theories adopting this approach is the so called Access Control Evalutation Framework [12], which was also implemented as a tool within PVS by the author [13]. While ACEF was presenting an innovative approach to the problem, its goal was still the one of verifying the validity of candidate ACS's against an application system, i.e. *after* the application system design was finished. Unfortunately, this made ACEF still prone to the integration phase issues discussed before when used in real-world scenarios.

From the previous experience with ACEF, ISAAC was designed with three main goals in mind:

- Making the design of application and authorization systems integrated, in order to minimize the risk of integration issues and forcing the system designer to consider security since the beginning.
- Offering an incremental approach to system design, i.e. refinement, and offering the tools to verify the consistency of the refinements.
- Being designed in such a way to be general purpose and easy to implement inside automated verification systems.

Most of the structure of ACEF was borrowed in ISAAC, but the meaning behind the elements of the models have been radically changed, with the goal of creating a perspective in line with the design principles mentioned. Other

elements were created anew, starting from the previous experience with automated verification systems, in order to facilitate future works towards the automation of ISAAC.

2.3 Related works

Most of the recent proposal for access control system languages either impose some sort of restrictions on the implementation of access control policies or are not considering the co-design of application and authorization as a crucial issue. Two notable exceptions are the works of Bertino et Al. [11] and of Crampton and Morriset [14]. In particular, the latter abstracts an access control system to its essential properties and is able to define a language that is almost completely free of restrictions. It is interesting to note that, albeit the end results are different, there are a number of similarities between the work of Crampton and Morriset and ISAAC, such as the distinction between authorization and application queries and the explicit definition of evaluation functions as a key element of the analysis. Nonetheless, their work focuses on the analysis of a single instance of system model, limiting the possibility to apply their method to the design of big and complicated systems, as it does not offer tools for an incremental approach to the design of the system.

The need for tools for incremental refinement verification, on the other hand, is a well studied problem approached in many different ways by many different authors. Notable examples of this are the techniques of model driven engineering [15], model transformation [16], refinement calculus [17, 18] and refinement for components and object systems [19].

The goal of ISAAC is to bring together these two aspects of generic system application and security co-design and system model refinement, in a such a way that the formal approach does not get too cumbersome to the point of being an obstacle to its application. None of the surveyed approaches shared this same dual goal, either focusing on the former or latter aspect only.

One-level mode in ISAAC 3

ISAAC is designed to work in two different modes: one-level modeling and two-levels modeling. In one-level modeling the system is represented at a single level of abstraction: there is a unique model of the system and the reasoning focuses on finding design flaws related to the presence of circular dependencies between the system interfaces.

In the he first section of this chapter the content of a one-level model is illustrated in detail, explaining the meaning and the structure of all its components.
The second section explains how reasoning is carried out to analyze the presence of circular dependencies between the system interfaces.

3.1 One-level modeling

When approaching the way ISAAC models system, there are a few things to remember in order to understand the design decisions taken. First of all, ISAAC has been designed with the goal of being as general as possible. The trade-off for this kind of comprehensiveness is a slightly large number of elements that must be specified in the model. Additionally, since formal proofs can be tedious in their details and many in numbers, ISAAC was designed trying to make it easy to reproduce the methodology within formal proving systems available off-the-shelf. For this purpose, ISAAC requires the model to be self-contained: *all* the elements of the system must be in the model, and no outside information must be provided. This, apparently strong, constraint guarantees that the behavior of all the sources of information is known in detail, so that this knowledge can be used to carry out the verification operations.

The elements of a system model in ISAAC can be split in two categories: the mutable elements and the immutable elements. The mutable elements are those data structure that may change their content during the execution of the system. The set of all the mutable elements is called the state of the system. The set of all possible states will be indicated by the symbol $\mathbb{S}$, and it contains any possible combination of values of the mutable elements. There are three categories of mutable elements in any ISAAC model:

- The users set contains the list of all the *active* user ID's currently in the system. It is indicated by the symbol $\mathbb{U}$.
- The application data structures set is the set of all those data structure containing information related to the functioning of the system with respect to its goal. It is indicated by the symbol $\mathbb{X}$.
- The authorization data structures set is the set of all those data structure containing information related to the authorization policy of the system.

It is indicated by the symbol $\mathbb{A}$.

The sharp separation between application and authorization data comes from the idea that, when co-designing the authorization and application parts of your system, it is desirable to have separate structures for the two: albeit it is possible to store both kind of information in the same structures, doing so would increase the complexity of the information management, increasing the risk of bugs.
The set of active users is shared between application and authorization data. Albeit it is possible to formally model systems that do not interact with users, ISAAC focuses on those systems that require access control capabilities, and users are therefore included as a standard element in any ISAAC model. The list of the users of the system is intuitively shared between the authorization and application part, i.e. the users (more specifically, their ID's) are the same when referred by the application data structures and the authorization data structures.

In addition to the state, a system is described also by its immutable elements. These elements describe the behavior, the capabilities and the interfaces of the system with the outer world. Immutable elements can, in theory, change during the life-cycle of the system, such as for the need of adding a new feature, but they are supposed to stay constant during the normal execution of the system. Changing an immutable elements must be thought of as a radical, structural change which compromises the validity of all the proofs carried out on the model so far, as modifying one of the immutable elements is effectively equivalent to creating a new system. In ISAAC there are six immutable elements:

- The domains set, indicated by the symbol $\mathbb{D}$. It specifies all the data types used in the system. These can be natural numbers, enumerations,

ID's or anything required. It is important to note that no piece of information of a type not in the domains set can exist in the system model.

- The commands set, indicated by the symbol $\mathbb{C}$. Commands are those interfaces of the system that may modify its state. As for methods in object oriented programming, their goal is to prevent direct access to the inner state and control how it is manipulated.
- The queries set, indicated by the symbol $\mathbb{Q}$. Queries are those interfaces of the system that are guaranteed to not modify its state, and which return a value. As for properties in object oriented programming, their goal is to prevent direct access to the inner state and control how it is accessed. The queries set is the disjoint union of two subsets: application and authorization queries.
  - The application queries set, indicated by $\mathbb{Q}_{\mathbb{X}}$, contains all the queries that can be performed by the users of the system. These may also relate to the authorization policy, but are distinct from authorization queries.
  - The authorization queries set, indicated by the symbol $\mathbb{Q}_{\mathbb{A}}$, contains one query for each command or application query in the system. It contains queries that must be imagined as performed internally by the system and are not accessible by its users.
- The transition function, indicated by the symbol $\mathbb{T}$. Its goal is to contain the semantics of all the system commands, and define how they are performed.
- The interpretation function, indicated by the symbol $\mathbb{I}$. Its goal is to contain the semantics of all the system queries, and define how they are performed.

A visual representation of a system model is given in figure 3.1. After pre-

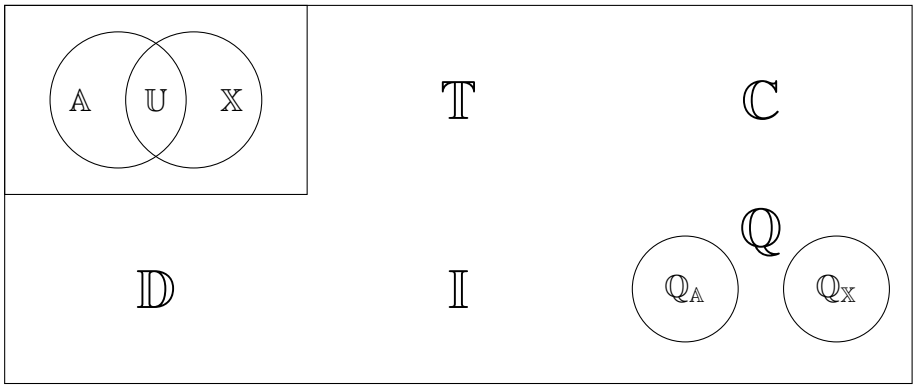

FIGURE 3.1: System model in ISAAC

senting in an informal way the elements of a ISAAC one-level model, the remainder of this section gives a detailed description and formalization of each of them. To facilitate comprehension, Appendix A and Appendix B contain the complete specification of an example scenario related to the modeling of a simplified file system manager, which will be used as a reference in this and in the next chapter. In this narrative, each file has an owner, and users have the possibility to read or write (no execute) a file. There is no possibility to create a file: writing to a non-existing file will simply fail. Only the owner of a file can read or write it. The authorization policy is specified using an access matrix, i.e. a set of <subject, object, right> triplets. In our example there are two users and two files: *Alice*, owner of file p, and *Bob*, owner of file q. The model must also consider the content of each file. Trying to read a non-existing file will return a default failure value.

3.1.1 Domains set

Definition 1 *The* domains set $\mathbb{D}$ *is a non-empty set of sets, containing at least the sets UIDs, which contains all the possible user ID's, and the set BOOLs, which contains the two boolean values.*
The domains set contains the data types used in the model.

In addition to the user ID's and the boolean values, the domains set also contains all the sets of all the data types that are present in the model: the flattened version of the domains set contains all the possible values that any non-composed field of information in the model may assume at any given time during the execution. The set of domains contains only the data types that model information regarding the mutable elements of the state, i.e. it does not contain the queries or commands types.

In our example, $\mathbb{D}$ is defined as follows:

$$\mathbb{D} = \{\text{UIDs}, \text{BOOLs}, \text{FIDs}, \text{FCs}, \text{AUTHs}\}$$

where UIDs (User ID's) contains the set of all non-empty strings of alphabetic character, BOOLs is the {TRUE, FALSE} set, FIDs (File ID's) contains the set of all the valid absolute file paths, FCs (File Contents) contains all the binary strings and AUTHs (Authorizations) contains the "read" and "write" constants representing the authorization rights.

3.1.2 Users set

Definition 2 *The* users set $\mathbb{U}$ *contains the user ID's of all the* active *users of the system, i.e. those capable of issuing commands or queries in the current state.*
$\mathbb{U} \subseteq \mathbb{D}.$*UIDs, as the latter contains* all *the possible user ID's.*

In our example, $\mathbb{U}$ is defined as follows:

$$\mathbb{U} = \{\text{Alice}, \text{Bob}\}$$

3.1.3 Application data

Definition 3 *The* application data structures set $\mathbb{X}$ *contains all those data structure related to application-specific information.*

These vary according to the specification of the system modeled, and therefore cannot be formalized precisely in the general case.

One important thing to notice is that application data structures are part of the state of the model, and as such they can only contain data. The procedures to alter and access them should be included in the commands and application queries sections. As for all the data in the model, the data types contained in the application data structure must also appear in the domains set.

In our example, the application data structures are composed of: the set of the files present in the system, which is a subset of $\mathbb{D}$.FIDs, a function from the files to their contents and a function from the files to their owners. Note that it is possible to make the co-domain of the content function either the totality of $\mathbb{D}$.FCs, or to model in the application data structure also a set of the current available contents, a subset of $\mathbb{D}$.FCs, and make that the target co-domain. As usual, there is more than one option. In this example, the first option is preferred.

$$\mathbb{X} = \{\text{FILES}, \text{CONTENT}, \text{OWNER}\}$$

where:

$$\text{FILES} \subseteq \mathbb{D}.\text{FIDs} = \{\text{p}, \text{q}\}$$

$$\text{CONTENT} : \text{FILES} \to \mathbb{D}.\text{FCs} = \{(\text{p}, 0), (\text{q}, 1)\}$$

$$\text{OWNER} : \text{FILES} \to \mathbb{U} = \{(\text{p}, \text{Alice}), (\text{q}, \text{Bob})\}$$

3.1.4 Authorization data

Definition 4 *The* authorization data structures set $\mathbb{A}$ *contains all those data structure related the current authorization policy of the system.*
Their structure depends on the access control system used, and therefore cannot be formalized precisely in the general case.

As for application data structures, authorization data structures are just information containers that can be modified and interpreted only by using the associated commands and authorization queries, and their data types must also appear in the domains section. Authorization data structures may also be absent from the model, in the case of immutable authorization policies. In this case, all the information regarding the authorization policy will be contained in the interpretation function.

In our example, the chosen access control system will be an access matrix, i.e. a set of <subject, object, right> tuples that specify the rights of each user.

$$\mathbb{A} = \{M\}$$

where:

$$M = \{$$

$$\langle \text{Alice, p, read} \rangle,$$

$$\langle \text{Alice, p, write} \rangle,$$

$$\langle \text{Bob, q, read} \rangle,$$

$$\langle \text{Bob, q, write} \rangle$$

$$\}$$

3.1.5 Interactions and actions

Definition 5 *An* interaction *is either a command or an application query. In other words, interactions are something that can be issued by an active user of the system.*

Commands compose, together with application queries, the interfaces of the system accessible from the active users of the system. The intuitive difference between the two is that application queries represent questions asked to the

system regarding its inner state, and as such they are guaranteed to not modify the state, while commands represent interfaces that are used to manipulate or alter the state of the system.

Definition 6 *An* action *is either an interaction or an authorization query. In other words, actions are anything that can be issued within the system either from users or from the system itself.*

Authorization queries, on the other hand, cannot be issued by the users, but are special queries issued by the system, in response to an user issuing an interaction, used to verify whether the user possesses the required rights.

3.1.5.1 Commands

Definition 7 *The* commands set $\mathbb{C}$ *contains the list of all the available commands of the model.*
Commands are those interaction that may alter the state of the model when executed.

Neither queries nor commands contain any semantics by themselves: the $\mathbb{Q}$ and $\mathbb{C}$ elements of the model merely contain definitions and type structures. They can be thought as equivalent as empty function prototypes, the semantics of which is specified by the transition and interpretation functions.

Definition 8 *A* command *c is defined as:*
$c = \langle n, P \rangle$
where:
n is the name of the command,
$P = P_1 \times \cdots \times P_j$ *is the set of parameters space from which j parameters of the command are drawn* ($P_1, \ldots, P_j \in \mathbb{D}$)

In our example, the only available command is the command to write a file:

$$\mathbb{C} = \{\langle \text{write}, \mathbb{D}.\text{FIDs} \times \mathbb{D}.\text{FCs} \rangle\}$$

3.1.5.2 Queries

Definition 9 *The* queries set $\mathbb{Q}$ *contains the list of all the available queries of the model, both application and authorization ones.*
Queries are those actions that are guaranteed not to alter the state of the model when executed.

Queries are of two disjoint types: application and authorization queries. Application queries are all those queries *that can be issued by the users of the system.* Authorization queries are inner queries, *which cannot be issued from the users*, used by the system to answer the authorization requests. There exist one authorization query for each application query and command. Authorization queries always return boolean values. In principle, application queries could be defined in such a way to return only boolean answers without reducing their expressive power: advanced ones, such as counting all the users which satisfy a given property, may be performed by sweeping the boolean query over each single user. Albeit this approach simplifies the definition of the methodology, it feels very distant from what happens in practice, where application queries may return any type of data. To allow the possibility of generic return types, the returned value type must also be specified in their definition.

