A Framework for Organisational Control Principles

Andreas Schaad

Submitted for the degree of
Doctor of Philosophy.

The University of York

Department of Computer Science

July 2003

In Gedanken an meinen zu früh verstorbenen Vater Dr. med. Gerhard Schaad und meine Mutter Christiane Schaad, die mir durch ihre unermüdliche Unterstützung diese Arbeit ermöglicht hat.

Abstract

Just as organisations have goals describing their primary business objectives, they also have goals with respect to controlling how these objectives are met. These are the control goals of an organisation which are enforced through a system of internal control. Such a system enables them to adhere to external laws and internal regulations, prevent and detect fraud and continuously enhance the overall quality of the business. Independent of the type of organisation, these internal control systems use common underlying principles to establish and achieve control over business activities.

Some of these principles have been previously described and analysed within the context of existing role- and policy-based frameworks. These descriptions, however, never explicitly consider the organisational background of these principles. Other principles have not received any attention at all. What is required is to provide a unifying representation of control principles allowing for their extended analysis and exploration.

This thesis presents a framework suitable to express and analyse a set of selected control principles. It consists of three main parts:

1. A review and discussion of organisational control principles, their origin, relationships, and existing role- and policy-based frameworks that partially support their expression;
2. A formal model for organisational control principles on the basis of which we define, analyse, discuss and explore:
   - separation controls;
   - delegation and revocation controls;
   - review and supervision controls;
3. A case study consisting of two parts, the first part describing the access control system and applied control principles of a major European bank, the second part using the domain example of a branch of that bank for the validation of our framework.

The Alloy specification language and its facilities for automated analysis are used to specify the components of the framework and explore their relationships.

The conclusion of this thesis is that a general framework for control principles is feasible, and provides valuable insights into their relationships and use.

Contents

List of Figures

List of Tables

Acknowledgements

I want to thank my supervisor Dr. Jonathan Moffett for his constant advice and help. The Engineering and Physics Research Council has provided funding under grant number 99311141. Additional financial support was provided by Prof. John McDermid throughout my research. The collaboration with Dr. Axel Kern and other staff from Betasystems AG contributed to my understanding of enterprise access control management. The staff at Dresdner Bank further assisted in clarifying aspects of real-world access control and provided access to the material the case studies of this thesis are based on. My colleagues Dr. Karsten Loer and Michael Hildebrandt provided valuable feedback to this work.

Declaration

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Philosophy of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references.

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and abstract to be made available to outside organisations.

Signed ..... (Andreas Schaad)

Date .....

Chapter 1

Introduction

1.1 Background and motivation

Error and fraud can lead to losses of any kind. Identifying, establishing and maintaining the appropriate controls can help to prevent and detect such errors and fraud. When Nick Leeson brought down Barings Bank after running up losses of £862 million on unauthorised trading in derivatives, a lack of control became painfully obvious [George, 1995]. However, at least in the financial area little seems to have been done, as is documented in the case of John Rusnak, alleged to be responsible for a £500 million fraud when trading currencies at an US subsidiary of Allied Irish Bank in early 2002 [The Times, 2002]. When interviewed about Rusnak's case, Mr. Leeson said that the deals at Allied Irish Banks' US subsidiary showed similarities to his own 1995 scandal in Singapore. Leeson claimed that many financial institutions had failed to impose effective checks, saying that "The checks that should be in place to stop this sort of thing happening are extremely basic and people haven't been doing them.". Allied Irish Bank's president Susan Keating stated at that time that it was clear that the alleged fraud had gone on for more than a year, and Mr. Rusnak had worked out how to overcome internal safeguards against unauthorised trading. The suspected fraud was uncovered during a management review by the treasury department.

What is characteristic of this and similar examples of fraud and error [Anderson, 2001], is that the principals conducting the fraud had been legitimately delegated the needed authority, which they then exploited to their favour. This illustrates the general dilemma in which organisations find themselves: On the one hand principals need to be given sufficient authority to perform their duties, while on the other hand they must be controlled to prevent them from (in)advertently using this authority in any way adverse to the organisational goals.

Looking at the fraud studies conducted by KPMG from 1996 to 2002 [KPMG, 2002], it can be seen that internal fraud may take various forms including insider trading of shares, conflict of interest in advising a client, purchase for personal use, cheque forgery and others. More interesting is the fact that the main reasons for fraud are the poor segregation of duties, poor access control, and easy override of internal controls. However, most fraud is claimed to be discovered by internal controls.

