A Compositional Framework for the Development of Secure Access Control Systems

PhD Thesis

François Siewe

Software Technology Research Laboratory

Faculty of Compting Sciences and Engineering

De Montfort University

England

2005

To my wife Justine and

my children Lena, Hermann and Russel.

Abstract

Security requirements deal with the protection of assets against unauthorised access (disclosure or modification) and their availability to authorised users. Traditionally, security concerns are considered as an add-on to be implemented at a later stage of system development. Nowadays, it is well understood that adopting such belief can be difficult and error-prone. Therefore security *must* be considered as an integral part of the system requirements right from the early stages of the system development life cycle.

In this thesis, we develop a unified compositional formal framework for the specification of the functional, temporal and security requirements of systems. The framework uses a single formalism, Interval Temporal Logic (ITL in short), for expressing the three types of requirements and for reasoning about them in a uniform manner. This provides an effective way of integrating security concerns into the system requirements and to address them early (high level specifications) in the system development process so that security holes can be detected and fixed timely.

We propose a language for expressing access control policies and their composition. Especially, a set of operators is defined for expressing policies that can change dynamically in response to time or events. The resulting model provides a high flexibility to

support the specification of several protection requirements that cannot be expressed in traditional access control models. We investigate the algebraic properties of the operators and develop sound algebraic laws for the comparison and the refinement of security policies.

We take the view that a system is developed starting from a high level specification and transformed by a sequence of correctness preserving refinement steps down to a low level implementation. The low level system must implement a mechanism for enforcing security policies. In this respect, we develop a computational model, **Secure Action System (SAS)**, that allows the enforcement of dynamically changing security policies. It is an extension of the traditional action system paradigm to cater for security. SASs can be composed into a large system. We propose a rich set of sound compositional rules for the design and verification of SASs.

We develop a tool, **SPAT**, to animate security policy specifications and to analyse them. We evaluate our approach with a detailed case-study of a secure exam system and the formalisation of the British Medical Association's security policy for Electronic Patient Records (EPRs).

Declaration

The thesis presented here is mine and original. It is submitted for the degree of Doctor of Philosophy at De Montfort University. The work was undertaken between October 2002 and July 2005.

Acknowledgements

I thank my supervisor Professor Hussein Zedan for his constructive criticism, experienced guidance during the preparation of this thesis. Thanks Hussein for arranging for financial support during my PhD studies.

I am grateful to Dr. Antonio Cau whose guidance, support and critical analysis were extremely important for the completion of this thesis.

My gratitude to Professor Jonathan Bowen without whom this adventure would have never started. Thanks Jonathan for giving me such an opportunity.

Special thanks go to my wife Justine and to my children Russel, Hermann and Lena for their patience, love, affection and encouragements over the years. This thesis is dedicated to them.

My gratitude to my family for their encouragement and full support during the preparation of this thesis.

Many thanks go to all our colleagues at the STRL for the valuable discussions during all these years.

Publications

1. François Siewe, Antonio Cau and Hussein Zedan. A Compositional Framework for Access Control Policies Enforcement. In *proceedings of the ACM Workshop on Formal Methods in Security Engineering*, Washington, DC, USA, October 2003, ACM Press(2003), pp. 32-42.

2. François Siewe, Helge Janicke and Kevin Jones. Dynamic Access Control Policies and Web-Service Composition. In *proceedings of the 1st Young Researcher Workshop on Service-Oriented Computing*, Leicester, U.K., April 2005.

3. Helge Janicke, Francois Siewe, Kevin Jones, Antonio Cau and Hussein Zedan. Analysis and Run-time Verification of Dynamic Security Policies. In *Proceedings of the Workshop on Defense Applications of Multi-Agent Systems (DAMAS) 2005 (colocated with AAMAS 05)*, Ultrecht University, The Netherlands, July 2005, pp. 55–66.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Security requirements deal with the protection of assets against unauthorised access (disclosure or modification) and their availability to authorised users. Traditionally, security concerns are considered as an add-on to be implemented at a later stage of system development. Nowadays, it is well understood that adopting such belief can be difficult and error-prone. Therefore security *must* be considered as an integral part of the system requirements right from the early stages of the system development life cycle. The advantages are numerous as security is becoming a critical requirement in many applications such as health care, commerce and the military. To name a few, some of the advantages are:

- same attention is paid to the analysis and implementation of both functional and security requirements of systems;

- dependencies between functional requirements and security requirements are well understood so as to guide their correct implementation;
- security policies and their implementation are well documented for underpinning further system evolution;
- bugs and security holes may be detected and fixed as the system is developed;
- enables a formal approach to secure system development whereby a system is developed starting from a formal specification and transformed, through correctness preserving refinement steps, into a concrete implementation;
- increases the dependability, and hence the trustworthiness of systems so developed.

Formal methods have been used successfully in the development of safety-critical systems where failure can cause serious damage such as loss of life or assets [72, 43, 17]. For example, Stepney et al. [72] proposed a development of an electronic purse using refinement. As security is becoming a critical requirement for many organisations, formal methods are increasingly used to address security issues in the development of information systems where confidentiality, integrity and availability of information are paramount [68, 76, 48, 37, 28, 1]. For example, many security protocols have been revealed insecure using model-checking techniques or interactive theorem provers [53, 54].

1.2 Original Contribution

In this thesis, we define a unified compositional formal framework for the specification of the functional, temporal and security requirements of secure systems and their implemen-

tation through step-wise correctness preserving refinements. The framework uses a single well-defined formalism, Interval Temporal Logic (ITL in short), for expressing the three types of requirements and for reasoning about them in a uniform manner. This provides an effective way of integrating security concerns into the system requirements early in the system development life cycle. Thus, bugs and security holes can be found and removed in a timely fashion during the system development process.

Our approach is novel, for it provides:

1. a formal model that is **compositional** and unifies functional, security and temporal requirements;

2. a policy language which has a sound semantics (with respect to the model) and has a rich set of operators that allow dynamic changing (i.e. at run-time) of policies.

The resulting model provides a high degree of flexibility to support the specification of several protection requirements that cannot be expressed in traditional access control models. We investigate the algebraic properties of the operators and develop sound algebraic laws for the comparison and the refinement of (dynamically changing) security policies. Compositionality is paramount for the design of (security- or safety-) critical or large systems in that it allows us to infer properties of a whole from the properties of its immediate constituents without additional information about their internal structures.

Fundamental to security requirements is the access control policy that allows a *subject* to perform an *action* on a given *object*. In this thesis we focus on the specification of access control policies. We address the completeness problem for access control policies. An access control policy is *complete* if it determines at any time the access rights of each

subject with respect to any object and any action. This ensures that any authorisation request has a deterministic answer. Furthermore, reasoning about liveness properties of incomplete policies is likely to be difficult. However, writing a complete policy specification can be very complex and cumbersome. Woo and Lam [76] proposed *default rules* as a way to provide complete specification. The drawback of this approach is that default rules might not be conclusive. As a consequence the model can lead to a situation in which an authorisation request has no answer. Logic-based approaches [40, 27, 42, 48] restrict the policy language to a variant of Datalog [33] to overcome this problem using the *closed world assumption* [33, 37] (i.e. if a positive literal cannot be proved to be true then it is assumed to be false).

We adopt an algorithmic approach. We propose an algorithm which transforms automatically an incomplete policy into a complete one that grants *exactly* the rights explicitly stated in the input policy and denies everything else. The correctness of the algorithm is established.

We take the view that a system is developed starting from a high level specification and transformed by a sequence of correctness preserving refinement steps down to a low level implementation. The low level system must implement a mechanism for enforcing security policies. In this respect, we develop a computational model, **Secure Action System (SAS)**, that allows the enforcement of (dynamically changing) security policies. It is an extension of the traditional action system paradigm [6] to cater for security. In addition to the security features provided by SASs, the novelty of our approach leans also on the fact that the development of SASs is compositional. SASs can be composed into a

system that enforces a specific (dynamically changing) security policy. We propose a rich set of sound compositional rules for the design and verification of SASs.

Our choice of ITL is very much motivated by the availability of an executable subset of the logic, known as *Tempura*, which allows us to validate ITL specifications through simulation before their actual implementation. *AnaTempura* is an integrated workbench for run-time verification of systems using Tempura. We extend AnaTempura architecture with a new component, known as *SPAT* (Security Policy Analysis Tool) to cater for the analysis of (dynamically changing) security policies. The current version of SPAT

(i) allows visualisation of the access control matrix, access control lists and capability lists in each state of the simulation period;

(ii) allows users to specify queries, in *Query By Example (QBE)* format, for extracting desired information about the security policy being executed;

(iii) computes the information flow allowed by a security policy;

(iv) draws the graphs of authorisation rights, the graphs of delegation rights (i.e. rights to pass a right on to someone else) and the graphs of information flow allowed by a security policy.

(v) provides a user-friendly graphical interface for visualising security policies during their execution by the Tempura interpreter;

As proof of concept, we specify and simulate in SPAT the Anderson's security policy for Electronic Patient Records (EPRs) [5]. Anderson's security model is seen by many authors [63, 16, 4] as a separate security model that combines confidentiality and integrity

policies to protect patient privacy and record integrity. This model cannot be specified in most policy languages [4]. We demonstrate our development technique for secure systems with a simple yet interesting example of a United Kingdom based examination setting systems. The example shows how to develop a system that enforces a dynamically changing security policy.

1.3 Thesis Outline

In Chapter 2 we give a survey of security policy specification models and languages, and related work on the integration of security requirement into system requirements. We start by introducing basic security concepts, and follow with a description of security policy models and policy specification languages. Then related work on the integration of security concerns in system development processes is presented.

Chapter 3 presents the formal foundation of our approach and a computational model for supporting the enforcement of security policies. We introduce Interval Temporal Logic (ITL) which is the unified formal notation for reasoning about properties, whether functional, temporal or security, of systems. The computational model extends the well-defined *action systems* paradigm to cater for security requirements. It is used as a design language for secure systems.

In Chapter 4, we present notations for expressing authorisation and delegation policy rules. Positive and negative authorisations and delegations can be specified, and conflicts among them resolved using conflict resolution rules. Then we introduce the concept of a *simple policy* which is merely a finite set of policy rules. The problem of completeness of

simple policies is addressed and an algorithm for solving this problem is proposed.

Chapter 5 is devoted to policy composition. We introduce operators for expressing dynamically changing security policies that can change at run-time in response to *time* and *events*. An algebra of security policies is proposed and allows us to compare policies and to refine them.

In Chapter 6, we propose a development method for secure systems. We focus on the enforcement of dynamically changing security policies in action systems. We propose sound compositional rules for the design and verification of *secure* action systems.

In Chapter 7, we evaluate our approach with two case studies. In the first case study, we give a formal specification of Anderson's security policy for electronic patient records in our security policy language. The second case study illustrates our development technique for secure systems.

Chapter 8 presents our security analysis tool SPAT, and Chapter 9 concludes and suggests directions for future work.

Chapter 2

Security Policy Overview

Objectives

- To define basic security concepts.
- To present an overview of security models.
- To present an overview of security policy languages.
- To present related work on the integration of security and system requirements.

2.1 Introduction

In this chapter we give an account of related work in the area of security policy specification and their integration into system requirements. We start by introducing basic security concepts, and follow with a description of security policy models and policy specification languages. Then related work on the integration of security concerns in system development processes is presented. Although a variety of security policy models and languages

have been developed, little work has been done concerning how security policies can be integrated into the system requirement from the early stages of system development life cycle. It is widely agreed nowadays [28, 57, 68, 63] that ad hoc solutions that implement security after the system has been developed are error-prone and therefore are just not acceptable for the design of security-critical systems.

2.2 Basic Security Concepts

2.2.1 Security Requirements, Policies, Models and Mechanisms

Security requirements are concerned with the protection of assets from threats. They specify *constraints* about who is allowed to access the system's resources. As such security requirements are seen by some authors as *non-functional* requirements, where non-functional requirements are defined as constraints to be imposed on the use of functions of the systems [57]. The definitions most frequently proposed for computer security identify three primary requirements: *confidentiality*, *integrity* and *availability*. Confidentiality (sometimes called *secrecy*) requires that the information or resources in a computer system only be disclosed to authorised parties. Integrity refers to the protection of information against improper or unauthorised modifications. Integrity includes data integrity (the content of the information) and origin integrity (the source of the data, often called *authentication*). As pointed out by Bishop [16], the source of the information may bear on its accuracy and credibility and on the trust that people place in the information. Availability is concerned with the ability to use the information or resource desired. Availability is an

important aspect of reliability as well as of system design because an unavailable system is at least as bad as no system at all [16]. The aspect of availability that is relevant to security is that someone may deliberately arrange to deny access to data or to a service by making it unavailable to legitimate parties (denial of service) or just to slug it (covert channels).

To enforce the above requirements, three mutually supportive technologies are used: *authentication, access control* and *auditing*. Authentication deals with identification of users. Access control is concerned with limiting the activity of users who have been successfully authenticated by ensuring that every access to information or resources is controlled and that only authorised accesses can take place. Auditing is the process of recording information (such as user name and time of access) about accesses to resources so as to be able to establish responsibilities in the event of a security breach.

A security policy is a specification of the security requirements of a system. It describes rules about who is allowed to do what within a system. Considerable body of work is devoted to developing languages for expressing security policies. A review of these works is given in Section 2.4. Note that in the literature, the term security policy is also used to refer to security requirement.

A security model is a way of formalising security policies so as to enable formal reasoning about them. A variety of access control models have been defined to handle different kinds of security policies such as confidentiality policies or integrity policies or a mixture of both. An account of these models is given in Section 2.3.

A security mechanism defines the low level (software and hardware) functions that

implement the controls imposed by the policy and formally stated in the model. It is a method, tool, or procedure for enforcing a security policy. It detects and prevents attacks and recover from those that succeed. Analysing the security of a system requires an understanding of the mechanism that enforces its security policy.

2.2.2 Security Threats

A threat is a potential violation of security [16]. The violation need not actually occur for there to be a threat. The fact that the violation might occur means that those actions that could cause it to occur must be guarded against (or prepare for) [16]. Those actions are called *attacks*. Those who execute such actions, or cause them to be executed, are called *attackers*. Following are common threats discussed in the literature [16, 75].

Interception, the unauthorised disclosure of information. This is a threat to confidentiality. It is passive, suggesting simply that some entity is listening to (or reading) communication or browsing through file or system information. *Passive wiretapping* is a form of interception in which a network is monitored. This threat is also known as *snooping* or *eavesdropping*.

Modification or *alteration*, an unauthorised change of information. This is a threat to (data) integrity. Unlike interception, modification is active. *Active wiretapping* is a form of modification in which data moving across a network is altered. An example is the *man-in-the-middle* attack, in which an intruder reads messages from the sender and sends (possibly modified) versions to the recipient, in hopes that the recipient and the sender will not realised the presence of the intermediary.

Masquerading or *spoofing*, an impersonation of an entity by another, is a threat to origin integrity or authentication. It lures a victim believing that the entity with which it is communicating is a different entity.

Interruption, an unauthorised blocking of access to information or resources. It is a threat to availability. An example is the *denial-of-service* attack, in which the attacker prevents a server providing a service. The denial may occur at the source (by preventing the server from obtaining the resources needed to perform its function), at the destination (by blocking the communication from the server), or along the intermediate path (by discarding messages from either the client or the server, or both).

Fabrication, an unauthorised insertion or creation of (possibly counterfeit) information or resources. This is a threat to integrity. Examples include the insertion of spurious messages in a network or the addition of records to a file.

2.3 Security Policy Models

Access control has been traditionally divided into *discretionary* access control (DAC) and *mandatory* access control (MAC) depending on whether the control is exercised by the owner (of the object to access) or the operating system. In discretionary access controls, access rights are based on the identity of subjects and objects, and the owner of an object may decide who can or cannot access the object. Mandatory access controls base the access rights on fixed regulations determined by a central authority. The operating system controls access and the owner cannot override the controls. While discretionary access controls are concerned with controlling the access of users to information, mandatory

access controls are mostly concerned with controlling information flow between objects in a system.

In addition to the two aforementioned types of access control policies, two others types of policies have attracted attention for their effectiveness in military and commercial applications: *information flow* policies and *role-based access control* (RBAC) policies. Information flow policies mostly address the issue of data confidentiality and are more closely related to the principles of mandatory access control. Role-based access control policies control access based on the roles that users have within the system. Discretionary and role-based access control policies are usually coupled with (or include) an *administrative* policy that defines who can specify authorisations governing access control.

2.3.1 Discretionary Access Control

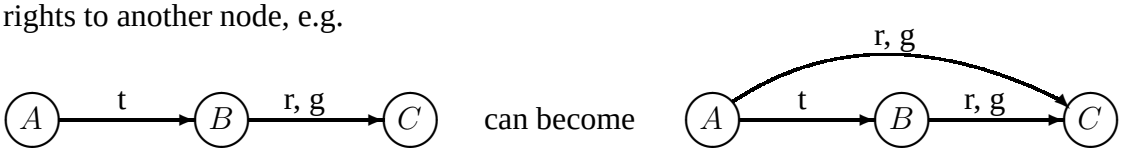
Discretionary access control policies restrict access to objects based on the identity of the subjects and/or groups to which they belong. Access control is at the discretion of users as they can be given the ability to pass on their access rights to other users.

Lampson [47] proposed the *access matrix model* as a framework for describing discretionary access control. In the access matrix model, the state of the system is defined by a triple (S, O, A), where S is the set of subjects, O is the set of objects and A is the access matrix, where rows correspond to subjects, columns correspond to objects and entry $A[s, o]$ records the privileges of s on o. Later refined by Graham and Denning [34], the access matrix model was subsequently formalised by Harrison, Ruzzo, and Ullmann [38]

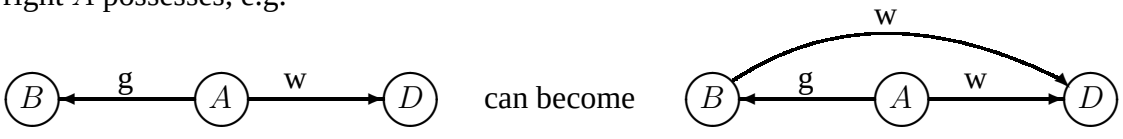
(HRU model) for analysing the complexity of access control policies.

The HRU formalisation identifies six primitive operations that describe changes to the state of a system. It turned out that the *safety* problem of determining whether or not a given subject can ever acquire a given access right is undecidable in general. Progress in safety analysis was made in a later extension of the HRU model by Sandhu [64], who proposed the *TAM* (Typed Access Matrix) model. Like HRU, monotonic TAM is undecidable. However, a rich subset of the model is decidable in polynomial time.

Although the access matrix remains a framework for reasoning about DAC policies, several extensions have been proposed. Take-grant models [15, 71] use graphs to model access control. The protection state of a system is described by a directed graph that represents the same information found in the access matrix. Nodes in the graph are of two types, one corresponding to subjects and the other to objects. An arc directed from a node A to another node B indicates that A has some access right(s) to B. The arc is labelled with the set of A's rights to B. The possible access rights are "read" (r), "write" (w), "take" (t), and "grant" (g). "Take" access implies that node A can take node B's access rights to another node, e.g.



Conversely, if the arc from A to B is marked with a "g", B can be granted any access right A possesses, e.g.



Abadi et al. [1] present a calculus for access control that combines authentication

(i.e. identity check) and authorisation control, taking also into account delegation of privileges among parties. The calculus is based on a notion of principals as the source of request, and provides an algebra for composing them. For example $B\ for\ A$ denotes the principal obtained when a principal B speaks on behalf of another principal A, with appropriate delegation certificate. The process of determining whether a request from a principal should be granted or denied is based on a modal logic that extends the algebra of principles. In that logic, $A\ says\ s$ represents the informal statement that the principal A says s, where s could be e.g. "B's public key is K". However, the calculus is not able to support temporal constraints on authorisations or delegations.

As pointed out in [12], permissions are often limited in time or may hold for specific period of time. A temporal model for access control has been proposed by Bertino et al. [11]. A time expression is associated with authorisations to restrict their validity to specific time periods. Other work suggests using both positive and negative authorisations to conveniently specify exceptions in authorisation rules [63, 76]. However, the combined use of both positive and negative authorisations gives rise to the problem of *incompleteness* (when no authorisation is specified for a given access) and potential *inconsistencies* (when there is a positive authorisation and a negative authorisation for the same access).

Finally, DAC policies present some limitations. DAC policies [63, 32] do not distinguish between users and processes running on their behalf. This makes them vulnerable from processes executing malicious codes which exploit the access rights of the user who initiates their execution. In particular, the access control system can be bypassed by *Trojan Horses* embedded in programs. A Trojan horse is a computer program that

masquerades as a useful service but surreptitiously leaks or alters data. Moreover, DAC policies cannot control information flow among users' processes. Mandatory access control resolves these vulnerabilities through the use of labels.

2.3.2 Mandatory Access Control

Mandatory security policies [63, 56, 16, 55] enforce control on the basis of regulations mandated by a central authority. The most common form of mandatory policy is the *multilevel security policy*, based on the classifications of *subjects* and *objects*. Unlike in DAC policies, MAC policies make a clear distinction between *users* and *subjects*. Users are human beings who can access the system, while subjects are processes operating on behalf of users. This distinction allows the policy to control the indirect accesses (leakages or modifications) of information caused by the execution of the processes.

In multilevel mandatory policies, each object and subject are assigned an access class that is defined as consisting of two components: a *security level* and a *set of categories*. The security level is an element of a hierarchical ordered set $\mathcal{L}$, while the set of categories is a subset of an unordered set $\mathcal{C}$, whose elements reflect functional or competence areas. The set $\mathcal{AC} \mathrel{\widehat{=}} \mathcal{L} \times \mathcal{P}(\mathcal{C})$ of access classes form a lattice together with the *dominance* relationship (here denoted by "$\succeq$") defined as follows:

$$\forall (L_1, C_1), (L_2, C_2) \in \mathcal{AC} : (L_1, C_1) \succeq (L_2, C_2) \Leftrightarrow (L_1 \geq L_2 \ \wedge \ C_1 \supseteq C_2),$$

i.e. an access class c_1 *dominates* an access class c_2 iff the security level of c_1 is greater or equal that of c_2 and the categories of c_1 include those of c_2. The semantics and use of the classifications assigned to objects and subjects is different depending on whether the

classification is intended for a *secrecy* or an *integrity* policy.

A secrecy mandatory policy controls the direct and *indirect* flows of information for preventing leakages to unauthorised subjects. The following two principles, first formulated by Bell and LaPadula (BLP model) [10], must be satisfied to protect information confidentiality:

No-read-up A subject is allowed read access to an object only if the access class of the subject dominates the access class of the object.

No-write-down A subject is allowed a write access to an object only if the access class of the subject is dominated by the access class of the object.

However, the BLP model aroused considerable controversy, with John MacLean's system Z [56].

An integrity mandatory policy controls the flows of information and prevents subjects from *indirectly* modifying information they cannot write. The following two principles, formulated by Biba [14], must be satisfied to protect information integrity:

No-read-down A subject is allowed read access to an object only if the access class of the object dominates the access class of the subject.

No-write-up A subject is allowed a write access to an object only if the access class of the subject is dominates the access class of the object.

The Biba's integrity principles are dual to the confidentiality principles formulated by Bell and LaPadula.

Although mandatory policies, unlike discretionary ones, provide protection against indirect information flow, they are still vulnerable to *covert channels* [63]. Covert channels are channels that are not intended for normal communication, but still can be exploited to infer information.

2.3.3 Role-Based Access Control

Role-based access control (RBAC) is an alternative to traditional discretionary and mandatory access control policies that is attracting increasing attention, particularly for commercial applications. The idea behind RBAC originated from the work of Baldwin [9], where the concept of *named protection domain* is introduced as a way of grouping the privileges needed to accomplish a specific task in database systems. Role-based policies regulate the access of users to the information on the basis of the organisational activities and responsibility that users have in a system. In RBAC, a role is defined as the set of privileges associated with a particular position within an organisation, or a particular working activity.

Unlike in DAC where authorisations are assigned to users, in RBAC authorisations are assigned to roles and roles are assigned to users. The user playing a role is allowed to execute all accesses for which the role is authorised. This mechanism greatly simplifies the management of security policies in that the assignment of roles to users is separated from the assignment of authorisations to roles. Therefore each of these assignments can be manipulated independently.

Sandhu et al. [67] have specified the following four conceptual RBAC models in

an effort to standardise RBAC. $RBAC_0$ contains *users*, *roles*, *permissions*, and *sessions*. Permissions are attached to roles and users can be assigned to roles to assume those permissions. A user can establish a session to activate a subset of the roles to which the user is assigned. $RBAC_1$ includes $RBAC_0$ and introduces *role hierarchies* [66]. In many applications there is a natural hierarchy of roles, based on the principles of generalisation and specialisation. Role *inheritance* enables reuse of permissions by allowing roles to be specified as junior from which senior roles can inherit permissions.

$RBAC_2$ includes $RBAC_0$ and introduces *constraints* to restrict the assignment of users or permissions to roles, or the activation of roles in sessions. Constraints are used to specify application-dependent conditions, and satisfy well-defined control principles such as *least privilege* (i.e. users are assigned only privileges necessary for the accomplishment of their tasks) and *separation of duties* (i.e. no user should be given enough privileges to misuse the system on their own). Finally, $RBAC_3$ combines $RBAC_1$ and $RBAC_2$, and provides both role hierarchies and constraints [2]. A number of variations of RBAC models have been proposed such as team-based access control (TBAC) [73].

2.3.4 Administrative Policies

Administrative policies [65] determine who is authorised to modify the allowed access rights and exist only within discretionary policies. In mandatory policies the access control is determined entirely on the basis of the security classification of subjects and objects. The security administrator is typically the only one who can change the security levels of subjects and objects.

Discretionary access control permits a wide range of administrative policies that can be divided into: *(i) Centralised* where a single authoriser (or group) is allowed to grant and revoke authorisations to the users. *(ii) Hierarchical* where a central authoriser is responsible for assigning administrative responsibilities to others administrators who can then grant and revoke authorisations to users. *(iii) Cooperative* where special authorisations on given resources cannot be granted by a single authoriser but needs cooperation of several authorisers. *(iv) Ownership* where users can grant and revoke authorisations on the objects they own. *(v) Decentralised* where the owner (or its administrators) can delegate other users the privilege of specifying authorisations, possibly with the ability of further delegating it.

Decentralised administration is convenient since it allows users to delegate administrative privileges to others. However, delegation and revocation of delegated authorisations complicate a lot the management of policies. Usually, authorisations can be revoked only by the user who granted them (or, possibly, by the object's owner). When an administrative authorisation is revoked, the problem arises of dealing with the authorisations granted by the users from whom the administrative privilege is being revoked. Two solutions are generally adopted to this problem: *(1) cascade* revocation where authorisations being revoked are deleted and, if they are administrative authorisations, all the authorisations granted by the revokee in virtue of the authorisations being revoked are recursively revoked. This solution has been considered in an initial implementation of System R [35]. *(2) non-cascade* revocation where all the authorisations granted by the revokee in virtue of the authorisations being revoked, are re-specified to be under the authority of the revoker.

Later versions of System R allow non-cascade revocation as well [13].

2.3.5 Other Security Models

Several proposals have attempted a combination of mandatory flow control and discretionary authorisations. A typical example is the *Chinese Wall* [20] model that was developed as a formal model of a security policy applicable to financial information systems. The goal is to prevent information flows which cause conflict of interest for individual consultants. However, unlike in the Bell and LaPadula model, access to data is not constrained by the data classifications but by what data the subjects have already accessed. The model attempts to balance commercial discretionary and mandatory controls.

The Clark-Wilson model [23] is an attempt to ensure the integrity of an organisation's accounting system and to improve the robustness against insider fraud. The model recommends the enforcement of two principles, namely the principle of *well-formed transactions* and the principle of *separation of duty*. While the latter reduces the possibility of fraud or damaging errors by partitioning the tasks and associated privileges so cooperation of multiple users are required to complete sensitive tasks, the former constrains the way data are manipulated so as to preserve and ensure their integrity.

2.4 Security Policy Languages

Recent proposals have worked towards languages and models able to express, in a single framework, a variety of security policies. Logic-based languages, for their expressive power and formal foundations, represent a good candidate.

The first work investigating logic-based languages for the specification of authorisations is the work by Woo and Lam [76]. Their language is essentially a many-sorted first-order language with a rule construct, useful for expressing authorisation derivations. The rule construct is similar to the default construct in *default logic* [62] with a different semantics. A rule is written in the form

$$\frac{f : f'}{g}$$

where f is a formula called *prerequisite*, f' is formula called *assumption* and g a conjunctive formula called *consequent*. The intuitive meaning of the rule is as follows: if f holds and there is no evidence that the negation of f' holds (hence it is consistent to assume that f' holds), then g must also holds. E.g. the following rule expresses the propagation of write access from groups to their members:

$$\frac{s \in G \ \wedge \ write^{+}(G, o) : true}{write^{+}(s, o)}, \tag{2.1}$$

where s is a subject, G a group of subjects, o an object and $write^{+}(x, y)$ is a predicate meaning that there is a positive authorisation for subject x to perform write access on object y (the predicate $write^{-}$ expresses negative authorisations). Positive and negative authorisations can be specified in the model. However, the approach provides no mechanism to handle conflicting authorisations which might be derived using the rules.

Jajodia et al. [40] worked on a proposal that provides specific rules for resolving conflicts among positive authorisations and negative authorisations. Their policy language, called *Authorisation Specification Language* (ASL), provides a specific predicate, $cando$, for expressing explicit authorisations and a different predicate, $dercando$, for expressing

derived authorisations. For example the propagation of groups' rights to their members stated in (2.1) is specified in ASL by the following rule where the predicate in stands for group membership and the sign "$+$" indicates positive authorisations (the sign "$-$" is used for negative ones):

(2.2) $$dercando(s, o, +write) \longleftarrow in(s, G)\ \&\ cando(s, o, +write).$$

The left hand side of the rule operator "$\longleftarrow$" is called the ***head*** of the rule and the right hand side is called the ***body*** of the rule. A third predicate, do, is introduced to resolve conflicts that might occur among derived authorisations. Different types of conflict resolution rules can be specified. For example the rule that gives precedence to denials in the event of conflicts is specified as

$$do(s, o, +a) \longleftarrow dercando(s, o, +a)\ \wedge\ \neg dercando(s, o, -a),$$

where s stands for a subject, o for an object and a for an action.

The type of logic used in ASL is called ***stratified clause form logic*** which is implementable in Datalog [33]. For example, $cando$ can appear in the body of an dercando-rule (i.e. a rule whose head is a predicate $dercando$), yet $dercando$ is not allowed in the body of a cando-rule. Similarly do can appear in the body of none of these rules while both $cando$ and $dercando$ are allowed in the body of a do-rule. In addition, two predicates $done$ and $error$, can be used to specify history-dependent authorisations based on actions previously executed by a subject, and to check the integrity of the authorisation policies. However, the language cannot express temporal dependencies among authorisations. Neither can the approach allow the composition of policies.

Bertino et al. [11] propose a model supporting the specification of periodic authorisations, that is, authorisations that hold in specific periodic intervals. This work extends [12] with positive and negative authorisations and periodic expressions. An ***authorisation*** is a tuple (s, o, m, pn, g) where s is a subject (the grantee), o an object, m an access mode, $pn \in \{+, -\}$ indicates the sign of the authorisation (positive or negative), and g is a subject (the grantor). For instance, by the authorisation $(Ann, o_1, read, +, Tom)$, Tom grants Ann the $read$ access mode on object o_1. A ***periodic authorisation*** is a triple $([begin, end], P, auth)$, where $begin$ is a date expression (e.g. $1/1/94$), end is either the constant ∞, or a date expression denoting an instant greater than or equal to the one denoted by $begin$, P is a periodic expression (such as "Mondays" or "Workingdays"), and $auth$ is an authorisation. For example, the periodic authorisation defined by $([1/1/94, \infty], Mondays, (Ann, o_1, read, +, Tom))$, specified by Tom, states that Ann has the authorisation to read o_1 each Monday starting from $1/1/94$. They also introduce the concept of ***derivation rule*** to express temporal dependencies among authorisations.

A derivation rule is a triple $([begin, end], P, A\ \langle Op \rangle\ \alpha)$ where, $begin$, end, P are defined as above, A is an authorisation, α is a Boolean expression of authorisations and Op is one of the following operators: $WHENEVER$, $ASLONGAS$, $UPON$. With the operator $WHENEVER$, an authorisation A can be derived for each instant between $begin$ and end and satisfying the periodic expression P for which α is valid. Similar semantics are given for the operators $ASLONGAS$ and $UPON$. Although temporal dependencies among authorisation can be specified, the approach is not compositional. Also it is not clear how to derive authorisations (positive or negative) based on the system

state (functional behaviours) or occurrence of events.

Ponder [25], developed at the Imperial College, is a policy language for specifying security and management policies for distributed systems. This declarative language is for specifying different types of policies, grouping them into roles and relationships, and defining configurations and relationships as management structures. Its scope is RBAC and a general purpose management policies. Ponder includes the following types of policies: positive and negative authorisation policies; positive and negative delegation policies; obligation policies and refrain policies. While an obligation policy specifies what actions to take when certain events occur, a refrain policy can be thought off as a negative obligation specifying the actions that a subject must refrain from performing on a target object even though it may actually be permitted to do so. The following positive authorisation policy taken from [24] (key words are in **bold** font) authorises members of the NetworkAdmin domain to load, remove, enable or disable objects of type ProfileT in the Nregion/switches domain.

inst auth+ switchProfileOps {

subject /NetworkAdmin ;

target <ProfileT> /Nregion/switches ;

action load(), remove(), enable(), disable() ;

}

Ponder covers a wide range of security and management policies. However, support for formal reasoning about properties of policies is still lacking in Ponder.

LaSCO (Language for Security Constraints on Objects) [39] is a graphical approach

for specifying security constraints on objects. A policy graph is an annotated directed graph where nodes represents system objects and edges represent system events. Nodes and edges are decorated using *domain predicates* and *requirement predicates*. Domain predicates restrict what can match a node or edge while requirement predicates are constraints to be met on domain match. Although a graphical approach to specifying policies is attractive for human users, LaSCO cannot specify any form of obligation policies, nor support the composition of policies.

Open applications such as e-commerce and other Internet-enabled services require interaction between different, remotely located, parties that may know little about each other. In such a situation, traditional assumptions for establishing and enforcing access control regulations do not hold anymore, making the separation between authentication and access control difficult. A possible solution to this problem is the use of certificates (or credentials), representing statements certified by given entities (e.g. certification authorities), which can be used to establish properties of their holder (such as identity, accreditation, or authorisations).

Trust management systems (such as PolicyMaker [18], KeyNote [19], REFEREE [22], and DL [49]) use credentials to describe specific delegation of trusts among keys and to bind keys to authorisations. Access control decision of whether or not a party may execute an access dependent on properties that the party may have, and can prove by presenting one or more certificates. Many authors in the trust management community have based their policy languages on a tractable fragment of first-order logic. The standard approach (see e.g. Delegation Logic [48], the Role-based Trust-management framework

[50], Binder [27] and SD3 [42]) is to describe policies in such a way that they can be analysed using a variant of Datalog, typically either *safe stratified* Datalog [33] or Datalog with *constraints* [51]. Datalog is an efficient well-understood reasoning engine that was originally designed for function-free negation-free Horn clauses. The variants allow some use of functions and negation, while preserving tractability [37].

Proof-Carrying Code (PCC) [60] is a technique that can be used for safe execution of untrusted code. In a typical instance of PCC, a code receiver establishes a set of safety rules that guarantee safe behaviour of programs, and the code producer creates a formal safety proof that proves, for the untrusted code, adherence to the safety rules. Then, the host is able to use a simple and fast *proof validator* to check, with certainty, that the proof is valid and hence the foreign code is safe to execute. Although the approach can be effective for ensuring security of mobile code, there is still many problems such as how small shall the proof validator be, or how to encode formal proof effectively.

2.5 Integration of Security into System Requirement

In this section, we give an account of related works attempting to integrate security and system requirements. Eckert defines in [30] an action-based computational model for secure systems and a security requirement logic for expressing information flow policies. She then coarsens the model into an object-based computational model in which user representatives and system resources are modelled as objects. She defines on this new model a security requirement logic for expressing access control policies in terms of which user is authorised to perform what method on what object. While an action is assumed to be

atomic (i.e. its execution cannot be interrupted), a method can involve many atomic actions. She also proposes a high level programming language called $INSEL^+$ [29] for specifying a concrete implementation. However, the approach is not compositional and the high level programming language provides no construct for specifying information flow policies within a program. Rather, only information flow policies that can be expressed as access control policies using suitable labelling of objects can be specified in the language.

Devanbu and Stubblebine [28] suggested extending standards such as UML (Unified Modelling Language) to include modelling of security requirements. Following this idea, Alghathbar and Wijesekera [3] proposes *authUML* by integrating a customised version of Flexible Authorisation Framework (FAF) of Jajodia et al. [40] into UML. authoUML allows to specify access control policies in Use Cases which state actors and their intended usage of the envisioned system. The model is supposed to be used by requirements engineers to analyse access control policies for conflicts and incompleteness. However, the model suffers from the limitations of FAF (e.g. temporal security requirements cannot be specified) and the lack of a formal semantics for UML. Some other work [31, 44, 52] has been done on modelling system security using UML. Jürjens [44] proposes UMLsec, a UML profile for modelling and evaluating security aspects based on the multi-level security model. Lodderstedt et al. propose SecureUML [52], an extension of UML with RBAC policies.

2.6 Summary

We have presented an account of related work on models and languages for expressing security policies. The traditional discretionary access control (DAC) model has inspired many authors and new models such as Role-Based Access Control (RBAC) [65] or Team-Based Access Control (TBAC) [73] have been introduced. Several proposals (such as [20, 23, 68]) have attempted a combination of mandatory flow control and discretionary authorisations. Trust management models based access control decision on one or more certificates to be presented by an access requestor. This mechanism is useful in open systems such as the World Wide Web where interactions may take place between parties that do not know each other. All these security models cover a wide range of security policies encountered in practise. However, less work is done on development methods that allow the integration of such security policies into other system requirements in the early stages of system development life cycle. The approaches based on the extension of UML are limited because UML so far lacks a suitable formal semantics to enable formal reasoning.

This thesis proposes a unified compositional formal framework for the development of secure systems. It provides a single formalism (namely Interval Temporal Logic) for expressing the functional, temporal and security requirements of systems at different levels of abstraction. Our development technique is based on step-wise refinement from high level specification down to a concrete implementation as an (secure) action system.

Chapter 3

Computational Model

Objectives

- To present the Interval Temporal Logic (ITL).
- To present the paradigm of Secure Action Systems.
- To define a denotational semantics of secure action systems based on ITL.
- To give an example of secure action system.

3.1 Introduction

In this chapter we present a unified formal framework for the development of secure systems. Fundamental to our approach is the integration of the functional, temporal and security requirements of systems in a single uniform formalism. We take the view that a secure system is developed from a high level specification down to a concrete implementation through successive correctness preserving refinement steps. Such a *top-down*

approach ensures the correctness of the implementation with respect to the initial high level specification by construction. In this respect, Interval Temporal Logic (ITL) is our underlying logic. The functional, temporal and security requirements are expressed in ITL, then refined into a concrete program. We propose a computational model for secure systems based on the *action system* paradigm [6]. In order to reason about properties, whether functional, temporal or security, of secure action systems, we present a denotational semantics of a secure action system in ITL.

3.2 Interval Temporal Logic

Interval Temporal Logic (ITL) is a linear-time temporal logic with a discrete model of time. A system is modelled by a set of relevant state variables. An interval is considered to be a (in)finite, nonempty sequence of states $\sigma_0\sigma_1\ldots$, where a state σ_i is a mapping from a set of variables to a set of values. The length, $|\sigma|$, of a finite interval σ is equal to the number of states in the interval minus one. An empty interval has exactly one state and its length is equal to 0. A *behaviour* is a sequence of states. As such, a behaviour is represented as an interval in our semantic domain. Therefore, the specification of a system represents the set of all acceptable behaviours of the system. This set is denoted by an ITL formula whose syntax and semantics are given below.

3.2.1 Syntax and Informal Semantics

The syntax of ITL is defined as follows, where a is a static variable (does not change within an interval), A is a state variable (can change within an interval), v a static or state

variable, g is a function symbol, and p is a relation symbol.

- Expressions

  $$exp \ ::= \ a \mid A \mid g(exp_1, \ldots, exp_n) \mid \imath \, a : f$$

- Formulae

  $$f \ ::= \ p(exp_1, \ldots, exp_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \cdot f \mid skip \mid f_1; f_2 \mid f^*$$

A *static variable* is global, i.e. its value remains constant throughout the reference interval, while a *state variable* is local, i.e. its value can change within the interval. The function symbols include e.g. arithmetic operators such as $+, -$ and $*$ (multiplication). A constant is denoted by a function without parameter, e.g. $2, 3$ or 5. An expression of the form $\imath \, a : f$ is called a *temporal expression*. It returns a value a for which the formula f holds in the reference interval. For example, the expression

$$\imath \, a : skip; (a = A)$$

returns the value of the state variable A in the second state of the reference interval.

Atomic formulae are constructed using relation symbols such as $=$ and $\leq$. Formulae are then constructed by composing atomic formulae with first order connectives (e.g. $\neg, \wedge, \forall$) and the temporal modalities $skip$ (i.e. an interval of exactly two states), ";" (*chop*) and "*" (*chopstar*).