Definition 10 *An application query q_x is defined as:*
$q_x = \langle n, P, R \rangle$
where:
n is the name of the query,
$P = P_1 \times \ldots \times P_k$ is the set of parameter space from which the query's k parameters are drawn ($P_1 \ldots, P_k \in \mathbb{D}$),
$R \in \mathbb{D}$ is the query's return type

Definition 11 *An authorization query q_a is defined as:*
$q_a = \langle n, P \rangle$ *where: n is the name of the authorization query*

$P = P_1 \times \ldots \times P_k$ is the set of parameter space from which the query's k parameters are drawn ($P_1 \ldots, P_k \in \mathbb{D}$)

In our example, the only available application query is the query to read a file content. There are also two authorization queries, one for each command or application query.

$$\mathbb{Q} = \{\mathbb{Q}_{\mathbb{X}} \dot{\cup} \mathbb{Q}_{\mathbb{A}}\}$$

where:

$$\mathbb{Q}_{\mathbb{X}} = \{\langle \text{read}, \mathbb{D}.\text{FIDs}, \text{FCs}\rangle\},$$

$$\mathbb{Q}_{\mathbb{A}} = \{$$

$$\langle \text{auth-read}, \mathbb{D}.\text{FIDs}\rangle,$$

$$\langle \text{auth-write}, \mathbb{D}.\text{FIDs} \times \mathbb{D}.\text{FCs}\rangle$$

$$\}$$

3.1.6 Semantics functions

Definition 12 *The semantics functions are the transition function $\mathbb{T}$ and the interpretation function $\mathbb{I}$.*
The semantics functions specify the semantics of actions: he transition function specifies the semantics of commands, the interpretation function specifies the semantics of queries.

Semantics functions are defined using a simple pseudo-programming language. This language is algorithmic and has the strong constraint of only possessing two flow control statements: 'if-then-else" branches and "for-each" loops. The reason for this constraint is to simplify the approach to one-level reasoning, as will be illustrated in the next section. It is possible to specify the semantics functions using *any* approach, but in that case ISAAC will not be able to operate in one-level mode, but only in two-levels mode.
Using an algorithmic language to specify the semantics functions implies the

possibility that an interaction will call other actions upon which its execution depends. For example, when a user tries to rent a book at a library, he is also querying the catalog of the library for the presence of such book. To actually perform an interaction, a user must be entitled not only to the right of the interaction itself, *but also to every other interaction in its closure*, i.e. to all interactions present in the call stack of the execution code.

3.1.6.1 Transition function

Definition 13 *The transition function* $\mathbb{T}$ *specifies the semantics of the commands:*

$\mathbb{T} : \mathbb{S} \times \mathbb{C} \times \mathbb{D}.UIDs \rightarrow \mathbb{D}.BOOLs \times \mathbb{D}.BOOLs \times \mathbb{S}$

where:

$\mathbb{S}$ *is the set of all the possible states of the model,*

the first return value is the authorization success flag,

the second return value is the semantic success flag,

the last return value is the state obtained after the issue of the command.

Starting from a system state, a command and the ID of the user issuing the command, the transition function returns a triplet containing two boolean values and a system state. The first boolean value indicates whether the command had success according to authorization constraints, i.e. whether the user had the right to issue the command.

The second boolean value indicates whether the command had success from the point of view of semantics: for some combination of parameters, certain commands may be semantically meaningless, such as trying to delete a non-existing file. The distinction between authorization and semantic success is aimed at trying to reduce information leak via covert channels: the transition function should be designed in such a way that, when the command fails for authorization reasons, no additional information is provided. In other words, in all the cases where the authorization return flag is false, the semantics return

flag should always be false. This does not prevent information leak completely, but forces the designer to think about the issue during the modeling of the system. The command should be considered as successfully executed if and only if both the return flags are true.

The returned state is the new state of the system after the command was issued. Albeit it may seem reasonable that an authorization or semantics failure should return a non-modified state, this is not strictly enforced: for example, the state may model also a log facility which registers the failure.

Another important point is that, as the transition function is defined over any possible user ID, it is required to specify what happens when a user ID not present in the state issues a command. Albeit this is apparently just a notation issue, due to the fact that it is required that the transition function does not change during the execution while the set of active users does, it is possible to give different semantics to the transition function where these cases may be relevant, for example to model intruders trying to issue commands.

In our example, the transition function may be defined as follows:

```
1 T(s, write(fid, fc), u) = {
2         if (!I(s, auth-write(fid, fc), u)) {
3                 //Authorization failure
4                 return <FALSE, FALSE, s>;
5         }
6         if (!s.X.FILES.contains(fid)) {
7                 //Semantic failure: non-existing file
8                 return <TRUE, FALSE, s>;
9         }
10        //Valid request: can write, file exists
11        s.X.CONTENT(fid) = fc;
12        return <TRUE, TRUE, s>;
13 }
```

3.1.6.2 Interpretation function

Definition 14 *The interpretation function* $\mathbb{I}$ *specifies the semantics of the queries.*

$\mathbb{I} : \mathbb{S} \times \mathbb{Q} \times \mathbb{D}.UIDs \rightarrow \mathbb{D}.BOOLs \times \mathbb{D}.BOOLs \times (\mathbb{D} \cup \{\perp\})$

where:

$\mathbb{S}$ *is the set of all the possible states of the model,*

the first return value is the authorization success flag,

the second return value is the semantic success flag,

the last return value is the result of the query or a default failure value.

The interpretation function is defined for both application and authorization queries. The definition and purpose of the interpretation function is very similar to the one of the transition function for commands, and the interpretation function also returns two boolean flags in addition to the query result. The meaning of those flags is the same of those in the transition function: the first one relates to authorization success, the second to semantics success. Also for the interpretation function, in all the cases where the authorization return flag is false, the semantics return flag should always be false and the returned value should be $\perp$.

The return value can be either the queried datum or a default failure symbol, indicated by the symbol $\perp$. The default failure symbol should be returned in all and only the cases where either an authorization or semantics failure is present, and should be distinguished from all the normal data values. In other words, $\perp$ should not appear in the domains set.

The only difference between the interpretation function for application and authorization queries is that the former are defined over a generic domain, and therefore the returned type must match the one of the application query, while

the latter can only return boolean values.

$$\forall s \in \mathbb{S}, q_x \in \mathbb{Q}_{\mathbb{X}}, u \in \mathbb{D}.\text{UIDs} :$$
$$\text{type}(\mathbb{I}(s, q_x, u).\text{retValue}) = q_x.\text{retType}$$

$$\forall s \in \mathbb{S}, q_a \in \mathbb{Q}_{\mathbb{A}}, u \in \mathbb{D}.\text{UIDs} :$$
$$\text{type}(\mathbb{I}(s, q_a, u).\text{retValue}) = \mathbb{D}.\text{BOOLs}$$

In our example, the interpretation function may be defined as:

```
1 I(s, read(fid), u) = {
2         if (!I(s, auth-read(fid), u)) {
3                 //Authorization failure
4                 return <FALSE, FALSE, ⊥>;
5         }
6         if (!s.X.FILES.contains(fid)) {
7                 //Semantic failure: non-existing file
8                 return <TRUE, FALSE, ⊥>;
9         }
10        //Valid request: can read, file exists
11        return <TRUE, TRUE, s.X.CONTENT(fid)>;
12 }
13
14 I(s, auth-read(fid), u)) = {
15        if (!s.FILES.contains(fid)) {
16                //Non-existing file
17                return FALSE;
18        }
19        //File exists
20        return s.A.M.contains(<u, fid, READ>);
21 }
```

```
22
23 I(s, auth-write(fid, fc), u) = {
24         if (!s.FILES.contains(fid)) {
25                 //Non-existing file
26                 return FALSE;
27         }
28         //File exists
29         return s.A.M.contains(<u, fid, WRITE>);
30 }
```

3.1.7 Additional aspects

Now that the structure of a one-level ISAAC model and the purpose of each of its elements are well defined, it is possible to point out a few peculiarities of this approach.

First of all, it is possible to specify a state of the system either by describing a state instance or just by defining the structure of the state. This allows to draw conclusion on the properties of the system for either any possible state, for a completely defined state of interest or anything in between, as the state can also only be partially instantiated.

Another point of interest regards the relationship between data structure and the interpretation and transition functions: it is rarely the case that any system of interest has no application data or an empty set of active users, but it is possible to have a system model with no authorization data structures. This is, for example, the case for systems where the authorization policy can be directly derived from the application data structures. For example, the interpretation function of the example in Appendix A could have been defined as:

```
1 I(s, auth-read(fid), u)) = {
2         if (!s.FILES.contains(fid)) {
3                 //Non-existing file
4                 return FALSE;
```

```
5         }
6         //File exists
7         return (s.X.OWNER(fid) == u);
8 }
9
10 I(s, auth-write(fid, fc), u) = {
11        if (!s.FILES.contains(fid)) {
12                //Non-existing file
13                return FALSE;
14        }
15        //File exists
16        return (s.X.OWNER(fid) == u);
17 }
```

The trade-off is that having authorization data structures, while increasing the complexity of the state, allows to modify the authorization policy without modifying the immutable elements of the model, which would invalidate all the reasoning carried out before the change. Always on the previous example, should the authorization policy change so that anyone can read files but only the owner can write them, it is trivial to modify the access control matrix in the former case, while it would be necessary to modify the interpretation function in the latter.

Finally, the nature of the call traces of commands, application queries and authorization queries is also intriguing. The call trace of a command is never empty, as it contains at least the respective authorization query, and it may contain calls to other commands, application queries or authorization queries; the call trace of an application query is never empty, as it contains at least the respective authorization query, and it may contain calls to other application queries or authorization queries, but not to commands. This is because queries are guaranteed not to modify the state, while commands may. The call trace of an authorization query may be empty, as it could work directly on the state

data structures, and it may contain calls exclusively to other authorization queries, not to commands or application queries. The possibility to call other authorization queries comes from the desire to avoid code duplication, but it should always be possible to unfold an authorization query call stack, as their call stack should *never* be recursive. This structure hints to the possibility to analyze some sort of fixed point property of the action call traces. As will be shown in the next section, this is the main goal of the one-level mode of ISAAC.

3.2 One-level reasoning

Reasoning within one-level models focuses on the verification of circular dependencies among actions. If these circular dependencies are present, they must be analyzed one-by-one to guarantee that no parameter combination ever triggers an infinite call stack, which would be equivalent to having an undecidable situation. The main goal of ISAAC is to work in the two-levels mode which will be illustrated in the next chapter, but if the semantics functions are specified according to the flow control constraints mentioned before, ISAAC is also capable to help out with the analysis of one-level models. While not able to automatically analyze all the scenarios, ISAAC provides a simple method to check the presence of circular dependencies. It is important to remember that this method *does not guarantee* the absence of circular dependencies in all the scenarios, and can only offer positive answers, but it can still help for some of the common cases.

As already mentioned, the execution of an action may rely on other actions. More specifically, commands may call upon other commands, application queries and authorization queries; application queries may call upon other application queries and authorization queries; authorization queries may call upon other authorization queries, but only in an unfoldable way. Therefore, each action possesses a "closure", i.e. a list of other actions it relies upon to be executed. For the purposes of the analysis, it is not necessary to compute the transitive closure of all the action calls, but is rather necessary to only consider the actions directly called at the first level of depth of the call stack. This is due to the fact that, in order to analyze the presence of potential infinite loops, it is possible to create a direct graph with interactions as nodes and call dependencies as edges. Once the graph is complete, it is possible to analyze the presence of loops in the graph using standard techniques of graph analysis. If no loops are present, then the analysis is over. Otherwise, it is required to analyze each loop individually: if an infinite loop scenario is present, it must appear as a

looping path in the graph. ISAAC offers a methodology to detect the common looping cases, but it is still required to check all the loops manually.

Definition 15 *A* call graph *is a labeled direct graph where each node represents an interaction, and each edge from A to B represents a direct call to the semantic functions of action B in the semantic functions of action A.*
The label of an edge identifies the code path *that led to the call.*

Definition 16 *The* code paths *of the semantic function of an action are all the possible paths that can be followed during the execution of the semantic function.*
Each code path has an associated code path context.

Definition 17 *A* code path context *is a set of logical conjunctions with variables bind to the parameters of the semantic functions.*
It represents the conditions under which the code path is executed.

3.2.1 Call graph computation

Semantic functions in ISAAC are specified using a simplified programming language, which only has two kind of flow control structures: "if-then-else" branches and "for-each" loops. The reason for this constraint is to be able to mechanically create the code paths for the call graph of an action. While it is possible to ignore this constraint, if the semantics functions of the model are specified using a different language, ISAAC can only be operated in two-levels mode. To compute the call graph of a model it is necessary to identify, for each action, all the possible code paths that lead to a return value. These are finite in number and can be computed easily according to the following rules:

- For a conditional branch, two code paths are generated: one for when the "then" branch is executed, one for when the "else" branch is executed.

- For a looping construct, two code paths are generated: one for when at least one item is a valid candidate for the loop and one for when the selected set is empty.

These code paths are identified by an unique ID composed by the name of the action they belong to and a progressive number. Each code path has an associated context, a set of logical conjunction with variables bind to the parameters of the semantics function.

The pseudo-code of the algorithm for the computation of the call graph is the as follows:

```
1 Set<Node> N = ℂ ∪ ℚ_𝕏;
2 foreach (Node n in N) {
3         n.edges = ∅;
4         foreach (Codepath c in n.codepaths) {
5                 foreach (SemanticFunctionCall sfc in c) {
6                         Edge e = new Edge(label = c.context);
7                         n.edges.add(e);
8                 }
9         }
10 }
11 return N;
```

Suppose to have a model with four commands: A(pa_1, pa_2), B(pb_1), C(pc_1, pc_2), D(). For the sake of simplicity, this example disregards authorization queries and return values, as they are not of interest for the understanding of the algorithm. The specification of the transition function is as follows:

```
1 T(s, A(pa1, pa2), u) = {
2         if (pa1 > 0) {
3                 foreach (x in pa2) {
4                         B(x);
5                 }
6                 return;
```

```
7         }
8         else {
9                 D();
10                return;
11        }
12}
13
14T(s, B(pb1), u) = {
15        if (pb1 < 0) {
16                C(pb1, pb1);
17                return;
18        }
19        else {
20                D();
21                return;
22        }
23}
24
25T(s, C(pc1, pc2), u) = {
26        if (pc1 > pc2) {
27                D();
28                return;
29        }
30        else {
31                A(pc1, {pc2});
32                return;
33        }
34}
35
36T(s, D(), u) = {
37        return
38}
```

The code paths of each action possess both an identifier and a logical conjunction which indicates the conditions required to activate that path:

- A:
  - A_1: Context = $[\mathrm{pa}_1 > 0, \mathrm{x} \in \mathrm{pa}_2]$; Calls = [B(x)]
  - A_2: Context = $[\mathrm{pa}_1 >, \mathrm{pa}_2 = \emptyset]$; Calls = []
  - A_3: Context = $[\mathrm{pa}_1 \leq 0]$; Calls = [D()]
- B:
  - B_1: Context = $[\mathrm{pb}_1 < 0]$; Calls = $[\mathrm{C}(\mathrm{pb}_1, \mathrm{pb}_1)]$
  - B_2: Context = $[\mathrm{pb}_1 1 \geq 0]$; Calls = [D()]
- C:
  - C_1: Context = $[\mathrm{pc}_1 > \mathrm{pc}_2]$; Calls = [D()]
  - C_2: Context = $[\mathrm{pc}_1 \leq \mathrm{pc}_2]$; Calls = $[\mathrm{A}(\mathrm{pc}_1, \mathrm{pc}_2)]$
- D:
  - D_1: Context = [ ]; Calls = [ ]

The complete call graph of this example model is indicated in figure 3.2

3.2.2 Call graph loops analysis

Once the call graph of the model is computed, the next step is to analyze the presence of loops. If the call graph does not present loops, then the analysis is over and model is guaranteed to not have circular dependencies among interactions. Otherwise, each loop must be analyzed for the presence of infinite looping scenarios.