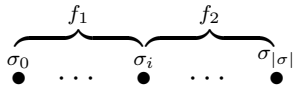
Informal Semantics of the Temporal Modalities

An informal semantics of the temporal modalities can be defined as follows:

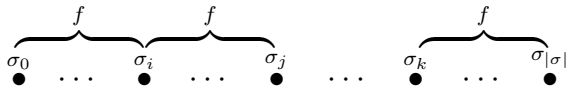
- The formula $skip$ denotes a unit interval (length equal to 1).

  $skip$: 

- The formula $f_1; f_2$ holds for an interval $\sigma = \sigma_0\sigma_1 \ldots \sigma_{|\sigma|}$ if the interval can be decomposed ("chopped") into two parts, a prefix interval $\sigma_0 \ldots \sigma_i$ and a suffix interval $\sigma_i \ldots \sigma_{|\sigma|}$, such that f_1 holds over the prefix and f_2 holds over the suffix, or if the interval is infinite and f_1 holds for that interval.

  $f_1; f_2$: 

- Finally the formula f^* holds for an interval if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

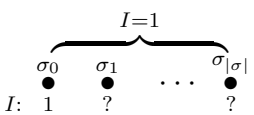
  f^* : 

A formula with no temporal operators in it is called a *state formula*. Such a formula holds for an interval if it holds in the first state of the interval.

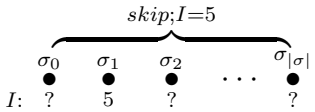
Samples of Formulae

Following are some samples of formulae with their informal meaning, where I is a state variable and the symbol "?" matches any integer value.

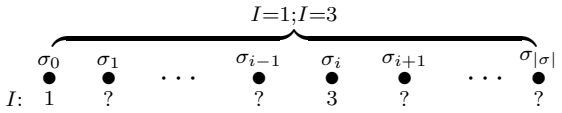
- $I = 1$ holds for an interval if I's value in the interval's initial state is equal to 1.

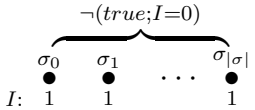
- $skip; I = 5$ holds for an interval if I's value in the interval's second state is equal to 5.



- $I = 1; I = 3$ holds for an interval if I's value in the initial state is equal to 1 and the value of I is equal to 3 in some other state (not necessary the second) of the interval.



- $\neg(true; I = 0)$ holds for an interval if I's value is never equal to 0 within the interval.



Derived Constructs

The following constructs will be used for simplicity.

- The predicates $true$ and $false$ are defined by: $true \mathrel{\widehat{=}} 0 = 0$ and $false \mathrel{\widehat{=}} \neg true$.
- The logical implication is denoted by $\supset$, i.e. $f_1 \supset f_2 \mathrel{\widehat{=}} (\neg f_1) \vee f_2$.
- The logical equivalence is denoted by $\equiv$, i.e. $f_1 \equiv f_2 \mathrel{\widehat{=}} (f_1 \supset f_2) \wedge (f_2 \supset f_1)$.
- Operator $next$ is defined by: $\bigcirc f \mathrel{\widehat{=}} skip; f$ and $\bigcirc exp \mathrel{\widehat{=}} \imath a : \bigcirc(a = exp)$.
- Nonempty interval ($more$) and empty interval ($empty$): $more \mathrel{\widehat{=}} \bigcirc true$ and $empty \mathrel{\widehat{=}} \neg more$.

- Infinite interval : $inf \mathrel{\widehat{=}} true; false$.

- Finite interval: $finite \mathrel{\widehat{=}} \neg inf$.

- Stability of expressions: $stable\ exp \mathrel{\widehat{=}} \Box(more \supset (\bigcirc exp) = exp)$.

- Interval length: $len \mathrel{\widehat{=}} \imath\, a : (\exists I.(I = 0 \wedge (skip \wedge (\bigcirc I) = I + 1)^{*} \wedge fin\ (I = a)))$

- Useful temporal modalities:

$\Diamond f$	$\widehat{=}$	$finite; f$	f holds for some suffix subinterval.
$\Box f$	$\widehat{=}$	$\neg\Diamond\neg f$	f holds for all suffix subintervals.
$\Diamond_i f$	$\widehat{=}$	$f; true$	f holds for some prefix subinterval.
$\Box_i f$	$\widehat{=}$	$\neg\Diamond_i\neg f$	f holds for all prefix subintervals.
$\Diamond_a f$	$\widehat{=}$	$\Diamond(f; true)$	f holds for some subinterval.
$\Box_a f$	$\widehat{=}$	$\neg\Diamond_a\neg f$	f holds for all subintervals.
$fin\ f$	$\widehat{=}$	$\Box(empty \supset f)$	f holds in the final state.
$halt\ f$	$\widehat{=}$	$\Box(empty \equiv f)$	f holds exactly in the final state.
$\lceil f \rceil^{0}$	$\widehat{=}$	$f \wedge empty$	f holds for a point interval.
$\lceil f \rceil^{1}$	$\widehat{=}$	$f \wedge skip$	f holds for a unit interval.
$keep\ f$	$\widehat{=}$	$\Box_a(skip \supset f)$	f holds for all unit subintervals.

3.2.2 Formal Semantics

The meaning of terms and formulae are defined in this section. We are interested only in the functions and relations over integer numbers. Let $\mathbb{Z}$ stand for the set of integer numbers, and Var the set of integer variables. In the sequel, we denote by $E \rightarrow F$ the set of all *total* functions from E to F.

We assume that a total function

$$\hat{g} \in \mathbb{Z}^n \to \mathbb{Z}$$

is associated with each n-ary function symbol g, and a total function

$$\hat{p} \in \mathbb{Z}^n \to \{\text{tt}, \text{ff}\}$$

is associated with each n-ary relation symbol p.

Function symbols, e.g. $+$ and $-$, and relation symbols, e.g. $\geq$ and $=$, are assumed to have their standard meanings. In particular, the truth-values tt and ff are associated with $true$ and $false$, respectively.

Expressions and formulae are evaluated over intervals. Remember that an interval is defined to be a (in)finite nonempty sequence of states $\sigma \mathrel{\widehat{=}} \sigma_0\sigma_1\ldots$, where each state σ_i is a *value assignment* which associates an integer number with each variable:

$$\sigma_i \in \Sigma \mathrel{\widehat{=}} Var \to \mathbb{Z}.$$

We denote by Σ^+ and Σ^ω the sets of finite intervals and the set of infinite intervals respectively. If σ is an infinite interval, we take $|\sigma| = \infty$ and write $\sigma = \sigma_0\sigma_1\ldots\sigma_\infty$. Furthermore, for any $i, j \in \mathbb{N}$ and an interval σ such that $i \leq j \leq |\sigma|$, we write $\sigma[i, j]$ to denote the subinterval $\sigma_i \ldots \sigma_j$ of σ. Given two intervals $\sigma, \sigma' \in (\Sigma^+ \cup \Sigma^\omega)$, we write $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mappings for the variable v, i.e $|\sigma| = |\sigma'|$ and $v \neq v' \Rightarrow \sigma_i(v') = \sigma'_i(v')$ for $i = 0, 1, \ldots, |\sigma|$.

The *semantics of an expression* exp is a function

$$\mathcal{E}[\![exp]\!] \in (\Sigma^+ \cup \Sigma^\omega) \to \mathbb{Z},$$

defined inductively on the structure of expressions by

$$\begin{aligned}
\mathcal{E}[\![v]\!](\sigma) &= \sigma_0(v) \\
\mathcal{E}[\![g(exp_1,\ldots,exp_n)]\!](\sigma) &= \hat{g}(\mathcal{E}[\![exp_1]\!](\sigma),\ldots,\mathcal{E}[\![exp_n]\!](\sigma)) \\
\mathcal{E}[\![\imath a : f]\!](\sigma) &= \begin{cases} \chi(u) \text{ if } u \neq \emptyset \\ \chi(\mathbb{Z}) \text{ otherwise} \end{cases}
\end{aligned}$$

where $u = \{\sigma'(a) \mid \sigma \sim_a \sigma' \text{ and } \sigma' \models f\}$, χ denotes a choice function which maps any nonempty set to some element in the set, and $\sigma' \models f$ means that the interval σ' satisfies the formula f.

The *semantics of a formula* f is a function

$$\mathcal{M}[\![f]\!] \in (\Sigma^+ \cup \Sigma^\omega) \to \{\text{tt}, \text{ff}\},$$

defined inductively on the structure of formulae below, where the following abbreviation is used:

$$\sigma \models f \;\widehat{=}\; \mathcal{M}[\![f]\!](\sigma) = \text{tt}$$

$$\sigma \not\models f \;\widehat{=}\; \mathcal{M}[\![f]\!](\sigma) = \text{ff}$$

The definition of $\mathcal{M}[\![f]\!]$ is

$$
\begin{array}{lll}
\sigma \models p(exp_1, \ldots, exp_n) & \text{iff} & \hat{p}(\mathcal{E}[\![exp_1]\!](\sigma), \ldots, \mathcal{E}[\![exp_n]\!](\sigma)) \\
\sigma \models \neg f & \text{iff} & \sigma \not\models f \\
\sigma \models f_1 \wedge f_2 & \text{iff} & \sigma \models f_1 \text{ and } \sigma \models f_2 \\
\sigma \models \forall v.\ f & \text{iff} & \sigma' \models f, \text{ for all } \sigma' \text{ such that } \sigma \sim_v \sigma' \\
\sigma \models skip & \text{iff} & |\sigma| = 1 \\
\sigma \models f_1; f_2 & \text{iff} & (\sigma[0,k] \models f_1 \text{ and } \sigma[k,|\sigma|] \models f_2, \\
 & & \text{for some } k \in \mathbb{N},\ 0 \leq k \leq |\sigma|) \text{ or } (|\sigma| = \infty \text{ and } \sigma \models f_1). \\
\sigma \models f^* & \text{iff} & (\text{exist } l_0, \ldots, l_n \in \mathbb{N} \text{ such that } l_0 = 0 \leq \ldots \leq l_n = |\sigma| \\
 & & \text{and } \sigma[l_i, l_{i+1}] \models f, 0 \leq i \leq n-1) \text{ or} \\
 & & (\text{exist } l_0, \ldots, l_\infty \in \mathbb{N} \text{ such that } l_0 = 0 \\
 & & \text{and } l_i \leq l_{i+1} \text{ and } \sigma[l_i, l_{i+1}] \models f,\ i \in \mathbb{N})
\end{array}
$$

A formula f is *valid*, written

$$\models f$$

iff $\sigma \models f$ for every interval $\sigma \in (\Sigma^+ \cup \Sigma^\omega)$. A formula f is *satisfiable* iff $\sigma \models f$ for some interval σ.

ITL has also got a sound compositional proof system. Interested readers are referred to [21, 59] for the proof system and further details about the logic. In the sequel, we write

$$\vdash f$$

to mean that the formula f is provable using the axioms and the proof rules of the ITL's proof system.

3.3 Operator Always-followed-by

We define the operator *always-followed-by*, denoted by the symbol "$\mapsto$", as follows:

$$f \mapsto w \mathrel{\widehat{=}} \boxdot((\Diamond f) \supset fin\, w) \tag{3.1}$$

where f stands for any ITL formula, and w is a state formula. The definition (3.1) states that whenever the formula f holds in a (finite) subinterval then the state formula w must hold in the final state of that subinterval. That is, f is always followed by w.

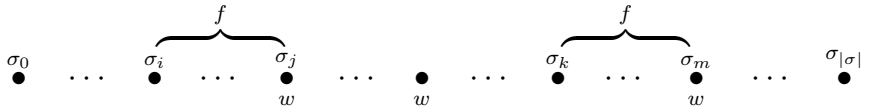


Figure 3.1: Graphical illustration of $f \mapsto w$.

Informally, the meaning of the operator can be portrayed as in Figure 3.1, where each bullet represents a state. Note that in the definition (3.1), w can be true in a state even if f does not hold in the left neighbourhood of the state.

This operator will be used in the subsequent chapters to express security policy rules. The following theorems state useful properties of the operator.

Theorem 3.1

1. $\vdash (true \mapsto w) \equiv \Box w$.

2. $\vdash false \mapsto w$.

3. $\vdash f \mapsto true$.

4. $\vdash (\lceil w_1 \rceil^0 \mapsto w_2) \equiv \Box(w_1 \supset w_2)$.

Theorem 3.1 states some special forms of formula that are useful for specification and to simplify proofs. For example, a formula of the form $true \mapsto w$ holds in an interval if w holds in every state of the interval, while a formula of the form $false \mapsto w$ holds in all intervals (i.e. is valid) regardless the form of w. We give the proofs of (1) and (4). The proofs of others are similar.

Proof.

- Proof of (1)

  $true \mapsto w$

  $\equiv$ {by definition}

  $\boxdot_{\mathrm{i}}((\Diamond true) \supset fin\ w)$

  $\equiv$ $\{(\Diamond true) \equiv (true; true) \equiv true\}$

  $\boxdot_{\mathrm{i}}(true \supset fin\ w)$

  $\equiv$ {ITL}

  $\boxdot_{\mathrm{i}}(false \vee fin\ w)$

  $\equiv$ {ITL}

  $\boxdot_{\mathrm{i}}(fin\ w)$

  $\equiv$ {ITL}

  $\Box w$

- Proof of (4)

  $\lceil w_1 \rceil^0 \mapsto w_2$

  $\equiv$ {by definition}

  $\boxdot_{\mathrm{i}}((\Diamond \lceil w_1 \rceil^0) \supset fin\ w_2)$

$\equiv$ {by definition}

$\boxdot_{\mathrm{i}}((\Diamond(w_1 \wedge empty)) \supset fin\ w_2)$

$\equiv$ $\{(\Diamond(w_1 \wedge empty)) \equiv fin\ w_2\}$

$\boxdot_{\mathrm{i}}(fin\ w_1 \supset fin\ w_2)$

$\equiv$ {ITL}

$\boxdot_{\mathrm{i}}(w_1 \supset w_2)$

$\equiv$ {ITL}

$\Box(w_1 \supset w_2)$

Theorem 3.2 $\vdash ((f \mapsto w_1) \wedge (\lceil w_1 \rceil^0 \mapsto w_2)) \supset (f \mapsto w_2)$.

Theorem 3.2 says that any interval that satisfies $f \mapsto w_1$ and $\lceil w_1 \rceil^0 \mapsto w_2$ also satisfies $f \mapsto w_2$.

Proof.

$(f \mapsto w_1) \wedge (\lceil w_1 \rceil^0 \mapsto w_2)$

$\equiv$ {by definition}

$\boxdot_{\mathrm{i}}(\Diamond f \supset fin\ w_1) \wedge \boxdot_{\mathrm{i}}(\Diamond(w_1 \wedge empty) \supset fin\ w_2)$

$\equiv$ $\{(\Diamond(w_1 \wedge empty)) \equiv fin\ w_1\}$

$\boxdot_{\mathrm{i}}(\Diamond f \supset fin\ w_1) \wedge \boxdot_{\mathrm{i}}(fin\ w_1 \supset fin\ w_2)$

$\equiv$ {ITL}

$\boxdot_{\mathrm{i}}((\Diamond f \supset fin\ w_1) \wedge (fin\ w_1 \supset fin\ w_2))$

$\supset$ {ITL}

$\boxdot_{\mathrm{i}}(\Diamond f \supset fin\ w_2)$

$\equiv$ {by definition}

$f \mapsto w_2$

Theorem 3.3 If $\vdash f_2 \supset f_1$ and $\vdash w_1 \supset w_2$ then $\vdash (f_1 \mapsto w_1) \supset (f_2 \mapsto w_2)$.

Theorem 3.3 states that the operator $\mapsto$ is monotonic with respect to implication.

Proof.

We assume $\vdash f_2 \supset f_1$ and $\vdash w_1 \supset w_2$ and prove $\vdash (f_1 \mapsto w_1) \supset (f_2 \mapsto w_2)$.

$(f_1 \mapsto w_1)$

$\equiv$ {by definition}

$\boxdot(\Diamond f_1 \supset fin\ w_1)$

$\supset$ $\{\vdash w_1 \supset w_2\}$

$\boxdot(\Diamond f_1 \supset fin\ w_2)$

$\supset$ $\{\vdash f_2 \supset f_1\}$

$\boxdot((\Diamond f_2 \supset fin\ w_2)$

$\equiv$ {by definition}

$f_2 \mapsto w_2$

Theorem 3.4 $\vdash (f \mapsto (w_1 \wedge w_2)) \equiv ((f \mapsto w_1) \wedge (f \mapsto w_2))$.

Theorem 3.4 asserts that the operator "$\mapsto$" is left distributive over "$\wedge$".

Proof.

$f \mapsto (w_1 \wedge w_2)$

$\equiv$ {by definition}

$\boxdot(\Diamond f \supset (w_1 \wedge w_2))$

$\equiv$ {distribution of $\supset$ over $\wedge$}

$\boxdot_1((\Diamond f \supset w_1) \wedge (\Diamond f \supset w_2))$

$\equiv$ {distribution of $\boxdot_1$ over $\wedge$}

$(\boxdot_1(\Diamond f \supset w_1)) \wedge (\boxdot_1(\Diamond f \supset w_2))$

$\equiv$ {by definition}

$(f \mapsto w_1) \wedge (f \mapsto w_2)$

Theorem 3.5 $\vdash ((f_1 \vee f_2) \mapsto w) \equiv ((f_1 \mapsto w) \wedge (f_2 \mapsto w))$.

Theorem 3.5 states an interesting property between the operator "$\mapsto$" and the Boolean connectors "$\vee$" and "$\wedge$".

Proof.

$(f_1 \vee f_2) \mapsto w$

$\equiv$ {by definition}

$\boxdot_1(\Diamond(f_1 \vee f_2) \supset w)$

$\equiv$ {distribution of $\Diamond$ over $\vee$}

$\boxdot_1(((\Diamond f_1) \vee (\Diamond f_2)) \supset w)$

$\equiv$ {property of $\supset$ and $\vee$}

$\boxdot_1((\Diamond f_1 \supset w) \wedge (\Diamond f_2 \supset w))$

$\equiv$ {distribution of $\boxdot_1$ over $\wedge$}

$(\boxdot_1(\Diamond f_1 \supset w)) \wedge (\boxdot_1(\Diamond f_2 \supset w))$

$\equiv$ {by definition}

$(f_1 \mapsto w) \wedge (f_2 \mapsto w)$

Note that a formula $f \mapsto w$ determines the value of w only in the final states of the

subintervals in which f holds and does not constrain its value elsewhere. For this reason, the operator $\mapsto$ is called *weak always-followed-by*.

3.3.1 Strong always-followed-by

The operator *strong always-followed-by*, denoted by the symbol "$\leftrightarrow$", is defined as follows:

$$f \leftrightarrow w \mathrel{\widehat{=}} \boxed{\text{i}}((\Diamond f) \equiv fin\ w) \tag{3.2}$$

where f stands for any ITL formula, and w is a state formula. The definition (3.2) states that the state formula w holds in a state if and only if the formula f holds in some left neighbourhood of that state in the reference interval. Informally, the meaning of the operator can be portrayed as in Figure 3.2, where each bullet represents a state.

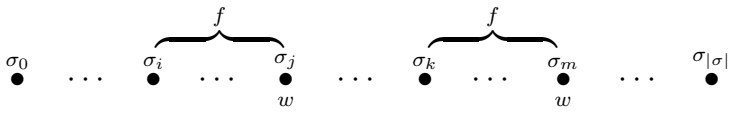


Figure 3.2: Graphical illustration of $f \leftrightarrow w$.

Unlike the operator weak always-followed-by, a rule of the form $f \leftrightarrow w$ determines in any state the value of the state formula w. If f holds in some left neighbourhood of a state in the reference interval, then w must hold in that state otherwise w must not hold in that state. Example 3.1 illustrates the difference between the two operators.

Example 3.1 *Let σ, σ' and σ'' be three intervals defined as follows, where X is a state variable:*

$$\begin{array}{lcccccccccc} \sigma: & \overset{\sigma_0}{\bullet} & \overset{\sigma_1}{\bullet} & \overset{\sigma_2}{\bullet} & \overset{\sigma_3}{\bullet} & \overset{\sigma_4}{\bullet} & \overset{\sigma_5}{\bullet} & \overset{\sigma_6}{\bullet} & \overset{\sigma_7}{\bullet} & \overset{\sigma_8}{\bullet} & \overset{\sigma_9}{\bullet} \\ X: & 1 & 3 & 2 & 0 & 7 & 2 & 0 & 2 & 0 & 1 \end{array}$$

$$\begin{array}{lcccccccccc} \sigma': & \overset{\sigma'_0}{\bullet} & \overset{\sigma'_1}{\bullet} & \overset{\sigma'_2}{\bullet} & \overset{\sigma'_3}{\bullet} & \overset{\sigma'_4}{\bullet} & \overset{\sigma'_5}{\bullet} & \overset{\sigma'_6}{\bullet} & \overset{\sigma'_7}{\bullet} & \overset{\sigma'_8}{\bullet} & \overset{\sigma'_9}{\bullet} \\ X: & 1 & 3 & 2 & 0 & 7 & 3 & 0 & 2 & 0 & 1 \end{array}$$

$$\begin{array}{lcccccccccc} \sigma'': & \overset{\sigma''_0}{\bullet} & \overset{\sigma''_1}{\bullet} & \overset{\sigma''_2}{\bullet} & \overset{\sigma''_3}{\bullet} & \overset{\sigma''_4}{\bullet} & \overset{\sigma''_5}{\bullet} & \overset{\sigma''_6}{\bullet} & \overset{\sigma''_7}{\bullet} & \overset{\sigma''_8}{\bullet} & \overset{\sigma''_9}{\bullet} \\ X: & 1 & 3 & 2 & 0 & 7 & 2 & 3 & 2 & 0 & 1 \end{array}$$

We consider the following formulae:

$\psi_1 \mathrel{\widehat{=}} (X = 2 \wedge skip) \mapsto X = 0$ *and*

$\psi_2 \mathrel{\widehat{=}} (X = 2 \wedge skip) \leftrightarrow X = 0$.

The formula ψ_1 holds in an interval if every state of that interval in which the value of the state variable X is 2 is followed by a state in which the value of X is 0. Similarly, the formula ψ_2 holds in an interval if for each state of that interval the value of the state variable X is 2 if and only if its value in the next state 0. It follows that:

- $\sigma \models \psi_1$ *and* $\sigma \models \psi_2$*;*
- $\sigma' \models \psi_1$ *and* $\sigma' \not\models \psi_2$ *(because of the subinterval $\sigma'_5\sigma'_6$);*
- $\sigma'' \not\models \psi_1$ *and* $\sigma'' \not\models \psi_2$ *(both because of the subinterval $\sigma''_5\sigma''_6$).*

Theorem 3.6 $\vdash (f \leftrightarrow w) \supset (f \mapsto w)$.

Theorem 3.6 establishes that the operator *strong always-followed-by* is a refinement of the operator weak always-followed-by.

Proof.

$$
\begin{aligned}
& (f \leftrightarrow w) \\
\equiv \quad & \{\text{by definition}\} \\
& \boxdot_{\mathrm{i}}((\Diamond f) \equiv w) \\
\equiv \quad & \{\text{ITL}\} \\
& \boxdot_{\mathrm{i}}(((\Diamond f) \supset w) \wedge (w \supset (\Diamond f))) \\
\equiv \quad & \{\text{distribution of } \boxdot \text{ over } \wedge\} \\
& (\boxdot_{\mathrm{i}}((\Diamond f) \supset w)) \wedge (\boxdot_{\mathrm{i}}(w \supset (\Diamond f))) \\
\supset \quad & \{\text{ITL}\} \\
& \boxdot_{\mathrm{i}}(\Diamond f \supset w) \\
\equiv \quad & \{\text{by definition}\} \\
& f \mapsto w
\end{aligned}
$$

The following theorems are similar to those defined for the operator $\mapsto$. Their proofs are also similar and are omitted here.

Theorem 3.7

1. $\vdash (true \leftrightarrow w) \equiv \Box w$.

2. $\vdash (false \leftrightarrow w) \equiv \Box \neg w$.

3. $\vdash (f \leftrightarrow true) \equiv \boxdot_{\mathrm{a}} f$.

4. $\vdash (f \leftrightarrow false) \equiv \diamondsuit_{\mathrm{a}} \neg f$.

5. $\vdash (\lceil w_1 \rceil^0 \leftrightarrow w_2) \equiv \Box(w_1 \equiv w_2)$.

Theorem 3.7 states some interesting forms of formulae, similar to those of Theorem 3.1. A formula of the form $true \leftrightarrow w$ is equivalent to $true \mapsto w$. Unlike for the operator weak always-followed-by, a formula of the form $false \leftrightarrow w$ holds for an interval if w is false in every state of the interval. Similarly, a formula of the forms $f \leftrightarrow true$ is interesting and it expresses that f holds for all subintervals of the reference interval. A formula of the form $f \leftrightarrow false$ means f does not hold in some subinterval of the reference interval.

Theorem 3.8 $\vdash ((f \leftrightarrow w_1) \wedge (\lceil w_1 \rceil^0 \leftrightarrow w_2)) \supset (f \leftrightarrow w_2)$.

Theorem 3.8 says that any intervals satisfying $f \leftrightarrow w_1$ and $\lceil w_1 \rceil^0 \leftrightarrow w_2$ also satisfies the formula $f \leftrightarrow w_2$.

Theorem 3.9 $\vdash ((f \leftrightarrow w_1) \wedge (f \leftrightarrow w_2)) \supset (f \leftrightarrow (w_1 \wedge w_2))$.

Theorem 3.9 asserts that a formula of the form $f \leftrightarrow (w_1 \wedge w_2)$ can be refined into the conjunction of the two formulas $f \leftrightarrow w_1$ and $f \leftrightarrow w_2$.

Theorem 3.10 $\vdash ((f_1 \leftrightarrow w) \wedge (f_2 \leftrightarrow w)) \supset ((f_1 \vee f_2) \leftrightarrow w)$.

Theorem 3.10 says that a formula of the form $(f_1 \vee f_2) \leftrightarrow w$ can be refined into the conjunction of the two formulas $f_1 \leftrightarrow w$ and $f_2 \leftrightarrow w$.

3.4 Secure Action Systems

Action systems were first introduced by Back [6] as a paradigm to describe distributed systems. Here we adapt the formalism to cater for the specification of secure agent systems. In a Secure Action System (SAS) paradigm, a system is viewed as

- a collection of agents,
- a collection of actions and
- a security policy.

An agent may be a user representative on behalf of whom it performs actions. An agent may also represent an object (e.g. a file) in the system. An agent (or group of agents) may perform an action on another agent provided that it has the appropriate right to do so. The access rights of agents are defined by the security policy. In this section we give the syntax and the informal semantics of secure action systems. In the following syntax, symbols in **bold** are key-words and optional expressions are surrounded with the symbols '[' and ']'.

A secure action system is a tuple $SAS \mathrel{\widehat{=}} (\mathcal{P}, \mathcal{A}, Agents, Actions, Policy)$ where

- $\mathcal{P}$ is a nonempty set of *agent* names,
- $\mathcal{A}$ is a nonempty set of *action* names,
- $Agents$, is a set of agent definitions of the form

(3.3) $$\mathbf{agent}\ \ p\ [:\ \mathbf{var}\ \ y\ [; S]].$$

where $p \in \mathcal{P}$ is the agent name, y is a set of local variables and the statement S stands for the initialisation of the local variables in y.

In the sequel, we will let $Var(p)$ and $Init(p)$ to denote respectively the set y of local variables and the initialisation statement S of the agent p.

Let $Var \mathrel{\widehat{=}} \bigcup_{p \in \mathcal{P}} Var(p)$ be the set of all the variables declared in the system and let D be the range of these variables. We denote by

$$\Sigma \mathrel{\widehat{=}} Var \to D$$

the set of all possible *states* of the system, where a state is a mapping of variables to values.

- $Actions$ is a set of action definitions of the form

(3.4) $$\textbf{action } a \textbf{ on } R_a \textbf{ by } U_a \, [\textbf{async } W_a] : \; g_a \to S_a.$$

where $a \in \mathcal{A}$ is the action name, R_a, U_a and W_a are sets of agents, the Boolean condition g_a is *functional guard* of the action, and the statement S_a is the *body* of the action. The agents in the set R_a are called *target agents*. These agents are protected against other participating agents which must have an appropriate right for the action to execute. The set U_a contains the agents which must synchronise with the target agents for the action to be executed. In the sequel, we call an agent in the set $R_a \cup U_a$ a *synchronisation agent* of action a. Two actions which have a common synchronisation agent cannot be executed in parallel. On the contrary, the agents in the set W_a need not be synchronised during the execution of the action. That is, they can be involved in the execution of another action while action a is being executed. However, the state of an agent in W_a cannot be modified by the body of the action a, but can only be read.

We denote by $\mathcal{P}_a \mathrel{\widehat{=}} R_a \cup U_a \cup W_a$, the set of all agents that participate in action a. We also denote by $Var(a) \mathrel{\widehat{=}} \bigcup_{p \in \mathcal{P}_a} Var(p)$, the set of local variables of all the

agents participating in the action a. The functional guard and the body of an action a may only refer to the local variables in $Var(a)$.

For each agent $p \in \mathcal{P}$, we denote by $\mathcal{A}_p$ the set of actions the agent p participates in as synchronisation agent.

- $Policy \mathrel{\widehat{=}} \{autho^+, autho^-, autho\}$ is a set of three total functions each having the following signature:

$$(\mathcal{P} \times \mathcal{P} \times \mathcal{A}) \rightarrow \Sigma \rightarrow \{true, false\}.$$

Intuitively, each of these functions associates with each triple $(p_1, p_2, a) \in \mathcal{P} \times \mathcal{P} \times \mathcal{A}$ a predicate over the set Var of program variables. These three functions are used to specify the security policy of the system.

The functions $autho^+$ and $autho^-$ express positive and negative authorisations respectively. The predicate $autho^+(p_1, p_2, a)$ is true in a state if the agent p_1 is *explicitly* authorised to perform the action a on the agent p_2 and false otherwise. Similarly, the predicate $autho^-(p_1, p_2, a)$ is true in a state if the agent p_1 is *explicitly* denied the right to perform the action a on the agent p_2 and false otherwise. The function $autho$ is used to resolve conflicts that may occur among positive and negative authorisations, i.e. when $autho^+(p_1, p_2, a)$ and $autho^-(p_1, p_2, a)$ hold in the same state. For example, denial takes precedence if

$$autho(p_1, p_2, a) \mathrel{\widehat{=}} (autho^+(p_1, p_2, a) \wedge \neg autho^-(p_1, p_2, a)).$$

Definition 3.1 *A guard is said to be* local *to an agent if it refers only to the local variables of that agent.*

Definition 3.2 *A guard is* separable *if it is a Boolean combination of local guards.*

The following restrictions are considered to avoid accidental exposure or modification of agent private information:

R1: The functional guard of an action is assumed to be separable. This restriction avoids implicit information flows between agents through the action functional guard.

R2: In the body of an action a, information can flow only from the agents in the set $U_a \cup W_a$ to the agents in the set R_a, or from the agents in the set R_a to the agents in the set U_a; no information flow is allowed between distinct agents in the set $R_a \setminus U_a$ nor between distinct agents in the set $U_a \setminus R_a$.

R3: The body of an action a should not contain statements that change the state of an agent in W_a.

Definition 3.3 *An action that involves more that one distinct participating agent is called a* joint action*, otherwise it is called a* private action*.*

Definition 3.4 *If $a \in \mathcal{A}$ is an action defined as in (3.4), we call the* security guard *of the action a the expression*

$$h_a \mathrel{\widehat{=}} (\bigwedge_{p_1 \in (U_a \cup W_a)} \bigwedge_{p_2 \in R_a} autho(p_1, p_2, a)).$$

Definition 3.5 *We say that an agent is* ready *for the execution of an action if it is not a synchronisation agent for the action or it is not currently involved in the execution of another action for which the agent is a synchronisation agent.*

Definition 3.6 *We say that an action is* enabled *if both its functional guard and its security guard evaluate to* $true$ *and all the participating agents are ready.*

The execution of a SAS proceeds as follows. The initialisation statements of the agents are executed to create an initial state of the system. We assume that a variable which is not explicitly initialised is assigned an arbitrary value (in its range). Starting from the initial state, the actions are repeatedly executed as long as there are actions that are enabled. The initialisation statements and the bodies of the actions are assumed to be deterministic. We also assume that an initialisation statement always terminates, and the body of an action terminates when the guard is true.

At any time, an agent is involved in the execution of at most one action for which it is a synchronisation agent. When many actions involving a common synchronisation agent are enabled, one of them is chosen nondeterministically for execution. As long as this restriction is obeyed, any number of actions may be executed in parallel.

A joint action is the only mechanism to model synchronisation and communication among agents. No explicit synchronisation or communication primitives are needed in the language. Synchronisation is achieved through the guard (that must be true for the action to be executed) and the readiness of the participating agents. Communication may take place in the body of the action by updating an agent's local state using information from other agents.

Note that SAS is a conservative extension of Back's Action System paradigm [7]. Indeed, a SAS for which the function $autho$ denotes the predicate $true$ and the actions do not contain the clause $\mathbf{async}$ is equivalent to a Back's action system.

We adopt the following syntax for the statements S, where x is a (list of) variable(s), exp denotes an (list of) expression(s), b stands for a Boolean expression, K is an integer variable and n a natural number.

Definition 3.7

$$S \;::=\; x := exp \mid \texttt{skip} \mid S_1; S_2$$
$$\mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi} \mid \texttt{for } K := 1 \texttt{ to } n \texttt{ do } S \texttt{ od}$$

It follows that a statement is a sequential process of any granularity provided it is deterministic and terminates in finite time. In an assignment statement $x := exp$, the list exp of expressions must match the list x of recipient variables. We assume that the initialisation statement $Init(p)$ of an agent $p \in \mathcal{P}$, if not empty, has the form $y := c$, where c is a list of constants.

Example 3.2 *Suppose a transfer of a large amount of money say $£10,000$ in a bank requires the cooperation of two employees who have each the authority to perform such a transaction. The two employees must agree for the transaction to take place. Indeed, this security measure is usually taken to limit frauds by insiders. The policy is enforced by the following specification, where $acc[i],\ i = 0, \ldots, m$ are bank accounts and $emp[i]$, $i = 1, \ldots, n$ are employees.*

agent $acc[i = 1, \ldots, m]$: **var** $secLevel, balance;\ secLevel, balance := i \bmod 3, 0.$

agent $emp[i = 1, \ldots, n]$: **var** $secClearance, ok;\ secClearance, ok :=$
$i \bmod 4, false.$

action $transfer[i, j = 1, \ldots, m; k, l = 1, \ldots, n]$ **on** $acc[i], acc[j]$ **by** $emp[k], emp[l]$:

$(acc[i].balance >= 10000 \ \wedge \ emp[k].ok \wedge emp[l].ok \ \wedge \ k \neq l) \rightarrow$

$emp[k].ok, emp[l].ok, acc[i].balance, acc[j].balance := false, false,$

$acc[i].balance - 10000, acc[j].balance + 10000.$

The authorisation predicates are defined as follows, where x and y range over agents and z over actions:

$autho^{+}(emp[i], acc[j], transfer) \mathrel{\widehat{=}} emp[i].secClearance \geq acc[j].secLevel,$ *for* $i = 1, \ldots, n$ *and* $j = 1, \ldots, m.$

$autho^{+}(x, y, transfer) \mathrel{\widehat{=}} false$ *elsewhere.*

$autho^{-}(emp[i], acc[j], transfer) \mathrel{\widehat{=}} emp[i].secClearance \leq acc[j].secLevel,$ *for* $i = 1, \ldots, n$ *and* $j = 1, \ldots, m.$

$autho^{-}(x, y, transfer) \mathrel{\widehat{=}} true$ *elsewhere.*

$autho(x, y, z) \mathrel{\widehat{=}} autho^{+}(x, y, z) \wedge \neg autho^{-}(x, y, z).$

In Example 3.2, the variable $secLevel$ stores the security level of the corresponding bank account. We assume there are three security levels from 0 to 2 corresponding respectively to *low*, *medium*, and *high*. Similarly, each employee is assigned a security clearance. The security policy says that an employee is authorised to perform a money transfer from or to a bank account if his/her security clearance is greater than the security level of the bank account. Furthermore, a transfer can take place only under the agreement of two authorised employees. The policy is enforced by implementing the action $transfer$ as a joint action that synchronises the bank account source, the bank account destination and two employees. Each of the employee expresses his/her agreement for a transfer by setting the local variable ok to true.

The formal semantics of SASs is presented in the following section.

3.5 Formal Semantics of SAS

In order to reason about properties of secure action systems, we present in this section their formal semantics in ITL. Let $SAS \mathrel{\widehat{=}} (\mathcal{P}, \mathcal{A}, Agents, Actions, Policy)$ be a secure action system. Firstly, the set of variables is extended with a collection of (control) Boolean variables $ready_p$, for each agent $p \in \mathcal{P}$, indicating whether the agent is ready or currently involved in the execution of some action. Let $Var' \mathrel{\widehat{=}} Var \cup \{ready_p \mid p \in \mathcal{P}\}$ be the new set of variables. The semantics of a statement S (described by the grammar given in Definition 3.7) relatively to a set V of variable symbols is an ITL formula $\mathcal{C}_V[\![S]\!]$ defined inductively on the structure of statements as follows:

$$\mathcal{C}_V[\![x := exp]\!] \mathrel{\widehat{=}} skip \wedge ((\bigcirc x) = exp) \wedge \bigwedge_{u \in (V \setminus x)} stable(u).$$

$$\mathcal{C}_V[\![\texttt{skip}\ ]\!] \mathrel{\widehat{=}} \mathcal{C}_V[\![V := V]\!].$$

$$\mathcal{C}_V[\![S_1; S_2]\!] \mathrel{\widehat{=}} \mathcal{C}_V[\![S_1]\!]; \mathcal{C}_V[\![S_2]\!].$$

$$\mathcal{C}_V[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] \mathrel{\widehat{=}} (b \wedge \mathcal{C}_V[\![S_1]\!]) \vee (\neg b \wedge \mathcal{C}_V[\![S_2]\!]).$$

$$\mathcal{C}_V[\![\texttt{for}\ K := 1\ \texttt{to}\ n\ \texttt{do}\ S]\!] \mathrel{\widehat{=}} \mathcal{C}_V[\![K := 1]\!]; (\mathcal{C}_V[\![S; K := K + 1]\!])^n.$$

Note that the semantics $\mathcal{C}_V[\![S]\!]$ of a statement S relatively to a set of variables V controls only the behaviours of the variables in the set V. Therefore, any variable not in V can change independently during the execution of the statement S. For example, the semantics of an assignment $x := exp$ relatively to a set of variable V is the set of all

intervals (behaviours) $\sigma \mathrel{\widehat{=}} \sigma_0\sigma_1$ of length one for which the value of x in the next state is equal to the value of the expression exp in the initial state, and the other variables in V remain unchanged. Figure 3.3 illustrates the semantics of $x := 2$ relatively to the set $\{x, y, z\}$ which is a subset of the system state $\{x, y, z, u\}$. The value of x is equal to 2 in the next state and y and z are kept unchanged. However, the variable u is controlled by the environment and therefore can take any value in the next state.

σ_0	σ_1
•	•
$x : 0$	$x : 2$
$y : 0$	$y : 0$
$z : 0$	$z : 0$
$u : 0$	$u : 5$

Figure 3.3: Illustration of $\mathcal{C}_{\{x,y,z\}}[\![x := 2]\!]$

Remember that the body of an action $a \in \mathcal{A}$ defined as in Equation 3.4 (see page 49), cannot contain statements that change the state of an agent in the set W_a. Only variables in the states of the synchronisation agents of the action can be modified by the execution of the action. Let $V_a \mathrel{\widehat{=}} \bigcup_{p \in R_a \cup U_a} Var'(p)$ be the set of variables that are allowed to be modified by the execution of the action a. We are now ready to give the semantics of an action in ITL.

An action a is enabled if both its functional guard g_a and its security guard h_a holds and all its synchronisation agents are ready. This is formulated in ITL by the formula

$$enabled_a \mathrel{\widehat{=}} g_a \wedge h_a \wedge \bigwedge_{p \in R_a \cup U_a} ready_p.$$

The semantics of an action a is given by the formula

$$\psi_a \;\widehat{=}\; enabled_a \;\wedge\; \mathcal{C}_{V_a}[\![ready^a := false^a; S_a; ready^a := true^a]\!],$$

where $ready^a$ is the list of all the variable $ready_p,\ p \in \mathcal{P}_a$ and $false^a$ (resp. $true^a$) is a list of constants $false$ (resp. $true$) that matches the list $ready^a$. Note that the variables in V_a which are not modified in the body S_a of the action a are kept unchanged during the execution of the action.

The behaviour of an agent $p \in \mathcal{P}$ is defined in terms of the semantics of the actions the agent p participates in. At any instant, an agent $p \in \mathcal{P}$ is involved in the execution of exactly one action in $\mathcal{A}_p$ which is enabled or p is idle (doing nothing) if none of the actions in $\mathcal{A}_p$ is enabled. When the agent is idle its state is kept unchanged. This is formulated in ITL by the formula

$$\varphi_p \;\widehat{=}\; ((\underset{a\in\mathcal{A}_p}{\uplus} \psi_a) \;\vee\; ((\underset{a\in\mathcal{A}_p}{\wedge} \neg enabled_a) \;\wedge\; idle_p))^*,$$

where $\uplus$ denotes the operator *exclusive-or* and

$$idle_p \;\widehat{=}\; empty \;\vee\; (skip \;\wedge\; stable(Var(p))).$$

The initial state of the system is described by the state formula

$$initial \;\widehat{=}\; (\underset{p\in\mathcal{P}}{\wedge} ready_p) \;\wedge\; (\underset{p\in\mathcal{P}'}{\wedge} (y_p = c_p)), \tag{3.5}$$

where $\mathcal{P}'$ is the set of agents $p \in \mathcal{P}$ such that $Init(p) \;\widehat{=}\; y_p := c_p$. Initially all the agents are ready for participating in the execution of actions.

The execution of the system terminates if all agents are ready and no guard is true. This is expressed by the state formula

$$final \;\widehat{=}\; (\underset{p\in\mathcal{P}}{\wedge} ready_p) \;\wedge\; (\underset{a\in\mathcal{A}}{\wedge} \neg(g_a \wedge h_a)).$$

Remember that if two or more actions sharing a common synchronisation agent are enabled at the same time, only one of them is chosen (nondeterministically) for execution. In order to ensure that computations are *fair*, we use the notion of *action fairness* which states that if an action is enabled infinitely often then the action is also executed infinitely often. The action fairness requirement is specified by the following formula, where inf specifies an infinite interval:

$$actionFair \;\widehat{=}\; inf \supset \Box(\underset{a\in\mathcal{A}}{\wedge}((\Box\Diamond(g_a \wedge h_a)) \supset \Diamond\psi_a)).$$

It is also important that computations are *agent fair*. The notion of *agent fairness* requires that no agent may be indefinitely held up while some of the joint actions it participates in are infinitely often enabled. This requirement is formalised as

$$agentFair \;\widehat{=}\; inf \supset \Box(\underset{p\in\mathcal{P}}{\wedge}((\Box\Diamond \underset{a\in\mathcal{A}_p}{\vee} enabled_a) \supset \Diamond \underset{a\in\mathcal{A}_p}{\vee} \psi_a)).$$

The semantics of the concurrent execution of a secure action system SAS is defined by the formula

$$\mathcal{C}[\![SAS]\!] \;\widehat{=}\; \bigcirc(initial \;\wedge\; halt(final) \;\wedge\; (\underset{p\in\mathcal{P}}{\wedge}\varphi_p) \;\wedge\; actionFair \;\wedge\; agentFair).$$

The formula $\mathcal{C}[\![SAS]\!]$ states that the execution of an action system starts with an initialisation step (or skip if no initialisation statements is specified), and repeatedly all the enabled

actions are executed in parallel. However, at any time an agent is participating in the execution of at most one action. The execution terminates if a state is reached where all the agents are ready and all the action guards are false (this is specified as $halt(final)$). If the computation is infinite then both the *action fairness* and the *agent fairness* requirements are enforced.

3.6 A Simple Example

Let us consider a simple example of a control system. The functional requirement of the system is to keep the temperature of a room above 16^oC. The system is composed of a heater, a temperature sensor and a controller. For the sake of simplicity, we assume that when the heater is on, the temperature of the room increases at constant rate 0.50^oC per minute. When the heater is switched off after being on for up to 20 minutes, the temperature decreases at constant rate 0.25^oC per minute. In reality, the variation of the temperature may be much more complicated due to many environmental factors which are out of the scope of this thesis.

We take the design decision that when the temperature of the room reaches a *safety level* of 16.25^oC, the heater is turned on for 20 minutes and then it is turned off. For security reason, only the controller is authorised to turn the heater off.

We consider the following agents: $heater$, $sensor$ and $controller$. The state of the heater (on/off) is modelled with a Boolean variable $heaterIsOn$ which equals $true$ if the heater is on and $false$ if it is off. We assume that initially the heater is off. So the agent $heater$ is specified by

agent $heater:$ **var** $heaterIsOn; heaterIsOn := false.$

The controller observes the temperature sensor and switches the heater on or off accordingly. The agent $sensor$ stores the temperature readings in the variable $temp$. To make our model more realistic, the initial value of $temp$ is unspecified. This means that the model functions correctly regardless the initial temperature of the room. Remember that in SAS, a variable takes an arbitrary initial value (in its range) if it is not explicitly initialised. The definition of agent $sensor$ is

agent $sensor:$ **var** $temp.$

The controller monitors the period in which the heater is on while the temperature is above 16^oC using a variable $time$. Initially $time$ equals 0 since the heater is off. The definition of agent $controller$ is

agent $controller:$ **var** $time; time := 0.$

We also define the following actions which describe the behaviours of the system:

$incrTemp$, $decrTemp$, $tick$, $turnOn$, and $turnOff$.

The actions $incrTemp$ and $decrTemp$ models the dynamics of the temperature within the room. We assume that when the heater is on the temperature of the room increases at constant rate 0.5^oC per minute and, when it is off the temperature decreases with a lower rate of 0.25^oC per minute. So, these actions are specified as follows:

action $incrTemp$ **on** $sensor$ **by** $sensor$ **async** $heater$:

$heaterIsOn \rightarrow temp := temp + 0.5.$

action $decrTemp$ **on** $sensor$ **by** $sensor$ **async** $heater$:

$\neg heaterIsOn \rightarrow temp := temp - 0.25.$

The action $tick$ monitors the time period during which the temperature is above 16^oC and the heater is on. It is executed jointly by the agents $heater$, $sensor$ and $controller$. Its definition is

action $tick$ **on** $controller$ **by** $controller$ **async** $heater, sensor$:

$(temp \geq 16 \wedge heaterIsOn) \rightarrow time := time + 1.$

Note that the time spent in the initialisation phase when the room temperature is below 16^oC and the heater is switched on does not count. To execute this action, the controller (which is the action's target agent) need not synchronise with the other participating agents $heater$ and $sensor$. It reads the current room temperature from the sensor, checks the state of the heater and increments the monitoring time by 1 if the room temperature is above 16^oC and the heater is on. While the action is being executed, the sensor can concurrently sample the room temperature (e.g by executing the action $incrTemp$ described above). Another version of the action in which $sensor$ is a synchronisation agent can be defined by

action $tick1$ **on** $controller$ **by** $controller, sensor$ **async** $heater$:

$(temp \geq 16 \wedge heaterIsOn) \rightarrow temp, time := temp + 0.5, time + 1.$

During the execution of the action $tick1$, the agent $sensor$ cannot be involved in the execution of any other action. That is why the body of $tick1$ must, at the same time, sample

the room temperature as it changes. To limit such a replication, the set of synchronisation agents of the actions can be minimised.

The action $turnOn$ is executed jointly by the agents $heater$, $sensor$ and $controller$. Its effect is to turn the heater on whenever the temperature of the room is less than or equal 16.25^oC and the heater is off. If initially the temperature of the room is less than or equal 16.25^oC then the heater must be turned on, otherwise the heater is kept off until the room temperature reaches the low safety temperature of 16.25^oC. Its specification is

action $turnOn$ **on** $heater$ **by** $controller$ **async** $sensor$:

$(temp \leq 16.25 \wedge \neg heaterIsOn) \rightarrow heaterIsOn := true.$

The action $turnOff$ is executed jointly by the agents $heater$ and $controller$. Its effect is to turn the heater off and reset the monitoring time to 0 whenever it has been on for 20 time units (while the room temperature is above 16^oC). Its specification is

action $turnOff$ **on** $heater$ **by** $controller$:

$(time = 20 \wedge heaterIsOn) \rightarrow heaterIsOn, time := false, 0.$

Note that, e.g. the action $turnOn$ and the action $incrTemp$ can execute in parallel because they do not share a common synchronisation agent. By contrast, the actions $turnoff$ and $tick$ cannot execute in parallel because the two actions are synchronised by the agent $controller$.

The access control predicates in this case are defined as follows, where p_1 and p_2 range over $\{heater, sensor, controller\}$ and a stands for any of the actions specified above. The requirement that the controller is allowed to turn the heater on is specified as

$autho^+(controller, heater, turnOff) = true.$

The agent $sensor$ is allowed to participate in the execution of the action $turnOn$ at any time. However the action can oly take place if the room temperature is less or equal 16.25^oC. It is not necessary to switch the heater on when the room temperature is already above the safety level. So the policy here is

$autho^+(sensor, heater, turnOn) = true$.

Apart from the controller and the sensor, no other agent is allowed to perform an action on the heater, viz for $p \notin \{controller, sensor\}$,

$autho^-(p, heater, turnOn) = true$, and

$autho^-(p, heater, turnOff) = true$.

In a similar vein, the agent $heater$ is allowed to increase or decrease the room temperature, i.e.

$autho^+(heater, sensor, incrTemp) = true$, and

$autho^+(heater, sensor, decrTemp) = true$.

The agent $heater$ and the agent $sensor$ are allowed to perform (or to participate in the execution of) the action $tick$ on the agent $controller$. This is specified by

$autho^+(heater, controller, tick) = true$, and

$autho^+(sensor, controller, tick) = true$.

For all other cases not treated above, $autho^+(p_1, p_2, a)$ and $autho^-(p_1, p_2, a)$ are taken to be equal to $false$.

The access control condition $autho(p_1, p_2, a)$ for an agent p_1 to perform an action a on another agent p_2 is expressed by

$$autho(p_1, p_2, a) = autho^{+}(p_1, p_2, a) \ \wedge \ \neg autho^{-}(p_1, p_2, a).$$

This means that in the event of conflicts between positive authorisations and negative authorisations, the latter take precedence.

3.7 Summary

We have presented the syntax and the semantics of the Interval Temporal Logic (ITL). ITL is the foundation of our uniform formal framework for integrating the functional, the temporal and the security requirements of secure systems. The operators *weak always-followed-by* ($\mapsto$) and *strong always-followed-by* ($\leftrightarrow$) are defined. These operators are used in the next chapter to express security policy rules. Interesting properties of the operators have been established that are useful in reasoning about policy rules.

We have presented SAS, a computational model for describing secure systems. SAS is a conservative extension of Back's action system paradigm to cater for security. It allows us to specify the functionalities and the security policies of systems at the implementation level. Security is enforced by strengthening the guard of actions with an access control condition, derived from the system's security policy. In the next chapter, we focus on the formalisation of security policies, and in a later chapter we show how to develop a SAS that enforces a specific security policy.

Chapter 4

Basic Security Policies

Objectives

- To present the authorisation model.
- To present the delegation mechanism.
- To describe the syntax and the semantics of simple policies.
- To give the algebraic properties of simple policies.
- To address the completeness of simple policies.

4.1 Introduction

In this chapter, we focus on the specification of security policies. A security policy describes rules about who is allowed to perform what action. Our computation model is a secure action system as described in the previous chapter. So a system is viewed as a collection of agents, a collection of actions and a security policy. An Agent can perform an

action on another agent. In this chapter we refer to the latter as an *object* and to the former as a *subject*. A security policy is defined by three Boolean functions $autho^+$, $autho^-$ and $autho$ as presented in the previous chapter. In the sequel, we model these functions as state variables and use them for specifying authorisation policies and delegation policies. A delegation policy specifies the ability of subjects to delegate some of their rights to other subjects to perform actions on their behalf. This is modelled as a special kind of authorisation policies. We assume a finite set $\mathcal{S}$ of subjects, a finite set $\mathcal{O}$ of objects and a finite set $\mathcal{A}$ of actions.

4.2 Authorisation Policy

4.2.1 Authorisations

A positive authorisation variable is a Boolean variable of the form $autho^+(s, o, a)$ used in the specification to *explicitly* grant the subject s the right to perform the action a on the object o. Similarly, a negative authorisation variable is a Boolean variable of the form $autho^-(s, o, a)$ used in the specification to *explicitly* deny the subject s the right to perform the action a on the object o. Hence, a positive authorisation expresses a permission while a negative authorisation expresses a denial. The use of positive and negative authorisation variables increases the expressiveness and the flexibility of the policy specification language.

In fact, different kinds of policy can be expressed using positive and negative authorisations. In a *closed policy*, only positive authorisations can be used in specifications.

Therefore a right is granted if explicitly stated, otherwise it is denied. This kind of policy is also called a *default negative* policy. On the contrary, only negative authorisations can occur in an *open policy*. Thus a right is granted if it is not explicitly denied. This kind of policy is also called a *default positive* policy. The third kind of policy that can be specified is a *Hybrid policy* where positive and negative authorisations can be used in specifications. In this case, conflicts may occur within specifications when a right is explicitly granted and explicitly denied to a subject at the same time. How conflicts are resolved leads to different kinds of hybrid policies. For example, conflicts can be resolved by giving denials precedence over permissions or the other way around.

The authorisation variable $autho(s, o, a)$ is used to resolve conflicts that may involve subject s w.r.t. action a on object o. The final decision whether or not the right to perform action a on object o is granted or denied to subject s is calculated in the variable $autho(s, o, a)$. For example, if denial takes precedence in the event of conflict then the value of $autho(s, o, a)$ is calculated as follows. If $autho^{-}(s, o, a)$ is true then $autho(s, o, a)$ is false otherwise the value of $autho(s, o, a)$ is equal to the value of $autho^{+}(s, o, a)$.

On the other hand, relationships such as membership between subjects and their groups or roles, and ownership between objects and their owners can be modelled by means of state variables. The use of state variables to model these relationships enables the dynamic change of subject memberships and object ownerships. We use the Boolean state variable $in(s_1, s_2)$ to state the membership of the subject s_1 to the group s_2. Similarly, the assignment of a subject s_1 to a role s_2 is modelled by the Boolean state variable

$role(s_1, s_2)$. We denote by the Boolean state variable $owner(o, s)$ the ownership of object o by subject s. New (state) variables can be introduced in a similar manner to model application specific features and used in policies specifications.

4.2.2 Authorisation Rules

An authorisation policy specifies the behaviour of authorisation variables introduced in the previous section. In ITL, such a behaviour can be described at different levels of abstraction. For example the policy

$$\bigwedge_{s\in\mathcal{S}}\bigwedge_{o\in\mathcal{O}}\bigwedge_{a\in\mathcal{A}}\Box(autho^{+}(s,o,a)) \supset len \leq \delta$$

states that a permission cannot last for more than δ time unit(s). This specification does not say anything about when an authorisation is granted or denied, but constrains the duration of any permissions. Similarly, for $s \in \mathcal{S}$, $o \in \mathcal{O}$ and $a \in \mathcal{A}$, the policy

$$\boxdot_{a}((\neg autho^{-}(s,o,a); \bigcirc\Box(autho^{-}(s,o,a)); \bigcirc\neg autho^{-}(s,o,a)) \supset len > \delta + 2)$$

specifies that any denial must be kept stable for at least δ time unit(s).

These examples show how real-time properties about policies can be expressed in our framework. However, in security settings flexibility as much as expressiveness is a desirable attribute for any policy framework. For this reason, policies are commonly expressed in terms of rules which namely state conditions under which authorisations are granted or denied. Moreover, the use of rules makes the specification clearer and easier to understand.

We use the operator *always-followed-by* (denoted by $\mapsto$) defined in Section 3.3 to

express policy rules, and distinguish three kinds of rules: *positive authorisation rules*, *negative authorisation rules* and *conflict resolution rules*.

Positive authorisation rules

A positive authorisation rule has the form

$$\bigcup_{X \in \mathcal{S}'} \bigcup_{Y \in \mathcal{O}'} \bigcup_{Z \in \mathcal{A}'} \{f \mapsto autho^+(X, Y, Z)\}$$

where $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{O}' \subseteq \mathcal{O}$, $\mathcal{A}' \subseteq \mathcal{A}$ are nonempty sets and f is an ITL formula. The variable symbols X, Y, Z may occur in the formula f. This rule says that for each subject $X \in \mathcal{S}'$, each object $Y \in \mathcal{O}'$ and each action $Z \in \mathcal{A}'$, subject X is explicitly allowed to perform action Z on object Y whenever a behaviour satisfying f is observed. The formula f is called the *premise* or the *body* of the rule and the expression $autho^+(X, Y, Z)$ is called the *conclusion* of the rule. Note that negation is not allowed in consequences of rules to avoid contradiction between them.

The formula f can express a property on the history of the execution that must hold before the authorisation is actually granted. For the sake of simplicity, we write

$$f \mapsto autho^+(X, Y, Z)$$

to mean

$$\bigcup_{X \in \mathcal{S}} \bigcup_{Y \in \mathcal{O}} \bigcup_{Z \in \mathcal{A}} \{f \mapsto autho^+(X, Y, Z)\}.$$

If, for example, $\mathcal{S}' = \{x\}$ (singleton set) then we write

$$f \mapsto autho^+(x, Y, Z)$$

to mean

$$\bigcup_{X\in\{x\}}\bigcup_{Y\in\mathcal{O}}\bigcup_{Z\in\mathcal{A}}\{f \mapsto autho^{+}(X,Y,Z)\}.$$

Similar notations are adopted when $\mathcal{O}'$ or $\mathcal{A}'$ is a singleton set. This means that we use lowercase to represent constant symbols and uppercase for variable symbols, by analogy to logic programming languages such as Prolog, Datalog [51] or Tempura [36]. For example, the following rule explicitly grants owners write access:

$$\lceil owner(Y,X)\rceil^{0} \mapsto autho^{+}(X,Y,write).$$

Negative authorisation rules

A negative authorisation rule has the form

$$\bigcup_{X\in\mathcal{S}'}\bigcup_{Y\in\mathcal{O}'}\bigcup_{Z\in\mathcal{A}'}\{f \mapsto autho^{-}(X,Y,Z)\}$$

where $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{O}' \subseteq \mathcal{O}$, $\mathcal{A}' \subseteq \mathcal{A}$ are nonempty sets and f is an ITL formula. The variable symbols X, Y, Z may occur in the formula f. This rule says that for each subject $X \in \mathcal{S}'$, each object $Y \in \mathcal{O}'$ and each action $Z \in \mathcal{A}'$, subject X is explicitly denied the right to perform action Z on object Y whenever a behaviour satisfying f is observed. We adopt the same notations as for the positive authorisation rules. For example, the following rule explicitly denies all but owners write access:

$$\lceil \neg owner(Y,X)\rceil^{0} \mapsto autho^{-}(X,Y,write).$$

Conflict resolution rules

A conflict resolution rule has the form

$$\bigcup_{X\in\mathcal{S}'}\bigcup_{Y\in\mathcal{O}'}\bigcup_{Z\in\mathcal{A}'}\{f \mapsto autho(X,Y,Z)\}$$

where $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{O}' \subseteq \mathcal{O}$, $\mathcal{A}' \subseteq \mathcal{A}$ are nonempty sets and f is an ITL formula. We adopt the same notations as for the positive authorisation rules.

Conflict resolution rules are devised to specify how the access rights are derived from positive authorisation and negative authorisation specifications. Inconsistencies amongst authorisations are resolved using these rules. The conflict resolution rules for closed policies have the form

$$\lceil autho^{+}(X, Y, Z) \rceil^{0} \mapsto autho(X, Y, Z)$$

which means that only privileges explicitly stated are granted. For open policies where only negative authorisations are allowed in the specifications, the conflict resolution rules have the form

$$\lceil \neg autho^{-}(X, Y, Z) \rceil^{0} \mapsto autho(X, Y, Z).$$

Therefore privileges are granted if not explicitly denied. Hybrid policies allow positive and negative authorisations to be specified. Conflicts among authorisations in these policies can be resolved in several ways. For example,

- permissions may have precedence over denials. This is formalised by rules of the form

  $$\lceil autho^{+}(X, Y, Z) \rceil^{0} \mapsto autho(X, Y, Z),$$

  i.e. a privilege is granted if explicitly stated by a positive authorisation.

- denials may have precedence over permissions. This is specified by rules of the form

  $$\lceil autho^{+}(X, Y, Z) \wedge \neg autho^{-}(X, Y, Z) \rceil^{0} \mapsto autho(X, Y, Z),$$

i.e. a privilege is granted if it is explicitly stated by a positive authorisation and there is not negative authorisation that forbids it.

- or no conflicts is allowed at all. In this case a privilege is granted if explicitly stated by a positive authorisation, i.e.

$$\lceil autho^{+}(X, Y, Z) \rceil^{0} \mapsto autho(X, Y, Z),$$

and the policy must be conflict free, i.e.

$$\lceil \neg autho^{-}(X, Y, Z) \rceil^{0} \leftrightarrow autho^{+}(X, Y, Z).$$

(We adopt the same declarative notation for rules using the operator strong always-followed, $\leftrightarrow$.)

More sophisticated enforcement mechanisms such as subgroups overriding (i.e if conflicting authorisations propagate to a member, authorisations given to a subgroups override the authorisations given to a super-group) or path overriding (i.e. if conflicting authorisations propagate to a member because of its membership in two groups g_1 and g_2, authorisations of a subgroup g_1 override the authorisations of a super-group g_2 if there is paths from the member to g_2 that passes from g_1) can be specified. Moreover, note that different conflict resolution rules can be specified within the same system for different triples (s, o, a) of subject s, object o and action a. For example, the following two rules can be part of the same policy:

(i) $\quad autho^{+}(alice, music, play) \mapsto autho(alice, music, play).$

$$(ii) \quad \left(\begin{array}{l} autho^{+}(alice, movie, watch) \wedge \\ \\ \neg autho^{-}(alice, movie, watch) \end{array} \right) \mapsto autho(alice, movie, watch).$$

The first rule (i) allows Alice to play music if she is explicitly authorised to do so (no matter if she is also explicitly denied). The second rule specifies a different mechanism that allows Alice to watch movie if she is explicitly authorised but not explicitly denied the right to do so.

In [63] Samarati et al. suggested to attach more general conditions to authorisation rules in order to specify their validity based on the system state, the state of objects or the history of authorisations. In our model, the premise of a rule can be any ITL formula, and therefore can specify properties about the states of the system, the states of subjects and objects, and the execution history. For example, suppose the boolean state variable $access(s, o, a)$ is $true$ in the states where subject s starts the execution of action a on object o. The policy rule

$$((access(X, Y, Z); len > 0; access(X, Y, Z); len > 0) \wedge len < \delta) \mapsto autho^{-}(X, Y, Z)$$

states that if a subject performs an action twice on the same object in less than δ time unit(s), then the right must be withdrawn from it. The premise of this rule specifies a property about the history of execution.

Another important issue in (discretionary) access control concerns the delegation of access rights which is addressed in the following section.

4.3 Delegation Policy

Delegation is a mechanism which enables a subject to delegate some of its rights to another subject for it to act on its behalf. In discretionary access control delegation of rights is at the discretion of subjects. This means that the initiative to delegate is taken by subjects and not the policy manager. However it should be possible to control delegations through access control policy to ensure security, especially in systems allowing cascaded delegations. A delegation policy specifies the ability of subjects (the grantors) to delegate access rights to other subjects (the grantees) to perform actions on their behalf.

4.3.1 Delegations

Similarly to the authorisation model, we model a delegation by a Boolean state variable $deleg(s_1, s_2, o, a)$ which holds in a state if subject s_1 delegates to s_2 the right to perform action a on object o in that state. The set

$$deleg \mathrel{\widehat{=}} \{deleg(s_1, s_2, o, a) \mid s_1, s_2 \in \mathcal{S}, o \in \mathcal{O}, a \in \mathcal{A}\}$$

of all delegation variables is called a *delegation matrix*. The *delegation list* of an object o is defined by

$$deleg_o \mathrel{\widehat{=}} \{deleg(s, s', o, a) \mid s, s' \in \mathcal{S}, a \in \mathcal{A}\},$$

i.e. the set of delegation variables for delegating the access rights over object o. Additionally, we associate with each object o a *delegation object* D_o that can be accessed only by performing a *delegation action* of the form $delegate(s_1, s_2, o, a)$ or a *revocation action* of the form $revoke(s_1, s_2, o, a),\ s_1, s_2 \in \mathcal{S}, a \in \mathcal{A}$. For each object o, the delegation object

D_o stores the delegation list $deleg_o$ of object o, and is protected by the security policy.

The effect of a delegation action $delegate(s_1, s_2, o, a),\ s_1, s_2 \in \mathcal{S}, o \in \mathcal{O}, a \in \mathcal{A}$ is to set the delegation variable $deleg(s_1, s_2, o, a)$ to true while a revocation action $revoke(s_1, s_2, o, a)$ sets the delegation variable $deleg(s_1, s_2, o, a)$ to false.

Let $\mathcal{O}' \widehat{=} \mathcal{O} \cup \{D_o \mid o \in \mathcal{O}\}$ and

$\mathcal{A}' \widehat{=} \mathcal{A} \cup \{delegate(s_1, s_2, o, a), revoke(s_1, s_2, o, a) \mid s_1, s_2 \in \mathcal{S}, o \in \mathcal{O}, a \in \mathcal{A}\}$.

Note that in a delegation action $delegate(s_1, s_2, o, a)$, the delegated action a cannot be another delegation action nor a revocation action. That is delegation actions of the form $delegate(s_1, s_2, o, delegate(\ldots))$ or of the form $delegate(s_1, s_2, o, revoke(\ldots))$ are not allowed. Similarly, revocation actions of the form $revoke(s_1, s_2, o, revoke(\ldots))$ or of the form $revoke(s_1, s_2, o, delegate(\ldots))$ are forbidden. It follows that the ability to delegate cannot be passed on from one subject to another, but is controlled by an administrative policy which states who is allowed to delegate and who is not. This mechanism gives the security administrator more control over the propagation of rights through cascaded delegations.

4.3.2 Delegation Rules

Delegation rules are expressed as a special kind of authorisation rules. A *positive delegation rule* is a positive authorisation rule of the form

$$\bigcup_{X_1 \in \mathcal{S}'} \bigcup_{X_2 \in \mathcal{S}'} \bigcup_{Y \in \mathcal{O}'} \bigcup_{Z \in \mathcal{A}'} \{f \mapsto autho^+(X_1, D_Y, op(X_1, X_2, Y, Z))\}$$

where $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{O}' \subseteq \mathcal{O}$, $\mathcal{A}' \subseteq \mathcal{A}$ are nonempty sets and $op \in \{delegate, revoke\}$. A *negative delegation rule* is defined in a similar manner as a special kind of negative

authorisation rules. So do *delegation conflict resolution rules*.

Following are some samples of rules related to health care services, where the state variable $Time$ stands for a global time, and an object is a medical record.

- Bob can delegate to Alice the right to read Bob's medical record, viz

$$\lceil owner(X, bob)\rceil^0 \mapsto autho^+(bob, D_X, delegate(bob, alice, X, read)).$$

  However, Alice cannot delegate this right on her turn unless she is explicitly allowed to do so.

- Alice cannot delegate to Bob the right to read her medical record between times t_1 and t_2:

$$\left(\begin{array}{c} t_1 \leq Time \leq t_2 \\ \wedge \\ \lceil owner(X, alice)\rceil^0 \end{array}\right) \mapsto autho^-(alice, D_X, delegate(alice, bob, X, read)).$$

- At any time Bob is allowed to revoke from Alice read access to Bob's medical record:

$$\lceil owner(X, bob)\rceil^0 \mapsto autho^+(bob, D_X, revoke(bob, alice, X, read)).$$

4.3.3 Delegation Mechanism

The effect of delegating a right is to pass the right on to the grantee. However, this is done only when the grantor has the right and is allowed to delegate it. The grantor delegates a right by performing a delegation action on a delegation object. A delegated right is

activated by a conflict resolution rule of the form

$$\left(\bigvee_{X' \in \mathcal{S}} \left(\begin{array}{c} autho(X', Y, Z) \wedge deleg(X', X, Y, Z) \wedge \\ autho(X', D_Y, delegate(X', X, Y, Z)) \end{array} \right) \right) \mapsto autho(X, Y, Z). \tag{4.1}$$

This rule says that the right of performing action Z on object Y is granted to a grantee X if some subject X' that has the right to perform action Z on object Y and the right to delegate that right to X, delegates the right to X. This means that if a grantor X' (i) revokes the right from X or (ii) loses the right to delegate it to X or (iii) loses the right itself then automatically the right is withdrawn from the grantee X. However, if the grantee X is delegated the same right by several grantors then the right is withdrawn only if at least one of the conditions (i), (ii) and (iii) above holds for each of the grantors. Any policy allowing delegation must contain the rule (4.1).

4.4 Simple Policy

4.4.1 Syntax

We define a *simple policy* to be a collection of policy rules. The following syntax is used to describe a simple policy, where p and q range over simple policies and r stands for a policy rule.

$$p ::= \emptyset \mid r \mid p \cup q \mid p \cap q \mid p \setminus q \mid -p \mid \pi^{+}(p) \mid \pi^{-}(p)$$

The empty policy $\emptyset$ contains no rules. The set operators "$\cup$", "$\cap$", and "$\setminus$" have their usual meaning. In particular, the operator "$\cup$" can be used to add rules to a policy while

the operator "$\setminus$" can be used to remove rules from a policy. The unary operator "$-$" changes positive authorisation rules into negative ones and negative authorisation rules into positive ones and leaves conflict resolution rules unchanged if there is any in the policy argument. The operators π^+ and π^- return respectively the set of positive authorisation rules and the set of negative authorisation rules contained in the policy argument. Note that the set of the conflict resolution rules of a policy p is determined by the expression $p \setminus (\pi^+(p) \cup \pi^-(p))$.

Let us illustrate the meaning of the unary operators by an example. The policy p_0 defined by

$$p_0 = \bigcup \left\{ \begin{array}{l} \lceil owner(Y, X) \rceil^0 \mapsto autho^+(X, Y, read), \\ (\lceil patient(X', X) \wedge owner(Y, X') \rceil^0) \mapsto autho^+(X, Y, read), \\ \lceil role(X, nurse) \rceil^0 \mapsto autho^-(X, Y, read), \\ \lceil autho^+(X, Y, Z) \rceil^0 \mapsto autho(X, Y, Z) \end{array} \right\}$$

is a security policy for a health care service, where Y stands for a medical record and the predicate $patient(X', X)$ means that the patient X' is being treated by the physician X. The first rule says that patients are allowed read access to their medical records. The second rule states that physicians are allowed to read their patients' medical records. On the contrary the third rule stipulates that nurses are forbidden read access to patient records. Note that conflicting authorisations can be derived, for instance when a nurse is at the same time a patient in the clinic. In this case the first rule states that (s)he is allowed to read his/her record while the third rule forbids him to do so. The conflict resolution mechanism is specified by the fourth rule that gives precedence to permission in the event of such conflict. So, the policy p_0 does allow nurses read access to their own personal

health information.

Following are samples of policy expressions:

$$-p_0 \quad = \quad \bigcup \left\{ \begin{array}{l} \lceil owner(Y,X) \rceil^0 \mapsto autho^-(X,Y,read), \\ (\lceil patient(X',X) \wedge owner(Y,X') \rceil^0) \mapsto autho^-(X,Y,read), \\ \lceil role(X,nurse) \rceil^0 \mapsto autho^+(X,Y,read), \\ \lceil autho^+(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z) \end{array} \right\}$$

$$\pi^+(p_0) \quad = \quad \bigcup \left\{ \begin{array}{l} \lceil owner(Y,X) \rceil^0 \mapsto autho^+(X,Y,read), \\ (\lceil patient(X',X) \wedge owner(Y,X') \rceil^0) \mapsto autho^+(X,Y,read) \end{array} \right\}$$

$$\pi^-(p_0) \quad = \quad \lceil role(X,nurse) \rceil^0 \mapsto autho^-(X,Y,read)$$

$$p_0 \setminus (\pi^+(p_0) \cup \pi^-(p_0)) \quad = \quad \lceil autho^+(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z)$$

$$p_0 \cap (-p_0) \quad = \quad \lceil autho^+(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z)$$

We believe that this set of operators is minimal and rich enough to specify different ways of merging policies. For example, if p and q are two policies then the policy

$$\pi^+(p) \cup \pi^-(q) \cup (\lceil autho^+(X,Y,Z) \wedge \neg autho^-(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z))$$

grants all rights explicitly granted in the policy p and not explicitly denied in the policy q. Similarly, the policy

$$\pi^-(p) \cup \pi^-(q) \cup (\lceil \neg autho^-(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z))$$

grants all rights not explicitly denied in the policy p or in the policy q, while the policy

$$\pi^+(p) \cup \pi^+(q) \cup (\lceil autho^+(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z))$$

grants all rights explicitly granted in the policy p or in the policy q. Yet another example is the policy

$$\pi^+(p \cup q) \cup \pi^-(p \cup q) \cup (\lceil autho^+(X,Y,Z) \wedge \neg autho^-(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z))$$

that grants all the rights explicitly stated in the policy p or in the policy q but not explicitly denied in any of the two policies. This means that if a subject is explicitly granted a right in the policy p and is explicitly denied the right in the policy q then the subject is denied the right, and vice-versa.

4.4.2 Algebraic Properties

In addition to the well-known algebraic properties of the set operators "$\cup$", "$\cap$" and "$\setminus$" such as associativity and monotonicity, we define in this section the properties of the policy operators "$-$", "π^+" and "π^-". The operators "$-$", "π^+" and "π^-" are defined as follows, where p is a policy.

$$\begin{aligned} -p &= \{f \mapsto autho^-(t_1,t_2,t_3) \mid f \mapsto autho^+(t_1,t_2,t_3) \in p\} \cup \\ &\quad \{f \mapsto autho^+(t_1,t_2,t_3) \mid f \mapsto autho^-(t_1,t_2,t_3) \in p\} \cup \\ &\quad \{f \mapsto autho(t_1,t_2,t_3) \in p\} \\ \pi^+(p) &= \{f \mapsto autho^+(t_1,t_2,t_3) \in p\} \\ \pi^-(p) &= \{f \mapsto autho^-(t_1,t_2,t_3) \in p\} \end{aligned}$$

These operators satisfy the following properties, where $op \in \{+,-\}$.

Law 1 *(*Distributivity*)*

- $\pi^{op}(p \cup q) = \pi^{op}(p) \cup \pi^{op}(q)$
- $\pi^{op}(p \cap q) = \pi^{op}(p) \cap \pi^{op}(q)$
- $\pi^{op}(p \setminus q) = \pi^{op}(p) \setminus \pi^{op}(q)$
- $-(p \cup q) = (-p) \cup (-q)$
- $-(p \cap q) = (-p) \cap (-q)$
- $-(p \setminus q) = (-p) \setminus (-q)$

Law 2 *(*Monotonicity*)*

if $p \subseteq q$ *then*

- $\pi^{op}(p) \subseteq \pi^{op}(q)$
- $-p \subseteq -q$

Law 3 *(*Duality*)*

- $\pi^{+}(-p) = -\pi^{-}(p)$
- $\pi^{-}(-p) = -\pi^{+}(p)$

Law 4 *(Miscellaneous)*

- $\pi^{op}(\pi^{op}(p)) = \pi^{op}(p)$
- $-(-p) = p$

- $\pi^+(\pi^-(p)) = \emptyset$
- $\pi^-(\pi^+(p)) = \emptyset$
- $\pi^+(p) \cap \pi^-(p) = \emptyset$
- $\pi^{op}(\emptyset) = \emptyset$
- $-p = p \Leftrightarrow (\pi^+(p) = -\pi^-(p))$

The proofs of these properties are straight forward from the definitions of the operators. The formal semantics of simple policies is given in the following section.

4.4.3 Semantics

We say that a policy rule $f \mapsto op(s, o, a)$ is closed if s, o and a are ground (i.e. constant) terms, where $op \in \{autho, autho^+, autho^-\}$. Let $\mathcal{R}(\mathcal{S}, \mathcal{O}, \mathcal{A})$ stand for the set of all closed rules over a set $\mathcal{S}$ of subjects, a set $\mathcal{O}$ of objects and a set $\mathcal{A}$ of actions. Syntactically a policy over a set $\mathcal{S}$ of subjects, a set $\mathcal{O}$ of objects and a set $\mathcal{A}$ of actions is a subset of $\mathcal{R}(\mathcal{S}, \mathcal{O}, \mathcal{A})$. In this section we give the formal semantics of policies. For any policy $p \subseteq \mathcal{R}(\mathcal{S}, \mathcal{O}, \mathcal{A})$, we denote by $\mathcal{M}(p)$ the formal semantics of the policy p which is defined by:

$$\mathcal{M}(p) \widehat{=} \begin{cases} true & \text{if } p = \emptyset \\ \bigwedge\limits_{r \in p} r & \text{otherwise} \end{cases}$$

This semantics is compositional in the sense that, e.g. the semantics of the union of two policies can be defined in terms of the semantics of the two policies without additional

information about their internal structures:

$$\mathcal{M}(p_1 \cup p_2) = \mathcal{M}(p_1) \wedge \mathcal{M}(p_2).$$

This is due to the properties of the operator $\mapsto$, especially the fact that the conjunction of rules is always satisfiable. In fact two rules cannot contradict each other because negation is not allowed in the consequences of rules. Although this approach provides high flexibility and expressiveness power for writing specifications that are contradiction-free, it is not suitable for reasoning about liveness properties of policies. For example, we cannot prove that a policy satisfies a property of the form $\Diamond\neg autho(s, o, a)$, i.e. eventually a subject s is denied the right to perform action a on object o.

Moreover, a policy may not be complete in the sense that the access rights of a subject w.r.t. some object might not be specified. Therefore an authorisation request whether the subject is allowed or not to access the object has no answer. It is the case, e.g. for the policy $\emptyset$ or a policy without conflict resolution rule. Recall that, in our model the authorisation decision for a subject s to perform an action a on an object o is determined by the value of the state variable $autho(s, o, a)$. The policy $\emptyset$ and any policy without conflict resolution rules leave those state variables unspecified; therefore they can take any boolean value.

To solve the problem while preserving the flexibility and the expressiveness power of the policy language, we develop a rewriting technique to transform a simple policy as described above into a complete policy that grants *exactly* the rights specified in the input policy and denies everything else. For example the output corresponding to the policy $\emptyset$ denies every rights to every subjects in the system. The algorithm is presented in the

following section.

4.4.4 Complete Specification

A security policy must determine at any time the access rights of each subject with respect to any object and any action. Writing a complete specification to state this can be very complex and cumbersome. Default rules have been used in [76] as a way to provide complete specification. The drawback of this approach is that default rules might not be conclusive. As a consequence the model can lead to a situation in which an authorisation request has no answer. Most of the logic-based approaches [40, 27, 42, 48] restrict the policy language to a variant of Datalog [33] to overcome this problem. The inference mechanism in Datalog makes the *closed world assumption*: if a positive literal cannot be proved to be true then it is considered to be false.

We say that a policy specification is *complete* if it determines at any time the access right of each subjects with respect to any object and any action. This ensures that any authorisation request is deterministic. Our policy language allows us to state policy rules in a declarative manner. This makes it easy to write policy specifications. Moreover, policy specifications are contradiction-free. By contradictory specification we mean a specification that is equivalent to false, thus not satisfiable. For example, the two rules $true \mapsto autho^+(john, x, read)$ and $true \mapsto \neg autho^+(john, x, read)$ contradict one another. The first rule set the variable $autho^+(john, x, read)$ to true everywhere (see Theorem 3.1) while the second set the same variable to false everywhere. So a policy containing the two rules is unsatisfiable. For this reason, we do not allow negation in the

consequences of the rules i.e. rules of the form, e.g. $f \mapsto \neg autho^+(s, o, a)$ are forbidden.

One of the advantages of our approach is that it provides high flexibility in writing policy specifications: rules can be added to and removed from a policy; conflicts among positive and negative authorisations are resolved using conflict resolution rules; policies can be merged in several manners to form new ones. However, policy specifications might still not be complete.

In this section, we present an algorithm $Comp(p)$ to construct a complete policy from a simple policy p. The policy $Comp(p)$ grants exactly the rights specified in the policy p and denies everything else. Recall that any rule in the input policy p has the form $f \mapsto op(s, o, a)$ where $op \in \{autho, autho^+, autho^-\}$ and s, o, a are constants. The algorithm is outlined in page 86.

The step 1 in the algorithm constructs a policy p_1 which contains all rules in p plus rules of the form $false \mapsto op(s, o, a)$, for each subject $s \in \mathcal{S}$, each object $o \in \mathcal{O}$ and each action $a \in \mathcal{A}$. Note that from Theorem 3.1, a rule of the form $false \mapsto op(s, o, a)$ is semantically equivalent to $true$. So the policy p_1 is equivalent to the policy p, i.e.

$$\mathcal{M}(p_1) \equiv \mathcal{M}(p).$$

In step 2, for each subject $s \in \mathcal{S}$, each object $o \in \mathcal{O}$, each action $a \in \mathcal{A}$ and each $op \in \{autho, autho^+, autho^-\}$, all the rules of the form $f_i \mapsto op(s, o, a)$ in p_1, $0 \leq i \leq k_{op}$, having the same consequence $op(s, o, a)$ are grouped into a single rule of the form $f_1 \vee f_2 \vee \ldots \vee f_{k_{op}} \mapsto op(s, o, a)$ in the policy p_2. From Theorem 3.5 it follows that the policy p_1 and the policy p_2 so constructed are equivalent, i.e.

$$\mathcal{M}(p_2) \equiv \mathcal{M}(p_1).$$

The step 3 constructs the output policy p' whose rules are the rules in p_2 with the operator weak always-followed-by ($\mapsto$) replaced by the operator strong always-followed-by ($\leftrightarrow$). Since the operator strong always-followed-by is a refinement of the operator weak always-followed-by as established by Theorem 3.6, it follows that the policy p' is a refinement of the policy p_2, i.e.

$$\mathcal{M}(p') \supset \mathcal{M}(p_2).$$

Note that the policy p' contains *exactly* one rule of the form $(f_1 \vee f_2 \vee \ldots \vee f_{k_{op}}) \leftrightarrow op(s,o,a)$, for each subject $s \in \mathcal{S}$, each object $o \in \mathcal{O}$, each action $a \in \mathcal{A}$ and each $op \in \{autho, autho^+, autho^-\}$. This means that, from the definition of the operator "$\leftrightarrow$", the value of the state variable $op(s,o,a)$ is equal to true in a state if the formula $f_1 \vee f_2 \vee \ldots \vee f_{k_{op}}$ holds for a left neighbourhood of the state and is equal to false otherwise, for each subject $s \in \mathcal{S}$, each object $o \in \mathcal{O}$, each action $a \in \mathcal{A}$ and each $op \in \{autho, autho^+, autho^-\}$. This give the proof of Theorem 4.1.