A very important thing to remember is that the simple ISAAC approach is just a first-hand tool that can fail in many instances, and therefore should not

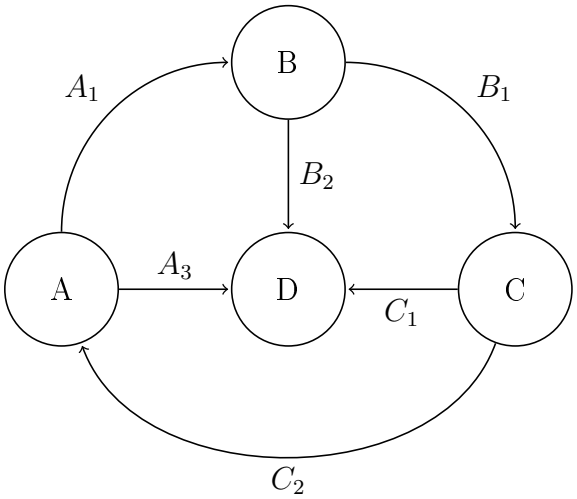

FIGURE 3.2: Example call graph

be used alone for this task, but rather as a support to the analysis performed using human insight and domain knowledge of the model.

The analysis of a loop is performed as follows: one of the nodes of the loop is chosen as the "pivot" of the analysis. The contexts of all the code paths in the loop are put together as a set of logical conjunctions and all the variable in the context of the loop are rewritten as a function of the parameters of the semantic functions of the pivot node. At this point, the logical conjunction of the loop context is checked for satisfiability: if the context of the loop is unsatisfiable, then the loop is not an infinite loop. Otherwise, it must be analyzed using more sophisticated techniques and human insight. The pseudo-code for the loop analysis is as follows:

```
1 LoopContext lc = ∅;
2 foreach (Codepath cp in loop) {
3         lc.add(cp.context);
4 }
5 Node pivot = random(loop.nodes);
6 lc.rewrite(pivot);
7 return AnalyzeSat(lc);
```

Let's take a look at our previous call graph example in figure 3.2: the only loop is given by A-B-C-A Suppose A is chosen as the pivot of the analysis. The next step consists in rewriting the context of the loop, i.e. the conjunction of the contexts of each code path, so that each variable is derived from the parameters of A. In our case, the loop is composed by code paths A_1, B_1, C_2 and again A_1. Its full context is [$\mathrm{pa}_1 > 0$, $\mathrm{x} \in \mathrm{pa}_2$, $\mathrm{pb}_1 < 0$, $\mathrm{pc}_1 \leq \mathrm{pc}_2$, $\mathrm{pa}_1 > 0$, $\mathrm{x} \in \mathrm{pa}_2$]. Looking at the call trace of the loop, the context can be rewritten as a function of the parameters of A as [$\mathrm{pa}_1 > 0$, $\mathrm{x} \in \mathrm{pa}_2$, $\mathrm{x} < 0$, $\mathrm{x} \leq \mathrm{x}$, $\mathrm{x} > 0$, $\mathrm{x}' \in \{\mathrm{x}\}$]. This conjunction is trivially unsatisfiable, as it presents both the conditions $\mathrm{x} < 0$ and $\mathrm{x} > 0$. Therefore, the call graph does not present infinite loops.

It is important to note that this method can offer the guarantee that the analyzed loop is not an infinite loop if and only if the analysis yields an unsatisfiability result. Since it is not known beforehand how deep the recursion of a call loop is, it is possible that a loop is not infinite, but just iterates for a big number of times. ISAAC is only capable to reliably detect all the loops and their relative code paths, but it will not always be able to answer whether a loop is infinite or not. In those cases, is up to the designer to use intuition, hindsight and more advanced proving techniques to verify the absence of infinite loops. Improvements in the field of one-level reasoning are one of the subjects of future works within ISAAC.

Two-levels mode in ISAAC 4

In two-levels mode, ISAAC assists with the verification of the correctness of a refinement step of the model of the system: there are two different models of the system, ideally one at a more abstract level and one refining the previous to a more concrete level, and the goal is to verify whether the more concrete model correctly implements the more abstract one. The two models are both instances of a one-level model, and a number of pairing elements are specified.

In the first section of the chapter, the pairing elements that bind together the two models are illustrated in detail.
The second section defines what correctness of the refinement is and shows how to prove it using the example from the previous chapter.
For both sections, the complete specification of the refined level of the example can be found in Appendix B.

4.1 Two-levels modeling

The rationale behind two-levels mode is that it is often the case that models are developed in a top-down fashion, starting from a very abstract description of the system, refining the model step-by-step towards a model that is concrete enough for the purpose at hand. These refinement steps must be correct, in the sense that they cannot introduce incoherence between the two levels of abstraction they pair. Additionally, it is sometimes the case that a system must be proved as functionally equivalent to another one, for example in the case of a replacement: albeit counter-intuitively, it is possible to model this equivalence in the same fashion used to model a refinement step, by imagining the old system as the abstract one and the new system as a "refinement" of the previous. Two-levels mode offers the possibility of formally specifying a refinement step in the form of three different elements: the state-mapping function σ, the action-mapping function α and the query-mapping function π. A visual representation of a two-levels setting is shown in figure 4.1. Once these three functions are properly defined, it is possible to check that the two models and the refinement are consistent with each other. For the sake of clarity, this section will be illustrated using the same narrative of the previous one: a simplified file system model. The refinement step in this case is given by splitting the, previously atomic, read and write operations in a series of "open file, do something, close file" operations, as well as by adding the modeling of the lock state of the files. For the sake of notation, the elements of the more abstract level will be indicated by the subscript 1, while those of the more concrete one will be indicated by the subscript 2.

4.1.1 State-mapping function

As already mentioned in the previous chapter, the state of a model is defined as the union of the active users set $\mathbb{U}$, the authorization data structures $\mathbb{A}$ and the application data structures $\mathbb{X}$. A model refinement may operate on any of

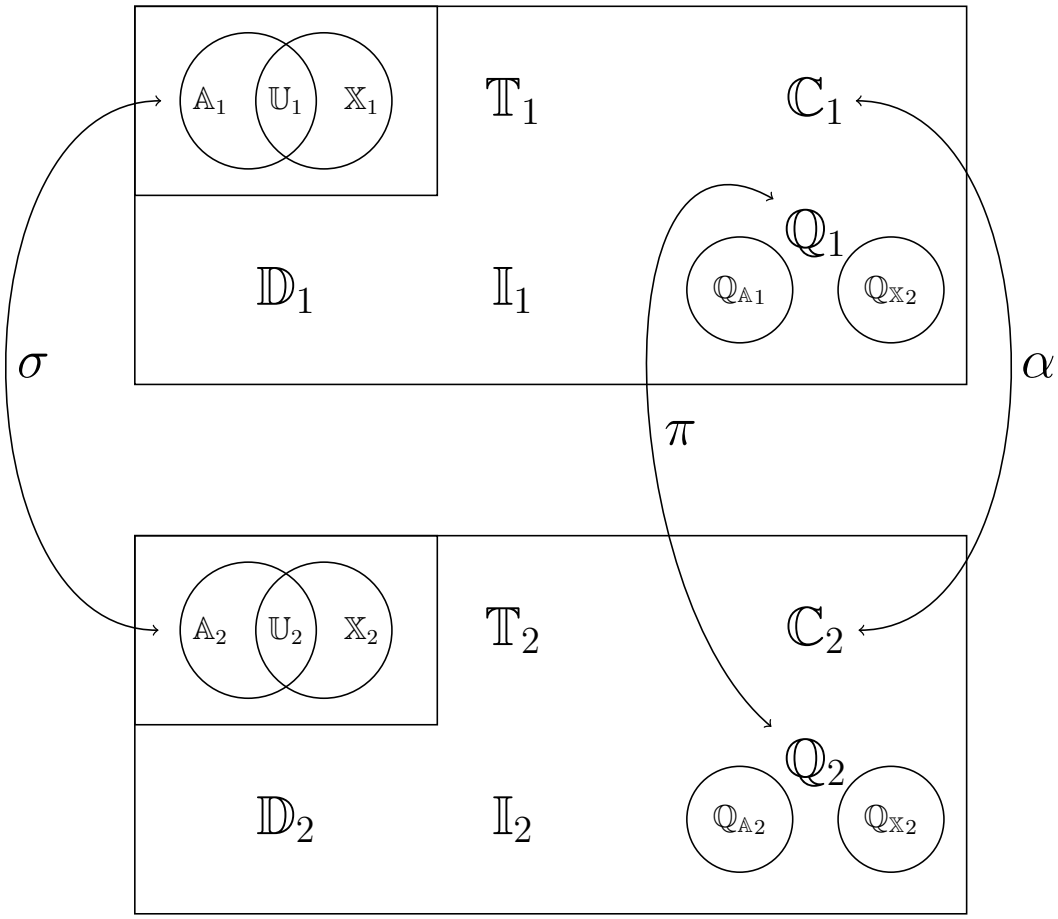

FIGURE 4.1: Two-levels mode

these state elements. For example, it is possible that the refined model splits an abstract user into multiple concrete users, translate an abstract authorization policy expressed in logical formulas into a commercial-grade access control system or represents the application database in a more realistic way. Therefore, a precise mapping from the more abstract state to the more concrete state must be defined.

Definition 18 *A state-mapping function σ specifies how the higher level model states are translated into lower level model states.*

$\sigma : \mathbb{S}_1 \rightarrow \mathbb{S}_2$

where $\mathbb{S}_1$ *is the set of states of the higher level model,* $\mathbb{S}_2$ *is the set of states of the lower level model.*

In a refinement scenario, the lower level concrete model is "the only one existing", while the higher level one is an abstraction that the designer made up to help reasoning. The reasoning during the analysis of the refinement flows from the more abstract model to the more concrete one. Therefore, the abstract level is the domain of the function, while the concrete level is the codomain. After defining the state-mapping function for the example narrative, the remainder of the subsection analyzes the properties that a state-mapping function should satisfy.

4.1.1.1 Example

In our example, the refinement of the model alters the state in two ways: it adds an additional application data element modeling file lock states and it adds the rights to "open" and "close" a file in the authorization data. Since in the higher level model the "read" and "write" operations were atomic, it is reasonable to map a higher level state into a lower level state where no file is locked. Also, it is reasonable to give "open" and "close" rights in the lower level model to anyone who could either read or write a file in the higher level model. Therefore, the state-mapping function of our example could be defined as follows:

```
1 σ(s1) = {
2         s2 = new lower level state;
3         //Users are the same
4         s2.U = s1.U;
5         //Read and write rights are the same
6         //Open and close right are derived
7         s2.A = s1.A;
8         foreach (u: s2.U) {
```

```
9                 if (s2.A.M.contains(<u, f, read>) || s2.A.M.
                      contains(<u, f, write>)) {
10                        s2.A.M.add(<u, f, open>);
11                        s2.A.M.add(<u, f, close>);
12                }
13        }
14        //Shared application data are the same
15        s2.X = s1.X;
16        //All files are unlocked
17        s2.X.LOCKED = \emptyset;
18        return s2;
19 }
```

4.1.1.2 Unique image

The state-mapping function has, so far, always been referred to as a function. Nonetheless, it is not guaranteed that each state of the higher level model is mapped in only one lower level state: since the lower level model is, potentially, expressing things in more detail, it is reasonable to assume that two or more lower level states can equivalently represent the same higher level state. Therefore, in general, lower level states form a set of equivalence classes with respect to state-mapping of the higher level states. In other words, there is an equivalence relation among lower level states that represents the fact that two or more lower level states can equivalently be used to represent the same higher level state.

Definition 19 *The state-mapping equivalence relation σ_{eq} is an equivalence relation over the set of lower level states $\mathbb{S}_2$.*
$\sigma_{eq}(p, q)$ if and only if both p and q can both be used as image for the state-mapping function for the same higher level state.

Definition 20 *Two lower level states are* state-mapping equivalent *if and only*

if they belong to the same equivalence class for the images of the state-mapping function, i.e. if and only if they can both represent the state-mapping of the same higher level state.

It is important to define σ in such a way that each starting state has only one image, i.e. a representative of each equivalence class is chosen as the image of the state-mapping function. The choice about which lower level state is picked as image of a higher level state is an arbitrary choice of the analyst, very similar to choosing what a default value is. As will be shown later, equivalence classes of the state-mapping function are also important in terms of correctness.

4.1.1.3 Injectivity

Injectivity of the state-mapping function is another important issue. Consider the case when the state-mapping function is not injective: two different states of model A become equivalent when mapped in the representation of a model B. This means that model B is expressing something less than model A, as it cannot distinguish anymore between the two states. Therefore, if the state-mapping is not injective, it is likely that what is being performed is not a refinement, as a refinement should have non-decreasing expressive power. Therefore, in all the "normal" refinement cases, it is reasonable to admit that the state-mapping function should be injective.
Nonetheless, when the two-levels mode ISAAC is used to analyze functional equivalence of two different systems rather than a proper refinement, it is possible that the state-mapping function is not injective.

4.1.1.4 Bijectivity

Since the lower level state has, potentially, a cardinality higher than the higher level state, the state-mapping function may or may not be bijective. Nonetheless, this is not a problem, as the reasoning in two-levels mode always flows

from the higher level to the lower level model, and therefore it is not important for the state-mapping function to be invertible.

4.1.2 Action-mapping function

After defining how to translate a higher level model state in a lower level model state, it becomes necessary to specify how higher level commands are converted in lower level commands. This specification is given by the action-mapping function. Since higher level commands are usually more abstract than lower level ones, a single higher level command is translated in an ordered sequence of lower level commands. The mapping can also depend on the current higher level state, as different situations may require a command to be translated in different ways, as well as on the user performing the command. Since a higher level user can be represented by multiple lower level users, it is important also to specify which lower level user is supposed to be issuing the returned lower level commands.

Definition 21 *An action-mapping function α is a function translating higher level commands in a sequence of lower level commands-users pairs.*
$\alpha : \mathbb{S}_1 \times \mathbb{C}_1 \times \mathbb{D}_1.UIDs \rightarrow (\mathbb{C}_2 \times \mathbb{D}_2.UIDs)^*$

Once again, the reasoning flows from higher level models to lower level models. As for the previous subsection, after defining the action-mapping function for our example narrative, one of the interesting aspects of action-mapping functions is analyzed.

4.1.2.1 Example

In our example, the higher level model contains the single “write” command, while the lower level model introduces the “open” and “close” commands. Since the state-mapping was defined such that higher level states always maps in

lower level states where no file is locked, it is possible to define the action-mapping function as follows:

```
1 α(s1, write1(fid, fc), u1) = {
2         result = new List<C2, U2>();
3         //Open the file
4         result.append(<open2(fid), u2(u1)>);
5         //Write the content
6         result.append(<write2(fid, fc), u2(u1)>);
7         //Close the file
8         result.append(<close2(fid), u2(u1)>);
9         return result;
10 }
```

where user $u_2(u_1)$ is the lower level version of user u_1.