---

algorithm Comp

input: a simple policy p.

output: a complete policy p' that refines p.

begin

1. Construct the set

$$p_1 = p \cup \bigcup_{\substack{s\in\mathcal{S}\\ o\in\mathcal{O}\\ a\in\mathcal{A}}} \left\{ \begin{array}{l} false \mapsto autho^+(s,o,a), \\ false \mapsto autho^-(s,o,a), \\ false \mapsto autho(s,o,a) \end{array} \right\}.$$

2. Apply Theorem 3.5 as many times as necessary to group rules in p_1 into an equivalent policy p_2:

$$p_2 = \bigcup_{\substack{s\in\mathcal{S}\\ o\in\mathcal{O}\\ a\in\mathcal{A}}} \left\{ \begin{array}{l} f_{s,o,a} \mapsto autho^+(s,o,a), \\ g_{s,o,a} \mapsto autho^-(s,o,a), \\ h_{s,o,a} \mapsto autho(s,o,a) \end{array} \right\},$$

where

- $f_{s,o,a} = f_1 \vee \ldots \vee f_n$ is the disjunction of all the f_i such that

$$f_i \mapsto autho^+(s,o,a) \in p_1,\ i = 1,\ldots,n;$$

- $g_{s,o,a} = g_1 \vee \ldots \vee g_m$ is the disjunction of all the g_i such that

$$g_i \mapsto autho^-(s,o,a) \in p_1,\ i = 1,\ldots,m;$$

- $h_{s,o,a} = h_1 \vee \ldots \vee h_k$ is the disjunction of all the h_i such that

$$h_i \mapsto autho(s,o,a) \in p_1,\ i = 1,\ldots,k.$$

3. Refine p_2 into p' using Theorem 3.6 by simply replacing in p_2 the operator weak always-followed-by $\mapsto$ with the operator strong always-followed-by $\leftrightarrow$:

$$p' = \bigcup_{\substack{s\in\mathcal{S}\\ o\in\mathcal{O}\\ a\in\mathcal{A}}} \left\{ \begin{array}{l} f_{s,o,a} \leftrightarrow autho^+(s,o,a), \\ g_{s,o,a} \leftrightarrow autho^-(s,o,a), \\ h_{s,o,a} \leftrightarrow autho(s,o,a) \end{array} \right\}.$$

4. Return p'.

end

Theorem 4.1 *For any policy $p \in \mathcal{R}(\mathcal{S}, \mathcal{O}, \mathcal{A})$, the policy $Comp(p)$ grants exactly the rights specified in the policy p and denies everything else.*

Example 4.1 *In this example we compute the complete policy associated to the policy $\emptyset$.*

step 1:

$$p_1 = \bigcup_{\substack{s\in\mathcal{S}\\ o\in\mathcal{O}\\ a\in\mathcal{A}}} \left\{ \begin{array}{l} false \mapsto autho^{+}(s,o,a), \\ false \mapsto autho^{-}(s,o,a), \\ false \mapsto autho(s,o,a) \end{array} \right\}.$$

step 2:

$$p_2 = \bigcup_{\substack{s\in\mathcal{S}\\ o\in\mathcal{O}\\ a\in\mathcal{A}}} \left\{ \begin{array}{l} false \mapsto autho^{+}(s,o,a), \\ false \mapsto autho^{-}(s,o,a), \\ false \mapsto autho(s,o,a) \end{array} \right\}.$$

step 3:

$$p' = \bigcup_{\substack{s\in\mathcal{S}\\ o\in\mathcal{O}\\ a\in\mathcal{A}}} \left\{ \begin{array}{l} false \leftrightarrow autho^{+}(s,o,a), \\ false \leftrightarrow autho^{-}(s,o,a), \\ false \leftrightarrow autho(s,o,a) \end{array} \right\}.$$

So $Comp(\emptyset) = p'$. Note that the policy $Comp(\emptyset)$ forbids every right to every subject in the system.

Example 4.2 *Let $\mathcal{S} = \{john, paul\}$, $\mathcal{O} = \{doc\}$ and $\mathcal{A} = \{read, write\}$. In this*

example we compute the complete policy associated to the policy

$$p = \bigcup \left\{ \begin{array}{l} true \mapsto autho^{+}(X, doc, read), \\ true \mapsto autho^{-}(paul, doc, read), \\ (autho^{+}(X, Y, Z) \wedge \neg autho^{-}(X, Y, Z)) \mapsto autho(X, Y, Z) \end{array} \right\}.$$

The input policy to the algorithm contains only ground terms, no variables. So we rewrite the input policy p in the form:

$$p = \left\{ \begin{array}{ll} 1 & true \mapsto autho^{+}(john, doc, read), \\ 2 & true \mapsto autho^{+}(paul, doc, read), \\ 3 & true \mapsto autho^{-}(paul, doc, read), \\ 4 & \left[\begin{array}{l} autho^{+}(john, doc, read) \wedge \\ \neg autho^{-}(john, doc, read) \end{array} \right]^{0} \mapsto autho(john, doc, read), \\ 5 & \left[\begin{array}{l} autho^{+}(john, doc, write) \wedge \\ \neg autho^{-}(john, doc, write) \end{array} \right]^{0} \mapsto autho(john, doc, write), \\ 6 & \left[\begin{array}{l} autho^{+}(paul, doc, read) \wedge \\ \neg autho^{-}(paul, doc, read) \end{array} \right]^{0} \mapsto autho(paul, doc, read), \\ 7 & \left[\begin{array}{l} autho^{+}(paul, doc, write) \wedge \\ \neg autho^{-}(paul, doc, write) \end{array} \right]^{0} \mapsto autho(paul, doc, write) \end{array} \right\}.$$

step 1: In addition to the rules in p, the policy p_1 contains the rules 8 to 19.

$$p_1 = p \bigcup \left\{ \begin{array}{rl} 8 & false \mapsto autho^{+}(paul, doc, read), \\ 9 & false \mapsto autho^{+}(john, doc, read), \\ 10 & false \mapsto autho^{+}(paul, doc, write), \\ 11 & false \mapsto autho^{+}(john, doc, write), \\ 12 & false \mapsto autho^{-}(paul, doc, read), \\ 13 & false \mapsto autho^{-}(john, doc, read), \\ 14 & false \mapsto autho^{-}(john, doc, write), \\ 15 & false \mapsto autho^{-}(paul, doc, write), \\ 16 & false \mapsto autho(john, doc, read), \\ 17 & false \mapsto autho(john, doc, write), \\ 18 & false \mapsto autho(paul, doc, read), \\ 19 & false \mapsto autho(paul, doc, write) \end{array} \right\}.$$

step 2: In this step rules of the same consequences are grouped into one as follows:

$$p_2 = \left\{ \begin{array}{rl}
1, 9 & true \mapsto autho^+(john, doc, read), \\
2, 8 & true \mapsto autho^+(paul, doc, read), \\
3, 12 & true \mapsto autho^-(paul, doc, read), \\
10 & false \mapsto autho^+(john, doc, write), \\
11 & false \mapsto autho^+(paul, doc, write), \\
13 & false \mapsto autho^-(john, doc, read), \\
14 & false \mapsto autho^-(john, doc, write), \\
15 & false \mapsto autho^-(paul, doc, write), \\
4, 16 & \left[\begin{array}{l} autho^+(john, doc, read) \wedge \\ \neg autho^-(john, doc, read) \end{array} \right]^0 \mapsto autho(john, doc, read), \\
5, 17 & \left[\begin{array}{l} autho^+(john, doc, write) \wedge \\ \neg autho^-(john, doc, write) \end{array} \right]^0 \mapsto autho(john, doc, write), \\
6, 18 & \left[\begin{array}{l} autho^+(paul, doc, read) \wedge \\ \neg autho^-(paul, doc, read) \end{array} \right]^0 \mapsto autho(paul, doc, read), \\
7, 19 & \left[\begin{array}{l} autho^+(paul, doc, write) \wedge \\ \neg autho^-(paul, doc, write) \end{array} \right]^0 \mapsto autho(paul, doc, write)
\end{array} \right\}.$$

step 3: The operator "$\mapsto$" in p_2 is replaced by the operator "$\leftrightarrow$" in p':

$$p' = \left\{ \begin{array}{rl}
1,9 & true \leftrightarrow autho^{+}(john, doc, read), \\
2,8 & true \leftrightarrow autho^{+}(paul, doc, read), \\
3,12 & true \leftrightarrow autho^{-}(paul, doc, read), \\
10 & false \leftrightarrow autho^{+}(john, doc, write), \\
11 & false \leftrightarrow autho^{+}(paul, doc, write), \\
13 & false \leftrightarrow autho^{-}(john, doc, read), \\
14 & false \leftrightarrow autho^{-}(john, doc, write), \\
15 & false \leftrightarrow autho^{-}(paul, doc, write), \\
4,16 & \left[\begin{array}{l} autho^{+}(john, doc, read) \wedge \\ \neg autho^{-}(john, doc, read) \end{array} \right]^{0} \leftrightarrow autho(john, doc, read), \\
5,17 & \left[\begin{array}{l} autho^{+}(john, doc, write) \wedge \\ \neg autho^{-}(john, doc, write) \end{array} \right]^{0} \leftrightarrow autho(john, doc, write), \\
6,18 & \left[\begin{array}{l} autho^{+}(paul, doc, read) \wedge \\ \neg autho^{-}(paul, doc, read) \end{array} \right]^{0} \leftrightarrow autho(paul, doc, read), \\
7,19 & \left[\begin{array}{l} autho^{+}(paul, doc, write) \wedge \\ \neg autho^{-}(paul, doc, write) \end{array} \right]^{0} \leftrightarrow autho(paul, doc, write)
\end{array} \right\}.$$

The complete policy associated with the policy p above is then $Comp(p) = p'$.

4.5 Summary

We have presented a formal model of authorisation and delegation policies. Boolean state variables have been used to model authorisations. Delegation policies are specified as spe-

cial cases of authorisation policies that state who is allowed to delegate rights. Similarly to the Ponder policy framework [24], a delegation object is used for the enforcement of delegation policies. Policy rules are formulated as safety properties upon those variables, stating constraints on their behaviours. The model supports the specification of positive and negative authorisations/delegations, and therefore can specify different kind of policies such as closed, open, and hybrid policies. Conflicts among positive and negative authorisations are resolved using rules.

We have also addressed the completeness of policies. A policy is complete if it determines the access rights of each subject w.r.t. each object. An algorithm is proposed that transforms an incomplete policy into a complete one which grants exactly the rights specified in the input policy and denies everything else. The correctness of the algorithm is proved. In Datalog like policy languages, completeness is achieved by a mechanism known as the closed world assumption. Therefore if it cannot be inferred that someone has a given right then it is assumed that it does not have the right.

The advantage of using a temporal logic for the specification of security policies is that the same formalism can be used to specify the functional and temporal requirements. This provides an effective way of integrating the functional requirements and the security requirements and to reason about them uniformly throughout the design process. More importantly, Interval Temporal Logic offers more flexibility for policy composition as we show in the next chapter.

Chapter 5

Compound Security Policies

Objectives

- To describe the syntax of compound policies.
- To give a formal semantics of compound policies.
- To define an algebra for policy composition.
- To prove the soundness of algebraic laws.
- To give examples of derivation in the algebra.

5.1 Introduction

The algebra of simple policies defined in Chapter 4 provides operators for rules management and selection. However it cannot model dynamic changes in policies such as switching from one policy to another at some point in time. This chapter introduces additional operators for policy composition to cater for the dynamics in policies. For example,

a policy may have a duration (i.e. how long the policy can be enforced for) or can be a sequence of policies which apply one after the others. Another interesting example is a policy that is intended to change if some guard is triggered by the environment or a time-out has elapsed. The formal semantics of the operators is given. Their properties are established in terms of sound algebraic laws which define equational and refinement relationships among policies. We give some examples of derivations to illustrate the use of the algebra.

5.2 Syntax and Semantics

The syntax of policy is given by the the following BNFs where P, Q and S range over policies, p denotes a simple policy, t stands for a duration, C an ITL formula, and w for a condition (or predicate).

$$\begin{aligned} P ::= \;& p \mid NULL \mid CHAOS \mid P \sqcap Q \mid w?P : Q \mid P^\frown Q \mid P^\oplus \\ & \mid P; Q \mid P^+ \mid t : P \mid [w]P \mid \langle w \rangle P \mid P \rhd_w^t Q : S \mid C :: P \end{aligned}$$

An example of compound policy is

$$(\delta : p)^\frown (p \cup r)$$

where p is a simple policy and r a policy rule. This policy behaves like p for a duration of δ time units, then a new rule r is added and enforced together with p. This simple example shows yet how changes in policies can be modelled and enforced.

$NULL$ and $CHAOS$ are singular policies. $NULL$ behaves like the empty policy $\emptyset$ but terminates immediately. So it enforces no policy rules and takes no time. $CHAOS$

denotes an infeasible policy, i.e. a policy that cannot be implemented in our model.

Two policies P and Q can be composed in sequence to form a policy $P;Q$ (read P chop Q) that behaves like P for some time, then behaves like Q. However, the final state of the subinterval that satisfies P is the initial state of the subinterval satisfying Q. Thus this operator is used if P and Q can agree on that state. Otherwise, the *weak chop* operator can be used, and $P^\frown Q$ denotes a policy that behaves like P for some time, then behaves like Q afterwards starting from the state next to the final state of the subinterval that satisfies P. In this case P and Q share no state. An informal illustration of the operators *chop* and *weak chop* is depicted in Figure 5.1. In both cases, the sequential composition can be used to specify policies that are intended to evolve over time by changing policy rules. In general, organisations apply different policies for specific periods of time. Universities distinguish between term time and vacations. Banks render restricted services in week-ends and holidays.

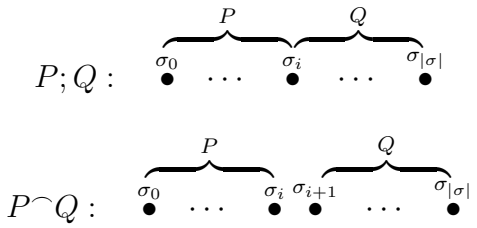


Figure 5.1: The operators Chop and Weak Chop

The conditional $w?P : Q$ specifies a policy that behaves like P when the condition w holds and behaves like Q otherwise. For example, in an organisation P may apply for staff and Q for non-staff. Note that P and/or Q might also be conditionals, refining staff and non-staff further into different subcategories.

On the contrary, the non-deterministic choice $P \sqcap Q$ specifies a policy which behaves either like P or like Q. This is a more abstract specification as the choice between the policies P and Q cannot be determined beforehand. However, the conditional for example is a refinement of the non-deterministic choice, that is a way of implementing such a specification at a lower level of abstraction.

The policy P^+ specifies a policy that behaves like a finite non-empty sequence of P using the chop operator. Similarly, the policy $P^{\oplus}$ denotes a finite non-empty sequence of P in the sense of the operator weak-chop. Both operators are used to express iteration of a policy.

The policy expression $t : P$ limits the scope of the policy P to a period of duration t. For example the policy

$$\delta : (true \mapsto autho^+(bob, movie, watch))$$

grants bob, for a specific amount of time δ, the right to watch movies.

The operator *as long as* defines a policy $[w]P$ (read: *P as long as* w) that enforces P as long as the condition w holds. The condition w can state a critical situation that requires a specific policy P to be enforced to ensure security throughout the critical period. For example, at the outbreak of epidemic diseases specific security policies are enforced to protect the population during the crisis.

On the other hand, the operator *unless* specifies a policy $\langle w \rangle P$ (read: P *unless* w) that enforces the policy P until the guard w is triggered. That is, the policy P is enforced as long as the guard is false. Therefore the policy $\langle w \rangle P$ behaves just like $[\neg w]P$. Suppose, in the military a policy P applies during peacetime and the policy Q applies during war

crisis. Then the policy of the whole system can be specified as

$$(peace?([peace]P) : (\langle peace\rangle Q))^{+}.$$

The *triangle* operator is devised to express policy of the form $P \rhd^{t}_{w} Q : S$ that behaves like the policy P until the condition w is $true$ or a time-out t time unit(s) has elapsed. If the condition w becomes true then the policy behaves like the policy Q, otherwise it behaves like the police S in the event of time-out.

The *context* operator "::" allows to specify constraints on the environment in which a policy is deployed. So, an expression of the form $C :: P$ means that the environment in which the policy P is enforced must fulfil the constraint C. Constraints are used to specify application-dependent conditions such as group or role memberships. In RBAC [67, 8] for example, constraints are used to specify control principles such as *least privilege* (i.e. users are assigned only privileges necessary for the accomplishment of their tasks) and *separation of duties* (i.e. no user should be given enough privileges to misuse the system on their own).

The formal semantics of these policies is given in Definition 5.1.

Definition 5.1 *The formal semantics of a policy P is an ITL formula $\mathcal{M}_c(P)$ defined inductively on the structure of policies by:*

1. $\mathcal{M}_c(NULL) \mathrel{\widehat{=}} empty$

2. $\mathcal{M}_c(CHAOS) \mathrel{\widehat{=}} false$

3. $\mathcal{M}_c(p) \mathrel{\widehat{=}} \mathcal{M}(p)$

4. $\mathcal{M}_c(P \sqcap Q) \mathrel{\widehat{=}} \mathcal{M}_c(P) \vee \mathcal{M}_c(Q)$

5. $\mathcal{M}_c(w?P : Q) \mathrel{\widehat{=}} (w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge \mathcal{M}_c(Q))$

6. $\mathcal{M}_c(P^\frown Q) \mathrel{\widehat{=}} \mathcal{M}_c(P); skip; \mathcal{M}_c(Q)$

7. $\mathcal{M}_c(P^\oplus) \mathrel{\widehat{=}} \mathcal{M}_c(P); (skip; \mathcal{M}_c(P))^*$

8. $\mathcal{M}_c(P; Q) \mathrel{\widehat{=}} \mathcal{M}_c(P); \mathcal{M}_c(Q)$

9. $\mathcal{M}_c(P^+) \mathrel{\widehat{=}} \mathcal{M}_c(P); (\mathcal{M}_c(P))^*$

10. $\mathcal{M}_c(t : P) \mathrel{\widehat{=}} \mathcal{M}_c(P) \wedge len = t$

11. $\mathcal{M}_c(C :: P) \mathrel{\widehat{=}} C \wedge \mathcal{M}_c(P)$

12. $\mathcal{M}_c([w]P) \mathrel{\widehat{=}} (\mathcal{M}_c(P) \wedge \Box w) \vee (((\mathcal{M}_c(P) \wedge \Box w); skip) \wedge fin\ \neg w) \vee (empty \wedge \neg w)$

13. $\mathcal{M}_c(\langle w \rangle P) \mathrel{\widehat{=}} \mathcal{M}_c([\neg w]P)$

14. $\mathcal{M}_c(P \rhd_w^0 Q : S) \mathrel{\widehat{=}} w?\mathcal{M}_c(Q) : \mathcal{M}_c(S)$

15. $\mathcal{M}_c(P \rhd_w^\infty Q : S) \mathrel{\widehat{=}} \mathcal{M}_c((\langle w \rangle P); (w?Q : NULL))$

16. $\mathcal{M}_c(P \rhd_w^t Q : S) \mathrel{\widehat{=}} (w \wedge \mathcal{M}_c(Q)) \ \vee \ (\mathcal{M}_c(P) \wedge (\Box \neg w) \wedge len < t) \vee$

    $((\mathcal{M}_c(P) \wedge (\Box \neg w) \wedge len < t); skip; (\mathcal{M}_c(Q) \wedge w)) \vee$

    $((\mathcal{M}_c(P) \wedge (\Box \neg w) \wedge len = t); skip; \mathcal{M}_c(S))$, *provided* $0 < t < \infty$.

5.3 Algebra

Based on the formal semantics of policies given above, we define the following relationships among policies. Definition 5.3 establishes the equality of two policies while Definition 5.2 states the refinement relationship between them.

Definition 5.2 (Refinement)

A policy P is refined by a policy Q, denoted by $P \sqsubseteq Q$, if the semantics of Q implies the semantics of P, viz

$$P \sqsubseteq Q \text{ iff } \mathcal{M}_c(Q) \supset \mathcal{M}_c(P).$$

Definition 5.3 (Equation)

A policy P is equal to a policy Q, denoted by $P = Q$, if $P \sqsubseteq Q$ and $Q \sqsubseteq P$.

Definition 5.4 (Complete policy) *The complete policy associated with a policy P is denoted by $Comp_c(P)$ and is defined inductively on the structure of policies as follows, where $Comp$ stands for the algorithm presented in page 86:*

1. $Comp_c(NULL) \mathrel{\widehat{=}} Comp_c(0 :: \emptyset)$
2. $Comp_c(CHAOS) \mathrel{\widehat{=}} CHAOS$
3. $Comp_c(p) \mathrel{\widehat{=}} Comp(p)$
4. $Comp_c(P \sqcap Q) \mathrel{\widehat{=}} Comp_c(P) \sqcap Comp_c(Q)$
5. $Comp_c(w?P : Q) \mathrel{\widehat{=}} w?Comp_c(P) : Comp_c(Q)$
6. $Comp_c(P^\frown Q) \mathrel{\widehat{=}} Comp_c(P)^\frown Comp_c(Q)$

7. $Comp_c(P^{\oplus}) \mathrel{\widehat{=}} (Comp_c(P))^{\oplus}$

8. $Comp_c(P;Q) \mathrel{\widehat{=}} Comp_c(P); Comp_c(Q)$

9. $Comp_c(P^{+}) \mathrel{\widehat{=}} (Comp_c(P))^{+}$

10. $Comp_c(t : P) \mathrel{\widehat{=}} t : Comp_c(P)$

11. $Comp_c(C :: P) \mathrel{\widehat{=}} C :: Comp_c(P)$

12. $Comp_c([w]P) \mathrel{\widehat{=}} [w]Comp_c(P)$

13. $Comp_c(\langle w\rangle P) \mathrel{\widehat{=}} \langle w\rangle Comp_c(P)$

14. $Comp_c(P \rhd^t_w Q : S) \mathrel{\widehat{=}} Comp_c(P) \rhd^t_w Comp_c(Q) : Comp_c(S)$

Corollary 5.1 *If P is a policy in the language described above then*

$$P \sqsubseteq Comp_c(P).$$

This corollary of Theorem 4.1 says that the function $Comp_c$ defined in Definition 5.4 is a refinement. Its proof is straightforward from Theorem 4.1 and is constructed inductively on the structure of policies.

The following algebraic laws express properties of the operators over policies in term of equations and refinement relationships. Therefore policies can be compared and refined into simpler ones, easier to analyse and to implement. In the sequel P, Q, S and T range over policies while p and q stand for simple policies.

5.3.1 Nondeterministic Choice

The nondeterministic choice is idempotent, i.e.

Nondet– 1 $P \sqcap P = P$.

The nondeterministic choice is commutative and associative, i.e.

Nondet– 2 $P \sqcap Q = Q \sqcap P$.

Nondet– 3 $P \sqcap (Q \sqcap S) = (P \sqcap Q) \sqcap S$.

$CHAOS$ and the empty policy $\emptyset$ are respectively unit and zero element of the nondeterministic choice, i.e.

Nondet– 4 $P \sqcap CHAOS = P$.

Nondet– 5 $P \sqcap \emptyset = \emptyset$.

Theorem 5.1 *The laws Nondet-1, Nondet-2, Nondet-3, Nondet-4 and Nondet-5 are sound.*

Proof. The proofs are immediate from the semantics and are omitted here.

Furthermore the nondeterministic choice is distributive w.r.t. conditional i.e.

Nondet– 6 $P \sqcap (w?Q : S) = w?(P \sqcap Q) : (P \sqcap S)$.

Each (immediate) component of a nondeterministic choice is its possible implementation, i.e.

Nondet– 7 $P \sqcap Q \sqsubseteq P$.

The nondeterministic choice is monotonic w.r.t. refinement, i.e.

Nondet– 8 $$\frac{P \sqsubseteq P' \qquad Q \sqsubseteq Q'}{P \sqcap Q \sqsubseteq P' \sqcap Q'}$$

Theorem 5.2 *Laws Nondet-6, Nondet-7 and Nondet-8 are sound.*

Proof.

- Law Nondet-6

  $\mathcal{M}_c(P \sqcap (w?Q : S))$

  $\equiv$ {by definition of $\sqcap$}

  $\mathcal{M}_c(P) \vee \mathcal{M}_c(w?Q : S)$

  $\equiv$ {by definition of conditional}

  $\mathcal{M}_c(P) \vee ((w \wedge \mathcal{M}_c(Q)) \vee (\neg w \wedge \mathcal{M}_c(S)))$

  $\equiv$ $\{(w \vee \neg w) \equiv true\}$

  $((w \vee \neg w) \wedge \mathcal{M}_c(P)) \vee ((w \wedge \mathcal{M}_c(Q)) \vee (\neg w \wedge \mathcal{M}_c(S)))$

  $\equiv$ {distributivity of $\wedge$ over $\vee$}

  $(w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge \mathcal{M}_c(P)) \vee (w \wedge \mathcal{M}_c(Q)) \vee (\neg w \wedge \mathcal{M}_c(S))$

  $\equiv$ {distributivity of $\wedge$ over $\vee$}

  $(w \wedge (\mathcal{M}_c(P) \vee \mathcal{M}_c(Q))) \vee (\neg w \wedge (\mathcal{M}_c(P) \vee \mathcal{M}_c(S)))$

  $\equiv$ {by definition of $\sqcap$ twice}

  $(w \wedge \mathcal{M}_c(P \sqcap Q)) \vee (\neg w \wedge \mathcal{M}_c(P \sqcap S))$

  $\equiv$ {by definition of conditional}

  $\mathcal{M}_c(w?(P \sqcap Q) : (P \sqcap S))$

- Law Nondet-7

$$\mathcal{M}_c(P \sqcap Q)$$

$\equiv$ {by definition of $\sqcap$}

$$\mathcal{M}_c(P) \vee \mathcal{M}_c(Q)$$

$\subset$ {ITL}

$$\mathcal{M}_c(P)$$

- Law Nondet-8

$$\mathcal{M}_c(P' \sqcap Q')$$

$\equiv$ {by definition of $\sqcap$}

$$\mathcal{M}_c(P') \vee \mathcal{M}_c(Q')$$

$\supset$ $\{P \sqsubseteq P'\}$

$$\mathcal{M}_c(P) \vee \mathcal{M}_c(Q')$$

$\supset$ $\{Q \sqsubseteq Q'\}$

$$\mathcal{M}_c(P) \vee \mathcal{M}_c(Q)$$

$\equiv$ {by definition of $\sqcap$}

$$\mathcal{M}_c(P \sqcap Q)$$

5.3.2 Conditional

The conditional is idempotent, symmetric and associative, i.e.

Cond– 1 $w?P : P = P$.

Cond– 2 $w?P : Q = (\neg w)?Q : P$.

Cond– 3 $w_1?(w_2?P : Q) : S = (w_1 \wedge w_2)?P : (w_1?Q : S)$.

Cond– 4 $w_1?P : (w_2?P : Q) = (w_1 \vee w_2)?P : Q$.

Unaccessible components can be discarded as stipulated in the following laws.

Cond– 5 $true?P : Q = P$.

Cond– 6 $w?P : (w?Q : S) = w?P : S$.

Cond– 7 $w?(w?P : Q) : S = w?P : S$.

Theorem 5.3 *Laws Cond-1, Cond-2, Cond-3, Cond-4, Cond-5, Cond-6 and Cond-7 are sound.*

Proof. The proofs of laws Cond-1, Cond-2 and Cond-5 are straightforward from the definition. We detail here the proof of laws Cond-3 and Cond-6. The proofs of the law Cond-4 and the law Cond-7 are similar to the proofs of the law Cond-3 and the law Cond-6 respectively.

- Law Cond-3

  $\mathcal{M}_c(w_1?(w_2?P : Q) : S)$

  $\equiv$ {by definition of conditional}

  $(w_1 \wedge \mathcal{M}_c(w_2?P : Q)) \vee (\neg w_1 \wedge \mathcal{M}_c(S))$

  $\equiv$ {by definition of conditional}

  $(w_1 \wedge ((w_2 \wedge \mathcal{M}_c(P)) \vee (\neg w_2 \wedge \mathcal{M}_c(Q))) \vee (\neg w_1 \wedge \mathcal{M}_c(S))$

$\equiv$ {distributivity of $\wedge$ over $\vee$}

$((w_1 \wedge w_2) \wedge \mathcal{M}_c(P)) \vee ((w_1 \wedge \neg w_2) \wedge \mathcal{M}_c(Q)) \vee (\neg w_1 \wedge \mathcal{M}_c(S))$

$\equiv$ {ITL}

$((w_1 \wedge w_2) \wedge \mathcal{M}_c(P)) \vee (\neg(w_1 \wedge w_2) \wedge ((w_1 \wedge \mathcal{M}_c(Q)) \vee (\neg w_1 \wedge \mathcal{M}_c(S))))$

$\equiv$ {by definition of conditional}

$((w_1 \wedge w_2) \wedge \mathcal{M}_c(P)) \vee (\neg(w_1 \wedge \neg w_2) \wedge \mathcal{M}_c(w_1?Q : S))$

$\equiv$ {by definition of conditional}

$\mathcal{M}_c((w_1 \wedge w_2)?P : (w_1?Q : S))$

- Law Cond-6

  $\mathcal{M}_c(w?(w?P : Q) : S)$

  $\equiv$ {by definition of conditional}

  $(w \wedge \mathcal{M}_c(w?P : Q)) \vee (\neg w \wedge \mathcal{M}_c(S))$

  $\equiv$ {by definition of conditional}

  $(w \wedge ((w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge \mathcal{M}_c(Q)))) \vee (\neg w \wedge \mathcal{M}_c(S))$

  $\equiv$ {distributivity of $\wedge$ over $\vee$}

  $(w \wedge \mathcal{M}_c(P)) \vee (w \wedge \neg w \wedge \mathcal{M}_c(Q)) \vee (\neg w \wedge \mathcal{M}_c(S))$

  $\equiv$ {ITL}

  $(w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge \mathcal{M}_c(S))$

  $\equiv$ {by definition of conditional}

  $\mathcal{M}_c(w?P : S)$

The conditional is distributive w.r.t. nondeterministic choice.

Cond– 8 $w?P : (Q \sqcap S) = (w?P : Q) \sqcap (w?P : S)$.

Cond– 9 $w?(P \sqcap Q) : S = (w?P : S) \sqcap (w?Q : S)$.

Theorem 5.4 *Law Cond-8 and Cond-9 are sound.*

Proof. The proof of the law Cond-9 is similar to the proof of the law Cond-8.

- Law Cond-8

  $\mathcal{M}_c(w?P : (Q \sqcap S))$

  $\equiv$ {by definition of conditional}

  $(w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge \mathcal{M}_c(Q \sqcap S))$

  $\equiv$ {by definition of $\sqcap$}

  $(w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge (\mathcal{M}_c(Q) \vee \mathcal{M}_c(S)))$

  $\equiv$ {distributivity of $\wedge$ w.r.t. $\vee$}

  $(w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge \mathcal{M}_c(Q)) \vee (\neg w \wedge \mathcal{M}_c(S))$

  $\equiv$ {idempotency of $\vee$}

  $((w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge \mathcal{M}_c(Q))) \vee ((w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge \mathcal{M}_c(S)))$

  $\equiv$ {by definition of conditional twice}

  $\mathcal{M}_c(w?P : Q) \vee \mathcal{M}_c(w?P : S)$

  $\equiv$ {by definition of $\sqcap$}

  $\mathcal{M}_c((w?P : Q) \sqcap (w?P : S))$

The following refinement laws hold. The conditional is monotonic, i.e.

Cond– 10
$$\frac{P \sqsubseteq P' \qquad Q \sqsubseteq Q'}{w?P : Q \sqsubseteq w?P' : Q'}$$

The conditional is a refinement of nondeterministic choice, i.e.

Cond– 11 $P \sqcap Q \sqsubseteq w?P : Q$.

Theorem 5.5 *Laws Cond-10 and Cond-11 are sound.*

Proof. The proof of the law Cond-10 is similar to the proof of the law Nondet-8. On the other hand the proof of the law Cond-11 is straightforward from the semantics of the conditional which stipulates that $w?P : Q$ behaves like P if w holds in the initial state or behaves like Q otherwise.

5.3.3 Weak Chop

$CHAOS$ is the left zero element of the weak chop.

Wchop– 1 $CHAOS^\frown P = CHAOS$.

The weak chop operator is associative, i.e.

Wchop– 2 $P^\frown(Q^\frown S) = (P^\frown Q)^\frown S$

The weak chop operator is monotonic w.r.t. to refinement, i.e.

Wchop– 3
$$\frac{P \sqsubseteq P' \qquad Q \sqsubseteq Q'}{P^\frown Q \sqsubseteq P'^\frown Q'}$$

Theorem 5.6 *Laws Wchop-1, Wchop-2 and Wchop-3 are sound.*

Proof. The proof of this theorem is straightforward from the definition; so it is omitted here.

On the other hand, the operator weak chop is distributive w.r.t. the nondeterministic choice.

Wchop– 4 $P^\frown(Q \sqcap S) = (P^\frown Q) \sqcap (P^\frown S)$.

Wchop– 5 $(P \sqcap Q)^\frown S = (P^\frown S) \sqcap (Q^\frown S)$.

The operator weak chop is right distributive w.r.t. the conditional.

Wchop– 6 $(w?P : Q)^\frown S = w?(P^\frown S) : (Q^\frown S)$.

Theorem 5.7 *Laws Wchop-4, Wchop-5 and Wchop-6 are sound.*

Proof. The proof of laws Wchop-4 and Wchop-5 are similar. Here are the proofs of laws Wchop-5 and Wchop-6.

- Law Wchop-5

  $\mathcal{M}_c((P \sqcap Q)^\frown S)$

  $\equiv$ {by definition of weak chop}

  $\mathcal{M}_c(P \sqcap Q); skip; \mathcal{M}_c(S)$

  $\equiv$ {by definition of scope}

  $\mathcal{M}_c(P \sqcap Q); skip; \mathcal{M}_c(S)$

  $\equiv$ {by definition of $\sqcap$}

  $(\mathcal{M}_c(P) \vee \mathcal{M}_c(Q)); skip; \mathcal{M}_c(S)$

  $\equiv$ {distributivity of chop w.r.t. $\vee$}

  $(\mathcal{M}_c(P); skip; \mathcal{M}_c(S)) \vee (\mathcal{M}_c(Q); skip; \mathcal{M}_c(S))$

  $\equiv$ {by definition of weak chop twice}

  $\mathcal{M}_c(P^\frown S) \vee \mathcal{M}_c(Q^\frown S)$

  $\equiv$ {by definition of $\sqcap$}

  $\mathcal{M}_c((P^\frown S) \sqcap (Q^\frown S))$

- Law Wchop-6

$$\begin{aligned}
& \mathcal{M}_c((w?P:Q)^\frown S) \\
\equiv\ & \{\text{by definition of weak chop}\} \\
& \mathcal{M}_c(w?P:Q); skip; \mathcal{M}_c(S) \\
\equiv\ & \{\text{by definition of conditional}\} \\
& ((w \wedge \mathcal{M}_c(P)) \vee (\neg w \wedge \mathcal{M}_c(Q))); skip; \mathcal{M}_c(S) \\
\equiv\ & \{\text{distributivity of chop w.r.t. } \vee\} \\
& ((w \wedge \mathcal{M}_c(P)); skip; \mathcal{M}_c(S)) \ \vee\ ((\neg w \wedge \mathcal{M}_c(Q)); skip; \mathcal{M}_c(S)) \\
\equiv\ & \{w \text{ is a state formula}\} \\
& (w \wedge (\mathcal{M}_c(P); skip; \mathcal{M}_c(S))) \vee (\neg w \wedge (\mathcal{M}_c(Q); skip; \mathcal{M}_c(S))) \\
\equiv\ & \{\text{by definition of weak chop twice}\} \\
& (w \wedge \mathcal{M}_c(P^\frown S)) \vee (\neg w \wedge \mathcal{M}_c(Q^\frown S)) \\
\equiv\ & \{\text{by definition of conditional}\} \\
& \mathcal{M}_c(w?(P^\frown S) : (Q^\frown S))
\end{aligned}$$

5.3.4 Weak Chopstar

$CHAOS$ is a fix point of weak chopstar.

Wchopstar– 1 $CHAOS^{\oplus} = CHAOS$.

For any policy P, the policy $P^{\oplus}$ is a fix point of weak chopstar.

Wchopstar– 2 $P^{\oplus\oplus} = P^{\oplus}$.

The operator weak chopstar is monotonic, i.e.

Wchopstar– 3 $\dfrac{P \sqsubseteq Q}{P^{\oplus} \sqsubseteq Q^{\oplus}}$

Weak chopstar and chopstar distribute mutually.

Wchopstar– 4 $P^{+\oplus} = P^{\oplus+}$.

Theorem 5.8 *Laws Wchopstar-1, Wchopstar-2, Wchopstar-3 and Wchopstar-4 are sound.*

Proof. The proof of the laws Wchopstar-1, Wchopstar-2 and Wchopstar-3 are straightforward from the definition and therefore are omitted here. Following is the proof of Law Wchopstar-4.

- Law Wchopstar-4

  $\mathcal{M}_c(P^{\oplus+})$

  $\equiv$ {by definition of chopstar}

  $\mathcal{M}_c(P^{\oplus}); \mathcal{M}_c(P^{\oplus})^*$

  $\equiv$ {by definition of weak chopstar}

  $\mathcal{M}_c(P); (skip; \mathcal{M}_c(P))^*; (\mathcal{M}_c(P); (skip; \mathcal{M}_c(P))^*)^*$

  $\equiv$ {regular expression}

  $\mathcal{M}_c(P); (\mathcal{M}_c(P)^*; skip; \mathcal{M}_c(P))^*$

  $\equiv$ {regular expression}

  $\mathcal{M}_c(P); \mathcal{M}_c(P)^*; (skip; \mathcal{M}_c(P); \mathcal{M}_c(P)^*)^*$

  $\equiv$ {by definition of chopstar twice}

  $\mathcal{M}_c(P^+); (skip; \mathcal{M}_c(P^+))^*$

$\equiv$ {by definition of weak chopstar}

$\mathcal{M}_c(P^{+\oplus})$

5.3.5 Chop

$CHAOS$ is the left zero element of the Chop.

Chop– 1 $CHAOS; P = CHAOS.$

$NULL$ is the unit element of the Chop.

Chop– 2 $NULL; P = P; NULL = P.$

The Chop operator is associative, i.e.

Chop– 3 $P; (Q; S) = (P; Q); S$

The Chop operator is monotonic w.r.t. to refinement, i.e.

Chop– 4 $$\frac{P \sqsubseteq P' \qquad Q \sqsubseteq Q'}{(P; Q) \sqsubseteq (P'; Q')}$$

Theorem 5.9 *Laws Chop-2, Chop-1, Chop-3 and Chop-4 are sound.*

Proof. The proof of this theorem is straightforward from the definition; so it is omitted here.

On the other hand, the operator Chop is distributive w.r.t. the nondeterministic choice.

Chop– 5 $P; (Q \sqcap S) = (P; Q) \sqcap (P; S).$

Chop– 6 $(P \sqcap Q); S = (P; S) \sqcap (Q; S).$

The operator Chop is right distributive w.r.t. the conditional.

Chop– 7 $(w?P : Q); S = w?(P; S) : (Q; S)$.

Theorem 5.10 *Laws Chop-5, Chop-6 and Chop-7 are sound.*

Proof. The proof of this theorem is similar to the proof of Theorem 5.7.

5.3.6 Chopstar

$CHAOS$ and $NULL$ are fix points of Chopstar.

Chopstar– 1 $CHAOS^+ = CHAOS$.

Chopstar– 2 $NULL^+ = NULL$.

For any policy P, the policy P^+ is a fix point of Chopstar.

Chopstar– 3 $P^{++} = P^+$.

The operator Chopstar is monotonic, i.e.

Chopstar– 4 $$\frac{P \sqsubseteq Q}{P^+ \sqsubseteq Q^+}$$

Theorem 5.11 *Laws Chopstar-1, Chopstar-3 and Chopstar-4 are sound.*

Proof. The proof of laws Chopstar-1, Chopstar-3 and Chopstar-4 are straightforward from the definition, and therefore is omitted here.

5.3.7 Scope

$CHAOS$ is the zero element of the operator scope.

Scope– 1 $t : CHAOS = CHAOS.$

The policy $NULL$ enforces no rules and takes no time.

Scope– 2 $0 : \emptyset = NULL.$

$NULL$ cannot last inconsistently.

Scope– 3 $$\frac{t > 0}{t : NULL = CHAOS}$$

The duration of scope cannot change.

Scope– 4 $t : (t : P) = t : P.$

Scope– 5 $$\frac{t_1 \neq t_2}{t_1 : (t_2 : P) = CHAOS}$$

The operator scope is a refinement.

Scope– 6 $P \sqsubseteq t : P.$

Theorem 5.12 *Laws Scope-1, Scope-2, Scope-3, Scope-4, Scope-5 and Scope-6 are sound.*

Proof. The proof of this theorem is straightforward from the definition; so it is omitted here.

The operator scope is distributive w.r.t. the conditional and the nondeterministic choice.

Scope– 7 $t : (P \sqcap Q) = (t : P) \sqcap (t : Q)$.

Scope– 8 $t : (w?P : Q) = w?(t : P) : (t : Q)$.

The operator scope is monotonic.

Scope– 9 $$\frac{P \sqsubseteq Q}{t : P \sqsubseteq t : Q}$$

Theorem 5.13 *Laws Scope-7, Scope-8 and Scope-9 are sound.*

Proof. The proof of this theorem is straightforward from the definition of the operators involved. For example, here is the proof of the law Scope-7.

$\mathcal{M}_c(t : (P \sqcap Q))$

$\equiv$ {by definition of scope}

$len = t \wedge \mathcal{M}_c(P \sqcap Q)$

$\equiv$ {by definition of $\sqcap$}

$len = t \wedge (\mathcal{M}_c(P) \vee \mathcal{M}_c(Q))$

$\equiv$ {distributivity of $\wedge$ over $\vee$}

$(len = t \wedge \mathcal{M}_c(P)) \vee (len = t \wedge \mathcal{M}_c(Q))$

$\equiv$ {by definition of scope twice}

$\mathcal{M}_c(t : P) \vee \mathcal{M}_c(t : Q)$

$\equiv$ {by definition of $\sqcap$}

$\mathcal{M}_c((t : P) \sqcap (t : Q))$

5.3.8 As long as

The operators as long as and unless are symmetric each other.

As_long_as– 1 $\langle w\rangle P = [\neg w]P$.

If the guard cannot be triggered then as long as terminates immediately.

As_long_as– 2 $[false]P = NULL$.

On the contrary, if the guard is always true then it can be discarded.

As_long_as– 3 $[true]P = P$.

The operator as long as cannot enforce inconsistent policy.

As_long_as– 4 $[w]CHAOS = w?CHAOS : NULL$.

Theorem 5.14 *Laws As_long_as-1, As_long_as-2, As_long_as-3 and As_long_as-4 are sound.*

Proof. The proof of laws As_long_as-1, As_long_as-2, As_long_as-3 and As_long_as-4 are straightforward from the definition.

The operator as long as is distributive w.r.t. the nondeterministic choice and the conditional.

As_long_as– 5 $[w](P \sqcap Q) = ([w]P) \sqcap ([w]Q)$.

As_long_as– 6 $[w_1](w_2?P : Q) = w_2?([w_1]P) : ([w_1]Q)$.

The operator as long as is monotonic.

As_long_as– 7
$$\frac{P \sqsubseteq Q}{[w]P \sqsubseteq [w]Q}$$

Theorem 5.15 *Laws As_long_as-5, As_long_as-6 and As_long_as-7 are sound.*

Proof. The proof of the law As_long_as-7 is straightforward from the definition. The proof of others are similar. Here is the proof of the law As_long_as-5.

$\mathcal{M}_c([w](P \sqcap Q))$

$\equiv$ {by definition of as long as}

$(\mathcal{M}_c(P \sqcap Q) \wedge \Box w) \vee (((\mathcal{M}_c(P \sqcap Q) \wedge \Box w); skip) \wedge fin\neg w) \vee (empty \wedge \neg w)$

$\equiv$ {by definition of $\sqcap$ twice}

$((\mathcal{M}_c(P) \vee \mathcal{M}_c(Q)) \wedge \Box w) \ \vee \ (empty \wedge \neg w) \vee$

$((((\mathcal{M}_c(P) \vee \mathcal{M}_c(Q)) \wedge \Box w); skip) \wedge fin\neg w)$

$\equiv$ {distributivity of $\wedge$ over $\vee$}

$((\mathcal{M}_c(P) \wedge \Box w) \vee (\mathcal{M}_c(Q) \wedge \Box w)) \vee$

$((((\mathcal{M}_c(P)) \wedge \Box w) \vee (\mathcal{M}_c(Q) \wedge \Box w)); skip) \wedge fin\neg w) \vee (empty \wedge \neg w)$

$\equiv$ {distributivity of chop over to $\vee$}

$((\mathcal{M}_c(P) \wedge \Box w) \vee (\mathcal{M}_c(Q) \wedge \Box w)) \vee$

$((((\mathcal{M}_c(P) \wedge \Box w); skip) \vee ((\mathcal{M}_c(Q) \wedge \Box w); skip)) \wedge fin\neg w) \vee (empty \wedge \neg w)$

$\equiv$ {distributivity of $\wedge$ over $\vee$}

$(((\mathcal{M}_c(P) \wedge \Box w) \vee (((\mathcal{M}_c(P) \wedge \Box w); skip) \wedge fin\neg w) \vee (empty \wedge \neg w))) \wedge$

$(((\mathcal{M}_c(Q) \wedge \Box w) \vee (((\mathcal{M}_c(Q) \wedge \Box w); skip) \wedge fin\neg w) \vee (empty \wedge \neg w)))$

$\equiv$ {by definition of as long as twice}

$\mathcal{M}_c([w]P) \vee \mathcal{M}_c([w]Q)$

$\equiv$ {by definition of $\sqcap$}

$\mathcal{M}_c(([w]P) \sqcap ([w]Q))$

5.3.9 Unless

The properties of the operator unless can be deduced by symmetry from those of the operator as long as. Therefore the following laws are sound.

Unless– 1 $\langle false \rangle P = P.$

Unless– 2 $\langle true \rangle P = NULL.$

Unless– 3 $\langle w \rangle CHAOS = w?NULL : CHAOS.$

Unless– 4 $\langle w \rangle (P \sqcap Q) = (\langle w \rangle P) \sqcap (\langle w \rangle Q).$

Unless– 5 $\langle w_1 \rangle (w_2?P : Q) = w_2?(\langle w_1 \rangle P) : (\langle w_1 \rangle Q).$

Unless– 6
$$\frac{P \sqsubseteq Q}{\langle w \rangle P \sqsubseteq \langle w \rangle Q}$$

5.3.10 Triangle

Specific forms of the triangle operator can be expressed using the operators described above.

Triangle– 1 $P \rhd_w^0 Q : S = w?Q : S.$

Triangle– 2 $P \rhd_{true}^t Q : S = Q$

Triangle– 3 $P \rhd_w^\infty Q : S = \langle w \rangle P; (w?Q : NULL).$

Theorem 5.16 *Laws Triangle-1, Triangle-2 and Triangle-3 are sound.*

Proof. The proof of this theorem is straightforward from the definition.

The operator triangle is distributive w.r.t. the nondeterministic choice.

Triangle– 4 $P \rhd_w^t (Q \sqcap S) : T = (P \rhd_w^t Q : T) \sqcap (P \rhd_w^t S : T)$.

Triangle– 5 $P \rhd_w^t Q : (S \sqcap T) = (P \rhd_w^t Q : S) \sqcap (P \rhd_w^t Q : T)$.

Triangle– 6 $(P \sqcap Q) \rhd_w^t S : T = (P \rhd_w^t S : T) \sqcap (Q \rhd_w^t S : T)$.

The operator triangle is monotonic.

Triangle– 7 $$\frac{P \sqsubseteq P' \qquad Q \sqsubseteq Q' \qquad S \sqsubseteq S'}{P \rhd_w^t Q : S \sqsubseteq P' \rhd_w^t Q' : S'}$$

Theorem 5.17 *Laws Triangle-4, Triangle-5, Triangle-6 and Triangle-7 are sound.*

Proof. The proof of three first laws are similar. So we give here the proof of the law Triangle-4.

Case 1: $0 < t < \infty$

$$\begin{aligned}
& \mathcal{M}_c(P \rhd_w^t (Q \sqcap S) : T) \\
\equiv \quad & \{\text{by definition of triangle}\} \\
& (w \wedge \mathcal{M}_c(Q \sqcap S)) \vee (\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len < t) \vee \\
& ((\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len < t); skip; (w \wedge \mathcal{M}_c(Q \sqcap S))) \vee \\
& ((\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len = t); skip; \mathcal{M}_c(T))
\end{aligned}$$

$\equiv$ {by definition of $\sqcap$ twice}

$(w \wedge (\mathcal{M}_c(Q) \vee \mathcal{M}_c(S))) \vee (\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len < t) \vee$

$((\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len < t); skip; (w \wedge (\mathcal{M}_c(Q) \vee \mathcal{M}_c(S)))) \vee$

$((\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len = t); skip; \mathcal{M}_c(T))$

$\equiv$ {distributivity of $\wedge$ w.r.t. $\vee$ twice}

$(w \wedge \mathcal{M}_c(Q)) \vee (w \wedge \mathcal{M}_c(S)) \vee (\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len < t) \vee$

$((\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len < t); skip; ((w \wedge \mathcal{M}_c(Q)) \vee (w \wedge \mathcal{M}_c(S)))) \vee$

$((\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len = t); skip; \mathcal{M}_c(T))$

$\equiv$ {distributivity of chop w.r.t. $\vee$}

$(w \wedge \mathcal{M}_c(Q)) \vee (w \wedge \mathcal{M}_c(S)) \vee (\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len < t) \vee$

$((\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len < t); skip; (w \wedge \mathcal{M}_c(Q))) \vee$

$((\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len < t); skip; (w \wedge \mathcal{M}_c(S))) \vee$

$((\mathcal{M}_c(P) \wedge (\Box\neg w) \wedge len = t); skip; \mathcal{M}_c(T))$

$\equiv$ {by definition of triangle twice}

$\mathcal{M}_c(P \rhd_w^t Q : T) \vee \mathcal{M}_c(P \rhd_w^t S : T)$

$\equiv$ {by definition of $\sqcap$}

$\mathcal{M}_c((P \rhd_w^t Q : T) \sqcap (P \rhd_w^t S : T))$

Case 2: $t = \infty$

$P \rhd_w^\infty (Q \sqcap S) : T$

$\equiv$ {Triangle-3}

$\langle w \rangle P; (w?(Q \sqcap S) : NULL)$

$\equiv$ {Cond-9}

$\langle w \rangle P; ((w?Q : NULL) \sqcap ((w?S : NULL))$

$\equiv$ {Chop-5}

$(\langle w \rangle P; (w?Q : NULL)) \sqcap (\langle w \rangle P; (w?S : NULL))$

$\equiv$ {Triangle-3}

$(P \rhd_w^\infty Q : T) \sqcap (P \rhd_w^\infty S : T)$

5.3.11 Context

In this section we study the properties of the context operator. Following are basic properties of the operator.

Cont– 1 $C_1 :: (C_2 :: P) = (C_1 \wedge C_2) :: P$.

Cont– 2 $C :: CHAOS = CHAOS$.

Cont– 3 $false :: P = CHAOS$.

Cont– 4 $C :: \emptyset = C$.

Cont– 5 $true :: P = P$.

Theorem 5.18 *The laws Cont-1, Cont-2, Cont-3, Cont-4 and Cont-5 are sound.*

Proof. The proofs are immediate from the semantics and are omitted here.

Furthermore the context operator is distributive w.r.t. nondetreministic choice, conditional and scope, i.e.

Cont– 6 $C :: (Q \sqcap S) = (C :: Q) \sqcap (C :: S)$.

Cont– 7 $C :: (w?Q : S) = w?(C :: Q) : (C :: S)$.

Cont– 8 $C :: (t : Q) = t : (C :: Q)$.

Theorem 5.19 *Laws Cont-6, Cont-7 and Cont-8 are sound.*

The proof of this theorem is stated as follows.

Proof.

- Law Cont-6

  $\mathcal{M}_c(C :: (Q \sqcap S))$

  $\equiv$ {by definition of ::}

  $C \wedge \mathcal{M}_c(Q \sqcap S)$

  $\equiv$ {by definition of $\sqcap$}

  $C \wedge (\mathcal{M}_c(Q) \vee \mathcal{M}_c(S))$

  $\equiv$ {distribution of $\wedge$ over $\vee$}

  $(C \wedge \mathcal{M}_c(Q)) \vee (C \wedge \mathcal{M}_c(S))$

  $\equiv$ {by definition of :: twice}

  $\mathcal{M}_c(C :: Q) \vee \mathcal{M}_c(C :: S)$

  $\equiv$ {by definition of $\sqcap$}

  $\mathcal{M}_c((C :: Q) \sqcap (C :: S))$

- Law Cont-7

  $\mathcal{M}_c(C :: (w?Q : S))$

  $\equiv$ {by definition of ::}

  $C \wedge \mathcal{M}_c(w?Q : S)$

$\equiv$ {by definition of conditional}

$C \wedge ((w \wedge \mathcal{M}_c(Q)) \vee (\neg w \wedge \mathcal{M}_c(S)))$

$\equiv$ {distribution of $\wedge$ over $\vee$}

$(w \wedge (C \wedge \mathcal{M}_c(Q))) \vee (\neg w \wedge (C \wedge \mathcal{M}_c(S)))$

$\equiv$ {by definition of :: twice}

$(w \wedge \mathcal{M}_c(C :: Q)) \vee (\neg w \wedge \mathcal{M}_c(C :: S))$

$\equiv$ {by definition of conditional}

$\mathcal{M}_c(w?(C :: Q) : (C :: S))$

- Law Cont-8

$\mathcal{M}_c(C :: (t : Q))$

$\equiv$ {by definition of ::}

$C \wedge \mathcal{M}_c(t : Q)$

$\equiv$ {by definition of scope}

$C \wedge (len = t \wedge \mathcal{M}_c(Q))$

$\equiv$ {commutativity and associativity of $\wedge$}

$len = t \wedge (C \wedge \mathcal{M}_c(Q))$

$\equiv$ {by definition of ::}

$len = t \wedge \mathcal{M}_c(C :: Q)$

$\equiv$ {by definition of scope}

$\mathcal{M}_c(t : (C :: Q))$

The following refinement laws hold. The laws Cont-9 and Cont-10 state that the context operator is a refinement of its immediate components.

Cont– 9 $C \sqsubseteq C :: P$

Cont– 10 $P \sqsubseteq C :: P$

The parallel composition is monotonic, i.e.

Cont– 11 $$\frac{C' \supset C \qquad Q \sqsubseteq Q'}{C :: Q \sqsubseteq C' :: Q'}$$

Theorem 5.20 *Laws Cont-9 and Cont-11 are sound.*

Proof.

- Law Cont-9

  $\mathcal{M}_c(C :: Q)$

  $\equiv$ {by definition of ::}

  $C \wedge \mathcal{M}_c(Q)$

  $\supset$ {ITL}

  C

- Law Cont-11

  $\mathcal{M}_c(C' :: Q')$

  $\equiv$ {by definition of ::}

  $C' \wedge \mathcal{M}_c(Q')$

  $\supset$ $\{C' \supset C\}$

  $C \wedge \mathcal{M}_c(Q')$

$\supset$ $\{Q \sqsubseteq Q'\}$

$C \wedge \mathcal{M}_c(Q)$

$\equiv$ {by definition of ::}

$\mathcal{M}_c(C :: Q)$

5.4 Examples of Derivation

In this section we illustrate the use of algebraic laws described above to prove that two policy expressions are equal. The first example involves only the conditional operator.

Theorem 5.21 $w_2?(w_1?P : S) : (w_1?P : S) = w_1?P : (w_2?Q : S)$

Proof.

$w_2?(w_1?P : Q) : (w_1?P : S)$

$=$ {Cond-3}

$(w_1 \wedge w_2)?P : (w_2?Q : (w_1?P : S)$

$=$ {Cond-2}

$(w_1 \wedge w_2)?P : (\neg w_2?(w_1?P : S) : Q)$

$=$ {Cond-3}

$(w_1 \wedge w_2)?P : ((\neg w_2 \wedge w_1)?P : (\neg w_2)?S : Q))$

$=$ {Cond-4}

$w_1?P : ((\neg w_2)?S : Q)$

$=$ {Cond-2}

$w_1?P : (w_2?Q : S)$

Following is a valid equation for the triangle operator.

Theorem 5.22 $P \rhd^{\infty}_{false} Q : S = P$

Proof.

$$
\begin{aligned}
& P \rhd^{\infty}_{false} Q : S \\
= \quad & \{\text{Triangle-3}\} \\
& (\langle false \rangle P); (false?Q : NULL) \\
= \quad & \{\text{Cond-2}\} \\
& (\langle false \rangle P); (true?NULL : Q) \\
= \quad & \{\text{Cond-5}\} \\
& (\langle false \rangle P); NULL \\
= \quad & \{\text{Chop-1}\} \\
& \langle false \rangle P \\
= \quad & \{\text{Unless-1}\} \\
& P
\end{aligned}
$$

The following example combines the conditional operator and the triangle operator.

Theorem 5.23 $P \rhd^{\infty}_{w} (w?Q : S) : T = P \rhd^{\infty}_{w} Q : T$

Proof.

$$
\begin{aligned}
& P \rhd^{\infty}_{w} (w?Q : S) : T \\
= \quad & \{\text{Triangle-3}\} \\
& (\langle w \rangle P); (w?(w?Q : S) : NULL) \\
= \quad & \{\text{Cond-7}\} \\
& (\langle w \rangle P); (w?Q : NULL)
\end{aligned}
$$

$=$ {Triangle-3}

$P \rhd_w^\infty Q : T$

5.5 Summary

In this chapter an algebra for composing policies has been presented. A rich set of operators is defined to cater for the specification of complex policies in a compositional manner. Fundamental is the specification of dynamic changes in policies through policy composition. For example, the operators chop and triangle specify policies which are intended to change over time. For the latter, the change occurs when a guard is triggered or a time-out has elapsed. The former is more abstract as the point of change can be floating. However this point can be dictated by its first argument using the operators scope, as_long_as or unless. The operator scope specifies the duration of the policy while as_long_as and unless enforce the policy until a specific event occurs. Sound algebraic laws are established to state properties of the operators in terms of equational and refinement relationships among policies. Thus proofs of the equality of two policies or of the refinement of one policy by another can be carried out by derivations using the algebraic laws.

Chapter 6

Development of Secure Systems

Objectives

- To present an approach for developing secure systems.
- To define operators for composing secure action systems.
- To illustrate the enforcement of compound policies through system composition.
- To develop compositional rules for the verification of secure action systems.

6.1 Introduction

In this chapter, we present an approach for developing secure systems. Central to our objective is to provide a mechanism for the continual enforcement of dynamically changing security policies such as those presented in Chapter 5. The initial specification of a system has the form $\psi :: P$, where ψ is an ITL formula specifying the functional requirement

of the system, and P is the security policy of the system written in the policy language presented in Chapter 5. Then the algebra and the refinement calculus presented in the same chapter are used to transform the system specification into a more concrete one. We then use the compositional rules presented in this chapter to derive a secure action system that implements the system. A notion of *normal form* for secure action systems is defined. Then a set of closed operators are defined for composing normal forms. By closed operators we mean that the result of a composition still denotes a normal form. This technique provides a compositional way of devising complex systems from simpler ones. More importantly, the technique constitutes an effective mechanism for enforcing dynamically changing security policies that can change in response to time or events.

6.2 Composition of Secure Action Systems

In this section, we define operators for composing secure action systems. This is paramount for modelling complex systems by composing simpler ones. We define a *normal form* for secure action systems to be a tuple $(\mathcal{P}, \mathcal{A}, Agents, Actions, Policy)$ where the components are defined as in Section 3.4 of Chapter 3 (page 47). The following BNF describe the syntax of the operators over secure action systems' normal forms, where nf is a normal form, b an Boolean expression and SAS ranges over secure action systems.

$$\begin{aligned} SAS \quad ::= \quad & nf \mid SAS_1; SAS_2 \mid SAS_1 \| SAS_2 \mid \langle b \rangle SAS \mid t : SAS \\ & \mid b?SAS_1 : SAS_2 \mid b * SAS \mid SAS_1 \rhd_b^t SAS_2 : SAS_3 \end{aligned}$$

A sequential composition $SAS_1; SAS_2$ of secure action systems denotes a system that behaves like SAS_1 and when SAS_1 terminates it behaves like SAS_2. A parallel composition

$SAS_1 \| SAS_2$ denotes a superposition of two systems SAS_1 and SAS_2. The expression $\langle b \rangle SAS$ denotes a system that behaves like SAS until some event b occurs or the system SAS terminates. We call this operator "*unless*". A system $t : SAS$ behaves like SAS for a period no longer than t time units. If SAS terminates in less than t time units then the system terminates at the same time otherwise the execution of SAS is stopped after t time units. The conditional $b?SAS_1 : SAS_2$ behaves like SAS_1 if initially b is true and like SAS_2 otherwise. The expression $b * SAS$ denotes the iteration of SAS until b evaluates to false. The expression $SAS_1 \rhd_b^t SAS_2 : SAS_3$ denotes a system that behaves like SAS_1 until a condition b becomes true or a duration of t time units elapses or SAS_1 terminates. Then it changes to behave like SAS_2 if b is true or like SAS_3 otherwise. This operator is called "*time to change*" operator. We believe that this set of operators is rich enough to specify complex systems that may change dynamically in response to time or events.

We show that any of the expressions defined above can be implemented as a normal form. Let $SAS_1 \mathrel{\widehat{=}} (\mathcal{P}_1, \mathcal{A}_1, Agents_1, Actions_1, Policy_1)$,
$SAS_2 \mathrel{\widehat{=}} (\mathcal{P}_2, \mathcal{A}_2, Agents_2, Actions_2, Policy_2)$ and
$SAS_3 \mathrel{\widehat{=}} (\mathcal{P}_3, \mathcal{A}_3, Agents_3, Actions_3, Policy_3)$ be three systems in normal form. We assume that any action $a \in \mathcal{A}_i \cap \mathcal{A}_j$, $i \neq j$, has the same definition in $Actions_i$ and $Actions_j$. Let $final_i \mathrel{\widehat{=}} \bigwedge\limits_{a \in \mathcal{A}_i} \neg(g_a \wedge h_a)$ be a formula that holds in a state if no actions in system SAS_i are enabled in that state, for $i = 1, 2, 3$. Recall that g_a and h_a stand for the functional guard and the security guard of action a, respectively. Similarly, $Init_i(p)$ denotes the initialisation statements of an agent p in the system SAS_i.

6.2.1 Sequential Composition

To express $SAS_1; SAS_2$ as a normal form, we need a mechanism to detect the final state of SAS_1 and to start dynamically SAS_2 from that state. Let *watchdog* be an new agent that sets a local variable $flag$ to true when it detects the final state of SAS_1. Initially $flag$ is set to false. The agent *watchdog* participates in a single action called *switch* and both are defined as follows:

(6.1) $$\textbf{agent}\ \ watchdog: \ \textbf{var}\ \ flag;\ flag := false.$$

(6.2) $$\textbf{action}\ \ switch\ \textbf{on}\ \ \mathcal{P}_1, \mathcal{P}_2\ \textbf{by}\ \ watchdog:$$
$$\neg flag \rightarrow flag, y_{p_1}, \ldots, y_{p_n} := true, c_{p_1}, \ldots, c_{p_n}.$$

where $p_i \in \mathcal{P}_2$ and there exists an initialisation statement $Init_2(p_i) = (y_{p_i} := c_{p_i})$, for $i = 1, \ldots, n$.

A sequential composition $SAS_1; SAS_2$ denotes a normal form

$$(\mathcal{P}, \mathcal{A}, Agents, Actions, Policy)$$

where

- $\mathcal{P} \ \widehat{=}\ \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{watchdog\}$ and $\mathcal{A} \ \widehat{=}\ \mathcal{A}_1 \cup \mathcal{A}_2 \cup \{switch\}$;
- $Agents \ \widehat{=}\ Agents_1 \cup Agents'_2 \cup \{(6.1)\}$,

  where $Agents'_2$ is identical to $Agents_2$ without initialisation statements for agents.
- $Actions \ \widehat{=}\ Actions_1 \cup Actions_2 \cup \{(6.2)\}$;

- $Policy$ contains the functions $autho$, $autho^+$ and $autho^-$ defined as follows:

  1. $autho(x, y, z) = (\neg flag \wedge autho_1(x, y, z))$, $\forall x, y \in \mathcal{P}_1 \setminus \mathcal{P}_2,\ \forall z \in \mathcal{A}_1 \setminus \mathcal{A}_2$;
  2. $autho(x, y, z) = (flag \wedge autho_2(x, y, z))$, $\forall x, y \in \mathcal{P}_2 \setminus \mathcal{P}_1,\ \forall z \in \mathcal{A}_2 \setminus \mathcal{A}_1$;
  3. $autho(x, y, z) = (\neg flag \wedge autho_1(x, y, z)) \vee (flag \wedge autho_2(x, y, z))$, $\forall x, y \in \mathcal{P}_1 \cap \mathcal{P}_2,\ \forall z \in \mathcal{A}_1 \cap \mathcal{A}_2$;
  4. $autho(watchdog, x, switch) = final_1$, $\forall x \in \mathcal{P}_1 \cup \mathcal{P}_2$;
  5. $autho(x, y, z) = false$, otherwise.

  The functions $autho^+$ and $autho^-$ are defined similarly to $autho$.

Note that the actions in SAS_1 are not enabled if $flag$ evaluates to true and the actions in SAS_2 are not enabled if $flag$ evaluates to false. Initially $flag$ is equal to false and it is kept to false until SAS_1 terminates. At the termination of SAS_1 the action $switch$ is enabled because $flag$ is still false. Since it is the only enabled action in that state, it is selected for execution. Its execution sets $flag$ to true, performs all the initialisation statements provided in SAS_2 and preserves the postconditions of SAS_1 that are not affected by these initialisation statements (see the formal semantics of secure action systems in Chapter 3, page 30). From there on, only actions in SAS_2 can be enabled because $flag$ is true.

6.2.2 Parallel Composition

A parallel composition $SAS_1 \| SAS_2$ denotes a normal form

$$(\mathcal{P}, \mathcal{A}, Agents, Actions, Policy)$$

where

- $\mathcal{P} \mathrel{\widehat{=}} \mathcal{P}_1 \cup \mathcal{P}_2$ and $\mathcal{A} \mathrel{\widehat{=}} \mathcal{A}_1 \cup \mathcal{A}_2$;

  We assume that any agent in $\mathcal{P}_1 \cap \mathcal{P}_2$ has the same definition in $Agents_1$ and $Agents_2$.

- $Agents \mathrel{\widehat{=}} Agents_1 \cup Agents_2$ and $Actions \mathrel{\widehat{=}} Actions_1 \cup Actions_2$;

- $Policy$ contains the functions $autho$, $autho^+$ and $autho^-$ defined as follows:

  1. $autho(x, y, z) = autho_1(x, y, z), \forall x \in \mathcal{P}_1 \setminus \mathcal{P}_2,\ \forall y \in \mathcal{P}_1,\ \forall z \in \mathcal{A}_1$;
  2. $autho(x, y, z) = autho_1(x, y, z), \forall x \in \mathcal{P}_1,\ \forall y \in \mathcal{P}_1 \setminus \mathcal{P}_2,\ \forall z \in \mathcal{A}_1$;
  3. $autho(x, y, z) = autho_1(x, y, z), \forall x \in \mathcal{P}_1,\ \forall y \in \mathcal{P}_1,\ \forall z \in \mathcal{A}_1 \setminus \mathcal{A}_2$;
  4. $autho(x, y, z) = autho_2(x, y, z), \forall x \in \mathcal{P}_2 \setminus \mathcal{P}_1,\ \forall y \in \mathcal{P}_2,\ \forall z \in \mathcal{A}_2$;
  5. $autho(x, y, z) = autho_2(x, y, z), \forall x \in \mathcal{P}_2,\ \forall y \in \mathcal{P}_2 \setminus \mathcal{P}_1,\ \forall z \in \mathcal{A}_2$;
  6. $autho(x, y, z) = autho_2(x, y, z), \forall x \in \mathcal{P}_2,\ \forall y \in \mathcal{P}_2,\ \forall z \in \mathcal{A}_2 \setminus \mathcal{A}_1$;
  7. $autho(x, y, z) = autho_1(x, y, z) \wedge autho_2(x, y, z), \forall x, y \in \mathcal{P}_1 \cap \mathcal{P}_2,$ $\forall z \in \mathcal{A}_1 \cap \mathcal{A}_2$;
  8. $autho(x, y, z) = false$, otherwise.

  The functions $autho^+$ and $autho^-$ are defined similarly to $autho$.

In a parallel $SAS_1 \| SAS_2$, the security policy of a joint action which is common to SAS_1 and SAS_2 is the conjunction of the security policies of the action in each of the subsystems. The actions that are not common to the two subsystems keep their security policies.

6.2.3 Unless

A normal formal that implements an expression $\langle b \rangle SAS_1$ contains an agent $detector$ that monitors the value of b and performs an action $stop$ to end the execution of the system once b is true. The agent $detector$ and the action $stop$ are defined as follows, where the local variable $flag$ is set to true when the condition b is true:

(6.3) $$\textbf{agent}\ \ detector:\ \textbf{var}\ \ flag;\ flag := false.$$

(6.4)

$$\textbf{action}\ \ stop\ \textbf{on}\ \ detector\ \textbf{by}\ \ detector\ \textbf{async}\ \ \mathcal{P}_1:\ (\neg flag \wedge b) \rightarrow flag := true.$$

The expression $\langle b \rangle SAS_1$ denotes a normal form

$$(\mathcal{P}_1 \cup \{detector\}, \mathcal{A}_1 \cup \{stop\}, Agents_1 \cup \{(6.3)\}, Actions_1 \cup \{(6.4)\}, Policy)$$

where

1. $autho(x, y, z) = \neg b \wedge \neg flag \wedge autho_1(x, y, z), \forall x, y \in \mathcal{P}_1,\ \forall z \in \mathcal{A}_1$;
2. $autho(detector, detector, stop) = true$;
3. $autho(x, y, z) = false$, otherwise.

The functions $autho^+$ and $autho^-$ are defined similarly to $autho$. The security guard of each action is strengthened with the condition $\neg b \wedge \neg flag$ so that the action is not enabled when the condition b becomes true. The system will then terminate.

6.2.4 Duration

In order to express $t : SAS_1$ in normal form, we need an agent $clock$ that observes the execution of SAS_1 and stops it if it does not terminate before t time units. The agent $clock$ has a state variable $time$ initialised to 0, and participates in the execution of a single action $tick$ that increments the value of the variable $time$ by 1 at each transition of the system.

(6.5) $$\textbf{agent}\ \ clock : \ \textbf{var}\ \ time;\ time := 0.$$

(6.6) $$\textbf{action}\ \ tick\ \textbf{on}\ \ clock\ \textbf{by}\ \ clock :\ time < t \rightarrow time := time + 1.$$

The expression $t : SAS_1$ denotes a normal form

$$(\mathcal{P}_1 \cup \{clock\}, \mathcal{A}_1 \cup \{tick\}, Agents_1 \cup \{(6.5)\}, Actions_1 \cup \{(6.6)\}, Policy)$$

where

1. $autho(x, y, z) = time < t \wedge autho_1(x, y, z), \forall x, y \in \mathcal{P}_1,\ \forall z \in \mathcal{A}_1$;
2. $autho(clock, clock, tick) = true$;
3. $autho(x, y, z) = false$, otherwise.

The functions $autho^+$ and $autho^-$ are defined similarly to $autho$. Note that no action (including the action $tick$) is enabled when the value of $time$ equals t and thereafter. So the system will stop.

6.2.5 Conditional

To express a conditional $b?SAS_1 : SAS_2$ in normal form, we need an agent $choice$ and an action $select$ defined below. The Boolean variable $cond$ in the agent $choice$ stores the value of the Boolean expression b in the initial state while the control variable $lock$ is used to keep the value of $cond$ stable through the action $select$. The action $select$ is executed in the initial state to assign $cond$ the value of the expression b. If b holds in the initial state, the action $select$ performs the initialisation statements of SAS_1, otherwise it performs those of SAS_2. After that it will never execute again because $lock = 1$, keeping so the values of $cond$ and $lock$ unchanged till termination. The variables $cond$ and $lock$ are used to control the execution of the actions in SAS_1 and SAS_2. Only actions in SAS_1 are executed when $cond$ is true, otherwise only those in SAS_2 are executed.

(6.7) $$\textbf{agent}\ \ choice:\ \ \textbf{var}\ \ cond, lock;\ cond, lock := false, 0.$$

(6.8)
$$\begin{array}{l}\textbf{action}\ \ select\ \textbf{on}\ \ \mathcal{P}_1, \mathcal{P}_2\ \textbf{by}\ \ choice:\\ lock = 0 \rightarrow \quad cond, lock := b, 1\\ \qquad\qquad \| \ \texttt{if}\ \ b\ \texttt{then}\ \ y_{p_1}, \ldots, y_{p_n} := c_{p_1}, \ldots, c_{p_n}\\ \qquad\qquad\quad \texttt{else}\ \ y_{q_1}, \ldots, y_{q_m} := c_{q_1}, \ldots, c_{q_m}\ \texttt{fi}\ .\end{array}$$

where $p_i \in \mathcal{P}_1$, $q_j \in \mathcal{P}_2$ and there exists an initialisation statement $Init_1(p_i) = (y_{p_i} := c_{p_i})$ and an initialisation statement $Init_2(q_j) = (y_{q_j} := c_{q_j})$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

A conditional $b?SAS_1 : SAS_2$ denotes a normal form

$$(\mathcal{P}, \mathcal{A}, Agents, Actions, Policy)$$

where

- $\mathcal{P} \mathrel{\widehat{=}} \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{choice\}$ and $\mathcal{A} \mathrel{\widehat{=}} \mathcal{A}_1 \cup \mathcal{A}_2 \cup \{select\}$;

- $Agents \mathrel{\widehat{=}} Agents'_1 \cup Agents'_2 \cup \{(6.7)\}$,

  where $Agents'_i$ is identical to $Agents_i$ without the initialisation statements, $i = 1, 2$;

- $Actions \mathrel{\widehat{=}} Actions_1 \cup Actions_2 \cup \{(6.8)\}$;

- $Policy$ contains the functions $autho$, $autho^+$ and $autho^-$ defined as follows:

  1. $autho(x, y, z) = (lock = 1 \wedge cond \wedge autho_1(x, y, z))$, $\forall x, y \in \mathcal{P}_1 \setminus \mathcal{P}_2$, $\forall z \in \mathcal{A}_1 \setminus \mathcal{A}_2$;

  2. $autho(x, y, z) = (lock = 1 \wedge \neg cond \wedge autho_2(x, y, z))$, $\forall x, y \in \mathcal{P}_2 \setminus \mathcal{P}_1$, $\forall z \in \mathcal{A}_2 \setminus \mathcal{A}_1$;

  3. $autho(x, y, z) = (lock = 1 \wedge cond \wedge autho_1(x, y, z)) \vee (lock = 1 \wedge \neg cond \wedge autho_2(x, y, z))$, $\forall x, y \in \mathcal{P}_1 \cap \mathcal{P}_2,\ \forall z \in \mathcal{A}_1 \cap \mathcal{A}_2$;

  4. $autho(choice, choice, select) = true$;

  5. $autho(x, y, z) = false$, otherwise.

  The functions $autho^+$ and $autho^-$ are defined similarly to $autho$.

6.2.6 Iteration

The expression of an iteration $b*SAS_1$ in normal form requires an agent $iterator$ responsible for scheduling an execution of SAS_1 if b evaluates to true. This agent participates in a single action $loop$ which is enabled in the initial state if b holds in that state, and is enabled each time the system SAS_1 terminates in a state where the condition b is true. The body of the action executes the initialisation statements of SAS_1 to initiate a new iteration. Similarly to the sequential composition, the postconditions of SAS_1 which are not modified by the initialisation statements are preserved during the execution of the action $loop$. The Boolean variable $flag$ is used to distinguish the first iteration from subsequent ones.

(6.9) $$\textbf{agent}\ \ iterator:\ \textbf{var}\ \ flag; flag := false.$$

(6.10) $$\begin{array}{l}\textbf{action}\ \ loop\ \textbf{on}\ \ \mathcal{P}_1\ \textbf{by}\ \ iterator:\\ \qquad b \rightarrow\ flag, y_{p_1}, \ldots, y_{p_n} := true, c_{p_1}, \ldots, c_{p_n}.\end{array}$$

where $p_i \in \mathcal{P}_1$ and there exists an initialisation statement $Init_1(p_i) = (y_{p_i} := c_{p_i})$, for $i = 1, \ldots, n$.

An expression $b * SAS_1$ denotes a normal form

$$(\mathcal{P}, \mathcal{A}, Agents, Actions, Policy)$$

where

- $\mathcal{P} \ \widehat{=}\ \mathcal{P}_1 \cup \{iterator\}$ and $\mathcal{A} \ \widehat{=}\ \mathcal{A}_1 \cup \{loop\}$;

- $Agents \mathrel{\widehat{=}} Agents'_1 \cup \{(6.9)\}$,

  where $Agents'_1$ is identical to $Agents_1$ without the initialisation statements;

- $Actions \mathrel{\widehat{=}} Actions_1 \cup \{(6.10)\}$;

- $Policy$ contains the functions $autho$, $autho^+$ and $autho^-$ defined as follows:

  1. $autho(x, y, z) = flag \wedge autho_1(x, y, z), \forall x, y \in \mathcal{P}_1, \forall z \in \mathcal{A}_1$;
  2. $autho(iterator, x, loop) = final_1 \vee \neg flag, \forall x \in \mathcal{P}_1$;
  3. $autho(x, y, z) = false$, otherwise.

  The functions $autho^+$ and $autho^-$ are defined similarly to $autho$.

6.2.7 Time to Change

The expression $SAS_1 \rhd^t_b SAS_2 : SAS_3$ denotes a normal form defined as follows, where $observer$ is a new agent and $observe$ a new action. The Boolean variables $cond$ and $timeout$ are used to detect respectively when the condition b becomes true and when a duration of t time units elapses since the execution of SAS_1 started. If b occurs during the execution of SAS_1 then SAS_1 is stopped and SAS_2 is executed, otherwise SAS_3 is executed if SAS_1 terminates or a duration of t time units has elapsed.

$$\begin{array}{l} \textbf{agent}\ \ observer : \ \textbf{var}\ \ flag, cond, timeout; \\ \qquad flag, cond, timeout := true, false, false. \end{array} \tag{6.11}$$

(6.12)

action $observe$ **on** $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ **by** $observer$ **async** $clock$:

$flag \rightarrow$ `if` $(b \vee final_1)$

`then` $flag, cond, y_{p_1}, \ldots, y_{p_n} := false, true, c_{p_1}, \ldots, c_{p_n}$

`else` $flag, timeout, y_{q_1}, \ldots, y_{q_m} := false, true, c_{q_1}, \ldots, c_{q_m}$ `fi` .

where $p_i \in \mathcal{P}_2$, $q_j \in \mathcal{P}_3$ and there exists an initialisation statement $Init_2(p_i) = (y_{p_i} := c_{p_i})$ and an initialisation statement $Init_3(q_j) = (y_{q_j} := c_{q_j})$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

Similarly to the action $switch$ in the sequential composition, the action $observe$ preserves the postconditions of SAS_1 that are not affected by the initialisation steps of SAS_2 and SAS_3.

The normal form denoted by an expression $SAS_1 \rhd_b^t SAS_2 : SAS_3$ is

$$(\mathcal{P}, \mathcal{A}, Agents, Actions, Policy)$$

where

- $\mathcal{P} \mathrel{\widehat{=}} \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \{clock, observer\}$ and $\mathcal{A} \mathrel{\widehat{=}} \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3 \cup \{tick, observe\}$;
- $Agents \mathrel{\widehat{=}} Agents_1 \cup Agents'_2 \cup Agents'_3 \cup \{(6.5), (6.11)\}$,

  where $Agents'_i$ is identical to $Agents_i$ without the initialisation statements, $i = 2, 3$;
- $Actions \mathrel{\widehat{=}} Actions_1 \cup Actions_2 \cup Actions_3 \cup \{(6.6), (6.12)\}$;
- $Policy$ contains the functions $autho$, $autho^+$ and $autho^-$ defined as follows:

1. $autho(x, y, z) = \neg cond \wedge \neg timeout \wedge autho_1(x, y, z)$, $\forall x, y \in \mathcal{P}_1 \setminus (\mathcal{P}_2 \cup \mathcal{P}_3)$, $\forall z \in \mathcal{A}_1 \setminus (\mathcal{A}_2 \cup \mathcal{A}_3)$;

2. $autho(x, y, z) = (\neg cond \wedge \neg timeout \wedge autho_1(x, y, z)) \vee$ $(cond \wedge autho_2(x, y, z))$ $\forall x, y \in (\mathcal{P}_1 \cap \mathcal{P}_2) \setminus \mathcal{P}_3, \forall z \in (\mathcal{A}_1 \cap \mathcal{A}_2) \setminus \mathcal{A}_3$;

3. $autho(x, y, z) = (\neg cond \wedge \neg timeout \wedge autho_1(x, y, z)) \vee$ $(timeout \wedge autho_3(x, y, z))$ $\forall x, y \in (\mathcal{P}_1 \cap \mathcal{P}_3) \setminus \mathcal{P}_2, \forall z \in (\mathcal{A}_1 \cap \mathcal{A}_3) \setminus \mathcal{A}_2$;

4. $autho(x, y, z) = (\neg cond \wedge \neg timeout \wedge autho_1(x, y, z)) \vee$ $(cond \wedge \neg timeout \wedge autho_2(x, y, z)) \vee (\neg cond \wedge timeout \wedge autho_3(x, y, z))$ $\forall x, y \in \mathcal{P}_1 \cap \mathcal{P}_2 \cap \mathcal{P}_3, \forall z \in \mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3$;

5. $autho(clock, x, observe) = true$, $\forall x \in \mathcal{P}_1$;

6. $autho(observer, x, observe) = final_1 \vee b \vee time \geq t$, $\forall x \in \mathcal{P}_1$;

7. $autho(x, y, z) = false$, otherwise.