4.1.2.2 State dependence

Albeit it was not the case in the example, it is possible that the result of the action-mapping function depends on the state where the higher level command is considered, i.e. the same command must be translated differently according to its context. As a matter of fact, the action-mapping function can be imagined as some sort of context-sensitive grammar. This should not surprise the reader: ISAAC was designed to allow for a great degree of expressiveness, and this implies having to work with complex situations.

4.1.3 Query-mapping function

The final pairing elements between the models at the two levels of abstraction is the query-mapping function. As the lower level model is, ideally, the "only one existing", it must be possible to answer higher level queries by just looking at the lower level model. But higher level queries, i.e. interactions that do not modify the state, may require to alter the lower level state in order to be answered. For example, consider our narrative: the "read" query at the

higher level must, in the lower level, "open" and "close" the file. Therefore, the query-mapping function should not only return an answer to the higher level query based on the lower level model, but also the lower level state after the processing of the query, which may or may not have been modified in the process.
Similarly to what happens when answering queries at a single level of abstraction, the computation of a query answer may fail for either authorization or semantics reasons. The query-mapping function must therefore account also for those scenarios, and provide extra information in addition to just the result of the computation.

Definition 22 *A query-mapping function π defines how to answer higher level queries as a function of lower level actions.*
$\pi : \mathbb{S}_2 \times \mathbb{Q}_1 \times \mathbb{D}_1.UIDs \rightarrow \mathbb{D}_2.BOOLs \times \mathbb{D}_2.BOOLs \times (flatten(\mathbb{D}_1) \cup \{\perp\}) \times \mathbb{S}_2$
where:
the first boolean return value is the authorization success flag,
the second boolean return value is the semantics success flag,
the third return value is the answer of the query,
the final return value is the lower level state after the computation of the result.

The definition of a query-mapping function is a bit complicated and counter-intuitive. Its purpose will be clearer in the next subsection, when correctness of a refinement step is defined.

4.1.3.1 Example

In our example, the higher level model contains the single "read" application query and the two "auth-read" and "auth-write" authorization queries. One possible way to define the query-mapping function is:

```
1 π(s₂, read₁(fid), u₁) = {
2         <authFlag, semFlag, finalState> =
```

```
3                 T2(s2, open2(fid), u2(u1));
4         if (!(authFlag && semFlag)) {
5                 //User failed the open interaction
6                 return <authFlag, semFlag, ⊥, finalState>;
7         }
8         <authFlag, semFlag, result> =
9                 I2(finalState, read2(fid), u2(u1));
10        if (!(authFlag && semFlag)) {
11                //User failed the read interaction
12                finalState = T2(s2, close2(fid), u2(u1)).STATE;
13                return <authFlag, semFlag, ⊥, finalState>;
14        }
15        <authFlag, semFlag, finalState> =
16                T2(finalState, close2(fid), u2(u1));
17        if (!(authFlag && semFlag)) {
18                //User failed the close interaction
19                return <authFlag, semFlag, ⊥, finalState>;
20        }
21        return <TRUE, TRUE, result, finalState>;
22 }
23
24 π(s2, auth-read1(fid), u1) = {
25        return <I2(s2, auth-read2(fid), u2(u1)), s2>;
26 }
27
28 π(s2, auth-read1(fid, fc), u) = {
29        return <I2(s2, auth-write2(fid), u2(u1)), s2>;
30 }
```

where user $u_2(u_1)$ is the lower level version of user u_1.

4.2 Two-levels reasoning

After having presented the structure of the two-levels mode, this section focuses on what *correctness* means for a refinement, and how to verify it.
As shown in the previous section, there are three pairing elements between the two models: the state-mapping, the action-mapping and the query-mapping functions. These three elements are independently defined, but it is fundamental that their independent definitions do not introduce inconsistency with each other. More specifically, it is important that the state-mapping works consistently with both the action-mapping and query-mapping functions, i.e. the state-mapping preserves both the action-mapping and the query-mapping. Therefore, correctness is defined as the logical conjunction of two independent sub-properties: action-mapping preservation and query-mapping preservation.

Definition 23 *A two-levels refinement is* correct *if and only if both action-mapping preservation and query-mapping preservation are verified.*

4.2.1 Action-mapping preservation

Informally speaking, action-mapping preservation ensures that a higher level command is really equivalent to its lower level commands sequence counterpart. But a higher level command is, in general, mapped into an ordered sequence of lower level commands. In order to define action-mapping preservation, it is required to define a recursive version of the lower level transition function.

Definition 24 *A* recursive transition function $\mathbb{T}^*$ *is an extension to a non-recursive transition function* $\mathbb{T}$

$\mathbb{T}^* : \mathbb{S} \times (\mathbb{C} \times \mathbb{D}.UIDs)^* \rightarrow \mathbb{S}$

$$\mathbb{T}^*(s, list) = \left\{ \begin{array}{ll} s & if\ list = \emptyset \\ \mathbb{T}^*(\mathbb{T}(s, first(list)), rest(list)) & otherwise \end{array} \right\}$$

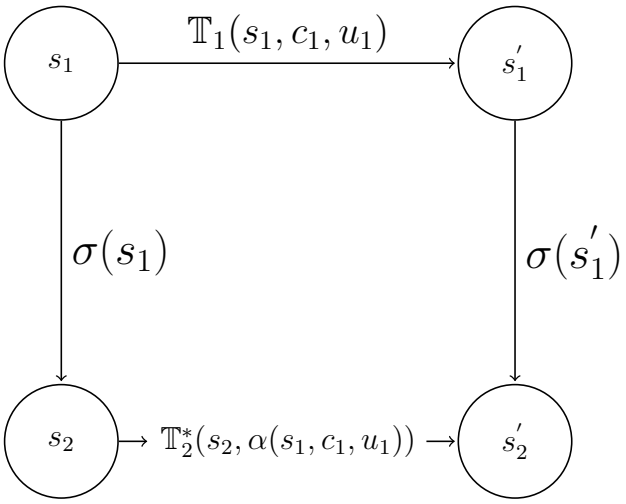

FIGURE 4.2: Action-mapping preservation: the two paths from s_1 to s_2' must be equivalent with respect to state-mapping

Definition 25 Action-mapping preservation *is verified in a two-levels refinement if and only if:*

$$\forall s : \mathbb{S}_1, c : \mathbb{C}_1$$
$$\sigma_{eq}(\sigma(\mathbb{T}_1(s, c)), \mathbb{T}_2{}^*(\sigma(s), \alpha(s, c)))$$

where σ_{eq} is the state-mapping equivalence relation.

A graphical representation of the property is shown in figure 4.2.

4.2.2 Action-mapping verification example

This subsection will illustrate how to verify, in a semi-formal way, action-mapping preservation for the two-levels scenario presented. With reference to the notation used in figure 4.2, it must be proved, for all states, commands and users, that:

$$\sigma_{eq}(\sigma(\mathbb{T}_1(s_1, c_1, u_1)), \mathbb{T}_2^*(\sigma(s_1), \alpha(s_1, c_1, u_1)))$$

In the specific case in analysis, the only available command is the "write" command. The action-mapping function will return, in all cases, a sequence

of open-write-close commands from the lower level version of the higher level user. These information allow, by reducing state-mapping equivalence to state equality and by ignoring the user element since it is the same on both sides of the equivalence and it does not interact with the other elements of the goal, to rewrite our target as:

$$\sigma(\mathbb{T}_1(s_1, \text{write(fid, fc)})) = \mathbb{T}_2^*(\sigma(s_1), \langle \text{open(fid)}, \text{write(fid, fc)}, \text{close(fc)} \rangle)$$

The state-mapping function will return, for every higher level state, a lower level state with the same users of the higher level state, the same files and file contents, all files unlocked, the same read-write rights and the possibility to open and close a file for each user who could either read or write the file.
The higher level transition function for the "write" command yields three different behaviors according the state content.

4.2.2.1 Case 1: Write is not authorized

In the case of a higher level write command failing the authorization check, the returned state is not modified. This rewrites the goal as:

$$\sigma(s_1) = \mathbb{T}_2^*(\sigma(s_1), \langle \text{open(fid)}, \text{write(fid, fc)}, \text{close(fc)} \rangle)$$

Since it is known that the user does not have write permissions, applying the lower level recursive transition function to the open-write-close sequence returns different scenarios, according to whether the user had read permissions to the file, and has therefore inherited open and close rights, or not, and according to whether the file exists or not:

- In the case the user has read permission and the file exists, the execution of the sequence locks and then unlocks the file but does not modify its content, as there is no write permission. This rewrites the target as $\sigma(s_1) = \sigma(s_1)$ which is trivially true.

- The case where the user has read permission but the file does not exist is impossible, as the authorization query for read permissions yields a negative result for non-existing files.
- In the case the user does not have read permission, the execution of the sequence does nothing, as the open and close operations are not permitted. This rewrites the target as $\sigma(s_1) = \sigma(s_1)$ which is trivially true.

In all the sub-cases action-mapping preservation is verified.

4.2.2.2 Case 2: Write is authorized, file does not exist

This scenario is simply impossible, as the authorization function for the write command yields a negative result for non-existing files.

4.2.2.3 Case 3: Write is authorized, file exists

In the case of a higher level write command for an existing file with write permission, the higher level state returned from the higher level transition function differs from the original only in the updated content of the file. This rewrites the goal as:

$$\sigma(s_1 \text{ with } \mathbb{X}.\text{CONTENT(fid)} : \text{fc}) =$$
$$\mathbb{T}_2^*(\sigma(s_1), \langle \text{open(fid)}, \text{write(fid, fc)}, \text{close(fc)} \rangle)$$

The presence of write permissions guarantees the inheritance of open and close permissions. Therefore, since the file exists, the execution of the open-write-close sequence is successful, and it alters the lower level state by modifying it according to the parameters of the write command. This rewrites the goal as:

$$\sigma(s_1 \text{ with } \mathbb{X}.\text{CONTENT(fid)} : \text{fc}) =$$
$$\sigma(s_1) \text{ with } \mathbb{X}.\text{CONTENT(fid)} : \text{fc}$$

which is true according to the definition of the state-mapping function. Therefore, action-mapping preservation is verified for all the sub-cases.

4.2.3 Query-mapping preservation

Informally speaking, query-mapping preservation guarantees that the answers to all the higher level queries of a given higher level state can be inferred by lower level queries of the equivalent lower level state, and that answering the query does not alter the lower level state "too much".

Definition 26 Query-mapping preservation *is verified in a two-levels refinement if and only if*

$$\begin{aligned}
&\forall s_1 : \mathbb{S}_1, q_1 : \mathbb{Q}_1, u_1 : \mathbb{D}_1.UIDs : \\
&\quad I_1(s_1, q_1, u_1) = \pi(\sigma(s_1), q_1, u_!).RESULT \\
&\quad \wedge \\
&\quad \sigma_{eq}(\sigma(s_1), \pi(\sigma(s_1), q_1, u_1).STATE)
\end{aligned}$$

where:

$\pi(\sigma(s), q, u).RESULT$ *represents the returned success flags and computed answer,*

$\pi(\sigma(s), q, u).STATE$ *is the returned state after the answer computation,*

σ_{eq} *is the state-mapping equivalence relation.*

The first part of the conjunction is intuitive: the answer derived in the higher-state must match the one derived in the correspondent lower level state. The second part is a bit trickier: the definition states that $\sigma(s)$ must be state-mapping equivalent to the state obtained after answering the query through query-mapping.

This constraint is required to guarantee that the behavior of the system is not affected by answering queries. State-mapping equivalence is a softer condition than requiring the two lower level states to be the same state: think of our

narrative example: a higher level "read" query issues the "open" and "close" commands which alter the lower level state. Luckily, the final state is exactly the same as the starting one. Now suppose to add in the lower level model some sort of logging facility which records the access to the file: now the final state is different from the starting one, namely for the added content in the log. But this difference does not affect the behavior of the system in the future. In general, the presence of logging facilities introduces extra elements of complexity. For example, if a logging facility is present, even an atomic "read" must be treated as a command, and the authorization failures of commands return states that are not the same as the starting ones. This pattern related to logging facilities is an element of interest, which should be examined further in future works.

4.2.4 Query-mapping verification example

This subsection will illustrate how to verify, in a semi-formal way, query-mapping preservation for the two-levels scenario presented. The higher level queries available are the "read" application query and the "auth-read" and "auth-write" authorization queries. In order to verify query-mapping preservation, it is better to start analyzing authorization queries, as they usually are simpler than application queries. Both "auth-read" and "auth-write" return FALSE for non-existing files and the answer contained in the access control matrix for existing files, at both levels of abstraction. The query-mapping function for the two authorization queries does not alter the lower level state in order to answer, therefore state-mapping equivalence is guaranteed. The higher level and lower level interpretation functions are identical for the two authorization queries and the query-mapping function simply calls the lower level version. This guarantees state-mapping equivalence, and therefore query-mapping preservation is verified for authorization queries.

The only available higher level application query is the "read" query. The goal

to be proved can be rewritten as:

$$I_1(s_1, \text{read(fid)}, u_1) = \pi(\sigma(s_1), \text{read(fid)}, u_1).\text{RESULT}$$
$$\wedge$$
$$\sigma_{eq}(\sigma(s_1), \pi(\sigma(s_1), \text{read(fid)}, u_1).\text{STATE})$$

The higher level interpretation function for the "read" query yields three different behaviors according the state content.

4.2.4.1 Case 1: Read is not authorized

In the case of a higher level read query failing the authorization check, the goal can be rewritten as:

$$\langle \text{FALSE}, \text{FALSE}, \perp \rangle = \pi(\sigma(s_1), \text{read(fid)}, u_1).\text{RESULT}$$
$$\wedge$$
$$\sigma_{eq}(\sigma(s_1), \pi(\sigma(s_1), \text{read(fid)}, u_1).\text{STATE})$$

The query-mapping function returns different scenarios, according to whether the user had write permissions on the file, and has therefore inherited open and close rights, or not, and according to whether the file exists or not:

- In the case the user has write permission and the file exists, the read action will fail after opening the file, returning an authorization failure result. The returned state is not modified, as the user has the right to close the file before completing the query-mapping sequence. This rewrites the goal as:

  $$\langle \text{FALSE}, \text{FALSE}, \perp \rangle = \langle \text{FALSE}, \text{FALSE}, \perp \rangle$$
  $$\wedge$$
  $$\sigma_{eq}(\sigma(s_1), \sigma(s_1))$$

  which is trivially true.

- The case where the user has write permission but the file does not exist is impossible, as the authorization query for read permissions yields a negative result for non-existing files.
- In the case the user does not have write permission, the open action will fail, as the open and close operations are not permitted since both write and read permissions are lacking. This will return an authorization failure result, allowing to rewrite the goal as:

$$\langle \text{FALSE}, \text{FALSE}, \perp \rangle = \langle \text{FALSE}, \text{FALSE}, \perp \rangle$$
$$\wedge$$
$$\sigma_{eq}(\sigma(s_1), \sigma(s_1))$$

  which is trivially true.

In all the sub-cases query-mapping preservation is verified.

4.2.4.2 Case 2: Read is authorized, file does not exist

This scenario is simply impossible, as the authorization function for the write command yields a negative result for non-existing files.

4.2.4.3 Case 3: Read is authorized, file exists

In the case of an existing file with read rights, the higher level interpretation function will be successful. The state-mapping function, according to its definition, will copy the file content from the higher level to the lower level. The query-mapping function will successfully complete as all the permissions to open, read and close the file are available. Since the open-read-close sequence is correctly executed, the final state is not modified. According to all these

information, the goal can be rewritten as:

$$s_1.\mathbb{X}.\text{CONTENT}(\text{fid}) = s_1.\mathbb{X}.\text{CONTENT}(\text{fid})$$
$$\wedge$$
$$\sigma_{eq}(\sigma(s_1), \sigma(s_1))$$

which is trivially true. Therefore, query-mapping preservation is verified for all the sub-cases.

Case study: hospital scenario 5

After having presented the various working modes of ISAAC using a simple narrative example, this chapter explores a more complex case study in order to demonstrate the usage of ISAAC in a realistic application.

In this chapter the modeling of a part of an hospital system is described. The domain knowledge on the topic was acquired by interviewing members of the medical staff. Since the actual details and procedures may vary a lot from hospital to hospital, or even from ward to ward within the same hospital, this case study was developed trying to focus only on the common aspects. This choice required to abstract away some of the details specific to each different location, but the end result is realistic and generic enough, as confirmed by the medical staff.

Since the goal of this case study is to show the potential of ISAAC in a real application, it was not necessary to model the entire system of an hospital structure, i.e. including the payroll system, work insurances, etc., as many subsystems share more or less the same structures and offer more or less the same insights. In order to avoid modeling many similar subsystems, the case study will focus only on modeling the subsystem related to the relationships

among physicians, nurses and patients, with respect to the prescription of therapies and exams.

The first section of the chapter will describe informally the various elements of the system in analysis.
This knowledge will then be applied in the second section of the chapter to the construction of a first model of the system.
In the third section, the model created will be refined by introducing the possibility of inter-ward consultancy.
Finally, a two-levels analysis of the scenario will be performed, demonstrating the correctness of the refinement.
One-level analysis will be neglected due to the lack of recursive calls among interactions in the model.

5.1 Informal descriptions

This section describes, informally, the elements of the system in analysis. Some aspects, such as the possibility for doctors to alter the personal data of patients, may appear counter-intuitive, while others, such as the assumption that patients belong to a single ward, may be oversimplifying. The information presented was acquired by interviewing actual medical staff, while the assumptions made come from the tentative of creating a general model that may represent the many different policies applied by different hospitals.

5.1.1 Wards

According to the information acquired interviewing the medical staff, the first relevant element of an hospital system is the division in wards. Wards are abstract organizations, usually physically separated in different buildings, that focus on different areas of medicine. For example, in an hospital may be present the pediatric ward, the oncology ward and the cardiology ward. For the purposes of the case study, the physical location of wards is not relevant and will be ignored. Doctors, nurses and patients are all belonging to a ward, according to their specialization, for the medical staff, or to the pathology that caused their admission to the hospital, for the patients.
Wards are an element of separation, in the sense that, normally, a doctor of ward A cannot modify the data of a patient of ward B. In the case study, the first assumption is that wards represent a partition of the union of the sets of doctors, nurses and patients, i.e. doctors, nurses and patients all belong to exactly one ward. This is usually true for the medical staff, as most of the medical staff has at most one specialization, and can be more or less true for the patients according to the way they are treated in each different hospital.

5.1.2 Patients

Patients are the ill peoples that are being taken care of in the hospital. In the model, patients are assigned to exactly one ward, which is assigned to them at the time of their admittance in the hospital, according to their pathology. Patients have associated two information sources. The first one, which will be called "personal data", contains the personal data of the patients such as, but not limited to, date of birth, sex, medical history previous to the admittance, and telephone number. The second one, which will be called "clinic diary", contains all the medical information related to exams, therapies and check-ups performed during the current stay at the hospital. In most of the real hospitals, the clinic diary only contains the summary of the therapies and is associated with another document that contains the detailed information regarding doses and application times etc. This distinction is not relevant to the goal of the case study and will be ignored.

5.1.3 Therapies

In this case study, the term "therapy" will be used to describe all sort of prescriptions assigned to a patient, both chemical, i.e. medicines, and of other nature, such as physiotherapy sessions. Therapies are registered in the clinic diary of a patient and can only be prescribed by doctors.

5.1.4 Exams

Exams are roughly separable in two categories: laboratory exams, such as blood or urine tests, and instrumental exams, such as NMR's. This distinction is very important: in most of the real hospitals, laboratory exams can be prescribed to patients by both nurses and doctors, while instrumental exams, being more costly and risky in certain cases, can only be prescribed by doctors. Exams are registered in the clinic diary of a patient.

5.1.5 Check-ups

During their period in the hospital, patients will often get controlled by doctors. The outcome of these check-ups is a fundamental piece of information and must be registered for future reference. Check-ups outcomes are registered in the clinic diary of a patient.

5.1.6 Nurses

Nurses are that part of the medical staff that is assigned to the care of the patients and to actually executing the prescribed therapies. Nurses cannot, in general, alter the therapies of a patient, but they can prescribe laboratory exams. In addition, they can also alter the personal data of a patient. This is allowed in order to make it easy to keep the personal data always up to date, since most of them are easily mutable, such as the address or the telephone number.

5.1.7 Doctors

Doctors are that part of the medical staff that is in charge of diagnosing the pathology of a patient and prescribe therapies. They can modify the therapies of a patient, register check-ups outcomes and prescribe all sort of exams. They can also modify the personal data of a patient, for the same reason of nurses.

5.2 Basic model

This section will present a first ISAAC model of the system in analysis. The first draft of the model will impose that doctor and nurses only have access to the patients of their own wards, i.e. it is impossible for a doctor of ward A to visit a patient of ward B. This constraint will be softened in the next section, where the refinement will introduce the possibility for inter-ward consulting.

5.2.1 Domains

To the goal of the case study, doctors and nurses can be grouped under the same "staff" set. Therefore, the data types present in the subsystem in analysis are:

- WARDs: the ID's of the wards. Can be modeled using the set of natural numbers.
- STAFFs: the ID's of nurses and doctors. Can be modeled using the set of natural numbers.
- PATIENTs: the ID's of patients. Can be modeled using the set of natural numbers.
- PDATAs: the information container ID's of the personal data of a patient. Can be modeled as the set of natural language strings.
- CDIARYs: the information container ID's of the medical data of a patient. Can be modeled using the set of natural numbers.
- THERAPYs: the description of a therapy. Can be modeled as the set of natural language strings.
- EXAMs: the ID's of an exams. Can be modeled using the set of natural numbers.

- CHECKUPs: the outcome of a checkup. Can be modeled as the set of natural language strings.
- DATEs: a calendar date, used to record start and end date of therapies or exams. It is a well-formed date string.

A very important thing to notice is that most of the data types are modeled using the same basic set of natural numbers or strings. This is not a problem, as the information of each datum is not only in its content, but also in its type. All these elements, together with the UIDs and BOOLs set, compose the domain set of the model. In the scenario in analysis, the users of the system are the staff of the hospital, as the case study focuses on the modification of the personal data and clinic diary of patients. Therefore, UIDs = STAFFs.

$$
\begin{aligned}
\mathbb{D} = \{ & \\
& \text{UIDs, BOOLs, WARDs, STAFFs,} \\
& \text{PATIENTs, PDATAs, CDIARYs,} \\
& \text{THERAPYs, EXAMs, CHECKUPs, DATEs} \\
\} &
\end{aligned}
$$

5.2.2 Users

Since the case study is modeling a generic hospital, and not a specific instance, the users set must not be defined by describing its elements, but rather by specifying its structure. In the model, the users of the system are the medical staff members.

$$\mathbb{U} \subseteq \mathbb{D}.\text{UIDs}$$

$$\mathbb{D}.\text{UIDs} = \mathbb{D}.\text{STAFFs}$$

5.2.3 Application data

The application data must be designed so that it can store the current state of the system. In particular, it must store the doctors, nurses and patients that are currently in the hospital. It must also be possible to distinguish between nurses and doctors.

In addition to the patients and the staff, the application data must also store the information regarding personal data and clinic diary of patients. The former can be represented as a mapping from patients to PDATAs, while the latter is a mapping from patients to CDIARYs. Application data is also required to have a mapping from the clinic diaries of patients to the elements of a clinic diary, i.e. check-ups, exams and therapies. Therapies should have a start and end date, while exams and check-ups should only register the date when they were performed.

$$
\begin{aligned}
\mathbb{X} = \{ & \\
& \text{WARDS, STAFFS, ISDOCTOR, ISNURSE,} \\
& \text{PATIENTS, CDIARYS, GETWARDS, GETPDATA, GETCDIARY,} \\
& \text{GETTHERAPYS, EXAMS, ISLABEXAM, ISINSTEXAM,} \\
& \text{GETEXAMS, GETCHECKUPS} \\
\} &
\end{aligned}
$$

where:

$\text{WARDS} \subseteq \mathbb{D}.\text{WARDs}$,

$\text{STAFFS} \subseteq \mathbb{D}.\text{STAFFs}$,

$\text{ISDOCTOR} : \text{STAFFS} \to \mathbb{D}.\text{BOOLs}$,

$\text{ISNURSE} : \text{STAFFS} \to \mathbb{D}.\text{BOOLs}$,

$\forall s \in \text{STAFFS} : \text{ISNURSE}(s) \oplus \text{ISDOCTOR}(s)$,

$\text{PATIENTS} \subseteq \mathbb{D}.\text{PATIENTs}$,

$\text{CDIARYS} \subseteq \mathbb{D}.\text{CDIARYs}$,

$\text{GETWARD} : \text{PATIENTS} \cup \text{STAFF} \to \text{WARDS}$,

GETPDATA : PATIENTS $\rightarrow \mathbb{D}$.PDATAs,

GETCDIARY : PATIENTS $\rightarrow$ CDIARYS,

GETTHERAPYS : CDIARYS $\rightarrow (\mathbb{D}.\text{THERAPYs} \times \mathbb{D}.\text{DATEs} \times \mathbb{D}.\text{DATEs})^*$,

EXAMS $\subseteq \mathbb{D}$.EXAMs,

ISLABEXAM : EXAMS $\rightarrow \mathbb{D}$.BOOLs,

ISINSTEXAM : EXAMS $\rightarrow \mathbb{D}$.BOOLs,

$\forall e \in$ EXAMS : ISLABEXAM$(e) \oplus$ ISINSTEXAM(e),

GETEXAMS : CDIARYS $\rightarrow (\mathbb{D}.\text{EXAMs} \times \mathbb{D}.\text{DATEs})^*$,

GETCHECCKUPS : CDIARYS $\rightarrow (\mathbb{D}.\text{CHECKUPs} \times \mathbb{D}.\text{DATEs})^*$,

5.2.4 Authorization data

In order to simplify the case study and to focus the attention of the reader on the aspects related to the proof of correctness of the system, rather than on the details of the formalization of commercial grade access control systems, the case study will approach a logic-based access control system that implements an immutable authorization policy by neglecting authorization data structures, a possibility already mentioned in Chapter 3.

5.2.5 Commands

The commands of interest with respect to the subsystem in analysis are related to the modification of the personal data and the clinic diary of patients. A full hospital model should also consider the operations related to the admission and discharge of patients, hiring and firing of staff, change of wards of patients and staff etc. Since all these extra commands would behave very similarly to the ones included in the case study and would not bring additional insight, the case study will ignore them.

More precisely, according to the application data structures described, there are four commands available to the hospital staff:

- The command used to modify the personal data of a patient.

- The command used to modify the therapies present in the clinic diary of a patient.
- The command used to modify the exams present in the clinic diary of a patient.
- The command used to modify the check-ups present in the clinic diary of a patient.

Therefore, the set of commands can be formalized as:

$$
\begin{aligned}
\mathbb{C} = \{ & \\
& \langle \text{MODIFYPDATA}, \mathbb{D}.\text{PATIENTs} \times \mathbb{D}.\text{PDATAs} \rangle, \\
& \langle \text{ADDTHERAPY}, \mathbb{D}.\text{PATIENTs} \times (\mathbb{D}.\text{THERAPYs} \times \mathbb{D}.\text{DATEs} \times \mathbb{D}.\text{DATEs}) \rangle, \\
& \langle \text{ADDEXAM}, \mathbb{D}.\text{PATIENTs} \times (\mathbb{D}.\text{EXAMs} \times \mathbb{D}.\text{DATEs}) \rangle, \\
& \langle \text{ADDCHECKUP}, \mathbb{D}.\text{PATIENTs} \times (\mathbb{D}.\text{CHECKUPs} \times \mathbb{D}.\text{DATEs}) \rangle \\
\} &
\end{aligned}
$$

Note that, for the sake of simplicity, the command to modify personal data has been designed in such a way to overwrite the previous content of the application data structure. This choice was taken in order to reduce the number of commands available and simplifying the proofs required to demonstrate system properties.