The functions $autho^+$ and $autho^-$ are defined similarly to $autho$.

6.3 Compositional Verification of SAS

We assume that system properties are expressed in ITL. These may be functional, temporal or security properties of the system. The verification of SASs can be done using the denotational semantics presented in Chapter 3. Indeed, a SAS satisfies a property ψ (expressed in ITL) if its semantics implies ψ. However, a syntactical approach for system verification is more useful in practise as it hides the complicated formal semantics of the

system to the system designers. In this section we provide compositional proof rules for the verification of SASs.

First, we define a satisfaction relation **sat** in Definition 6.1.

Definition 6.1 *A secure action system SAS satisfies a formula ψ, denoted by SAS* **sat** *ψ, if the semantics of SAS implies $\bigcirc\psi$, i.e.*

$$SAS \textbf{ sat } \psi \ \widehat{=}\ \mathcal{C}[\![SAS]\!] \supset \bigcirc\psi.$$

For example, if SAS models a bank system then SAS must satisfy the requirement that any transaction on a bank account leaves its balance not lower than the minimum threshold, i.e. for each bank account b,

$$SAS \textbf{ sat } \Box(b.balance \geq b.min).$$

Similarly, we can prove properties about the security policy enforced by a secure action system, using the predicates $autho^+$, $autho^-$ and $autho$ which are specified in the system. For example, one can verify whether a trusted agent $john$, say may eventually read the information stored in $file$, i.e.

$$SAS \textbf{ sat } \Diamond autho(john, file, read).$$

It can also be checked if a *bad* guy bob, say is never allowed to write on $file$, i.e.

$$SAS \textbf{ sat } \Box\neg autho(bob, file, write).$$

In the following rules, the symbol ψ stands for an ITL formula, w for a state formula, and b for a Boolean expression.

$$\textbf{SAT-1}\quad \frac{SAS\ \textbf{sat}\ \psi_1 \qquad \psi_1 \supset \psi_2}{SAS\ \textbf{sat}\ \psi_2}$$

The rule **SAT-1** is known as the *consequence rule*. It says that if a system satisfies a property, then it satisfies all the consequences of the property. Its proof is straightforward from Definition 6.1 and the transitivity of "$\supset$".

$$\textbf{SAT-2}\quad \frac{SAS\ \textbf{sat}\ \psi_1 \qquad SAS\ \textbf{sat}\ \psi_2}{SAS\ \textbf{sat}\ (\psi_1\ \wedge\ \psi_2)}$$

The rule **SAT-2** can be used to prove that a system satisfies a set of properties, e.g. a functional requirement and a security policy. Its proof is immediate from Definition 6.1 and the left distributivity of "$\supset$" over "$\wedge$".

$$\textbf{SAT-3}\quad \frac{\begin{array}{c} SAS_1\ \textbf{sat}\ (As_1 \supset \psi_1) \qquad SAS_2\ \textbf{sat}\ (As_2 \supset \psi_2) \\ (As\ \wedge\ \psi_1) \supset As_2 \qquad (As\ \wedge\ \psi_2) \supset As_1 \qquad As \supset (As_1\ \vee\ As_2) \end{array}}{(SAS_1 \| SAS_2)\ \textbf{sat}\ (As \supset (\psi_1\ \wedge\ \psi_2))}$$

The rule **SAT-3** allows to prove properties about the parallel composition of secure action systems. The formula As_i specifies an assumption about the environment of the system SAS_i, $i = 1, 2$ while the formula ψ_i stands for the commitment of the system. The interpretation is that if the environment fulfils the assumption then the execution of the system within that environment satisfies the commitment. Communication between two systems can be achieved through a joint action which agents from the two systems participate in. In this case each of the systems is part of the environment of the other, i.e. the commitment ψ_i is part of the assumption As_j, $i \neq j$. This is captured in the rule

SAT-3 by the premises $(As \ \wedge \ \psi_1) \supset As_2$ and $(As \ \wedge \ \psi_2) \supset As_1$, where As is the assumption about the environment in which the two systems are executed in parallel.

Proof.

1. SAS_1 **sat** $(As_1 \supset \psi_1)$ {assumption}
2. SAS_2 **sat** $(As_2 \supset \psi_2)$ {assumption}
3. $(As \ \wedge \ \psi_1) \supset As_2$ {assumption}
4. $(As \ \wedge \ \psi_2) \supset As_1$ {assumption}
5. $As \supset (As_1 \vee As_2)$ {assumption}
6. $\mathcal{C}[\![SAS_1 \| SAS_2]\!] \supset (\mathcal{C}[\![SAS_1]\!] \wedge \mathcal{C}[\![SAS_2]\!])$ {definition of $\|$}

   $\supset \bigcirc(As_1 \supset \psi_1) \wedge \bigcirc(As_2 \supset \psi_2)$ {1., 2. and Definition 6.1}

   $\supset \bigcirc((As_1 \supset \psi_1) \wedge (As_2 \supset \psi_2))$ {ITL}
7. $(\mathcal{C}[\![SAS_1 \| SAS_2]\!] \wedge \bigcirc As \wedge \bigcirc As_1)$

   $\supset \bigcirc(As \wedge As_1 \wedge (As_1 \supset \psi_1) \wedge (As_2 \supset \psi_2))$ {6. and ITL}

   $\supset \bigcirc(As \wedge \psi_1 \wedge (As_2 \supset \psi_2))$ {modus ponens}

   $\supset \bigcirc(\psi_1 \wedge As_2 \wedge (As_2 \supset \psi_2))$ {3.}

   $\supset \bigcirc(\psi_1 \wedge \psi_2)$ {modus ponens}
8. $(\mathcal{C}[\![SAS_1 \| SAS_2]\!] \wedge \bigcirc As \wedge \bigcirc As_2) \supset \bigcirc(\psi_1 \wedge \psi_2)$ {Similarly to 7. }
9. $(\mathcal{C}[\![SAS_1 \| SAS_2]\!] \wedge \bigcirc As) \supset \bigcirc(\psi_1 \wedge \psi_2)$ {5., 7. and 8.}
10. $(SAS_1 \| SAS_2)$ **sat** $(As \supset (\psi_1 \ \wedge \ \psi_2))$ {9. and Definition 6.1}

SAT-4 $$\frac{SAS_1 \textbf{ sat } \psi_1 \qquad SAS_2 \textbf{ sat } \psi_2}{(SAS_1; SAS_2) \textbf{ sat } (\psi_1; skip; \psi_2)}$$

The rule **SAT-4** is about the sequential composition of secure action systems. In the definition of a sequential composition $SAS_1; SAS_2$ (see Section 6.2.1), the variables in

SAS_1 which are not modified by the initialisation statements of SAS_2 are kept unchanged during the initialisation step of SAS_2.

Proof.

1. SAS_1 **sat** ψ_1 {assumption}
2. SAS_2 **sat** ψ_2 {assumption}
3. $\mathcal{C}[\![SAS_1; SAS_2]\!] \supset (\mathcal{C}[\![SAS_1]\!]; \mathcal{C}[\![SAS_2]\!])$ {sequential composition}

   $\supset \bigcirc\psi_1; \bigcirc\psi_2$ {1., 2. and Definition 6.1}
4. $(SAS_1; SAS_2)$ **sat** $(\psi_1; skip; \psi_2)$ {3. and Definition 6.1}

SAT-5-a $$\frac{SAS \textbf{ sat } \psi \qquad \psi \equiv\ keep\ \psi}{(t : SAS) \textbf{ sat } (len \leq t \wedge \psi)}$$

SAT-5-b $$\frac{SAS \textbf{ sat } len \geq t}{(t : SAS) \textbf{ sat } len = t}$$

The *duration* rule **SAT-5** can be used to prove properties of a system when the system is executed for a period no longer than t time units, say. The rule **SAT-5-a** can be used to prove properties which are fix-points of the operator $keep$ defined in Chapter 3 (see page 34). This operator can be used to specify an important class of systems' properties which has been investigated by Moszkowski [59]. We recall that the formula $keep\ \psi$, for some sub-formula ψ, is defined to be true on an interval if and only if ψ is true on every unit subinterval (i.e. consisting of exactly two adjacent states): $keep\ \psi\ \widehat{=}\ \boxdot(skip \supset \psi)$. The formula $keep((\bigcirc X) = X + 1)$ is an example of such a fix-point, where X is a variable. It states that X increases by 1 between every pair of adjacent states. The rule **SAT-5-b** determines the maximum duration an execution can last for. The proofs of these rules are straightforward.

SAT-6-a $$\frac{SAS \textbf{ sat } ((keep\ \neg b \wedge \psi \wedge fin\ b); true)}{(< b > SAS) \textbf{ sat } (keep\ \neg b \wedge \psi \wedge fin\ b)}$$

SAT-6-b $$\frac{SAS \textbf{ sat } \psi \qquad \psi \supset \Box\neg b}{(< b > SAS) \textbf{ sat } \psi}$$

Similarly to the rule **SAT-5**, the rule **SAT-6** can be used to establish properties of a system when the system is observed until a specific event occurs or it terminates. These rules are important, especially to reason about systems that are meant to change dynamically in response to time and events. The rule **SAT-6-a** states the case in which the condition (event) b becomes true in the course of the execution of the system, and the rule **SAT-6-b** treats the case it does not. Their proofs are straightforward.

SAT-7 $$\frac{SAS_1 \textbf{ sat } \psi_1 \qquad SAS_2 \textbf{ sat } \psi_2}{b?SAS_1 : SAS_2 \textbf{ sat } ((b \wedge \bigcirc\psi_1) \vee (\neg b \wedge \bigcirc\psi_2))}$$

The rule **SAT-7** allows to prove properties about a conditional. The proof is evident from the definition of conditional.

SAT-8 $$\frac{SAS \textbf{ sat } (w \wedge \psi \wedge fin\, w)}{b * SAS \textbf{ sat } ((b \wedge \bigcirc(w \wedge \psi \wedge \ fin\, w))^* \wedge fin\, (\neg b))}$$

In the rule **SAT-8** about iteration, the state formula w is called an *invariant* of the loop. It holds at the beginning and the end of each iteration of the system SAS_1. The next iteration starts with the execution of the initialisation statements which obviously preserves the invariant. The last iteration, if it exists, terminates in a state where the condition of the loop b is false.

SAT-9 $$\frac{(SAS_1) \textbf{ sat } \psi_1 \qquad SAS_2 \textbf{ sat } \psi_2 \qquad SAS_3 \textbf{ sat } \psi_3}{(SAS_1 \rhd_b^t SAS_2 : SAS_3) \textbf{ sat } (\theta_1 \vee \theta_2)}$$

where $\theta_1 \mathrel{\widehat{=}} (\psi_1 \wedge\ keep\ \neg b \wedge len \leq t \wedge fin\ b); skip; \psi_2$ and

$\theta_2 \mathrel{\widehat{=}} (\psi_1 \wedge\ \Box\neg b \wedge len \leq t); skip; \psi_3$.

The rule **SAT-9** allows to prove properties of the operator "$\rhd$". Let $\gamma \mathrel{\widehat{=}} \bigwedge\limits_{a \in \mathcal{A}_1} \neg(g_a \wedge h_a)$ be a formula that holds in the final state of system SAS_1.

Proof.

1. SAS_1 **sat** ψ_1 {assumption}
2. SAS_2 **sat** ψ_2 {assumption}
3. SAS_3 **sat** ψ_3 {assumption}
4. $\mathcal{C}[\![SAS_1 \rhd_b^t SAS_2 : SAS_3]\!]$

   $\supset ((\mathcal{C}[\![SAS_1]\!] \wedge \bigcirc(keep\ \neg b \wedge len \leq t)); (\mathcal{C}[\![SAS_2]\!] \wedge b))$

   $\vee((\mathcal{C}[\![SAS_1]\!] \wedge \bigcirc(\Box\neg b \wedge len = t)); \mathcal{C}[\![SAS_3]\!])$

   $\vee((\mathcal{C}[\![SAS_1]\!] \wedge \bigcirc(\Box\ \neg b \wedge len < t \wedge \gamma)); \mathcal{C}[\![SAS_3]\!])$ {definition of "$\rhd$"}

   $\supset ((\bigcirc\psi_1 \wedge \bigcirc(keep\ \neg b \wedge len \leq t)); (\bigcirc\psi_2 \wedge b))$

   $\vee((\bigcirc\psi_1 \wedge \bigcirc(\Box\neg b \wedge len \leq t)); \bigcirc\psi_3)$ {1., 2. and 3.}

   $\supset ((\bigcirc(\psi_1 \wedge\ keep\ \neg b \wedge len \leq t \wedge fin\ b)); skip; \psi_2)$

   $\vee((\bigcirc(\psi_1 \wedge\ \Box\neg b \wedge len \leq t)); skip; \psi_3)$ {ITL}

   $\supset \bigcirc(\theta_1 \vee \theta_2)$ {ITL}
5. $(SAS_1 \rhd_b^t SAS_2 : SAS_3)$ **sat** $(\theta_1 \vee \theta_2)$ {Definition 6.1}

6.3.1 Additional Rules for Security Policy Enforcement

In this section we present useful rules for verifying that a SAS enforces a security policy specified as in Chapter 5. These rules are *additional* as they can be established using

the rules presented above. In the sequel, $\mathcal{M}_c$ stands for the semantic function of security policies defined in Chapter 5. Moreover, P_1, P_2 and P_3 range over complete security policies.

Let us consider the following complete simple policy (the syntax of simple policies is given in Chapter 4) where $f_{s,o,a}$, $g_{s,o,a}$ and $h_{s,o,a}$ for $s, o \in \mathcal{P}$ and $a \in \mathcal{A}$ are boolean expressions, $\mathcal{P}$ is a finite set of subjects' and objects' names and $\mathcal{A}$ a finite set of actions' names.

(6.13)

$$\{\lceil f_{s,o,a}\rceil^0 \leftrightarrow autho^+(s,o,a), \lceil g_{s,o,a}\rceil^0 \leftrightarrow autho^-(s,o,a), \lceil h_{s,o,a}\rceil^0 \leftrightarrow autho(s,o,a) \mid s, o \in \mathcal{P}, a \in \mathcal{A}\}.$$

The rule **SAT-10** says that this policy is enforced by any secure action system of the form $(\mathcal{P}, \mathcal{A}, Agents, Actions, \{autho^+, autho^-, autho\})$ such that

(6.14)
$$\forall s, o \in \mathcal{P}, \forall a \in \mathcal{A} : \begin{cases} autho^+(s,o,a) \mathrel{\widehat{=}} f_{s,o,a} \\ autho^-(s,o,a) \mathrel{\widehat{=}} g_{s,o,a} \\ autho(s,o,a) \mathrel{\widehat{=}} h_{s,o,a} \end{cases}$$

This means that (6.14) implements the security policy (6.13) at the concrete level.

SAT-10
$$\frac{(6.14)}{(\mathcal{P}, \mathcal{A}, Agents, Actions, \{autho^+, autho^-, autho\})\ \textbf{sat}\ \mathcal{M}_c(6.13)}$$

The proof of the rule **SAT-10** is straightforward from Theorem 3.7 (5).

The following rules can be used to develop a secure action system that enforces a compound policy.

SAT-11 $$\frac{SAS_1 \textbf{ sat } \mathcal{M}_c(P_1) \qquad SAS_2 \textbf{ sat } \mathcal{M}_c(P_2)}{(SAS_1; SAS_2) \textbf{ sat } \mathcal{M}_c(P_1 \frown P_2)}$$

SAT-12 $$\frac{SAS_1 \textbf{ sat } \mathcal{M}_c(P_1)}{(\langle b \rangle SAS_1) \textbf{ sat } \mathcal{M}_c(\langle b \rangle P_1)}$$

SAT-13 $$\frac{SAS_1 \textbf{ sat } \mathcal{M}_c(P_1)}{(b * SAS_1) \textbf{ sat } \mathcal{M}_c(P_1^{\oplus} \sqcap NULL)}$$

SAT-14 $$\frac{SAS_1 \textbf{ sat } \mathcal{M}_c(P_1) \quad SAS_1 \textbf{ sat } \psi}{SAS_1 \textbf{ sat } \mathcal{M}_c(\psi :: P_1)}$$

SAT-15 $$\frac{SAS_1 \textbf{ sat } \mathcal{M}_c(P_1)}{(t : SAS_1) \textbf{ sat } \mathcal{M}_c(t : P_1)}$$

SAT-16 $$\frac{SAS_1 \textbf{ sat } \mathcal{M}_c(P_1) \qquad SAS_2 \textbf{ sat } \mathcal{M}_c(P_2)}{(b?SAS_1 : SAS_2) \textbf{ sat } \mathcal{M}_c(b?(NULL \frown P_1) : (NULL \frown P_2))}$$

SAT-17 $$\frac{SAS_1 \textbf{ sat } \mathcal{M}_c(P_1) \qquad SAS_2 \textbf{ sat } \mathcal{M}_c(P_2) \qquad SAS_3 \textbf{ sat } \mathcal{M}_c(P_3)}{(SAS_1 \rhd_b^t SAS_2 : SAS_3) \textbf{ sat } \mathcal{M}_c(P_1 \rhd_b^t P_2 : P_3)}$$

The proofs of these rules are straightforward using the rules presented in the previous section. We give below the proofs of some of them.

Proof.

- Rule **SAT-11**

  1. SAS_1 **sat** $\mathcal{M}_c(P_1)$ {assumption}
  2. SAS_2 **sat** $\mathcal{M}_c(P_2)$ {assumption}
  3. $SAS_1; SAS_2$ **sat** $\mathcal{M}_c(P_1); skip; \mathcal{M}_c(P_2)$ {1., 2. and **SAT-4**}
  4. $SAS_1; SAS_2$ **sat** $\mathcal{M}_c(P_1 \frown P_2)$ {3. and Definition 5.1}

- Rule **SAT-16**

1. SAS_1 **sat** $\mathcal{M}_c(P_1)$ {assumption}
2. SAS_2 **sat** $\mathcal{M}_c(P_2)$ {assumption}
3. $b?SAS_1 : SAS_2$ **sat** $(b \wedge \bigcirc \mathcal{M}_c(P_1)) \vee (\neg b \wedge \bigcirc \mathcal{M}_c(P_2))$ {1., 2., **SAT-7**}
4. $b?SAS_1 : SAS_2$ **sat** $((b \wedge skip; \mathcal{M}_c(P_1))$
   $\vee (\neg b \wedge skip; \mathcal{M}_c(P_2)))$ {3. and ITL}
5. $b?SAS_1 : SAS_2$ **sat** $((b \wedge \mathcal{M}_c(NULL); skip; \mathcal{M}_c(P_1))$
   $\vee (\neg b \wedge \mathcal{M}_c(NULL); skip; \mathcal{M}_c(P_2)))$ {Definition 5.1}
6. $b?SAS_1 : SAS_2$ **sat** $((b \wedge \mathcal{M}_c(NULL^\frown P_1))$
   $\vee (\neg b \wedge \mathcal{M}_c(NULL^\frown P_2)))$ {Definition 5.1}
7. $b?SAS_1 : SAS_2$ **sat** $\mathcal{M}_c(b?(NULL^\frown P_1) : (NULL^\frown P_2))$ {Definition 5.1}

- Rule **SAT-17**

1. SAS_1 **sat** $\mathcal{M}_c(P_1)$ {assumption}
2. SAS_2 **sat** $\mathcal{M}_c(P_2)$ {assumption}
3. SAS_3 **sat** $\mathcal{M}_c(P_3)$ {assumption}
4. $SAS_1 \rhd_b^t SAS_2 : SAS_3$ **sat**
   $(((\mathcal{M}_c(P_1) \wedge\ keep\ \neg b \wedge len \leq t \wedge fin\ b); skip; \mathcal{M}_c(P_2))$
   $\vee ((\mathcal{M}_c(P_1) \wedge\ \Box \neg b \wedge len = t); skip; \mathcal{M}_c(P_3)))$ {1., 2., 3., **SAT-9**}
5. $(((\mathcal{M}_c(P_1) \wedge\ keep\ \neg b \wedge len \leq t \wedge fin\ b); skip; \mathcal{M}_c(P_2))$
   $\vee ((\mathcal{M}_c(P_1) \wedge\ \Box \neg b \wedge len = t); skip; \mathcal{M}_c(P_3)))$
   $\supset \mathcal{M}_c(P_1 \rhd_b^t P_2 : P_3)$ {Definition 5.1}
5. $SAS_1 \rhd_b^t SAS_2 : SAS_3$ **sat** $\mathcal{M}_c(P_1 \rhd_b^t P_2 : P_3)$ {4., 5., **SAT-1**}

6.4 Summary

We have presented in this chapter an approach for developing secure systems. Starting from an abstract specification a system is refined into a secure action system using sound compositional rules. A mechanism for enforcing dynamically changing security policies is developed, based on the composition of secure action systems. The enforcement of dynamically changing policies requires the introduction of specific agents whose role is to detect the moment a change must take place and to take appropriate actions then. For instance, the agent that enforces a policy $P_1 \frown P_2$ detects the moment when the policy P_1 terminates and then loads the policy P_2. Compositional rules are devised to reason about the functional, temporal and security properties of a system in a uniform manner. In the next chapter, we illustrate the approach with a simple, yet interesting example of a secure exam system.

Chapter 7

Case Studies

Objectives

- To illustrate the specification of simple policies.
- To illustrate the composition of policies.
- To illustrate the composition of secure action systems.
- To illustrate the enforcement of dynamically changing security policies.
- To illustrate the compositional verification of secure action systems.

7.1 Introduction

In this chapter we evaluate our development technique of secure systems with two case studies. The first case study illustrates the use of our policy language to specify complex security policies such as that proposed by Anderson [5] for health care systems. Ander-

son's security model is seen by many authors [63, 16, 4] as a separate security model that combines confidentiality and integrity policies to protect patient privacy and record integrity. We give a formal specification of the Anderson's security model in our policy language. This specification will be animated and analysed in the next chapter using our policy analysis tool SPAT. The second case study illustrates the development technique from a high level specification of a secure system down to a concrete implementation as a secure action system. It also illustrates the specification and composition of dynamically changing security policies and their enforcement mechanism in the secure action system paradigm. The case study uses a simple yet interesting example of a secure exam system.

7.2 Case Study 1: A Secure Health Care System

Health care systems store sensitive information about their patients' health. This information used to be kept on paper and access to it controlled by the health care manager who is responsible for enforcing the security policy. With the advances of the technology, the move to Electronic Patient Records (EPRs) is increasingly accepted because it provides several advantages. Among those are the possibility to improve health care through timely access to information and decision-support aids; information sharing between health care personnel within organisations and across organisations boundaries; and better support of clinical research, to name a few. However, such a technology will be practical only if enough evidence can be provided that the security of personal health information is guaranteed. Indeed, patients may avoid getting needed health care if they believe their personal health information may be improperly disclosed.

In this section, we give a formal specification of the security policy elaborated by the British Medical Association (BMA) [5] and also known as the Anderson security model.

7.2.1 Information Security in Health care Systems

Medical records contain a great deal of information about patients, such as HIV status, blood pressures, psychiatric care and so forth. This information is meant to be confidential because the consequence of unauthorised disclosure of information may seriously affect a patient's health, social standing, and employment prospects. Once sensitive information about an individual is exposed and the resulting damage done to that person, the information cannot be withdrawn and made secret again. However, this information is only useful to the patients when it is shared with the medical providers and health care organisations from which they get care. Indeed, physicians need and expect access to the complete medical records in order to help diagnose diseases correctly, to avoid duplicative risky or expensive tests, and to design effective treatment plans that take into account many complicating factors. The desirable sharing goes beyond personal care and includes its relationships to the society as a whole through support of medical research, public health management, and law enforcement.

Guaranteeing the confidentiality of personal health information while allowing information sharing with different health care institutions make the information security in health care subtle. While some users such as doctors can be allowed access to the entire information contained in an EPR, others such as insurers, billing services or law enforcement authorities have a restricted access. Patients generally can read their own EPRs and

request corrections if they identify errors and mistakes. General practitioners must seek patients' consent for any use and sharing of their personal health information. Although this rule is overridden in emergency situations, the patient must be notified of any use and disclosure of information. Patients can also request that the use or disclosure of his personal health information is restricted.

7.2.2 BMA Security Policy

Anderson [5] proposed a policy model based on nine principles for the BMA security policy for health care systems in the United Kingdom. These principles are described below. To ease the presentation and to conform with Anderson's model, we will assume that a clinician is female and a patient male.

Principle 1 *Each EPR shall be marked with an access control list naming the people or groups of people who may read it and append data to it. The system shall prevent anyone not on the access control list from accessing the record in any way.*

Principle 2 *A Clinician may open an EPR with herself and the patient on the access control list. Where a patient has been referred, she may open a record with herself, the patient and the referring clinician(s) on the access control list.*

Principle 3 *One of the clinicians on the access control list must be marked as being responsible. Only she may alter the access control list, and only she may add other health care professionals to it.*

Principle 4 *The responsible clinician must notify the patient of the names on his record's access control list when it is opened, of all subsequent additions, and whenever responsibility is transferred. His consent must also be obtained, except in emergency or in statutory exemptions.*

Principle 5 *No one shall have the ability to delete clinical information until the appropriate time period has expired.*

Principle 6 *All accesses to clinical records shall be marked on the record with the subject's name, as well as the date and time. An audit trail must also be kept of all deletions.*

Principle 7 *Information derived from record A may be appended to record B if and only if B's access control list is contained in A's.*

Principle 8 *There shall be effective measures to prevent the aggregation of personal health information. In particular, patients must receive special notification if any person whom it is proposed to add to their access control list already has access to personal health information on a large number of people.*

Principle 9 *Computer systems that handle personal health information shall have a subsystem that enforces the above principles in an effective way. Its effectiveness shall be subject to evaluation by independent experts.*

7.2.3 Formalising the BMA Security Policy

In this section, we give the formal specification of the BMA security policy in our policy language presented in Chapter 5. We assume a finite set $\mathcal{S}$ of clinicians and patients, and

a finite set $\mathcal{O}$ of patients' EPRs. At this level of abstraction, the concrete structure of an EPR is not relevant. We use a predicate $clinician(x)$ to mean that $x \in \mathcal{S}$ assumes the role *clinician*. The elements of the set $\mathcal{A}$ of actions are defined in the sequel.

We consider for each EPR $i \in \mathcal{O}$ the following state variables:

- $Owner(i)$: denotes the patient owner of the EPR i.
- $Responsible(i)$: denotes the responsible clinician of the record. By default the clinician who creates the record assumes this role.
- $Referring(i)$: denotes the referring clinician if the owner of the record has been referred.
- $ExpTime(i)$: denotes the expiry time of the record.
- $Info(i)$: this attribute stores medical information.
- $Status(i)$: is equal -1 if the record is deleted, 0 if the record is not yet opened and 1 if it is opened.
- $Trail(i)$: stores all accesses to the record with the subject's name and the time.
- $Acl(i)$: denotes the access control list of the record. It is a set of entries of the form (s, a), where s is a subject and a an action.
- $Notif(i)$: denotes the set of notifications addressed to the owner of the record.
- $Consent(i)$: denotes the set of the record's owner consents.

- $Access(i, j, a)$: denotes a predicate which holds in the final state of an execution of action $a \in \mathcal{A}$ when the action is performed by subject $i \in \mathcal{S}$ on object $j \in \mathcal{O}$.

In the sequel we will use positive authorisations to express permissions and negative authorisations to express denials. In the event of conflicts, we assume that denials take precedence. So the conflict resolution rule is

(7.1) $$\lceil autho^{+}(X, Y, Z) \wedge \neg autho^{-}(X, Y, Z) \rceil^{0} \mapsto autho(X, Y, Z).$$

7.2.3.1 Principle 1

The principle 1 says that an EPR's access control list names the people or group of people who may read it or append data to it. This authorisation requirement is formalised as

(7.2) $$\lceil (X, read) \in Acl(Y) \rceil^{0} \mapsto autho^{+}(X, Y, read).$$

(7.3) $$\lceil (X, append) \in Acl(Y) \rceil^{0} \mapsto autho^{+}(X, Y, append).$$

The requirement that

> *the system shall prevent anyone not authorised from accessing the record in any way*

is formulated as

(7.4) $$\bigwedge_{i \in \mathcal{S}} \bigwedge_{j \in \mathcal{O}} \bigwedge_{a \in \mathcal{A}} \neg \Diamond (\neg autho(i, j, a) \wedge \bigcirc Access(i, j, a)),$$

i.e. it is never the case that a subject performs an action on an object when it is not authorised to do so (we assume that actions are atomic, therefore each action execution corresponds to a system transition).

7.2.3.2 Principle 2

A record is created by setting its status to 1. The following safety requirement says that a record cannot be recreated:

$$\bigwedge_{i \in \mathcal{O}} \neg \Diamond (Status(i) = 1 \wedge \Diamond Status(i) = 0), \tag{7.5}$$

i.e. it is never the case that a record's status changes from 1 (created) to 0 (noncreated). Similarly, a record cannot be deleted if it has not been created, i.e.

$$\bigwedge_{i \in \mathcal{O}} \neg \Diamond (Status(i) = 0 \wedge \bigcirc Status(i) = -1). \tag{7.6}$$

We assume that a deleted record cannot become available, i.e.

$$\bigwedge_{i \in \mathcal{O}} \neg \Diamond (Status(i) = -1 \wedge \bigcirc Status(i) \neq -1). \tag{7.7}$$

Clinicians are allowed to create EPRs, i.e.

$$\lceil Status(Y) = 0 \wedge clinician(X) \rceil^0 \mapsto autho^+(X, Y, create). \tag{7.8}$$

The clinician who creates an EPR for a patient is the *responsible clinician* of the record and the patient is the *owner*. She is allowed to read and to append to the record while the patient can only read it. This requirement is specified as follows:

$$\lceil X = Responsible(Y) \rceil^0 \mapsto autho^+(X, Y, append). \tag{7.9}$$

$$\lceil X = Responsible(Y) \vee X = Owner(Y) \rceil^0 \mapsto autho^+(X, Y, read). \tag{7.10}$$

Where a patient has been referred, the referring clinician can also access the patient record. This is specified by the following rules:

(7.11) $$\lceil X = Referring(Y) \rceil^0 \mapsto autho^+(X, Y, read).$$

(7.12) $$\lceil X = Referring(Y) \rceil^0 \mapsto autho^+(X, Y, append).$$

7.2.3.3 Principle 3

The principle 3 says that only the responsible clinician of an EPR may alter record's access control list by adding or removing entries from it. Such accesses are denied to anyone else. This requirement is formulated as follows, where add is an action for adding an entry to an access control list, $remove$ is an action for removing an entry from an access control list, and $transfer$ is an action for transferring responsibility.

(7.13) $$\lceil X = Responsible(Y) \rceil^0 \mapsto autho^+(X, Y, add).$$

(7.14) $$\lceil X = Responsible(Y) \rceil^0 \mapsto autho^+(X, Y, remove).$$

(7.15) $$\lceil X = Responsible(Y) \rceil^0 \mapsto autho^+(X, Y, transfer).$$

(7.16) $$\lceil X \neq Responsible(Y) \rceil^0 \mapsto autho^-(X, Y, add).$$

(7.17) $$\lceil X \neq Responsible(Y) \rceil^0 \mapsto autho^-(X, Y, remove).$$

(7.18) $$\lceil X \neq Responsible(Y) \rceil^0 \mapsto autho^-(X, Y, transfer).$$

7.2.3.4 Principle 4

The principle 4 states obligations for patient notifications and obligations for acquiring patients' informed consents. We designate by a *pre-notification* a notification that must take place before the notified action is performed. A *post-notification* takes place after the action is performed. The enforcement of these obligations requires the following authorisations:

1. The responsible clinician of an EPR should be authorised to notify the record's owner, viz

   (7.19) $$\lceil X = Responsible(Y) \rceil^0 \mapsto autho^+(X, Y, notify).$$

2. The owner of an EPR should be authorised to give and withdraw consents, viz

   (7.20) $$\lceil X = Owner(Y) \rceil^0 \mapsto autho^+(X, Y, giveConsent).$$

   (7.21) $$\lceil X = Owner(Y) \rceil^0 \mapsto autho^+(X, Y, withdrawConsent).$$

In the sequel, we consider for each patient i a Boolean state variable $Emergency(i)$ which indicates whether the patient is in an emergency situation (or in a statutory exemption) or not.

Patient Post-Notifications The responsible clinician must notify the patient of the names on his record's access control list when it is opened, viz.

$$\bigwedge_{i\in\mathcal{O}} \Box \left(\begin{array}{c} (Status(i) = 0 \wedge \bigcirc Status(i) = 1) \\ \supset \\ \bigwedge_{j\in\mathcal{S}} \bigwedge_{a\in\mathcal{A}} ((j,a) \in Acl(i) \supset (add, j, a) \in \bigcirc Notif(i)) \end{array} \right). \tag{7.22}$$

A record is opened if its status changes from 0 to 1. We assume that the owner and the responsible clinician of a record are determined at the opening of the record. Here the notification message (add, j, a) tells the owner of the record i that the subject j is added to the record's ACL with the action a. The formula (7.22) says that the owner is notified of the names in the record's ACL.

The principle 4 also requires the responsible clinician of a record to notify the patient of all subsequent additions in the record's access control list, viz.

$$\bigwedge_{i\in\mathcal{O}} \bigwedge_{j\in\mathcal{S}} \bigwedge_{a\in\mathcal{A}} \Box(((j,a) \notin Acl(i) \wedge (j,a) \in \bigcirc Acl(i)) \supset (add, j, a) \in \bigcirc Notif(i)). \tag{7.23}$$

Similarly to the addition of an entry to a record's access control list, the deletion of an entry from the list is specified as follows:

(7.24)

$$\bigwedge_{i\in\mathcal{O}} \bigwedge_{j\in\mathcal{S}} \bigwedge_{a\in\mathcal{A}} \Box(((j,a) \in Acl(i) \wedge (j,a) \notin \bigcirc Acl(i)) \supset (remove, j, a) \in \bigcirc Notif(i)).$$

Furthermore, a patient must be notified whenever responsibility is transferred, viz.

$$\bigwedge_{i\in\mathcal{O}}\ \bigwedge_{j\in\mathcal{S}}\ \Box\left(\begin{array}{c}(Responsible(i)\neq j\wedge j=\bigcirc Responsible(i))\\ \supset\\ (transferResp,j)\in\bigcirc Notif(i)\end{array}\right). \tag{7.25}$$

The notification message $(transferResp, j)$ means that responsibility is transferred to the subject j.

Patient's Consents Unlike the notification requirement, the patient's consent must be obtained before a name is added to or remove from his record's access control list and before a new responsible clinician is assigned to a patient's record. However, the patient's consent is not required in emergency situations or statutory exemptions. We assume that a consent has one of the following forms:

- the form $(transferResp, k)$ stands for the patient's consent to the transfer of responsibility to the subject k;
- the form (add, k, a) means that the patient consents to the addition of the subject k to an EPR's ACL with the action a;
- the form $(remove, k, a)$ is similar and means the patient consents to the removal of the subject k with the action a from an EPR's ACL.

So Principle 4 is formalised as follows:

- it is never the case that a subject is added to a record's ACL without the informed

consent of the record's owner, unless in emergency situation, i.e.

$$\bigwedge_{i\in\mathcal{O}}\ \bigwedge_{j,k\in\mathcal{S}}\ \bigwedge_{a\in\mathcal{A}} \neg\Diamond \left(\begin{array}{c} \left(\begin{array}{c} Owner(i) = k\ \wedge\ \neg Emergency(k) \wedge \\ (j,a) \notin Acl(i)\ \wedge\ (j,a) \in \bigcirc Acl(i) \end{array} \right) \\ \wedge \\ (add, j, a) \notin Consent(i) \end{array} \right), \tag{7.26}$$

- it is never the case that a subject is removed from a record's ACL without the informed consent of the record's owner, unless in emergency situation, i.e.

$$\bigwedge_{i\in\mathcal{O}}\ \bigwedge_{j,k\in\mathcal{S}}\ \bigwedge_{a\in\mathcal{A}} \neg\Diamond \left(\begin{array}{c} \left(\begin{array}{c} Owner(i) = k\ \wedge\ \neg Emergency(k) \wedge \\ (j,a) \in Acl(i)\ \wedge\ (j,a) \notin \bigcirc Acl(i) \end{array} \right) \\ \wedge \\ (remove, j, a) \notin Consent(i) \end{array} \right), \tag{7.27}$$

- it is never the case that responsibility for a record is transferred without the informed consent of the record's owner, unless in emergency situation, i.e.

$$\bigwedge_{i\in\mathcal{O}}\ \bigwedge_{j,k\in\mathcal{S}} \neg\Diamond \left(\begin{array}{c} \left(\begin{array}{c} Owner(i) = k\ \wedge\ \neg Emergency(k) \wedge \\ Responsible(i) \neq j \wedge j = \bigcirc Responsible(i) \end{array} \right) \\ \wedge \\ (transferResp, j) \notin Consent(i) \end{array} \right). \tag{7.28}$$

7.2.3.5 Principle 5

We assume there is a state variable $Time$ that determines the global time. The behaviour of this variable is specified by:

$$Time = 0 \wedge\ keep((\bigcirc Time) = Time + 1). \tag{7.29}$$

That is the variable $Time$ is initialised to zero and increases by one at every transition of the system. The variable $Time$ is used in the sequel to refer to the global time.

Principle 5 says that no one has the right to delete a record before its expiry time, i.e

$$\lceil Time < ExpTime(Y) \rceil^0 \mapsto autho^-(X, Y, delete). \tag{7.30}$$

When a record is expired, the responsible clinician is allowed to delete the record, i.e

$$\lceil Time \geq ExpTime(Y) \wedge X = Responsible(Y) \rceil^0 \mapsto autho^+(X, Y, delete). \tag{7.31}$$

No one is allowed to access a deleted record, i.e.

$$\lceil Status(Y) = -1 \rceil^0 \mapsto autho^-(X, Y, Z). \tag{7.32}$$

7.2.3.6 Principle 6

Keeping an audit trail of all accesses to sensitive data is an effective mechanism to protect these data against malicious users (namely insiders) as it helps to establish accountabilities in the event of a security breach. In this respect, the principle 6 states the obligations of recording all accesses to an EPR, with the subject's name and the time of access. This is specified as follows:

- every read access is recorded, viz.

$$\bigwedge_{i \in \mathcal{S}} \bigwedge_{j \in \mathcal{O}} \Box(Access(i, j, read) \supset (i, read, Time) \in Trail(j)), \tag{7.33}$$

- every append access is recorded, viz.

$$\bigwedge_{i \in \mathcal{S}} \bigwedge_{j \in \mathcal{O}} \Box(Access(i, j, append) \supset (i, append, Time) \in Trail(j)), \tag{7.34}$$

- every delete access is recorded, viz.

$$\bigwedge_{i\in\mathcal{S}}\bigwedge_{j\in\mathcal{O}}\Box(Access(i,j,delete)\supset(i,delete,Time)\in Trail(j)), \tag{7.35}$$

7.2.3.7 Principle 7

For each subject $i\in\mathcal{S}$, we use a state variable $Source(i)$ to store the object the subject i last read information from. The value of that state variable is determined by the following formula:

$$\bigwedge_{i\in\mathcal{S}}\bigwedge_{j\in\mathcal{O}}\Box(\bigcirc Access(i,j,read)\supset(\bigcirc Source(i))=j). \tag{7.36}$$

The requirement that information derived from record j may be appended to record k if and only if k's access control list is contained in j's is specified as follows:

$$\bigwedge_{i\in\mathcal{S}}\bigwedge_{k\in\mathcal{O}}\Box\left(\begin{array}{c}\bigcirc Access(i,k,append)\\ \supset\\ \bigwedge_{j\in\mathcal{S}}\bigwedge_{a\in\mathcal{A}}(autho(j,k,a)\supset autho(j,Source(i),a))\end{array}\right). \tag{7.37}$$

7.2.3.8 Principle 8

In order to formalise the *pre-notification* requirement specified in Principle 8 (by contrast to the *post-notification* requirement specified in Principle 4), we consider for each subject j, a state variable $Score(j)$ that stores the number of accesses to EPRs the subject currently has. This variable increases by 1 each time the subject is added to an EPR's ACL, and decreases by 1 when the subject is removed from an EPR's ACL. Let, for $j\in\mathcal{S}$,

$$isAdded(j)\mathrel{\widehat{=}}\bigvee_{i\in\mathcal{O}}\bigvee_{a\in\mathcal{A}}((j,a)\notin Acl(i)\wedge\ (j,a)\in\bigcirc Acl(i))$$

and

$$isRemoved(j) \mathrel{\widehat{=}} \bigvee_{i \in \mathcal{O}} \bigvee_{a \in \mathcal{A}} ((j,a) \in Acl(i) \wedge \ (j,a) \notin \bigcirc Acl(i)).$$

The formula $isAdded(j)$ holds for an interval if subject j is added to an EPR's ACL at the beginning of the interval. By contrast, the formula $isRemoved(j)$ holds for an interval if subject j is removed from an EPR's ACL at the beginning of the interval. The computation of subjects' scores is specify as follows:

- Initially, the number of EPRs a subject can currently access is zero because no record is opened yet, viz.

  (7.38) $$\bigwedge_{j \in \mathcal{S}} (Score(j) = 0).$$

- Whenever a subject j is added to an EPR's ACL, the subject's score increases by 1, i.e.

  (7.39) $$\bigwedge_{j \in \mathcal{S}} \Box(isAdded(j) \supset (\bigcirc Score(j)) = Score(j) + 1).$$

- Whenever a subject j is removed from an EPR's ACL, the subject's score decreases by 1, i.e.

  (7.40) $$\bigwedge_{j \in \mathcal{S}} \Box(isRemoved(j) \supset (\bigcirc Score(j)) = Score(j) - 1).$$

- If a subject j is not added or is not removed from an EPR's ACL, its score does not change, i.e.

  (7.41) $$\bigwedge_{j \in \mathcal{S}} \Box \left(\begin{array}{c} (\neg isAdded(j) \wedge \neg isRemoved(j)) \\ \supset \\ (\bigcirc Score(j)) = Score(j) \end{array} \right).$$

The following formula formalises the requirement that *patients must receive special notification if any person whom it is proposed to add to their access control list already has access to personal health information on a large number of people.*

$$\bigwedge_{i\in\mathcal{O}}\ \bigwedge_{j\in\mathcal{S}}\ \bigwedge_{a\in\mathcal{A}}\ \Box\left(\begin{array}{c} isAdded(j) \\ \supset \\ (add, j, a, Score(j)) \in Notif(i) \end{array}\right). \tag{7.42}$$

This formula states that the owner of a record i must be notified of the score of any subject j before the subject is added to i's ACL. Of course from (7.26), the subject is added to the record's ACL if the patient consent to it or there is an emergency.

7.2.3.9 Principle 9

A rationale for using formal techniques to develop a system is to increase the dependability (including the trustworthiness) of the final product. Our framework has got a sound foundation based on ITL. The policy specification language and the computational model that implements the security enforcement mechanism are given both a formal semantics in ITL. This enables formal reasoning about the design using ITL assistant tools such as Anatempura for run-time verification, SPAT for security policy analysis and the theorem prover ITL-PVS. In this respect, we believe that Principle 9 can be fulfilled by our approach with a suitable use of the accompanying tools.

Finally, a formal specification of the Anderson model in our policy language is denoted by $C :: P$, where $C \mathrel{\widehat{=}} (7.4)\wedge\cdots\wedge(7.7)\wedge(7.22)\wedge\cdots\wedge(7.29)\wedge(7.33)\wedge\cdots\wedge(7.42)$ specifies the obligation policies and $P \mathrel{\widehat{=}} (7.1)\cup\cdots\cup(7.3)\cup(7.8)\cup\cdots\cup(7.21)\cup(7.30)\cup\cdots\cup(7.32)$ denotes the authorisation policies. This policy is animated and analysed in

the following chapter.

7.3 Case Study 2: A Secure Exam System

We consider an exam system in which each exam is assigned an examiner, a moderator and an external examiner. The examiner prepares the exam questions which are reviewed by the moderator and the external examiner. During this process, security measures should be taken to ensure the secrecy and the integrity of the exam. In particular, the exam questions cannot be disclosed to students before the time the exam takes place.

The examiner writes the first draft of the exam questions and submits it to the moderator. The moderator reads the exam questions and comments them. This must be done within 10 days from the submission date. Till the moderator finishes his review, no change can be done on the exam. On the receipt of the moderator comments, the examiner revises the exam questions to include them. However, the examiner cannot change the moderator comments, neither can the external examiner. The revised version is then submitted to the external examiner for appraisal. The external examiner reviews the exam questions and releases his comments within 1 week. Based on these comments, the examiner produces the final version of the exam and submits it.

The final release of the exam questions as well as the moderator's comments and the external examiner's comments are kept securely. No one can access them until the time the exam takes place. For the sake of simplicity, we assume the exam takes place within 30 days from the time the final version is submitted. During the exam period, students who attend the exam can read it. So can the examiner, the moderator and the external

examiner. No write access is allowed.

7.3.1 Formalisation

We assume a finite set $\mathcal{S}$ of subjects s and a finite set $\mathcal{O}$ of exams o. Each subject has a role: student, examiner, moderator or external examiner. We write $isStudent(s)$ to mean that the subject s is a student. We also write $isExaminer(o, s)$, $isModerator(o, s)$ or $isExternal(o, s)$ to mean that the subject s is the examiner, the moderator or the external examiner of the exam o, respectively. The set $\mathcal{A}$ of actions is formed by the following actions with obvious meanings:

- $submit$: submit an exam.
- $readExam$: read the content of an exam.
- $writeExam$: modify the content of an exeam.
- $readModCmt$: read an exam's moderator comments.
- $writeModCmt$: modify an exam's moderator comments.
- $readExtCmt$: read an exam's external examiner comments.
- $writeExtCmt$: modify an exam's external examiner comments.

Based on these sets of subjects, objects and actions, we can now formulate in our policy language the security policy of the exam system. We use positive authorisation rules to state explicit permissions, and negative authorisation rules to state explicit denials.

For example, the requirement that examiners can write their exams is formalised by the positive authorisation rule:

$$\lceil isExaminer(X,Y)\rceil^0 \mapsto autho^+(Y,X,writeExam).$$

Similarly, the following negative authorisation rule denies students write access to exams:

$$\lceil isStudent(Y)\rceil^0 \mapsto autho^-(Y,X,writeExam).$$

If an examiner is also a student, the above two rules generate conflicting authorisations. We resolve conflict by giving precedence to denials using the conflict resolution rule:

$$\lceil autho^+(X,Y,Z) \wedge \neg autho^-(X,Y,Z)\rceil^0 \mapsto autho(X,Y,Z).$$

So an examiner who is at same time a student will be merely denied write access to exams.

We define the following simple policies in which denials take precedence in the event of conflicts.

$$P_1 \mathrel{\widehat{=}} \bigcup \left\{ \begin{array}{l} \lceil isExaminer(X,Y)\rceil^0 \mapsto autho^+(Y,X,readExam), \\ \lceil isExaminer(X,Y)\rceil^0 \mapsto autho^+(Y,X,writeExam), \\ \lceil isExaminer(X,Y)\rceil^0 \mapsto autho^+(Y,X,submit), \\ \lceil isExaminer(X,Y)\rceil^0 \mapsto autho^+(Y,X,readModCmt), \\ \lceil isExaminer(X,Y)\rceil^0 \mapsto autho^+(Y,X,readExtCmt), \\ true \mapsto autho^-(Y,X,writeModCmt), \\ true \mapsto autho^-(Y,X,writeExtCmt), \\ \lceil \neg isExaminer(X,Y)\rceil^0 \mapsto autho^-(Y,X,writeExam), \\ \lceil isStudent(Y)\rceil^0 \mapsto autho^-(Y,X,Z), \\ \lceil autho^+(X,Y,Z) \wedge \neg autho^-(X,Y,Z)\rceil^0 \mapsto autho(X,Y,Z) \end{array} \right\}$$

The policy P_1 allows the examiner read and write access to the exam he is assigned to. It also gives him the right to submit the exam and to read the moderator's and the external examiner's comments. No one is allowed to modify these comments. The policy explicitly forbids the moderator and the external examiner write access to the exam. Students are explicitly denied all accesses to the exam as well as to the comments of the moderator and the external examiner. The policy P_1 is enforced during the preparation of the initial draft of the exam and the subsequent revisions.

$$P_2 \widehat{=} \bigcup \left\{ \begin{array}{l} \lceil isModerator(X,Y) \rceil^0 \mapsto autho^+(Y,X,readExam), \\ \lceil isModerator(X,Y) \rceil^0 \mapsto autho^+(Y,X,readModCmt), \\ \lceil isModerator(X,Y) \rceil^0 \mapsto autho^+(Y,X,writeModCmt), \\ true \mapsto autho^-(Y,X,writeExam), \\ true \mapsto autho^-(Y,X,submit), \\ true \mapsto autho^-(Y,X,writeExtCmt), \\ \lceil \neg isModerator(X,Y) \rceil^0 \mapsto autho^-(Y,X,writeModCmt), \\ \lceil isStudent(Y) \rceil^0 \mapsto autho^-(Y,X,Z), \\ \lceil autho^+(X,Y,Z) \wedge \neg autho^-(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z) \end{array} \right\}$$

During the moderator's review of the exam, a different policy P_2 applies. It gives the moderator the right to read the exam questions and the right to read and to write comments about it. No one else can modify the moderator comments. The policy explicitly forbids the examiner and the external examiner write access to the exam. No one is allowed to write the external examiner's comments about the exam. Again, students are explicitly denied all accesses.

$$P_3 \mathrel{\widehat{=}} \bigcup \left\{ \begin{array}{l} \lceil isExternal(X,Y) \rceil^0 \mapsto autho^+(Y,X,readExam), \\ \lceil isExternal(X,Y) \rceil^0 \mapsto autho^+(Y,X,readExtCmt), \\ \lceil isExternal(X,Y) \rceil^0 \mapsto autho^+(Y,X,writeExtCmt), \\ true \mapsto autho^-(Y,X,writeExam), \\ true \mapsto autho^-(Y,X,submit), \\ true \mapsto autho^-(Y,X,writeModCmt), \\ \lceil isExaminer(X,Y) \rceil^0 \mapsto autho^-(Y,X,writeExtCmt), \\ \lceil isModerator(X,Y) \rceil^0 \mapsto autho^-(Y,X,writeExtCmt), \\ \lceil isStudent(Y) \rceil^0 \mapsto autho^-(Y,X,Z), \\ \lceil autho^+(X,Y,Z) \wedge \neg autho^-(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z) \end{array} \right\}$$

A similar policy P_3 applies during the external examiner's review of the exam. The policy P_3 gives the external examiner the right to read the exam questions and the right to read and write comments about it. No one else can modify these comments. The policy explicitly forbids the examiner and the moderator write access to the exam. No one is allowed to modify the moderator comments. Still, students are explicitly denied all accesses.

$$P_4 \mathrel{\widehat{=}} \bigcup \left\{ \begin{array}{l} isExaminer(X,Y) \mapsto autho^+(Y,X,submit), \\ true \mapsto autho^-(Y,X,Z), \\ \lceil autho^+(X,Y,Z) \wedge \neg autho^-(X,Y,Z) \rceil^0 \mapsto autho(X,Y,Z) \end{array} \right\}$$

The policy P_4 is enforced from the time the final version of the exam is produced till the time the exam takes place. It forbids all accesses to the exam and to the comments about it.

$$P_5 \mathrel{\widehat{=}} \bigcup \left\{ \begin{array}{l} true \mapsto autho^{+}(Y, X, readExam), \\ true \mapsto autho^{-}(Y, X, writeExam), \\ true \mapsto autho^{-}(Y, X, submit), \\ true \mapsto autho^{+}(Y, X, readModCmt), \\ true \mapsto autho^{-}(Y, X, writeModCmt), \\ true \mapsto autho^{+}(Y, X, readExtCmt), \\ true \mapsto autho^{-}(Y, X, writeExtCmt), \\ \lceil isStudent(Y) \rceil^0 \mapsto autho^{-}(Y, X, readModCmt), \\ \lceil isStudent(Y) \rceil^0 \mapsto autho^{-}(Y, X, readExtCmt), \\ \lceil autho^{+}(X, Y, Z) \wedge \neg autho^{-}(X, Y, Z) \rceil^0 \mapsto autho(X, Y, Z) \end{array} \right\}$$

During the exam period and afterwards, the policy P_5 applies. Any one can read the exam yet no one can modify it. This is the only time students are allowed to read the exam. However, they are not allowed to read the moderator's and the external examiner's comments.

The security policy of the exam system can then be specified as

(7.43) $$(\langle done \rangle P_1)^\frown(10 : P_2)^\frown(\langle done \rangle P_1)^\frown(7 : P_3)^\frown(\langle done \rangle P_1)^\frown(30 : P_4)^\frown P_5.$$

where the event $done$ occurs when all the exams are submitted. In the following section, we develop a secure action system that enforces the policy (7.43).

7.3.2 Implementation of the Exam System

In this section, we develop a secure action system that enforces the security policy (7.43). Subjects and objects are implemented as agents. For the sake of simplicity, we assume that there are as many examiners, moderators, external examiners as there are exams. Let m be the total number of exams and n be the total number of students. We let $\mathcal{P} \mathrel{\widehat{=}} \{exam[i], examiner[i], moderator[i], external[i], student[j] \mid i = 1, \ldots, m \wedge j = 1, \ldots, n\}$ be the set of the agents. We assign each exam i, an examiner, a moderator and an external examiner of the same index. So the predicates $isStudent$, $isExaminer$, $isModerator$ and $isExternal$ used in the previous section are implemented as follows:

- $isStudent(student[i]) \mathrel{\widehat{=}} true$, for $i = 1, \ldots, n$ and it is false elsewhere.
- $isExaminer(exam[i], examiner[j]) \mathrel{\widehat{=}} (i = j)$, for $i, j = 1, \ldots, m$ and it is false elsewhere.
- $isModerator(exam[i], moderator[j]) \mathrel{\widehat{=}} (i = j)$, for $i, j = 1, \ldots, m$ and it is false elsewhere.
- $isExternal(exam[i], external[j]) \mathrel{\widehat{=}} (i = j)$, for $i, j = 1, \ldots, m$ and it is false elsewhere.

7.3.2.1 Specification of Agents

The exam agents are defined as follows:

$$\textbf{agent}\ \ exam[i = 1, \ldots, m] : \ \textbf{var}\quad submitted, questions, modCmts, extCmts;$$
$$submitted := false.$$

The local Boolean variable $submitted$ is true if the corresponding exam is submitted for review or as final version. Initially, it is set to false because the exam is still in preparation. The local variables $questions$, $modCmts$ and $extCmts$ store respectively the exam questions, the moderator comments and the external examiner comments.

The other agents are defined as follows, where the local variable $text$ is used for communication.

agent $examiner[i = 1, \ldots, m]$: **var** $text$.

agent $moderator[i = 1, \ldots, m]$: **var** $text$.

agent $external[i = 1, \ldots, m]$: **var** $text$.

agent $student[i = 1, \ldots, n]$: **var** $text$.

For example, an examiner can prepare locally the exam questions in the variable $text$ then copy them into the corresponding exam agent. A student can read the exam questions into the variable $text$ before treating them. For the simplicity of the presentation, we do not address how an examiner locally edits an exam questions or how a student takes an exam.

7.3.2.2 Specification of Actions

We use indexes to differentiate distinct instantiations of a same action $a \in \mathcal{A}$. For example $submit[i]$ correspond to the action $submit$ performed on $exam[i]$ by $examiner[i]$, for $i = 1, \ldots, m$. The action $submit$ is implemented as follows:

action $submit[i = 1, \ldots, m]$ **on** $exam[i]$ **by** $examiner[i]$:

$true \rightarrow submitted := true.$

An action $submit$ is a joint action involving an exam and its examiner. This enforces the security policy that only examiners can be allowed to submit their exams. Its execution sets the local variable $submitted$ of the participating exam to true and terminates.

By contrast, moderators, external examiners and students as well as examiners can be allowed read access to an exam. So the action $readExam$ is defined as follows:

action $readExam[i = 1, \ldots, m; 1]$ **on** $exam[i]$ **by** $examiner[i]$:

$true \rightarrow text := questions.$

action $readExam[i = m + 1, \ldots, 2m; 1]$ **on** $exam[i - m]$ **by** $moderator[i - m]$:

$true \rightarrow text := questions.$

action $readExam[i = 2m + 1, \ldots, 3m; 1]$ **on** $exam[i - 2m]$ **by** $external[i - 2m]$:

$true \rightarrow text := questions.$

action $readExam[i = 1, \ldots, m; j = 2, \ldots, n + 1]$ **on** $exam[i]$ **by** $student[j - 1]$:

$true \rightarrow text := questions.$

For $i = 1, \ldots, m$, the agents $examiner[i]$, $moderator[i]$, $external[i]$ participates respectively in the action $readExam[i, 1]$, $readExam[i + m, 1]$ and $readExam[i + 2m, 1]$, jointly with the agent $exam[i]$. Similarly, each agent $student[j]$, $1 \leq j \leq n$ participates in the actions $readExam[i, j + 1]$, jointly with the agent $exam[i]$, for $i = 1, \ldots, m$. The effect of the action in each case is to copy the exam questions into the local variable $text$. The functional guard of the actions is $true$ meaning that the actions can be performed at any time provided that the participating agents have the appropriate rights. These rights are specified by the security policy which we present in Section 7.3.2.3.

Only the examiner assigned to an exam can modify it. This is specified by defining a joint action $write[i]$ involving the agents $exam[i]$ and $examiner[i]$, for $i = 1, \ldots, m$.

action $writeExam[i = 1, \ldots, m]$ **on** $exam[i]$ **by** $examiner[i]$:

$true \rightarrow questions := text.$

The effect of the action is to copy the new version $text$ of the exam questions into the local variable $questions$ of the exam agent.

Accesses to the moderators' and the external examiners' comments are specified in a similar fashion.

action $readModCmt[i = 1, \ldots, m]$ **on** $exam[i]$ **by** $examiner[i]$:

$true \rightarrow text := modCmts.$

action $readModCmt[i = 1, \ldots, m]$ **on** $exam[i]$ **by** $moderator[i]$:

$true \rightarrow text := modCmts.$

action $writeModCmt[i = 1, \ldots, m]$ **on** $exam[i]$ **by** $moderator[i]$:

$true \rightarrow modCmts := text.$

action $readExtCmt[i = 1, \ldots, m]$ **on** $exam[i]$ **by** $examiner[i]$:

$true \rightarrow text := extCmts.$

action $readExtCmt[i = 1, \ldots, m]$ **on** $exam[i]$ **by** $external[i]$:

$true \rightarrow text := extCmts.$

action $writeExtCmt[i = 1, \ldots, m]$ **on** $exam[i]$ **by** $external[i]$:

$true \rightarrow extCmts := text.$

Moderators and external examiners solely can modify their comments. The examiner of an exam can read the comments attached to this exam.

7.3.2.3 Enforcement of the Security Policy

We now specify the security policy of the system. For each policy P_i, $i = 1, \ldots, 5$ defined in Section 7.3.1, we develop a secure action system SAS_i that enforces it using the rule **SAT-10** (see Chapter 6, page 148).

Let $\mathcal{A} \mathrel{\widehat{=}} \{submit[i], readExam[i], writeExam[i, j+1],$ $writeExam[k, 1], readModCmt[i], writeModCmt[i], readExtCmt[i],$ $writeExtCmt[i] \mid 1 \leq i \leq m \ \wedge \ 1 \leq j \leq n \ \wedge \ 1 \leq k \leq 2m\}$. Let $Agents$ (resp. $Actions$) be the set of the agent definitions (resp. action definitions) given above.

We consider the following secure action system normal form

$$SAS_1 \mathrel{\widehat{=}} (\mathcal{P}, \mathcal{A}, Agents, Actions, \{autho^+, autho^-, autho\})$$

where the functions $autho^+$, $autho^-$ and $autho$ are defined as follows:

1. $autho^+(examiner[i], exam[i], readExam[i, 1]) \mathrel{\widehat{=}} true, \ \forall i : 1 \leq i \leq m$;
2. $autho^+(examiner[i], exam[i], writeExam[i]) \mathrel{\widehat{=}} true, \ \forall i : 1 \leq i \leq m$;
3. $autho^+(examiner[i], exam[i], submit[i]) \mathrel{\widehat{=}} true, \ \forall i : 1 \leq i \leq m$;
4. $autho^+(examiner[i], exam[i], readModCmt[i]) \mathrel{\widehat{=}} true, \ \forall i : 1 \leq i \leq m$;
5. $autho^+(examiner[i], exam[i], readExtCmt[i]) \mathrel{\widehat{=}} true, \ \forall i : 1 \leq i \leq m$;
6. $autho^+(x, y, z) \mathrel{\widehat{=}} false$, elsewhere;
7. $autho^-(moderator[i], exam[i], writeModCmt[i]) \mathrel{\widehat{=}} true, \ \forall i : 1 \leq i \leq m$;
8. $autho^-(external[i], exam[i], writeExtCmt[i]) \mathrel{\widehat{=}} true, \ \forall i : 1 \leq i \leq m$;

9. $autho^{-}(student[i], x, y) \widehat{=} true,\ \forall x \in \mathcal{P},\ \forall y \in \mathcal{A},\ \forall i : 1 \leq i \leq n$;

10. $autho^{-}(x, y, z) \widehat{=} false$, elsewhere;

11. $autho(x, y, z) \widehat{=} autho^{+}(x, y, z)\ \wedge\ \neg autho^{-}(x, y, z),\ \forall x, y \in \mathcal{P},\ \forall z \in \mathcal{A}$.

This security policy says that examiners are explicitly allowed to read and write the exams they are assigned to. They can also submit the exam at any time. In the meantime, they are explicitly allowed to read the moderator's and the external examiner's comment of their exams. On the contrary, they are explicitly denied write access to them. Students are forbidden all accesses to the exams.

In a similar fashion the normal forms SAS_2, SAS_3, SAS_4 and SAS_5 are defined as follows:

- $SAS_2 \widehat{=} (\mathcal{P}, \mathcal{A}, Agents, Actions, \{autho^{+}, autho^{-}, autho\})$ where the functions $autho^{+}$, $autho^{-}$ and $autho$ are defined as follows:

  1. $autho^{+}(moderator[i], exam[i], readExam[i + m, 1]) \widehat{=} true,\ \forall i : 1 \leq i \leq m$;

  2. $autho^{+}(moderator[i], exam[i], readModCmt[i]) \widehat{=} true,\ \forall i : 1 \leq i \leq m$;

  3. $autho^{+}(moderator[i], exam[i], writeModCmt[i]) \widehat{=} true,\ \forall i : 1 \leq i \leq m$;

  4. $autho^{+}(x, y, z) \widehat{=} false$, elsewhere;

  5. $autho^{-}(examiner[i], exam[i], writeExam[i]) \widehat{=} true,\ \forall i : 1 \leq i \leq m$;

  6. $autho^{-}(student[i], x, y) \widehat{=} true,\ \forall x \in \mathcal{P},\ \forall y \in \mathcal{A},\ \forall i : 1 \leq i \leq n$;

  7. $autho^{-}(x, y, z) \widehat{=} false$, elsewhere;

8. $autho(x, y, z) \widehat{=} autho^{+}(x, y, z) \ \wedge \ \neg autho^{-}(x, y, z), \ \forall x, y \in \mathcal{P}, \ \forall z \in \mathcal{A}$.

This security policy explicitly allows each moderator read access to the exam he is assigned to. A moderator is also allowed to read and write comments about his exam. By contrast, examiners are explicitly denied write access to the exams. So the exam questions cannot be modified when this policy is enforced. Again, students are forbidden all accesses to the exams.

- $SAS_3 \widehat{=} (\mathcal{P}, \mathcal{A}, Agents, Actions, \{autho^{+}, autho^{-}, autho\})$ where the functions $autho^{+}$, $autho^{-}$ and $autho$ are defined as follows:

  1. $autho^{+}(external[i], exam[i], readExam[i + 2m, 1]) \widehat{=} true, \ \forall i : 1 \leq i \leq m$;
  2. $autho^{+}(external[i], exam[i], readExtCmt[i]) \widehat{=} true, \ \forall i : 1 \leq i \leq m$;
  3. $autho^{+}(external[i], exam[i], writeExtCmt[i]) \widehat{=} true, \ \forall i : 1 \leq i \leq m$;
  4. $autho^{+}(x, y, z) \widehat{=} false$, elsewhere;
  5. $autho^{-}(examiner[i], exam[i], writeExam[i]) \widehat{=} true, \ \forall i : 1 \leq i \leq m$;
  6. $autho^{-}(student[i], x, y) \widehat{=} true, \ \forall x \in \mathcal{P}, \ \forall y \in \mathcal{A}, \ \forall i : 1 \leq i \leq n$;
  7. $autho^{-}(x, y, z) \widehat{=} false$, elsewhere;
  8. $autho(x, y, z) \widehat{=} autho_1^{+}(x, y, z) \ \wedge \ \neg autho^{-}(x, y, z), \ \forall x, y \in \mathcal{P}, \ \forall z \in \mathcal{A}$.

  This security policy explicitly allows each external examiner read access to the exam assigned to him. An external examiner is also allowed to read and write comments about his exam. By contrast, examiners are explicitly denied write access to the exams. Again, students are forbidden all accesses to the exams.

- $SAS_4 \mathrel{\widehat{=}} (\mathcal{P}, \mathcal{A}, Agents, Actions, \{autho^+, autho^-, autho\})$ where the functions $autho^+$, $autho^-$ and $autho$ are defined as follows:

  1. $autho^+(examiner[i], exam[i], submit[i]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
  2. $autho^+(x, y, z) \mathrel{\widehat{=}} false$, elsewhere;
  3. $autho^-(x, y, z) \mathrel{\widehat{=}} true,\ \forall x, y \in \mathcal{P},\ \forall z \in \mathcal{A}$;
  4. $autho(x, y, z) \mathrel{\widehat{=}} autho^+(x, y, z)\ \wedge\ \neg autho^-(x, y, z),\ \forall x, y \in \mathcal{P},\ \forall z \in \mathcal{A}$.

  Moderators, external examiners and students are denied all accesses to the exams. The exams and the comments about them are kept securely as long as this policy applies. However, the examiners are allowed to perform the actions $submit$ on the exams they are assigned to. These are the only actions enabled in the system. If examiners were also denied the right to perform these actions, then the system would terminate immediately because no actions were enabled.

- $SAS_5 \mathrel{\widehat{=}} (\mathcal{P}, \mathcal{A}, Agents, Actions, \{autho^+, autho^-, autho\})$ where the functions $autho^+$, $autho^-$ and $autho$ are defined as follows:

  1. $autho^+(examiner[i], exam[i], readExam[i, 1]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
  2. $autho^+(moderator[i], exam[i], readExam[i + m, 1]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
  3. $autho^+(external[i], exam[i], readExam[i + 2m, 1]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
  4. $autho^+(student[j], exam[i], readExam[i, j + 1]) \mathrel{\widehat{=}} true,\ \forall i, j : 1 \leq i \leq m\ \wedge\ j = 1, \ldots, n$;

5. $autho^{+}(examiner[i], exam[i], readModCmt[i]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
6. $autho^{+}(moderator[i], exam[i], readModCmt[i]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
7. $autho^{+}(examiner[i], exam[i], readExtCmt[i]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
8. $autho^{+}(external[i], exam[i], readExtCmt[i]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
9. $autho^{+}(x, y, z) \mathrel{\widehat{=}} false$, elsewhere;
10. $autho^{-}(examiner[i], exam[i], writeExam[i]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
11. $autho^{-}(moderator[i], exam[i], writeModCmt[i]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
12. $autho^{-}(external[i], exam[i], writeExtCmt[i]) \mathrel{\widehat{=}} true,\ \forall i : 1 \leq i \leq m$;
13. $autho^{-}(x, y, z) \mathrel{\widehat{=}} false$, elsewhere;
14. $autho(x, y, z) \mathrel{\widehat{=}} autho^{+}(x, y, z)\ \wedge\ \neg autho^{-}(x, y, z),\ \forall x, y \in \mathcal{P},\ \forall z \in \mathcal{A}$.

The security policy of the system SAS_5 grants examiners, moderators, external examiners read access to their exams and associated comments. However, no one is allowed to modify them. Students are allowed to read exams, but cannot read the comments associated to them. Neither can they modify any of these data.

The exam system is then implemented by the following secure action system, where $done \mathrel{\widehat{=}} \wedge_{i=1}^{m} exam[i].submitted$. The condition $done$ holds if all the exams are submitted.

$$(\langle done \rangle SAS_1); (10 : SAS_2); (\langle done \rangle SAS_1); (7 : SAS_3); \\ (\langle done \rangle SAS_1); (30 : SAS_4); SAS_5. \tag{7.44}$$

That is, the system starts by behaving like SAS_1 until all the exams are set and submitted by their respective examiners. During this period the security policy that applies is the one enforced by the secure action system SAS_1, namely the policy P_1. Then the system

changes and behaves like SAS_2 for 10 days. A different security policy P_2 is enforced, allowing the moderators read access to exams and to write comments about them. The system continues its execution by starting a new execution of SAS_1. So the local variable $submitted$ of each exam will be reinitialised to $false$ (see the initialisation statements of agents) and each examiner will have the opportunity to work out a new version of the exam assigned to him, taking into account the moderator's comments.

The execution of SAS_1 will terminate when all the exam are resubmitted, this time for external examiners' appraisals. The system launches then the execution of SAS_3 whose policy P_3 allows external examiners to read the exams assigned to them and to write comments about them. This process will last for 7 days and the examiners will be allowed to access the exams and to make a final revision on them, incorporating the external examiners' concerns. This is done by executing SAS_1 once again. Upon submission of all the exams, the system behaves like SAS_4 whose policy P_4 prohibits every access, except *submit*, to the exams. This situation will hold for 30 days and thereafter, the system will launch SAS_5 whose security policy P_5 allows everybody, including students (for the first time), read access to the exams yet forbids write access.

7.3.3 Proof of Correctness

The following lemma says that each secure action system SAS_i specified above enforces the policy P_i, $1 \leq i \leq 5$, where $\mathcal{M}_c$ stands for the semantic function of policies (see Chapter 5, page 98) and $Comp$ for the completeness algorithm presented in Chapter 4 (see page 86).

Lemma 7.1 SAS_i **sat** $\mathcal{M}_c(P_i), \qquad \forall i : 1 \leq i \leq 5.$

Proof. The proof is similar for $i = 1, \ldots, 5$. Here we give the proof for $i = 1$.

1- $\mathcal{M}_c(Comp(P_1)) \supset \mathcal{M}_c(P_1)$	Theorem 4.1
2- SAS_1 **sat** $\mathcal{M}_c(Comp(P_1))$	**SAT-10**
3- SAS_1 **sat** $\mathcal{M}_c(P_1)$	1, 2 and **SAT-1**

Theorem 7.1 establishes the correctness of the design.

Theorem 7.1 (7.44) **sat** $\mathcal{M}_c((7.43))$.

Proof.

1- SAS_i **sat** $\mathcal{M}_c(P_i), \qquad i = 1, \ldots, 5$	Lemma 7.1
2- $(\langle done \rangle SAS_1)$ **sat** $\mathcal{M}_c(\langle done \rangle P_1)$	1 and **SAT-12**
3- $(10 : SAS_2)$ **sat** $\mathcal{M}_c(10 : P_2)$	1 and **SAT-15**
4- $(7 : SAS_3)$ **sat** $\mathcal{M}_c(7 : P_3)$	1 and **SAT-15**
5- $(30 : SAS_4)$ **sat** $\mathcal{M}_c(30 : P_4)$	1 and **SAT-15**
6- $(\langle done \rangle SAS_1); (10 : SAS_2)$ **sat** $\mathcal{M}_c((\langle done \rangle P_1)^\frown(10 : P_2))$	2, 3 and **SAT-11**
7- $(\langle done \rangle SAS_1); (7 : SAS_3)$ **sat** $\mathcal{M}_c((\langle done \rangle P_1)^\frown(7 : P_3))$	2, 4 and **SAT-11**
8- $(\langle done \rangle SAS_1); (30 : SAS_4)$ **sat** $\mathcal{M}_c((\langle done \rangle P_1)^\frown(30 : P_4))$	2, 5 and **SAT-11**
9- (7.44) **sat** $\mathcal{M}_c((7.43))$	1, 6, 7, 8 and **SAT-11**

7.4 Summary

In this chapter, we have presented two case studies to evaluate our work. The Anderson security model is known to be difficult to specify in most existing policy frameworks [16, 4, 75]. For instance, [4] uses this policy model to compare the expressive power of three policy specification languages: ASL [40], LaSCO [39] and Ponder [25]. Only Ponder is able to express the first eight principles. LasCO can specify the authorisation policy part but does not support any form of obligation. ASL is not expressive enough to specify even all the authorisation requirements because of its stratified nature. Based on this result, we believe our policy language is expressive enough. In the next chapter, the policy is animated and analysed using policy analysis tool SPAT.

The second case study illustrates our development technique for secure systems. The security requirements of an exam system are formalised and refined into a secure action system that enforces them. The example also illustrates the specification of dynamically changing security policies using policy composition. Here policies change sequentially when exams are submitted or a deadline is reached. Such a policy composition is not supported by most policy frameworks. The enforcement of compound security policies is pretty modular, based on the composition of secure action systems. The correctness of the implementation of the exam system is established using the compositional verification rules presented in Chapter 6.

Chapter 8

Simulation Environment

Objectives

- To present Tempura, an executable subset of ITL.
- To present Anatempura, a run-time verification tool based on the Tempura interpreter.
- To present SPAT, a security policy analysis and simulation tool.
- To simulate the BMA security policy using SPAT.

8.1 Introduction

An important reason of choosing ITL is the availability of an executable subset, known as *Tempura* [58], of the logic. In this chapter, we give a short presentation of Tempura language and interpreter, and refer to appropriate references for further details. We also present the architecture of the simulation and run-time verification environment, known as *AnaTempura* [78]. SPAT is an additional tool in ITL workbench for security policy

analysis and visualisation. We simulate in this environment the Anderson security model for health care systems.

8.2 Tempura

An ITL formula is executable by the Tempura interpreter if (i) it is deterministic, (ii) the length of the corresponding interval is known and (iii) the values of the variables (in the formula) are known throughout the corresponding interval. The Tempura interpreter takes a Tempura formula and constructs the corresponding sequence of states, i.e., interval. For more technical details of the interpreter, we refer the reader to [58] which is available from the ITL home-page [21].

The syntax of Tempura resembles that of ITL. It has as data-structures integers and booleans and list construct to built more complex ones. Tempura offers a means for rapidly developing, testing and analysing suitable ITL specifications. As with ITL, Tempura can be extended to contain most imperative programming features and yet retain its distinct temporal feel. In particular, it contains control structures such as the conditional statement `IF THEN ELSE` and the iteration statement `WHILE`. The use of ITL, together with its subset of Tempura, offers the benefits of traditional proof methods balanced with the speed and convenience of computer-based testing through execution and simulation. The entire process can remain in one powerful logical and compositional framework.

8.2.1 Expressing Security Policies as Tempura Programs

The implementation of our policy language in Tempura is straightforward. The set $\mathcal{S}$ of subjects can be represented by a list S, the set of objects $\mathcal{O}$ by a list O and the set $\mathcal{A}$ of actions by a list A. Each subject is assigned a unique identifier from 0 to $|\mathrm{S}| - 1$. Similarly, each object (resp. action) is assigned a unique identifier form 0 to $|\mathrm{O}| - 1$ (resp. to $|\mathrm{A}| - 1$). The positive (esp. negative) authorisation matrix $Autho^+$ (resp. $Autho^-$) is implemented as a list $AuthoP$ (resp. $AuthoN$) of size $|\mathrm{S}| \times |\mathrm{O}| \times |\mathrm{A}|$. The authorisation matrix $Autho$ is also implemented as a list of the same size. A cell (i, j, k) of each of the matrices corresponds to the entry $index(i, j, k)$ of the associated list, where $index$ is an injective function from $\{0, \ldots, |S| - 1\} \times \{0, \ldots, |O| - 1\} \times \{0, \ldots, |A| - 1\}$ to $\{0, \ldots, (|S| \times |O| \times |A|) - 1\}$. For example, $index(i, j, k) \mathrel{\widehat{=}} (i * |S| + j) * |O| + k$ is such a function.

The completeness algorithm defined in Chapter 4 (see page 86) is implemented in Tempura using *default value*. If an authorisation cannot be derived from rules then that authorisation is denied by default. In other words, if the value of $AuthoP[index(i, j, k)]$ is undefined in a state then it is taken to be false by default. Similarly for $AuthoN$ and $Autho$. Tempura provides a primitive function $undefined$ which take in argument a variable and returns $true$ if the variable is undefined, and false otherwise. We implement the concept of default value for Boolean variable by defining a Boolean function val as follows:

$$val(X) \mathrel{\widehat{=}} \texttt{if } undefined(X) \texttt{ then } false \texttt{ else } X$$

The function val returns $false$ if the variable in argument is undefined, and the value of

the variable otherwise.

A policy rule of the form

$$f(X, Y, Z) \mapsto Autho^{+}(X, Y, Z)$$

is implemented as follows:

define ruleName() = {
  always {
    forall x $<$ |S|:{
      forall y $<$ |O|:{
        forall z $<$ |A|:{
          if $f'(x, y, z)$
          then AuthoP[index(x,y,z)] = true
        }
      }
    }
  }
}.

where f' is the implementation of f in Tempura, and *ruleName* is the name given to the rule. The keyword **define** is used to bind a name to a constant value or to define subprograms. The keywords **always** and **forall** correspond to the temporal modality $\Box$ and the universal quantifier $\forall$, respectively.

Negative authorisation rules and conflict resolution rules are implemented in the same fashion. For example, the conflict resolution rule

$$\lceil autho^{+}(X,Y,Z) \wedge \neg autho^{-}(X,Y,Z) \rceil^{0} \mapsto autho(X,Y,Z)$$

is implemented in Tempura as follows:

```
define denialTakePrecedence() = {
  always {
    forall x < |S|:{
      forall y < |O|:{
        forall z < |A|:{
          if val(AuthoP[index(x,y,z)]) and not val(AuthoN[index(x,y,z)])
          then Autho[index(x,y,z)] = true
        }
      }
    }
  }
}.
```

A simple policy is defined as a conjunction of rules, i.e.

```
define simplePolicyName() = {
  ruleName1() and
  ruleName2() and
⋮
```

ruleNamen()

}.

Compound policies are implemented as specified by the semantic function $\mathcal{M}_c$ (see Chapter 5). For example, the policy $P_2 \mathrel{\widehat{=}} 20 : P_1$ is implemented in Tempura as follows, where *policy_P1* denotes the specification of P_1 in Tempura:

define policy_P2() = {

policy_P1() **and len(**20**)**

}.

Similarly, $P_3 \mathrel{\widehat{=}} P_1^\frown P_2$ is translated into

define policy_P3() = {

policy_P1();**skip**;policy_P2()

}.

8.3 AnaTempura

AnaTempura is an integrated workbench for ITL that offers

- specification support,
- validation and verification support in the form of simulation and run-time testing in conjunction with formal specification.

Figure 8.1 portrays the general architecture of AnaTempura. There are two main components in AnaTempura, *Monitor* and *Tempura Interpreter*. Monitor allows users to

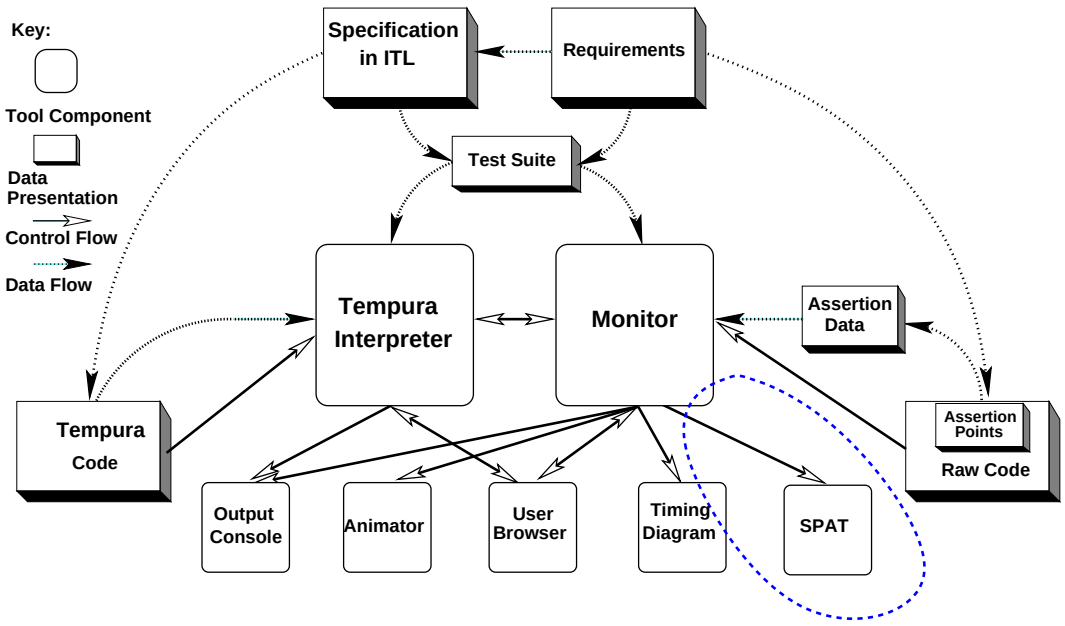


Figure 8.1: General System Architecture of AnaTempura

analyse the program at run-time with respect to a specification. Tempura Interpreter is used to execute Tempura files. AnaTempura also offers powerful visualisation function to enhance the ease of operation of the tool. It is an open architecture that allows new tool components to be easily "plugged in". Our addition to the workbench is SPAT (see Figure 8.1), a software component for security policy visualisation and analysis. A detailed presentation of SPAT is given in the next section.

An overview of the run-time analysis process in AnaTempura is depicted in Figure 8.2. AnaTempura automatically monitors and analyses time-critical systems. The starting point is formulating all behavioral properties of interest, such as safety and timeliness. These are stored in a Tempura file. An information generating mechanism, namely, Assertion Points, will be inserted into the body of the raw code (e.g. C code). These Assertion Points will generate run-time information (assertion data), such as state values,

time stamps, during the execution of the program. The sequence of assertion data will be a possible behaviour of the system and for which we check the satisfaction of our property. Of course, to check Tempura programs there is no need of Assertion Points because the run-time information is already known.

Figure 8.2: The Analysis Process

This checking is done as follows: start the *AnaTempura* and load the Tempura file. *AnaTempura* will then start the compiled raw code with Assertion Points. We then start the Tempura Interpreter to check whether the behaviours, generated by the Assertion Points, satisfy the properties. We note here that if the properties are not satisfied, AnaTempura will indicate the errors by displaying what is expected and what the current system actually provides. Therefore, the approach is not just a "keep-tracking" approach, i.e. giving the running results of certain properties of the system. There is also an facility to animate the received behaviours and to visualise timing properties. Issues of test *coverage* also apply here.