5.2.6 Queries

The application queries available to the staff of the hospital are all those needed to analyze the system state contained in the application data structure. These

are:

$\mathbb{Q}_{\mathbb{X}} = \{$

$\langle \text{ISDOC}, \mathbb{D}.\text{STAFFs}, \mathbb{D}.\text{BOOLs} \rangle,$

$\langle \text{ISNURSE}, \mathbb{D}.\text{STAFFs}, \mathbb{D}.\text{BOOLs} \rangle,$

$\langle \text{ISLABEXAM}, \mathbb{D}.\text{EXAMs}, \mathbb{D}.\text{BOOLs} \rangle,$

$\langle \text{ISINSTEXAM}, \mathbb{D}.\text{EXAMs}, \mathbb{D}.\text{BOOLs} \rangle,$

$\langle \text{GETPDATA}, \mathbb{D}.\text{PATIENTs}, \mathbb{D}.\text{PDATAs} \rangle,$

$\langle \text{GETWARD}, \mathbb{D}.\text{PATIENTs} \cup \mathbb{D}.\text{STAFFs}, \mathbb{D}.\text{WARDs} \rangle,$

$\langle \text{GETTHERAPIES}, \mathbb{D}.\text{PATIENTs}, (\mathbb{D}.\text{THERAPYs} \times \mathbb{D}.\text{DATEs} \times \mathbb{D}.\text{DATEs})^* \rangle,$

$\langle \text{GETEXAMS}, \mathbb{D}.\text{PATIENTs}, (\mathbb{D}.\text{EXAMs} \times \mathbb{D}.\text{DATEs})^* \rangle,$

$\langle \text{GETCHECKUPS}, \mathbb{D}.\text{PATIENTs}, (\mathbb{D}.\text{CHECKUPs} \times \mathbb{D}.\text{DATEs})^* \rangle$

$\}$

Additionally, the model includes the following authorization queries:

$\mathbb{Q}_{\mathbb{A}} = \{$

Commands:

$\langle \text{AUTH-MODIFYPDATA}, \mathbb{D}.\text{PATIENTs} \times \mathbb{D}.\text{PDATAs} \rangle,$

$\langle \text{AUTH-ADDTHERAPY}, \mathbb{D}.\text{PATIENTs} \times (\mathbb{D}.\text{THERAPYs} \times \mathbb{D}.\text{DATEs} \times \mathbb{D}.\text{DATEs} \rangle,$

$\langle \text{AUTH-ADDEXAM}, \mathbb{D}.\text{PATIENTs} \times (\mathbb{D}.\text{EXAMs} \times \mathbb{D}.\text{DATEs}) \rangle,$

$\langle \text{AUTH-ADDCHECKUP}, \mathbb{D}.\text{PATIENTs} \times (\mathbb{D}.\text{CHECKUPs} \times \mathbb{D}.\text{DATEs}) \rangle,$

App. Queries:

⟨AUTH-ISDOC, $\mathbb{D}$.STAFFs⟩,

⟨AUTH-ISNURSE, $\mathbb{D}$.STAFFs⟩,

⟨AUTH-ISLABEXAM, $\mathbb{D}$.EXAMs⟩,

⟨AUTH-ISINSTEXAM, $\mathbb{D}$.EXAMs⟩,

⟨AUTH-GETPDATA, $\mathbb{D}$.PATIENTs⟩,

⟨AUTH-GETWARD, $\mathbb{D}$.PATIENTs $\cup$ $\mathbb{D}$.STAFFs⟩,

⟨AUTH-GETTHERAPIES, $\mathbb{D}$.PATIENTs⟩,

⟨AUTH-GETEXAMS, $\mathbb{D}$.PATIENTs⟩,

⟨AUTH-GETCHECKUPS, $\mathbb{D}$.PATIENTs⟩

}

5.2.7 Transition function

The transition function for the specified command is rather simple and intuitive:

```
1 T(s, modifypdata(p, data), u) = {
2         if (!I(s, auth-modifypdata(p, data), u)) {
3                 //Authorization failure
4                 return <FALSE, FALSE, s>;
5         }
6         if (!s.X.PATIENTS.contains(p)) {
7                 //Semantic failure: patient does not exist
8                 return <TRUE, FALSE, s>;
9         }
10        //Valid request: can modify, patient exists
11        s.X.GETPDATA(p) = data;
12        return <TRUE, TRUE, s>;
13 }
14
```

```
15 T(s, addtherapy(p, t), u) = {
16         if (!I(s, auth-addtherapy(p, t), u)) {
17                 //Authorization failure
18                 return <FALSE, FALSE, s>;
19         }
20         if (!s.X.PATIENTS.contains(p)) {
21                 //Semantic failure: patient does not exist
22                 return <TRUE, FALSE, s>;
23         }
24         //Valid request: can add therapy, patient exists
25         s.X.GETTHERAPYS(s.X.GETCDIARY(p)).append(t);
26         return <TRUE, TRUE, s>;
27 }
28
29 T(s, addexam(p, e), u) = {
30         if (!I(s, auth-addexam(p, e), u)) {
31                 //Authorization failure
32                 return <FALSE, FALSE, s>;
33         }
34         if (!s.X.PATIENTS.contains(d)) {
35                 //Semantic failure: patient does not exist
36                 return <TRUE, FALSE, s>;
37         }
38         //Valid request: can add exam, patient exists
39         s.X.GETEXAMS(s.X.GETCDIARY(p)).append(e);
40         return <TRUE, TRUE, s>;
41 }
42
43 T(s, addcheckup(p, c), u) = {
44         if (!I(s, auth-addtherapy(p, c), u)) {
45                 //Authorization failure
46                 return <FALSE, FALSE, s>;
47         }
```

```
48          if (!s.X.PATIENTS.contains(p)) {
49                  //Semantic failure: patient does not exist
50                  return <TRUE, FALSE, s>;
51          }
52          //Valid request: can add checkup, patient exists
53          s.X.GETCHECKUPS(s.X.GETCDIARY(p)).append(c);
54          return <TRUE, TRUE, s>;
55  }
```

5.2.8 Interpretation function

The interpretation function for application queries is pretty trivial: it simply checks the content of the system state and returns it as a result. The interpretation function for authorization queries, on the other hand, depends on the access control system chosen. In order to simplify the case study and to focus the attention of the reader on the aspects related to the proof of correctness of the system, rather than on the details of the formalization of commercial grade access control systems, the case study will approach a logic-based access control system that implements an immutable authorization policy by neglecting authorization data structures, a possibility already mentioned in Chapter 3. Therefore, the interpretation function for the application queries can be defined as:

```
1 I(s, isdoc(s), u) = {
2           if (!I(s, auth-isdoc(s), u)) {
3                   //Authorization failure
4                   return <FALSE, FALSE, ⊥>;
5           }
6           if (!s.X.STAFFS.contains(s)) {
7                   //Semantic failure: staff member does not exist
8                   return <TRUE, FALSE, ⊥>;
9           }
10          //Valid request: can answer, staff member exists
```

```
11          return <TRUE, TRUE, s.X.ISDOC(s)>;
12  }
13
14  I(s, isnurse(s), u) = {
15          if (!I(s, auth-isnurse(s), u)) {
16                  //Authorization failure
17                  return <FALSE, FALSE, ⊥>;
18          }
19          if (!s.X.STAFFS.contains(s)) {
20                  //Semantic failure: staff member does not exist
21                  return <TRUE, FALSE, ⊥>;
22          }
23          //Valid request: can answer, staff member exists
24          return <TRUE, TRUE, s.X.ISNURSE(s)>;
25  }
26
27  I(s, islabexam(e), u) = {
28          if (!I(s, auth-islabexam(e), u)) {
29                  //Authorization failure
30                  return <FALSE, FALSE, ⊥>;
31          }
32          if (!s.X.EXAMS.contains(e)) {
33                  //Semantic failure: exam does not exist
34                  return <TRUE, FALSE, ⊥>;
35          }
36          //Valid request: can answer, exam exists
37          return <TRUE, TRUE, s.X.ISLABEXAM(e)>;
38  }
39
40  I(s, isinsexam(e), u) = {
41          if (!I(s, auth-isinsexam(e), u)) {
42                  //Authorization failure
43                  return <FALSE, FALSE, ⊥>;
```

```
44          }
45          if (!s.X.EXAMS.contains(e)) {
46                  //Semantic failure: exam does not exist
47                  return <TRUE, FALSE, ⊥>;
48          }
49          //Valid request: can answer, exam exists
50          return <TRUE, TRUE, s.X.ISINSEXAM(e)>;
51  }
52
53  I(s, getpdata(p), u) = {
54          if (!I(s, auth-getpdata(p), u)) {
55                  //Authorization failure
56                  return <FALSE, FALSE, ⊥>;
57          }
58          if (!s.X.PATIENTS.contains(p)) {
59                  //Semantic failure: patient does not exist
60                  return <TRUE, FALSE, ⊥>;
61          }
62          //Valid request: can answer, patient exists
63          return <TRUE, TRUE, s.X.GETPDATA(p)>;
64  }
65
66  I(s, getward(s), u) = {
67          if (!I(s, auth-getward(s), u)) {
68                  //Authorization failure
69                  return <FALSE, FALSE, ⊥>;
70          }
71          if (!s.X.PATIENTS.contains(s) && !s.X.STAFFS.contains(s
                )) {
72                  //Semantic failure: subject does not exist
73                  return <TRUE, FALSE, ⊥>;
74          }
75          //Valid request: can answer, subject exists
```

```
76          return <TRUE, TRUE, s.X.GETWARD(s)>;
77  }
78
79  I(s, gettherapies(p), u) = {
80          if (!I(s, auth-gettherapies(p), u)) {
81                  //Authorization failure
82                  return <FALSE, FALSE, ⊥>;
83          }
84          if (!s.X.PATIENTS.contains(p)) {
85                  //Semantic failure: patient does not exist
86                  return <TRUE, FALSE, ⊥>;
87          }
88          //Valid request: can answer, patient exists
89          return <TRUE, TRUE, s.X.GETTHERAPYS(s.X.GETCDIARY(p))>;
90  }
91
92  I(s, getexams(p), u) = {
93          if (!I(s, auth-getexams(p), u)) {
94                  //Authorization failure
95                  return <FALSE, FALSE, ⊥>;
96          }
97          if (!s.X.PATIENTS.contains(p)) {
98                  //Semantic failure: patient does not exist
99                  return <TRUE, FALSE, ⊥>;
100         }
101         //Valid request: can answer, patient exists
102         return <TRUE, TRUE, s.X.GETEXAMS(s.X.GETCDIARY(p))>;
103 }
104
105 I(s, getcheckups(p), u) = {
106         if (!I(s, auth-getcheckups(p), u)) {
107                 //Authorization failure
108                 return <FALSE, FALSE, ⊥>;
```

```
109         }
110         if (!s.X.PATIENTS.contains(p)) {
111                 //Semantic failure: patient does not exist
112                 return <TRUE, FALSE, ⊥>;
113         }
114         //Valid request: can answer, patient exists
115         return <TRUE, TRUE, s.X.GETCHECKUPS(s.X.GETCDIARY(p))>;
116 }
```

Authorization queries, instead, will not access any sort of authorization data structure, as it has already been mentioned. The definition of the interpretation function for authorization queries for commands is given by:

```
1  I(s, auth-modifypdata(p, d), u) = {
2         if (!s.X.PATIENTS.contains(p)) {
3                 //Patient does not exist
4                 return FALSE;
5         }
6         //Patient exists
7         if (!I(s, auth-getward(p), u) || !I(s, auth-getward(u),
              u)) {
8                 //Cannot verify wards
9                 return FALSE;
10        }
11        //Patient exists, can verify wards
12        return I(s, getward(p), u).RESULT == I(s, getward(u), u
              ))
13 }
14
15 I(s, auth-addtherapy(p, t), u) = {
16        if (!s.X.PATIENTS.contains(p)) {
17                //Patient does not exist
18                return FALSE;
19        }
```

```
20          //Patient exists
21          return (s.X.GETWARD(p) == s.X.GETWARD(u)) && s.X.
                ISDOCTOR(u)
22  }
23
24  I(s, auth-addexam(p, e), u) = {
25          if (!s.X.PATIENTS.contains(p)) {
26                  //Patient does not exist
27                  return FALSE;
28          }
29          //Patient exists
30          if (s.X.ISLABEXAM(e)) {
31                  return s.X.GETWARD(p) == s.X.GETWARD(u)
32          }
33          else {
34                  return (s.X.GETWARD(p) == s.X.GETWARD(u)) && s.
                        X.ISDOCTOR(u)
35          }
36  }
37
38  I(s, auth-addcheckup(p, c), u) = {
39          if (!s.X.PATIENTS.contains(p)) {
40                  //Patient does not exist
41                  return FALSE;
42          }
43          //Patient exists
44          return (s.X.GETWARD(p) == s.X.GETWARD(u)) && s.X.
                ISDOCTOR(u)
45  }
```

While the interpretation function for authorization queries related to application queries is given by:

```
1  I(s, auth-isdoc(s), u) = {
```

```
2         return TRUE
3 }
4
5 I(s, auth-isnurse(s), u) = {
6         return TRUE
7 }
8
9 I(s, auth-islabexam(e), u) = {
10        return TRUE
11 }
12
13 I(s, auth-isinsexam(e), u) = {
14        return TRUE
15 }
16
17 I(s, auth-getpdata(p), u) = {
18        if (!s.X.PATIENTS.contains(p)) {
19                //Patient does not exist
20                return FALSE
21        }
22        //Patient exists
23        return s.X.GETWARD(p) == s.X.GETWARD(u)
24 }
25
26 I(s, auth-getward(s), u) = {
27        return TRUE
28 }
29
30 I(s, auth-gettherapies(p), u) = {
31        if (!s.X.PATIENTS.contains(p)) {
32                //Patient does not exist
33                return FALSE
34        }
```

```
35          //Patient exists
36          return s.X.GETWARD(p) == s.X.GETWARD(u)
37  }
38
39  I(s, auth-getexams(p), u) = {
40          if (!s.X.PATIENTS.contains(p)) {
41                  //Patient does not exist
42                  return FALSE
43          }
44          //Patient exists
45          return s.X.GETWARD(p) == s.X.GETWARD(u)
46  }
47
48  I(s, auth-getcheckups(p), u) = {
49          if (!s.X.PATIENTS.contains(p)) {
50                  //Patient does not exist
51                  return FALSE
52          }
53          //Patient exists
54          return s.X.GETWARD(p) == s.X.GETWARD(u)
55  }
```

5.3 Refined model

This section will present a refined model of the system in analysis. The first draft of the model imposed that the medical staff could only access and modify data of patients of their own ward. The refinement introduces the possibility for staff members to require a consult with another staff member over a patient. As long as the consult is active, the consultant doctor can access and modify the patient data even if the patient is not of his same ward.
The consulting procedure works as follow: staff member s_1 issues a consulting request to staff member s_2 for one of his patients. Staff member s_2 can either reject or accept the request: if rejected, the request is removed and nothing happens. If accepted, the request becomes an active consulting. This active consulting exists as long as either s_1 or s_2 removes it.
For the sake of brevity, the specification of the lower level model will be given by specifying only the differences from the previous model.

5.3.1 Application data

The application data must now be able to store information about consulting requests and active consulting. Therefore, $\mathbb{X}$ now contains two additional sets:

$$\mathbb{X} = \{\ldots, \text{CONSREQ}, \text{ACTIVECONS}\}$$

where:
$\text{CONSREQ} \subseteq \text{STAFFS} \times \text{STAFFS} \times \text{PATIENTS}$,
$\text{ACTIVECONS} \subseteq \text{STAFFS} \times \text{STAFFS} \times \text{PATIENTS}$

5.3.2 Commands

The additional commands must allow the staff members to issue consulting requests, accept consulting requests and delete both consulting requests and

active consulting:

$$\mathbb{C} = \{\ldots,$$
$$\langle \text{CREATECONSREQ}, \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{PATIENTs} \rangle,$$
$$\langle \text{ACCEPTCONSREQ}, \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{PATIENTs} \rangle,$$
$$\langle \text{DELETECONS}, \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{PATIENTs} \rangle,$$
$$\}$$

5.3.3 Queries

Queries must now allow staff members to retrieve the consulting requests and active consulting they are involved with, so that they can then be used as parameters for the consulting commands:

$$\mathbb{Q}_{\mathbb{X}} = \{\ldots,$$
$$\langle \text{GETCONSREQS}, \mathbb{D}.\text{STAFFs}, (\mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{PATIENTs})^* \rangle,$$
$$\langle \text{GETACTIVECONS}, \mathbb{D}.\text{STAFFs}, (\mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{PATIENTs})^* \rangle,$$
$$\}$$

In addition to those, the model now includes the authorization queries for the new commands and application queries:

$$\mathbb{Q}_{\mathbb{A}} = \{\ldots,$$

Commands:

$$\langle \text{AUTH-CREATECONSREQ}, \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{PATIENTs} \rangle,$$
$$\langle \text{AUTH-ACCEPTCONSREQ}, \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{PATIENTs} \rangle,$$
$$\langle \text{AUTH-DELETECONS}, \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{STAFFs} \times \mathbb{D}.\text{PATIENTs} \rangle,$$

Application queries:

$$\langle \text{GETCONSREQS}, \mathbb{D}.\text{STAFFs} \rangle,$$
$$\langle \text{GETACTIVECONS}, \mathbb{D}.\text{STAFFs} \rangle,$$
$$\}$$