Runtime verification offered by AnaTempura differs from other verification techniques in that the system generates a behaviour at runtime and the system checks whether this behaviour satisfies a given property. An appropriate action should be taken depending on the chosen failure model, one can stop and "repair" the system once an error is found or emit an error message and continue. The difference with testing is that testing is used *before* the system is employed. The difference with model-checking and on-the-fly verifi-

cation is that these check every possible behaviour while runtime verification *only* checks the ones that are generated at runtime. Of course when a system generates all the possible behaviours during its lifetime then these techniques are equivalent. But the crucial difference is that for runtime verification one has only to check the runtime behaviours of the system which can be done in much more efficient way (no state explosion).

8.4 SPAT

SPAT is a prototype for security policy visualisation and analysis. The current version of SPAT allows a textual and a graphical visualisation of security policies. It also allows us to analyse the information flow permitted by an access control policy. An input to the system is a Tempura file containing the security policy of interest. Tempura interpreter executes the security policy and sends information to SPAT through Monitor (see Figure 8.1). SPAT provides an interface that allows a user to query a security policy. For example, a user may wish to know the access rights of an individual or a group of individuals at some state or over a specific period of time. Or a user may wish to observe the information flow allowed by a security policy. The result of a query can be displayed as a table (text format) of a graph. If the policy file contains also delegation policies, the tool can be used to navigate along cascaded delegations permitted by the policy.

In order to ease the presentation of the tool, we consider a component of the healthcare system that enforces Anderson's security policy model discussed in the previous chapter. We assume the following set of subject $\mathcal{S} \mathrel{\widehat{=}} \{$ Alice, Russel, Lena, Hermann$\}$ where Lena and Hermann are clinician and the others are patients. The set objects is $\mathcal{O} \mathrel{\widehat{=}} \{$aliceEPR1,

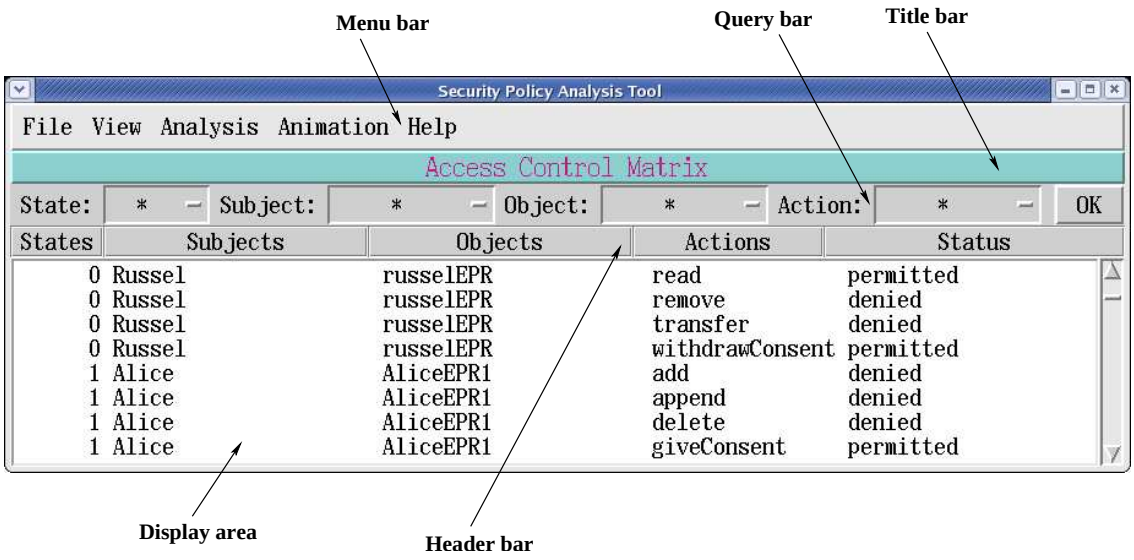


Figure 8.3: SPAT's user interface

aliceEpr2, russelEPR}. Alice has gotten two ERPs: aliceERP1 and aliceEPR2 both created by Hermann. However, Lena is the referring clinician of record aliceEPR2. Russel's record russelEPR is created by Lena. We assume that Hermann referred Russel to Lena.

8.4.1 Visualisation of Authorisation Policies

Figure 8.3 portrays a screen dump of SPAT main window. The window is divided into five zones:

- *Menu bar*, is used to activate the services provided by the tools. Here we focus on the main services for policy visualisation and information flow analysis. The menu **View** provides commands for visualising authorisation and delegation policies in text format. Authorisation policies can be displayed in the form of access control matrix, access control list or capability list. The menu **Analysis** is used to compute the information flow allowed by a security policy and to visualise the output in

text format. The menu **Animation** allows to configure graphical parameters and to represent the output of a query in the form of graph. So, the tool can draw authorisation graph, delegation graph and information flow graph. The graphic engine used is “dot”, which is part of the open source licensed software *Graphviz* developed by *AT&T* and available for download at [46].

- *Title bar*, displays the title of the services being used.

- *Query bar*, allows the user to query the security policy. Queries are specified in *Query By Example (QBE)* style. A user states its query by specifying a pattern (in the form of regular expression) for each column of interest. For example, the service *Access Control Matrix* displays four columns as shown in Figure 8.3: state, subject, object and action. Following are some samples of queries:

  – *Who is allowed to append on Alice’s EPR “aliceEPR1” and when*. This query is specified by:

    State: *

    Subject: *

    Object: aliceEpr1

    Action: *

  – *What are the access rights of Hermann in state 8*. This query is specified by:

    State: 8

    Subject: Hermann

    Object: *

Action: *

– *Which EPRs can be modified in state 5*. This query is specified by:

State: 5

Subject: *

Object: *

Action: append

- *Header bar*, states the titles of the columns displayed in the display area.

- *Display area*, displays the answer to a query in text format.

An authorisation graph is a directed graph similar to take-grant model [15, 71]. Nodes in the graph are of two types, rectangular one corresponding to subjects and oval one to objects. In addition, nodes corresponding to subjects are coloured in orange and those corresponding to objects are blue. An arc directed from a node A to a node B indicates that A has the right to perform some action(s) on B. The arc is labelled with the set of actions A can perform on B and/or the state in which the authorisation applies. The display of labels is optional. An example of authorisation graph is depicted in Figure 8.4. The simulation period is 100 time units long. We assume that the record aliceEPR1 expires after 43 time units, aliceEPR2 after 80 time units and russelEPR after 95 time units.

The graph states the access rights of subjects in state 70 (denoted by "[s70]" in the graph) of the simulation period. Alice and Russel can only access their respective records, and the actions allowed to them are $read$, $giveConsent$ and $withdrawConsent$ with re-

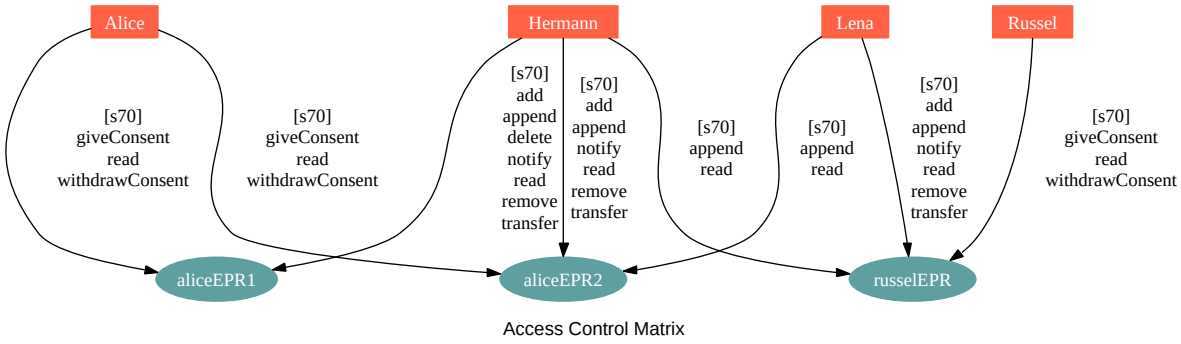


Figure 8.4: Authorisation graph

spect to the policy rules (7.10), (7.20), and (7.21). Lena, as responsible clinician, is allowed to perform the following actions on russelEPR: add, $append$, $notify$, $read$, $remove$ and $transfer$. These authorisations are generated by the rules (7.9), (7.10), (7.13), . . .,(7.19). However, Lena is denied the right to delete the record because the record is not expired yet, accordingly to the policy rule (7.30). The policy rules (7.11) and (7.12) authorise Lena, as the referring clinician, to read and append on aliceEPR. Similar interpretation can be done for Hermann. However, Hermann is allowed to delete aliceEPR1 in accordance with the policy rule (7.31).

8.4.2 Visualisation of Delegation Policies

The visualisation of delegation policies is similar to that of authorisation policies. It indicates for each state of the simulation period, which subject is allowed to delegate which access rights and to whom. The delegation window in SPAT displays the state in which the delegation authorisation is granted, the grantor, the grantee, the target object and the action granted. Cascaded delegations are more visible in a delegation graph. In addition to the two types of nodes used in an authorisation graph, a delegation graph

uses a third type of node which has the shape of a diamond. A diamond node is used to distinguish between the grantor and the grantee of a delegation. An arc goes from the grantor to a diamond node, a second arc from the diamond node to the grantee and a third one from the diamond node to the target object. The first arc is labelled with the state in which the delegation right is granted, and the third arc is labelled with the delegated action. An example of delegation graph is given in Figure 8.5. In this graph, e.g. Bob is allowed to delegate to Carol the right to download movies. This right can propagate through cascaded delegation from Bob to Carol, then from Carol to John and finally from John to Dave.

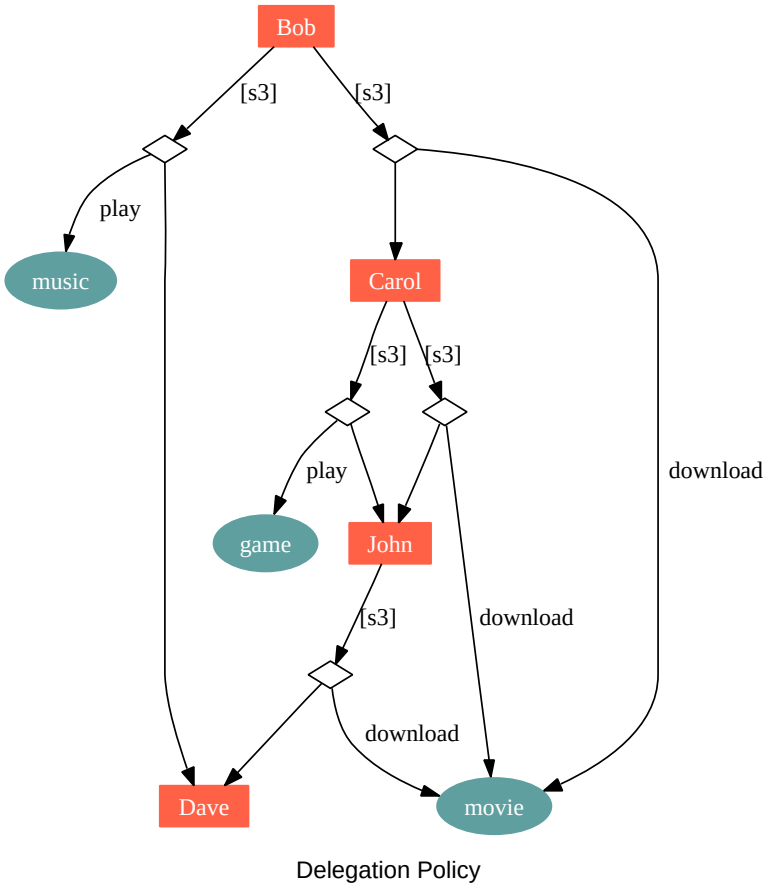


Figure 8.5: Delegation graph

8.4.3 Information Flow Analysis

The information flow analysis is based on two categories of actions: *read* actions and *write* actions. A read action is an action that can leak information from the object to the subject that performs the action. For example checking the balance of a bank account or reading a file leaks information from the bank account or the file to the subject that exercises the action. By contrast, a write action allows information to flow from the subject that exercises the action to the object recipient. For example crediting a bank account, or appending to a file. Actions that belong to neither of these categories are ignored. However some actions may belong to both categories. In the sequel we denote by $\mathcal{A}_r$ and by $\mathcal{A}_w$ the subset of $\mathcal{A}$ of all read actions and all write actions respectively.

Definition 8.1 and Definition 8.2 define our notion of *allowed direct information flow* from a subject to an object and from an object to a subject respectively.

Definition 8.1 *We say that there is an allowed direct information flow from a subject s to an object o if the subject s is allowed to perform a write action on the object o, viz*

$$s \rightsquigarrow o \mathrel{\widehat{=}} \bigvee_{a \in \mathcal{A}_w} autho(s, o, a).$$

Figure 8.6-a illustrates a direct flow from a subject to an object.

Definition 8.2 *We say that there is an allowed direct information flow from an object o to a subject s if the subject s is allowed to perform a read action on the object o, viz*

$$o \rightsquigarrow s \mathrel{\widehat{=}} \bigvee_{a \in \mathcal{A}_r} autho(s, o, a).$$

Figure 8.6-b illustrates a direct flow from an object to a subject.

(a) From a subject to an object (b) From an object to a subject

Figure 8.6: Direct information flow

As such, we have defined a relation $\rightsquigarrow$ over the set $\mathcal{P} = \mathcal{S} \cup \mathcal{O}$. We call elements of the set $\mathcal{P}$ agents. Note that $v \rightsquigarrow v'$ is a state formula, for any $v, v' \in \mathcal{P}$. For example the following formula holds for an interval if there is a direct information flow from agent v_1 to agent v_2 in that interval's second state:

$$\bigcirc(v_1 \rightsquigarrow v_2).$$

Yet another example is the formula

$$\Diamond(v_1 \rightsquigarrow v_2); \Diamond(v_2 \rightsquigarrow v_3),$$

which holds for an interval if information can flow from agent v_1 to agent v_2 at some point t in time, and from agent v_2 to another agent v_3 at some later time $t' \geq t$. This illustrates an implicit (possible) leakage of information from agent v_1 to agent v_3. For this reason, it is necessary to compute the transitive closure of the relation $\rightsquigarrow$ which lists out all possible flows of information allowed by an access control policy. The information flow transitive closure is formalised in Definition 8.3.

Definition 8.3 *Information can flow from $v \in \mathcal{P}$ to $v' \in \mathcal{P}$ if there exists a direct information flow from v to v' at some point in time or information can flow from v to some*

agent $u \in \mathcal{P}$ and from u to v' later on, i.e.

$$v \rightsquigarrow^+ v' \mathrel{\widehat{=}} \Diamond(v \rightsquigarrow v') \vee \bigvee_{u \in \mathcal{P}} ((v \rightsquigarrow^+ u); true; (u \rightsquigarrow^+ v')).$$

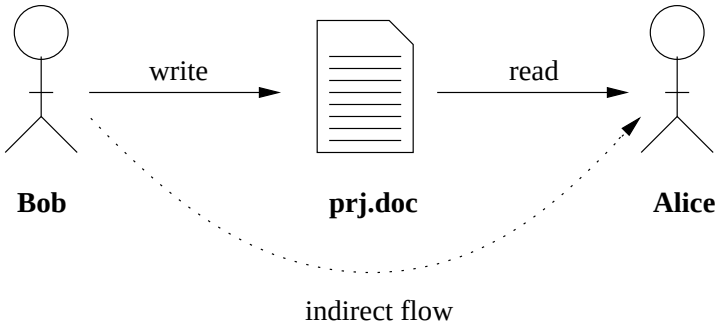


Figure 8.7: Indirect information flow

Figure 8.7 gives an example of transitive information flow from Bob to $Alice$ via the file $prj.doc$. The operator $\rightsquigarrow$ and its closure are implemented in SPAT.

For the health care example, the only read action is the action $read$ and the only write action is the action $append$. SPAT enables to visualise the information flow allowed by a security policy both as a table (text format) and as a graph. Figure 8.8 illustrates the textual presentation of allowed direct information flow. The presentation of the transitive closure is similar. It is a table of three columns indicating the *state* in which the information flow may take place, and *source* and *destination* of the information flow. Again, one can query the system to obtain desired information. For example, Figure 8.8 displays the allowed direct information flow in state 8. One can see that information can flow from aliceEPR2 to Alice who is the owner of the record, in accordance with the policy requirement (7.10). Hermann as the responsible clinician of the record aliceEPR2 is allowed by the security requirement (7.9) to append information on that record. This implies an allowed direct information flow from the subject Hermann to the object aliceEPR2.

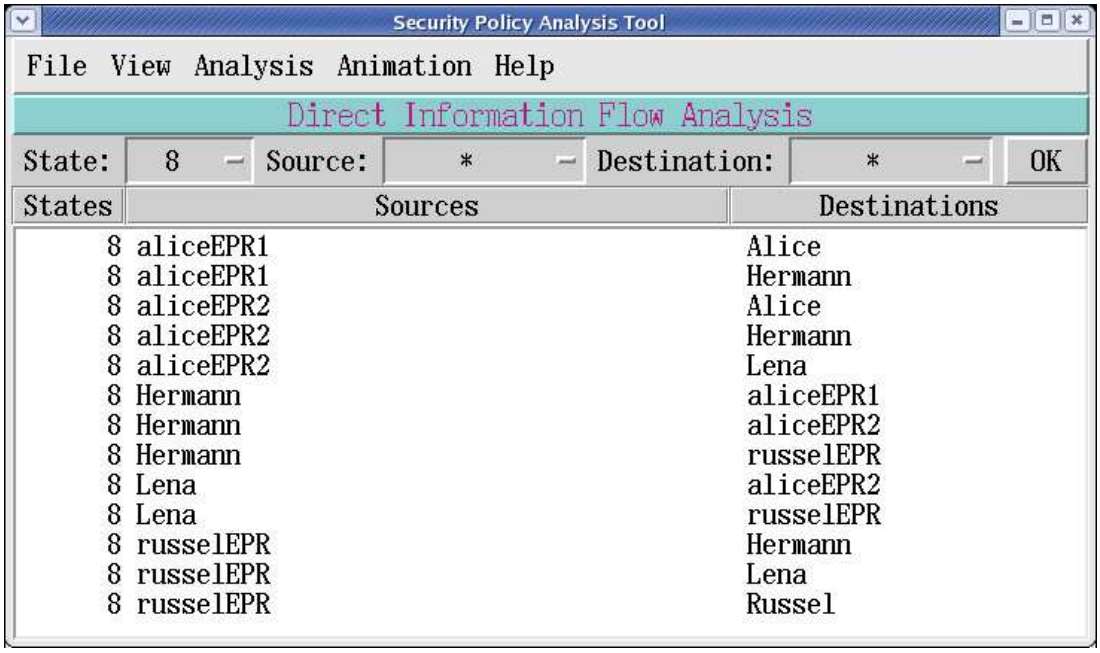


Figure 8.8: Visualisation of information flow

An allowed information flow graph is similar to an authorisation graph where an arc from a node A to a node B indicates that information can flow from A to B. An arc can be labelled with the state in which the flow can take place. There are two types of arcs: solid ones and dotted ones. A solid arc indicates a direct flow while a dotted one indicates an indirect flow. Dotted arcs appear only in the graphical representation of information flow closure.

Figure 8.9 presents the graph generated by SPAT for the output depicted in Figure 8.8. The allowed direct information flow from aliceEPR2 to Alice is represented by a solid arc from the node corresponding to the object aliceEPR2 to the one representing the subject Alice. Similarly, there is a solid arc from the node representing the subject Hermann to the one representing the object aliceEPR2. Figure 8.10 portrays the closure of the graph depicted in Figure 8.9. One can observe the indirect information flow from Hermann to

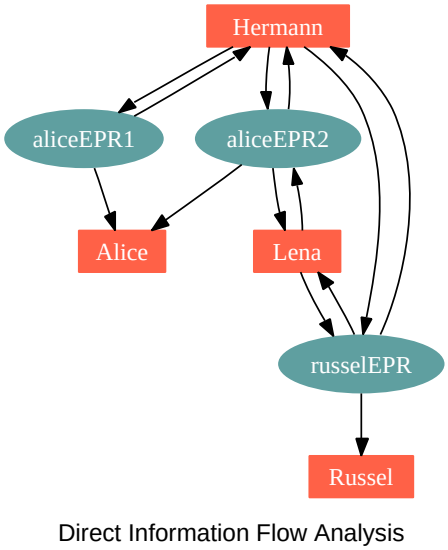


Figure 8.9: Direct information flow graph

Alice generated by the computation of the closure of direct information flow. This means that the security policy allows information leakage from Hermann to Alice. Similarly, the absence of arc between Alice and Russel indicates that information flow from one to another is forbidden by the security policy.

8.5 Summary

We have presented an environment for simulating, visualising and analysing security policies. The executable subset of ITL, Tempura, and its interpreter is the kernel of the simulation environment. We have shown how a security policy written in the policy language of Chapter 5 is translated into a Tempura program and executed. Run-time verification of secure systems can be done using AnaTempura with *assertion points*. We have developed a special tool called *SPAT* for security policy analysis and visualisation. SPAT is used

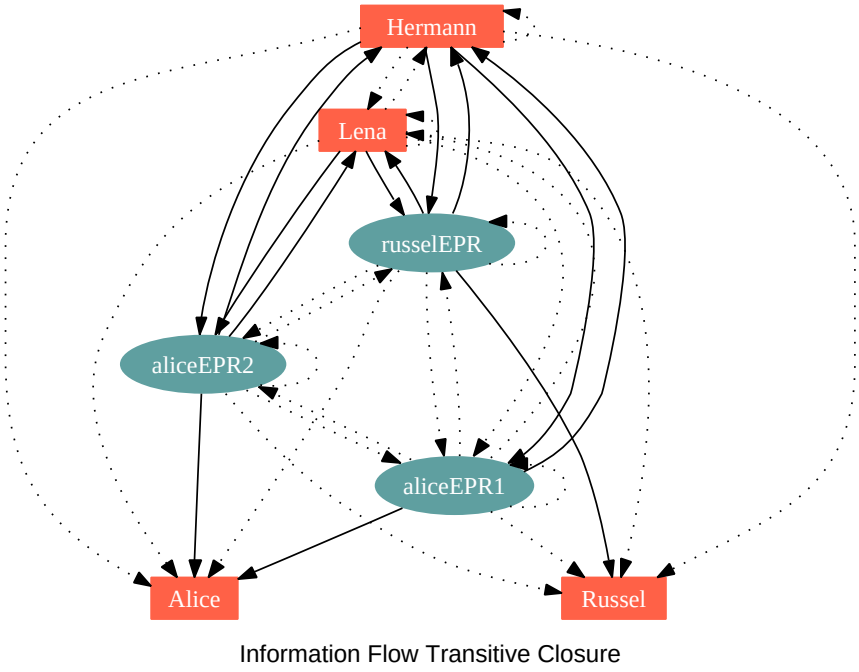


Figure 8.10: Information flow closure graph

to visualise the behaviour of a security policy over period of time and to compute the information flow allowed by a security policy. It provides a graphical user-interface that allows the users to specify queries for extracting information about a security policy. The result of a query can be displayed in text format or more friendly as a graph. The current version of the system can draw authorisation graphs, delegation graphs and information flow graphs. These are useful for policy validation and debugging. As a proof of concept, we have simulated the Anderson's security policy model for a simple sample of a health care system. SPAT has been used to observe the information flow allowed by the policy.

Chapter 9

Conclusion and Future Work

In this dissertation, we have defined a unified formal and compositional framework for the development of secure systems. Our approach uses a single formalism, namely Interval Temporal Logic (ITL), to specify and to reason about properties, whether functional, temporal and security, of systems. This makes it easy to model the dependencies and interactions between the three types of system requirements and provides uniform support for the design and verification of secure systems. More importantly, it becomes possible to address security concerns early just as an integral part of system requirements. Security requirements are as important as the functional counterpart of system requirements, therefore deserve similar attention.

We have proposed a security policy language that can express dynamically changing security policies. Such a policy allows a system switching from one policy to another after a specific period of time or when a specific event occurs. This kind of policy is not supported by most policy languages. We have defined a set of operators to express

changes among policies and to compose policies. An algebra of these operators is proposed. It allows policies to be compared and to be refined using sound algebraic laws. The language allows to specify *positive* and *negative* authorisation policies, and positive and negative delegation policies. Conflicts among positive and negative authorisations (resp. delegations) are resolved using special rules. A simple policy is defined to be a set of authorisation and delegation rules. Many policy languages can specify this kind of policy. The novelty of our approach comes from the way policies are composed.

The formalisation of dynamically changing security policies is challenging and is not supported by most policy frameworks. Following are some samples of dynamically changing policies that can be expressed in our policy language. A policy $(\langle b \rangle P); Q$ behaves like the policy P until the condition b turns true, them the policy changes to behave like the policy Q. Yet another example of dynamically changing policy is an expression $P \rhd_w^t Q : S$ that behaves like the policy P until the condition w is $true$ or a time-out t time unit(s) has elapsed. If the condition w becomes true then the policy behaves like the policy Q, otherwise it behaves like the police S in the event of time-out.

When using positive and negative authorisations, one of the problems arising in hybrid policies is the problem of handling *incomplete* specifications, i.e. what happens if no authorisation is specified for a specific access. This problem is solved for open and closed policies, where only positive authorisations are allowed in the former and only negative authorisations are allowed in the latter. However, open and closed policies are not expressive enough to model many protection requirements encountered in practise. We propose an algorithmic solution to that problem, in the scope of our policy model. The algorithm

transforms automatically an incomplete policy into a complete one that grants *exactly* the rights explicitly stated in the input policy and denies everything else. The correctness of the algorithm is established.

We have proposed a computational model, *Secure Action System (SAS)*, that allows the enforcement of (dynamically changing) security policies. It is an extension of the traditional action system paradigm [6] to cater for security. In addition to the security features provided by SASs, the novelty of our approach leans also on the fact that the development of SASs is compositional. SASs can be composed to build a system that enforces a specific (dynamically changing) security policy. We propose a rich set of sound compositional rules for the design and verification of SASs.

In order to help understand security policies and validate them, we have developed a software tool, SPAT, for analysing security policies. SPAT allows us to visualise security information extracted from a security policy during its simulation. The tool can display in text format or as a graph the access rights, the delegation rights and the information flow allowed by a security policy at certain point in time or during a specific period of time. A graphical representation of a security policy is much more friendly to casual and expert users.

As proof of concept we have considered two case studies. The first case study formalises the Anderson's security policy for electronic patient records. Aljareh and Rossiter [4] showed that many policy frameworks cannot specify the nine principles that compose this policy. The second case study illustrate our system development technique with a simple yet interesting example of a secure (United Kingdom based) examination setting

system.

In a related development, recent proposals have worked towards languages and models able to express, in a single framework, a variety of security policies. Logic-based languages, for their expressive power and formal foundations, represent a good candidate [76, 40, 37]. A temporal model of access control policies has been proposed by Bertino and al.[11], that allows the specification of periodic authorisations, that is, authorisations that hold in specific periodic intervals. In order to ease the specification of security policies and to help non specialist users understanding them, LaSCO a graphical language for expressing authorisation policies has been proposed [39].

The completeness problem has been addressed in many other works. Default rules have been used in [76] as a way to provide complete specification. The drawback of this approach is that default rules might not be conclusive. As a consequence the model can lead to a situation in which an authorisation request has no answer. Most of the logic-based approaches [40, 27, 42, 48] restrict the policy language to a variant of Datalog [33] to overcome this problem. Datalog is an efficient well-understood reasoning engine that was originally designed for function-free negation-free Horn clauses. The completeness result in the variants relies on the *closed world assumption*. According to this assumption, if we cannot prove that a positive literal is true, we assume that it is false.

Role-Based Access control (RBAC) models [67, 2, 50, 77, 8] have attracted attention, especially for commercial applications. Ponder [25] is a declarative language for specifying RBAC policies and management policies for distributed systems. Trust management systems (such as PolicyMaker [18], KeyNote [19], REFEREE [22], and DL [49]) are used

to express security policies in open applications such as e-commerce and other Internet-enabled services which require interaction between different, remotely located, parties that may know little about each other. None of these models allows the specification of dynamically changing security policies.

Although a variety of security policy models and languages have been developed, little work is done about how security policies can be integrated into the system requirement from the early stages of system development life cycle. It is widely agreed nowadays that ad hoc solutions that implement security after the system has been developed are error-prone and therefore are just not acceptable for the design of security-critical systems. An attempt towards the integration of security requirement with other system requirements has be done by [30]. Devanbu and Stubblebine [28] suggested extending standards such as UML to include modelling of security requirements. This idea has inspired many authors and models like authUML [3], UMLsec [44] and SecureUML [52] have been developed. These models do not support formal reasoning because there is not yet a well-defined formal semantics for UML.

Future research directions include:

- development of a high level *design language*, **SANTA**, for the specification of secure systems. The secure action system paradigm provides a low level programming language which is limited in terms of data structures and control structures. SANTA language is a *wide-spectrum* language that allows both abstract specification and concrete programming constructs. Abstract constructs in SANTA are expressed as ITL formulae while concrete ones are agent-based. Such a language

allows to mix abstract specifications and concrete programs, and to develop a system by applying successive *correctness-preserving* refinement steps.

- development of a *refinement calculus* that allows to transform an abstract specification in SANTA into a concrete SANTA program. To do this, we will need to define a formal semantics in ITL to the concrete part of the SANTA language. Sound refinement rules will then be developed with respect to this semantics. The calculus must allow to refine security enforcement architectures. We think of two main architectures: (i) **security enforcer**, where a central agent is responsible for the enforcement of the security policy of the system; (ii) **vigilant**, where each agent in the system is responsible for its own security (i.e. is vigilant). **Hybrid** architectures, allowing security enforcers and vigilant agents, will also be investigated.

- development of a SANTA *compiler* that translates SANTA programs into a concrete agent frameworks such as JADE [74] or Cougaar [26], which provide platforms for deploying multi-agent systems.

- the prototype SPAT needs to be updated to cope with SANTA specifications and to enable debugging of security policies.

On the other hand, when performing a security analysis of a system, both *overt* and *covert* channels of the system must be considered. Overt channels use the system's protected data objects to transfer information while covert channels, in contrast, use entities (such as file locks, busy device flags, and the passing of time) not normally viewed as data objects to transfer information from one subject to another. Although when identi-

fied, covert channels can be controlled in our security model, our approach does not deal with the discovery of such channels. We will investigate how our approach can be coupled with techniques such as *Shared Resource Matrix* methodology [45] for discovering storage and timing channels within systems.

Successful software systems always evolve as the environment in which these systems operate changes and requirements change [61]. Therefore managing changes is an important issue in software engineering. Typical changes to requirement specifications include adding or deleting requirements, and fixing errors. We will investigate how changes affect our system development method. We believe that the use of specification patterns for secure systems might be helpful.

References

[1] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(3):1–29, September 1993.

[2] Gail-Joon Ahn and Ravi Sandhu. Role-Based Authorization Constraints Specification. *ACM Transaction on Information and Systems Security*, 3(4):207–226, November 2000.

[3] Khaled Alghathbar and Duminda Wijesekera. authUML: A Three-phased Framework to Analyse Access Control Specifications in Use Cases. In *Proceedings of the ACM Workshop on Formal Method for Security Engineering (FMSE 2003)*, pages 77–86, Washington, DC, USA, October 2003. ACM Press.

[4] Salem Aljareh and Nick Rossiter. Towards security in multi-agent clinical information services. In *Proceedings of the Workshop on Dependability in Healthcare Informatics*, Edinburgh, March 2001.

[5] Ross Anderson. A Security Policy Model for Clinical Information Systems. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 34–48,

May 1996.

[6] R.J.R. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142. ACM, 1983.

[7] R.J.R. Back and R. Kurki-Suonio. Distributed Co-operation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10:513–554, 1984.

[8] Jean Bacon, Ken Moody, and Walt Yao. A Model of OASIS Role-Based Access Control and Its Support for Active Security. *ACM Transactions on Information and System Security*, 5(4):492–540, November 2002.

[9] W. Robert Baldwin. Naming and grouping privileges to simplify security management in large database. In *Proceedings of IEEE Computer Society Symposium on Research in Security nad Privacy*, pages 61–70, Oakland, CA, April 1990.

[10] D. E. Bell and L. J. LaPadula. Secure compter system: Unified exposition and multics interpretation. Technical report, MITRE Corp., Bedford, Massachusetts, July 1975.

[11] Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transaction on Database Systems*, 23(3):213–285, September 1998.

[12] Elisa Bertino, Claudio Bettini, and Pierangela Samarati. A Time Based Authorization Model. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 126–135, Fairfax, Va, November 1994.

[13] Elisa Bertino, Pierangela Samarati, and Sushil Jajodia. An extended authorization model for relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(1):85–101, January-February 1997.

[14] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp., 1977.

[15] M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *Proceedings of 7th Symposium on Operating Systems Principles, ACM SIGOPS Operating Systems Review*, volume 13, pages 45–54, December 1979.

[16] Matt Bishop. *Computer Security*. Addison-Wesley, 2002.

[17] Dines Bjoner. Trusted computing systems: the ProCos experience. In *Proceedings of the 14th International Conference on Software Engineering*, pages 15–34, Melbourne, Australia, 1992. ACM Press.

[18] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, CA, May 1996.

[19] M. Blaze, J. Feigenbaum, J. Loannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In *Secure internet Programming: Issue*

in Distributed and Mobile Object Systems. Springer verlag-LNCS State-of-the-Art series, 1998.

[20] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 215–228, Oakland, CA, 1989.

[21] Antonio Cau, Ben Moszkowski, and Hussein Zedan. *Interval Temporal Logic*. http://www.cse.dmu.ac.uk/~cau/itlhomepage/index.html.

[22] Y-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for Web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.

[23] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of IEEE Computer Society Symposium on Security and Privacy*, pages 184–194, Oakland, May 1987.

[24] Nicodemos C. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, February 2002.

[25] Nicodermos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Workshop on Policies for Distributed Systems and Networks*, number 1995 in LNCS, pages 18–39, Bristol, UK, 29-31 January 2001. Springer-Verlag.

[26] DARPA. *Cognitive Agent Architecture*. http://www.cougaar.org.

[27] John DeTreville. Binder, a Logic-based Security Language. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 95–105, 2002.

[28] Premkumar Devanbu and Stuart Stubblebine. Software engineering for security: a roadmap. In Anthony Finkelstein, editor, *The Future of Software Engineering*, pages 225–239. ACM Press, 2000. Special Volume (ICSE 2000).

[29] C. Eckert and D. Marek. Developing Secure Applications: A Systematic Approach. In *Proceedings of the IFIP 11th International Conference on Security (SEC'97)*, pages 267–279, Copenhagen, Denmark, May 1997. Chapman & Hall.

[30] Claudia Eckert. Matching security policies to application needs. In *Proceedings of the IFIP 11th International Conference on Information Security*, pages 237–254, Cape-Town, South Africa, May 1995.

[31] P. Epstein and R. S. Sandhu. Towards a UML Based Approach to Role Engineering. In *Proceedings of the 4th ACM Workshop on Role-Based Access Control*, pages 145–152, Fairfax, Virginia, USA, 1999. ACM Press.

[32] Patrick R. Gallagher. A Guide To Understanding Discretionary Access Control in Trusted Systems. Technical Report NCSC-TG-003, National Computer Security Center, September 1987.

[33] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, New Jersey, 2002.

[34] G. Graham and P. Denning. Protection–principles and practice. In *Proceedings of AFIPS Sprint Joint Computer Conference*, pages 417–429, 1972.

[35] P. P. Griffiths and B. W. Wade. An Authorization mechanism for a relational database system. *ACM Transactionon Database Systems*, 1(3):242–255, 1976.

[36] Roger William Stephen Hale. *Programming in Temporal Logic*. PhD thesis, Trinity College, University of Cambridge, October 1988.

[37] Joseph Halpern and Vicky Weissman. Using First-Order Logic to Reason about Policies. In *Proceedings of the 16th IEEE Computer Security Fundations Workshop*, pages 187–201, 2003.

[38] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, 1976.

[39] James Allen Hoagland. *Specifying and Implementing Security Policies Using LaSCO, the Language for Security Constraints on Objects*. PhD thesis, University of California, Davis, 2000.

[40] Sushil Jajodia, P. Samarati, V. S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. *ACM transaction on Database Systems*, 26(2):214–260, June 2001.

[41] Helge Janicke, Francois Siewe, Kevin Jones, Antonio Cau, and Hussein Zedan. Analysis and Run-time Verification of Dynamic Security Policies. In *Proceedings of the Workshop on Defence Applications of Multi-Agent Systems (DAMAS) 2005*

(colocated with AAMAS 05), pages 55–66, Ultrecht University, The Netherlands, July 2005.

[42] T. Jim. SD3: a Trust Management System with Certified Evaluation. In *Proceedings of the 22th IEEE Symposium on Security and Privacy*, pages 106–115, Oakland, California, May 2001.

[43] C. B. Jones. *Systematic Software Development– Using VDM*. Printice-Hall International, 2nd edition edition, 1989.

[44] J. Jurjens. UMLsec: Extending UML for Secure Systems development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 412–425, Dresden, Germany, October 2002.

[45] Richard A. Kemmerer. Shared resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels. *ACM Transactions an Computer Systems*, 1(3):256–277, August 1983.

[46] AT&T Labs-Research. *Graphviz Distribution*. http://www.research.att.com/sw/tools/graphviz/download.html.

[47] Bulter Lampson. Protection. In *Proceedings of the 5th Princeton Symposium of Information Science and Systems*, pages 437–443, March 1971.

[48] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, Febrary 2003.

[49] N. Li and J. Grosof, B. N.and Feigenbaum. A practical implementation and tractable delegation logic. In *Proceedings of the IEE Symposium on Security and privacy*, pages 27–42, Oakland, CA, 2000.

[50] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of 2002 IEEE Symposium on security and Privacy*, pages 114–130, Oakland, CA, 2002.

[51] Li, N. and Mitchell, J. C. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5h Intenational Symposium on Practical Aspects of Declarative Languages*, January 2003.

[52] T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 426–441, Dresden, Germany, October 2002.

[53] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In Margaria and Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996.

[54] Gavin Lowe. Analysis Protocols Subject to Guessing Attacks. *Journal of Computer Security*, 12(1), 2004.

[55] John MacLean. Algebra of security. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 2–7, Oakland, CA, 1998.

[56] John McLean. Security Models. In John Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley Press, 1994.

[57] Jonathan D. Moffett and Bashar A. Nuseibeh. A Framework for Security Requirements Engineering. Technical Report YCS 368, Department of Computer Science, University of York, 2003.

[58] Ben Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, England, 1986.

[59] Ben Moszkowski. Compositional reasoning using interval temporal logic and tempura. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1486 of *LNCS*, pages 439–464, Berlin, 1998. Springer Verlag.

[60] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, January 1997.

[61] Bashar Nuseibeh and Steve Easterbrook. Requirements Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, Limerick, Ireland, 2000. ACM Press.

[62] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, April 1980.

[63] P. Samarati and S. Vimercati. Access Control: Policies, Models, and Mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design (Tutorial Lectures)*, pages 137–196. Springer-Verlag, September 2000.

[64] R. Sandhu. The Type Access Matrix Model. In *Proceedings of the 1992 IEEE symposium on Security Conference*, pages 122–136, April 1992.

[65] R. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[66] R. S. Sandhu. Role Activation Hierarchies. In *Proceedings of the 3rd ACM/NIST Role Based Access Control Workshop*, Fairfax, Virginia, USA, October 1998. ACM Press.

[67] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[68] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[69] François Siewe, Antonio Cau, and Hussein Zedan. A Compositional Framework for Access Control Policies Enforcement. In Michael Backes, David Basin, and Michael Waidner, editors, *ACM Workshop on Formal Methods in Security Engineering (FMSE'03)*, pages 32–42, Washington, DC, October 2003. ACM Press.

[70] François Siewe, Helge Janicke, and Kevin Jones. Dynamic Access Control Policies and Web-Service Composition. In *Proceedings of the 1st Young Researcher Workshop on Service-Oriented Computing*, Leicester, U.K., April 2005.

[71] L. Snyder. Formal models of capability-based protection systems. Technical Report 151, Dept. computer Science, Yale University, New Haven, Conn, April 1979.

[72] Susan Stepney, David Cooper, and Jim Woodcock. An Electronic Purse: Specification, Refinement, and Proof. Technical Report PRG-126, Oxford University Computing Laboratory, July 2000.

[73] R. K. Thomas. Team-base Access Control (TBAC): A Primitive for Applying Role-based Access Control in Collaborative Environments. In *Proceedings of the 2nd ACM/NIST Role Based Access Control Workshop*, pages 13–19, November 1997.

[74] TILAB. *Java Agent Development Framework*. http://sharon.cselt.it/projects/jade.

[75] Harold F. Tipton and Micki Krause. *Information Security Management Handbook*. Auerbach Publications, fifth edition edition, 2004.

[76] Thomas Y. C. Woo and Simon S. Lam. Authorization in distributed systems: A new approach. *Journal of Computer Security*, 2(2,3):107–136, 1993.

[77] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. A rule-based framework for role-based delegation and revocation. *ACM Transactions on Information and System Security*, 6(3):404–441, August 2003.

[78] Shikun Zhou, Hussein Zedan, and Antonio Cau. Run-time Analysis of Time-critical Systems. *Journal of System Architecture*, 51(5):331–345, 2005.