5.3.4 Transition function

The refined transition function only differs from the previous with respect to the new commands:

```
1 T(s, createconsreq(s1, s2, p), u) = {
2         if (!I(s, auth-createconsreq(s1, s2, p), u)) {
3                 //Authorization failure
4                 return <FALSE, FALSE, s>;
5         }
6         if (!s.X.PATIENTS.contains(p) || !s.X.STAFFS.contains(
              s1) || !s.X.STAFFS.contains(s2)) {
7                 //Semantic failure: one of the parameter does
                      not exist
8                 return <TRUE, FALSE, s>;
9         }
10        //Valid request: can issue, parameters exist
11        s.X.CONSREQS.add(<s1, s2, p>);
12        return <TRUE, TRUE, s>;
13 }
14
15 T(s, acceptconsreq(s1, s2, p), u) = {
16        if (!I(s, auth-acceptconsreq(s1, s2, p), u)) {
17                //Authorization failure
18                return <FALSE, FALSE, s>;
19        }
20        if (!s.X.CONSREQS.contains(<s1, s2, p>)) {
21                //Semantic failure: the request does not exist
22                return <TRUE, FALSE, s>;
23        }
24        //Valid request: can issue, request exists
25        s.X.CONSREQ.remove(<s1, s2, p>);
26        s.X.ACTIVECONS.add(<s1, s2, p>);
27        return <TRUE, TRUE, s>;
```

```
28 }
29
30 T(s, deletecons(s1, s2, p), u) = {
31         if (!I(s, auth-deletecons(s1, s2, p), u)) {
32                 //Authorization failure
33                 return <FALSE, FALSE, s>;
34         }
35         if (!s.X.CONSREQS.contains(<s1, s2, p>) || !s.X.
               ACTIVEREQS.contains(<s1, s2, p>)) {
36                 //Semantic failure: the request does not exist
37                 return <TRUE, FALSE, s>;
38         }
39         //Valid request: can issue, request exists
40         s.X.CONSREQ.remove(<s1, s2, p>);
41         s.X.ACTIVECONS.remove(<s1, s2, p>);
42         return <TRUE, TRUE, s>;
43 }
```

5.3.5 Interpretation function

The refined interpretation function for the new application queries is given by:

```
1 I(s, getconsreq(s), u) = {
2         if (!I(s, auth-getconsreq(s), u)) {
3                 //Authorization failure
4                 return <FALSE, FALSE, ⊥>;
5         }
6         if (!s.X.STAFFS.contains(s)) {
7                 //Semantic failure: staff member does not exist
8                 return <TRUE, FALSE, ⊥>;
9         }
10        //Valid request: can answer, staff member exists
11        return <TRUE, TRUE, s.X.CONSREQS.where(s1 == s || s2 ==
              s)>;
```

```
}

I(s, getactivecons(s), u) = {
        if (!I(s, auth-getconsreq(d), u)) {
                //Authorization failure
                return <FALSE, FALSE, ⊥>;
        }
        if (!s.X.STAFFS.contains(s)) {
                //Semantic failure: staff member does not exist
                return <TRUE, FALSE, ⊥>;
        }
        //Valid request: can answer, staff member exists
        return <TRUE, TRUE, s.X.ACTIVECONS.where(s1 == s || s2
            == s)>;
}
```

While the interpretation function for the authorization queries of the new commands is given by:

```
I(s, auth-createconsreq(s1, s2, p), u) = {
        if (!s.X.STAFFS.contains(s1) || !s.X.STAFFS.contains(s2
            ) || !s.X.PATIENTS.contains(p) ) {
                //One of the parameters does not exist
                return FALSE;
        }
        //Parameters are valid
        return u == s1 && s.X.GETWARD(p) == s.X.GETWARD(s1);
}

I(s, auth-accept(s1, s2, p), u) = {
        if (!s.X.STAFFS.contains(s1) || !s.X.STAFFS.contains(s2
            ) || !s.X.PATIENTS.contains(p) ) {
                //One of the parameters does not exist
                return FALSE;
```

```
14          }
15          //Parameters are valid
16          return u == s2;
17  }
18
19  I(s, auth-deleteconsreq(s1, s2, p), u) = {
20          if (!s.X.STAFFS.contains(s1) || !s.X.STAFFS.contains(s2
                ) || !s.X.PATIENTS.contains(p) ) {
21                  //One of the parameters does not exist
22                  return FALSE;
23          }
24          //Parameters are valid
25          return u == s1 || u == s2;
26  }
```

Finally, the interpretation function for the authorization queries for the new application queries is given by:

```
1   I(s, auth-getconsreq(s), u) = {
2           if (!s.X.STAFFS.contains(s)) {
3                   //Staff member does not exist
4                   return FALSE
5           }
6           return s == u;
7   }
8
9   I(s, auth-getactivecons(s), u) = {
10          if (!s.X.STAFFS.contains(s)) {
11                  //Staff member does not exist
12                  return FALSE;
13          }
14          return s == u;
15  }
```

5.4 Pairing elements

The definition of a two-levels model in ISAAC requires the definition of the three pairing functions: the state-mapping function, the action-mapping function and the query-mapping function. This section of the case study focuses on the definition of those pairing elements.

5.4.1 State-mapping function

In Chapter 4, the state-mapping function was defined as the pairing element that describes how lower level states can be derived starting from higher level states:

$$\sigma : \mathbb{S}_1 \rightarrow \mathbb{S}_2$$

In the scenario in analysis, the lower level state expands the higher level state by adding the possibility to record consulting requests and active consulting. Since the consulting action is not allowed in the higher level state, a reasonable mapping simply creates a lower level state with no consulting requests or active consulting:

```
1 σ(s1) = {
2         s2 = new lower level state;
3         //Users are the same
4         s2.U = s1.U;
5         //There are no authorization data
6         //Shared application data are the same
7         s2.X = s1.X;
8         //Consulting requests and active consulting sets are
              empty
9         s2.X.CONSREQS = ∅;
10        s2.X.ACTIVECONS = ∅;
11        return s2;
12 }
```

5.4.2 Action-mapping function

In Chapter 4, the action-mapping function was defined as the pairing element that describes how a higher level command is translated in a series of lower level commands, specifying which lower level user should execute each of them:

$$\alpha : \mathbb{S}_1 \times \mathbb{C}_1 \times \mathbb{D}_1.\text{UIDs} \rightarrow (\mathbb{C}_2 \times \mathbb{D}_2.\text{UIDs})^*$$

The refinement described in the scenario in analysis does not alter the way higher level commands are executed, nor the users executing them: a higher level command simply calls the respective lower level command with the same user. Therefore, the action-mapping function for a generic command c_1 is defined by:

```
1 α(s1, c1, u1) = {
2         result = new lower level empty command-user pairs
              sequence;
3         result.append(<c2(c1), u2(u1)>);
4         return result;
5 }
```

where command $c_2(c_1)$ is the lower level version of command c_1, and user $u_2(u_1)$ is the lower level version of user u_1.

5.4.3 Query-mapping function

In chapter 4, the query-mapping function was defined as the pairing element that describes how to answer a higher level query as a function of lower level queries, possibly modifying the lower level state:

$$\pi : \mathbb{S}_2 \times \mathbb{Q}_1 \times \mathbb{D}_1.\text{UIDs} \rightarrow \mathbb{D}_2.\text{BOOLs} \times \mathbb{D}_2.\text{BOOLs} \times (\text{flatten}(\mathbb{D}_1) \cup \{\perp\}) \times \mathbb{S}_2$$

Similarly to what happens with the action-mapping function, the refinement described in the scenario in analysis does not alter the way a higher level query is answered, and answering higher level queries does not require altering

the lower level state: a higher level query is answered by simply calling the interpretation function of the corresponding lower level query. Therefore, the query-mapping function for a generic query q_1 is defined by:

```
1 π(s2, q1, u1) = {
2         return <I2(s2, q2, u2(u1).result, s2>;
3 }
```

where query $q_2(q_1)$ is the lower level version of query q_1 and user $u_2(u_1)$ is the lower level version of user u_1.

5.5 Proof of correctness

As shown in Chapter 4, two-levels mode reasoning focuses on verifying the correctness of a refinement, as specified in definition 23. Correctness is a conjunction of two independent sub-properties: action-mapping preservation and query-mapping preservation.

5.5.1 Action-mapping preservation

Informally speaking, action-mapping preservation guarantees that higher level commands and their action-mapped counterparts are really equivalent with respect to state-mapping. The formal definition of action-mapping preservation was given in Definition 25:

$$\forall s_1 : \mathbb{S}_1, c_1 : \mathbb{C}_1$$
$$\sigma_{eq}(\sigma(\mathbb{T}_1(s_1, c_1)), \mathbb{T}_2{}^*(\sigma(s_1), \alpha(s_1, c_1)))$$

In order to quickly verify action-mapping preservation for the scenario in analysis, it is useful to note that the action-mapping function simply calls the lower level version of the higher level command passed as its parameter. Additionally, it is possible to simplify state-mapping equivalence to state equality. Therefore, the action-mapping preservation definition can be simplified to:

$$\forall s_1 : \mathbb{S}_1, c_1 : \mathbb{C}_1$$
$$\sigma(\mathbb{T}_1(s_1, c_1)) = \mathbb{T}_2(\sigma(s_1), c_2(c_1))$$

where command $c_2(c_1)$ is the lower level version of command c_1.
Recall now that the lower level transition function differs from its higher level counterpart only with respect to lower level-exclusive commands. In other words, $\mathbb{T}_2(\sigma(s_1), c_2(c_1)) = \mathbb{T}_1(\sigma(s), c_1)$ for any higher level command c_1. It is therefore possible to further simplify the definition to:

$$\forall s_1 : \mathbb{S}_1, c_1 : \mathbb{C}_1$$
$$\sigma(\mathbb{T}_1(s_1, c_1)) = \mathbb{T}_1(\sigma(s_1), c_1)$$

According to its specification, the state-mapping function behaves like an identity function for the state with the exclusion of the consulting related elements, that are set to the empty set in the returned result. Since the higher level transition function is not dependent on the consulting elements nor alters them, as they are not part of the higher level state, the specific case in analysis makes it so that the higher level transition function and the state-mapping function are linearly exchangeable. Therefore, action-mapping preservation is verified.

5.5.2 Query-mapping preservation

Informally speaking, query-mapping preservation guarantees that the answers to all the higher level queries of a given higher level state can be inferred by lower level queries of the correspondent lower level state, and that answering the query does not alter the lower level state "too much". The formal definition of the query-mapping function was given in definition 26:

$$\begin{aligned}
&\forall s_1 : \mathbb{S}_1, q_1 : \mathbb{Q}_1, u_1 : \mathbb{D}_1.\text{UIDs} : \\
&\quad \mathbb{I}_1(s_1, q_1, u_1) = \pi(\sigma(s_1), q_1, u_1).\text{RESULT} \\
&\quad \wedge \\
&\quad \sigma_{eq}(\sigma(s_1), \pi(\sigma(s_1), q_1, u_1).\text{STATE})
\end{aligned}$$

In order to quickly verify query-mapping preservation for the scenario in analysis, it is useful to note that the query-mapping function simply interprets the lower level version of the higher level query passed as its parameter, and does not modify the state in its execution. Therefore, the query-mapping preservation definition can be simplified to:

$$\begin{aligned}
&\forall s_1 : \mathbb{S}_1, q_1 : \mathbb{Q}_1, u_1 : \mathbb{D}_1.\text{UIDs} : \\
&\quad \mathbb{I}_1(s_1, q_1, u_1) = \mathbb{I}_2(\sigma(s_1), q_2(q_1), u_2(u_1)) \\
&\quad \wedge \\
&\quad \sigma_{eq}(\sigma(s_1), \sigma(s_1))
\end{aligned}$$

where query $q_2(q_1)$ is the lower level version of query q_1 and $u_2(u_1)$ is the lower level version of user u_1.

At this point, since σ_{eq} is an equivalence relationship, state-mapping equivalence is trivially true. Similarly to what happens for action-mapping preservation, the state-mapping function behaves like an identity function for all the state elements but the ones related to consulting. The lower level interpretation function and its higher level counterpart are identical with respect to higher level queries. Therefore, it is possible to state that query-mapping preservation is also verified, using the same approach used for the proof of action-mapping preservation.

5.6 Conclusions

This chapter presented a case study based on a real-world hospital scenario. Albeit some of the elements were simplified and/or treated with a sometimes only semi-formal approach, for the sake of clarity and readability also for non specialized users, the case study shows how ISAAC can be applied to real cases in order to verify the correctness of the system design, and its refinements, with respect to both the application and authorization aspects. In particular, the analysis showed that, following the proposed approach, it is possible to safely enrich the basic hospital model in order to offer the possibility for inter-ward consulting. This analysis may take in place in one of two real-world contexts: a refinement of the previous, not yet implemented, system design with the goal of describing things at a lower level of abstraction, but also in the case of a modification of the "status quo" of an already implemented system, with the goal of verifying that the modification will not break the previous policies. In both cases, ISAAC offers a valid option to guarantee the reliability of the models and of the analysis carried on them.

Conclusions 6

This thesis presented ISAAC, a theoretical methodology for the formal co-design of the application and authorization parts of a system and refinement verification. Using the ISAAC approach, it is possible to design robust systems that are less likely to be affected by the usual problems that arise when plugging an access control system into an already completed application system, as the two parts are integrated since the early design stages. Additionally, it is possible to proceed in the refinement of an abstract design in a secure way, verifying at each refinement step that no inconsistency has been introduced in the models.

When modeling a system at a single level of abstraction, ISAAC offers the possibility to check for some simple call loops in the system interfaces. This approach is based on call graph analysis, and while being able to only offer partial answers, it is a starting point for the analysts in charge of detecting this kind of inconsistencies.

When working on the refinement of a previous model, ISAAC allows to ef-

fectively check that the refined model does not introduce inconsistencies with respect to the older version. This is done analyzing both the way how the new commands are executed and how the new queries of the system state are answered. The concept of refinement can also be applied, counter-intuitively, to the verification of the functional equivalence of two different systems, by imagining one as a refinement of the other.

The way ISAAC is designed presents two additional advantage. First, distinguishing between authorization and semantic failures forces the system designer to consider the risk of information leak via side channels during the design of the system. Second, ISAAC has been designed specifically with the goal of being easy to implement in most of the commonly used automated verification systems, in order to ease the burden of manually verifying system properties.

The feasibility of the ISAAC approach to system design and refinement was tested in a case study that modeled a sub-system of a real-world hospital scenario. In particular, the areas of the hospital system related to patient data and therapies were modeled. In the first instance of the model, only the medical personnel of a ward had access to the patients data of the ward. In a subsequent refinement, the model introduced the possibility to request consulting to medical personnel outside the ward of the patient. The case study showed the correctness of the proposed refinement, with respect to the definition of refinement correctness of ISAAC.

With respect to future works, there are three main possible directions of research. The first one relates to extending the working possibilities of ISAAC within one-level mode, both by enhancing the capabilities of the circular dependencies detection algorithm and by offering new tools to be applied to the

analysis of the one-level models.

Another option for future research is based on the fact that it was noted that the presence of logging facilities within the system often leads to peculiar behaviors, such as the fact that the state is not kept constant after a command failure and that preservation properties must rely on state-mapping equivalence rather than state equality. It is recommended to further explore this aspect, in order to offer a better integration of logging elements in the system designs, possibly by looking at the work already done on the topic in the field of aspect-oriented programming (AOP) [20].

A final suggestion for future research is based to the notion that the semantics functions of command and queries often tend to have the same abstract structure, based on "authorization check, semantic check, execution" sequences. The possibility of the development of a meta-language, using templates for the automated generation of semantic elements, could greatly help both in the description of the models and in the subsequent automated verification of properties, greatly increasing the likelihood of ISAAC becoming an approach used on the large scale.

Bibliography

[1] Stefania Gnesi and Tiziana Margaria. *Formal methods for industrial critical systems: A survey of applications.* John Wiley & Sons, 2012.

[2] Wolfgang Reisig. *Petri nets: an introduction.* Springer-Verlag New York, Inc., 1985.

[3] E Allen Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072, 1990.

[4] J Michael Spivey. The Z notation: a reference manual. International Series in Computer Science, 1992.

[5] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 2005.

[6] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[7] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Automated DeductionâĂŤCADE-11*, pages 748–752. Springer, 1992. URL `http://pvs.csl.sri.com`.

[8] Messaoud Benantar. *Access control systems: security, identity management and trust models.* Springer, 2006.

[9] Mahesh V Tripunitara and Ninghui Li. A theory for comparing the expressive power of access control models. *Journal of Computer Security*, 15(2):231–272, 2007.

[10] Ninghui Li, John C Mitchell, and William H Winsborough. Beyond proof-of-compliance: security analysis in trust management. *Journal of the ACM (JACM)*, 52(3):474–514, 2005.

[11] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):71–127, 2003.

[12] Timothy L Hinrichs, Diego Martinoia, C Garrison William III, Adam J Lee, Alessandro Panebianco, and Lenore Zuck. Application-sensitive access control evaluation using parameterized expressiveness. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 145–160. IEEE, 2013.

[13] Diego Martinoia. Proving correctness within an access control evaluation framework - M.Sc. thesis. 2013.

[14] Jason Crampton and Charles Morisset. Towards a generic formal framework for access control systems. *arXiv preprint arXiv:1204.2342*, 2012.

[15] Stuart Kent. Model driven engineering. In *Integrated formal methods*, pages 286–298. Springer, 2002.

[16] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[17] Ralph-Johan Back. *On the correctness of refinement steps in program development.* Department of Computer Science, University of Helsinki Helsinki, Finland, 1978.

[18] Carroll Morgan. *Programming from specifications.* Prentice-Hall, Inc., 1990.

[19] Zhenbang Chen, Zhiming Liu, Anders P Ravn, Volker Stolz, and Naijun Zhan. Refinement and verification in component-based model-driven design. *Science of Computer Programming*, 74(4):168–196, 2009.

[20] et Al. Kiczales, Gregor. *Aspect-oriented programming.* Springer Berlin Heidelberg, 1997.

List of Figures

Appendices

Simple file system manager - basic model A

Suppose that one wants to model a simplified file system management: each file has an owner, and users have the possibility to read or write (no execute) a file. There is no possibility to create a file: writing to a non-existing file will simply fail. Only the owner of a file can read or write it. The authorization policy is specified using an access matrix, i.e. a set of <subject, object, right> triplets. In our example there are two users and two files: *Alice*, owner of file p, and *Bob*, owner of file q. The model must also consider the content of each file. Trying to read a non-existing file returns a default failure symbol.

Domains

The set $\mathbb{D}$ is defined as follows:

$$\mathbb{D} = \{\text{UIDs}, \text{BOOLs}, \text{FIDs}, \text{FCs}, \text{AUTHs}\}$$

where UIDs (User ID's) contains the set of all non-empty strings of alphabetic character, BOOLs is the {TRUE, FALSE} set, FIDs (File ID's) contains the set of all the valid absolute file paths, FCs (File Contents) contains all the

binary strings and AUTHs (Authorizations) contains the "read" and "write" constants representing the authorization rights.

Users set

In our example, the set $\mathbb{U}$ is defined as follows:

$$\mathbb{U} = \{\text{Alice}, \text{Bob}\}$$

Application data

In our example, the application data structures are composed of: the set of the files present in the system, which is a subset of $\mathbb{D}$.FIDs, a function from the files to their contents and a function from the files to their owners. Note that it is possible to make the co-domain of the content function either the totality of $\mathbb{D}$.FCs, or to model in the application data structure also a set of the current available contents, a subset of $\mathbb{D}$.FCs, and make that the target co-domain. As usual, there is more than one option. In this example, the first option is preferred.

$$\mathbb{X} = \{\text{FILES}, \text{CONTENT}, \text{OWNER}\}$$

where:

$$\text{FILES} \subseteq \mathbb{D}.\text{FIDs} = \{\text{p}, \text{q}\}$$

$$\text{CONTENT} : \text{FILES} \to \mathbb{D}.\text{FCs} = \{(\text{p}, 0), (\text{q}, 1)\}$$

$$\text{OWNER} : \text{FILES} \to \mathbb{U} = \{(\text{p}, \text{Alice}), (\text{q}, \text{Bob})\}$$

Authorization data

In our example, the chosen access control system will be an access matrix, i.e. a set of <subject, object, right> tuples that specify the rights of each user.

$$\mathbb{A} = \{M\}$$

where:

$$M = \{$$
$$\langle \text{Alice, p, read} \rangle,$$
$$\langle \text{Alice, p, write} \rangle,$$
$$\langle \text{Bob, q, read} \rangle,$$
$$\langle \text{Bob, q, write} \rangle$$
$$\}$$

Commands

In our example, the only available command is the command to write a file. Therefore, the $\mathbb{C}$ set only contains one element.

$$\mathbb{C} = \{\langle \text{write}, \mathbb{D}.\text{FIDs} \times \mathbb{D}.\text{FCs} \rangle\}$$

Queries

In our example, the only available application query is the query to read a file content. There are also two authorization queries, one for each command or

application query.

$$\mathbb{Q} = \{\mathbb{Q}_{\mathbb{X}} \dot{\cup} \mathbb{Q}_{\mathbb{A}}\}$$

where:

$$\mathbb{Q}_{\mathbb{X}} = \{\langle \text{read}, \mathbb{D}.\text{FIDs}, \text{FCs}\rangle\},$$

$$\mathbb{Q}_{\mathbb{A}} = \{$$

$$\langle \text{auth-read}, \mathbb{D}.\text{FIDs}\rangle,$$

$$\langle \text{auth-write}, \mathbb{D}.\text{FIDs} \times \mathbb{D}.\text{FCs}\rangle$$

$$\}$$

Transition function

In our example, the transition function may be defined as follows:

```
1 T(s, write(fid, fc), u) = {
2         if (!I(s, auth-write(fid, fc), u)) {
3                 //Authorization failure
4                 return <FALSE, FALSE, s>;
5         }
6         if (!s.X.FILES.contains(fid)) {
7                 //Semantic failure: non-existing file
8                 return <TRUE, FALSE, s>;
9         }
10        //Valid request: can write, file exists
11        s.X.CONTENT(fid) = fc;
12        return <TRUE, TRUE, s>;
13 }
```

Interpretation function

In our example, the interpretation function may be defined as:

```
1 I(s, read(fid), u) = {
2         if (!I(s, auth-read(fid), u)) {
```

```
3                 //Authorization failure
4                 return <FALSE, FALSE, ⊥>;
5         }
6         if (!s.X.FILES.contains(fid)) {
7                 //Semantic failure: non-existing file
8                 return <TRUE, FALSE, ⊥>;
9         }
10        //Valid request: can read, file exists
11        return <TRUE, TRUE, s.X.CONTENT(fid)>;
12}
13
14I(s, auth-read(fid), u)) = {
15        if (!s.FILES.contains(fid)) {
16                //Non-existsing file
17                return FALSE;
18        }
19        //File exists
20        return s.A.M.contains(<u, fid, READ>);
21}
22
23I(s, auth-write(fid, fc), u) = {
24        if (!s.FILES.contains(fid)) {
25                //Non-existing file
26                return FALSE;
27        }
28        //File exists
29        return s.A.M.contains(<u, fid, WRITE>);
30}
```

Simple file system manager - refined model B

Let us now refine the model described in Appendix A. The previously atomic "read" and "write" commands will now be modeled as a series of "open, read, close" and "open, write, close" operations. Additionally, the lock status on a file must also be modeled. For the sake of simplicity, there is only one lock type, i.e. read and write lock are not distinguished. Trying to write into a locked file will not do anything, trying to read from a locked file will return a binary empty string. As all the reasoning of two-levels mode is carried out regarding the mapping of the higher level model into the lower-level model, the instantiation of the lower level model is not performed. The lower-level states will be derived by using the state-mapping function from the higher level model.

Domains

The refinement does not introduce new file types. Therefore the set $\mathbb{D}$ is:

$$\mathbb{D} = \{\text{UIDs}, \text{BOOLs}, \text{FIDs}, \text{FCs}, \text{AUTHs}\}$$

where UIDs contains the set of all non-empty strings of alphabetic character, BOOLs is the {TRUE, FALSE} set, FIDs contains the set of all the valid absolute file paths and FCs contains all the binary strings. AUTHs now contains the new "open" and "close" constants.

Users set

The refinement does not alter the users set construction. As before:

$$\mathbb{U} \subseteq \mathbb{D}.\text{UIDs}$$

Application data

Application data must now also model the lock on files. In order to do so, the *LOCKED* set contains a list of file-user pairs of the currently locked files.

$$\mathbb{X} = \{\text{FILES}, \text{CONTENT}, \text{OWNER}, \text{LOCKED}\}$$

where:
FILES $\subseteq \mathbb{D}$.FIDs,
CONTENT : FILES $\rightarrow \mathbb{D}$.FCs,
OWNER : FILES $\rightarrow \mathbb{U}$,
LOCKED $\subseteq$ FILES $\times \mathbb{U}$

Authorization data

For the sake of simplicity, the same access control system of the higher-level model (access matrix) is used.

$$\mathbb{A} = \{\text{M}\}$$

where M is a set of <subject, object, right> tuples.

Commands

In addition to the previously specified "write" command, the system now also offers the "open" and "close" commands.

$$\mathbb{C} = \{\langle \text{write}, \mathbb{D}.\text{FIDs} \times \mathbb{D}.\text{FCs}\rangle, \quad \langle \text{open}, \mathbb{D}.\text{FIDs}\rangle, \quad \langle \text{close}, \mathbb{D}.\text{FIDs}\rangle\}$$

Queries

Queries are unchanged by the refinement, but the model must now also offer the authorization queries for the "open" and "close" commands.

$$\mathbb{Q} = \{\mathbb{Q}_{\mathbb{X}} \cup \mathbb{Q}_{\mathbb{A}}\}$$

where:

$$\mathbb{Q}_{\mathbb{X}} = \{\langle \text{read}, \mathbb{D}.\text{FIDs}, \text{FCs}\rangle\},$$

$$\mathbb{Q}_{\mathbb{A}} = \{$$

$$\langle \text{auth-read}, \mathbb{D}.\text{FIDs}\rangle,$$

$$\langle \text{auth-write}, \mathbb{D}.\text{FIDs} \times \mathbb{D}.\text{FCs}\rangle,$$

$$\langle \text{auth-open}, \mathbb{D}.\text{FIDs}\rangle,$$

$$\langle \text{auth-close}, \mathbb{D}.\text{FIDs}\rangle$$

$$\}$$

Transition function

The transition function must now also be defined for the "open" and "close" commands:

```
1 T(s, write(fid, fc), u) = {
2         if (!I(s, auth-write(fid, fc), u)) {
3                 //Authorization failure
4                 return <FALSE, FALSE, s>;
5         }
6         if (!s.X.FILES.contains(fid)) {
```

```
7                 //Semantic failure: non-existing file
8                 return <TRUE, FALSE, s>;
9         }
10        if (!s.X.LOCKED.contains(<fid, u>)) {
11                //Semantic failure: user does not have the
                    lock
12                return <TRUE, FALSE, s>;
13        }
14        //Valid request: can write, file exists, user has the
            lock
15        s.X.CONTENT(fid) = fc;
16        return <TRUE, TRUE, s>;
17}
18
19T(s, open(fid), u) = {
20        if (!I(s, auth-open(fid, fc), u)) {
21                //Authorization failure
22                return <FALSE, FALSE, s>;
23        }
24        if (!s.X.FILES.contains(fid)) {
25                //Semantic failure: non-existing file
26                return <TRUE, FALSE, s>;
27        }
28        if (!s.X.LOCKED.contains(<fid, ANY>)) {
29                //Semantic failure: file is locked
30                return <TRUE, FALSE, s>;
31        }
32        //Valid request: can open, file exists, file is
            unlocked
33        s.X.LOCKED.add(<fid,  u>);
34        return <TRUE, TRUE, s>;
35}
36
```

```
37 T(s, close(fid), u) = {
38         if (!I(s, auth-close(fid, fc), u)) {
39                 //Authorization failure
40                 return <FALSE, FALSE, s>;
41         }
42         if (!s.X.FILES.contains(fid)) {
43                 //Semantic failure: non-existing file
44                 return <TRUE, FALSE, s>;
45         }
46         if (!s.X.LOCKED.contains(<fid, u>)) {
47                 //Semantic failure: user does not have the
                    lock
48                 return <TRUE, FALSE, s>;
49         }
50         //Valid request: can close, file exists, user has the
             lock
51         s.X.LOCKED.remove(<fid, u>);
52         return <TRUE, TRUE, s>;
53 }
```

Interpretation function

The refinement does not affect the interpretation function much. The only difference is that it must be now also specified for the new authorization query regarding the "open" and "close" commands.

```
1 I(s, read(fid), u) = {
2         if (!I(s, auth-read(fid), u)) {
3                 //Authorization failure
4                 return <FALSE, FALSE, ⊥>;
5         }
6         if (!s.X.FILES.contains(fid)) {
7                 //Semantic failure: non-existing file
8                 return <TRUE, FALSE, ⊥>;
```

```
9         }
10        if (!s.X.LOCKED.contains(<fid, u>)) {
11                //Semantic failure: user does not have the
                    lock
12                return <TRUE, FALSE, ⊥>;
13        }
14        //Valid request: can read, file exists
15        return <TRUE, TRUE, s.X.CONTENT(fid)>;
16 }
17
18 I(s, auth-read(fid), u) = {
19        if (!s.FILES.contains(fid)) {
20                //Non-existing file
21                return FALSE;
22        }
23        //File exists
24        return s.A.M.contains(<u, fid, READ>);
25 }
26
27 I(s, auth-write(fid, fc), u) = {
28        if (!s.FILES.contains(fid)) {
29                //Non-existing file
30                return FALSE;
31        }
32        //File exists
33        return s.A.M.contains(<u, fid, WRITE>);
34 }
35
36 I(s, auth-open(fid), u) = {
37        if (!s.FILES.contains(fid)) {
38                //Non-existing file
39                return FALSE;
40        }
```

```
41          //File exists
42          return s.A.M.contains(<u, fid, OPEN>);
43  }
44
45  I(s, auth-close(fid), u) = {
46          if (!s.FILES.contains(fid)) {
47                  //Non-existing file
48                  return FALSE;
49          }
50          //File exists
51          return s.A.M.contains(<u, fid, CLOSE>);
52  }
```