Two conclusions can be drawn from this. Poor internal controls and ineffective access controls are two main factors leading to fraud. Yet, internal controls are also the most efficient measure to prevent and detect fraud.

The concrete form and realisation of internal controls will vary, depending on the organisational context. However, we believe that independent of the type of organisation there are some common underlying principles used to establish and achieve control over business activities. Some of these principles have been previously described and analysed within the context of existing role- and policy-based frameworks. These descriptions, however, never explicitly consider the organisational background of these principles. Other principles have not received any attention at all. What is required is to provide a unifying representation of control principles allowing for their extended analysis and exploration.

1.2 Research hypotheses and methodical approach

We establish a framework for organisational control principles. This framework addresses the following theses:

1. Current systems management and access control research only partially acknowledges the presence of control principles and
   - some control principles have only been sparsely covered in the existing literature;
   - their general organisational background has not received sufficient attention yet.
2. The lack of a unified and integrated representation of control principles does not allow for their more detailed analysis and exploration.
3. Such a framework may be used for the analysis and application of control principles in an organisational context.

This framework will be established following an *explorative* approach where we understand explorative in two different but complementary ways. The first interpretation is that we seek to find out more about the characteristics of control principles that have not been described at all, or only insufficiently in the current literature. This underlines the conceptual character of our work which attempts to provide a simplified but coherent representation of control principles. The second interpretation of explorative is that we will discuss our findings by sometimes comparing different approaches that may have led to the same results.

With respect to the *framework* character of our work we refer to the definition of Jayaratna who defines a conceptual framework as "..a meta-model through which a range of concepts, models, techniques, or methodologies can either be clarified, compared, categorised, evaluated and/or integrated." [Jayaratna, 1994]. We will refer to this definition throughout this thesis. Specifically, our final discussion in section 11.3 will critically evaluate the established framework with respect to this definition.

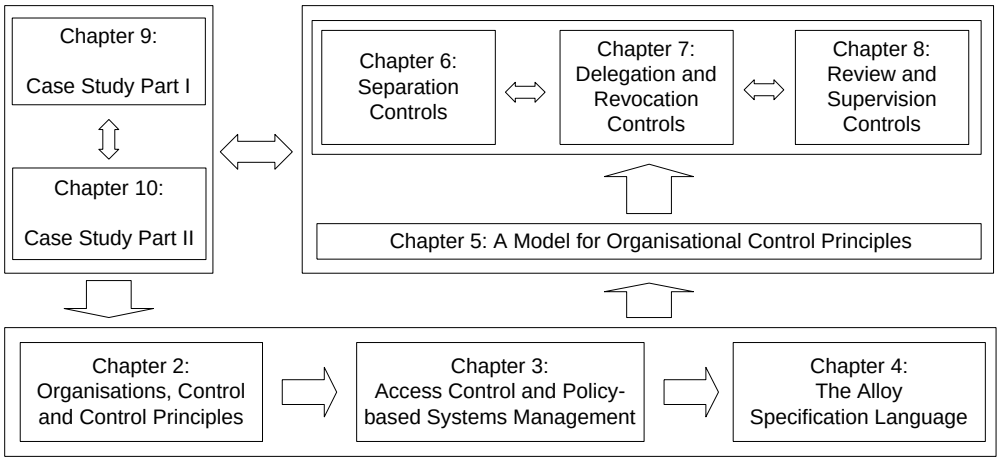

Figure 1.1: Thesis structure.

1.3 Thesis structure

This thesis is conceptually divided into three main blocks as displayed in figure 1.1, with the arrows indicating the relationships between the blocks and individual chapters. These blocks support our three stated hypotheses. Collaborative work with vendors of enterprise management software has influenced our understanding of the domain significantly.

Chapter 2 provides a review and discussion on the concepts of *organisation*, *control* and *control principles*. Organisations are determined by their goal-attainment character. *Organisational structure* will result from the efforts to reach the organisational goals. Structure may be achieved through the *division of work*, definition of *roles* and *positions*, and vertical and horizontal *relationships* between these. This structure needs to be controlled through *internal control systems*. These systems define control goals corresponding to the activity goals of the organisation. Independent of the type of organisation, there are common underlying *principles* used to establish and achieve control over these business activities.

Chapter 3 uses the established definitions and reviewed concepts of chapter 2 to investigate how these map to *access control* and more general *policy*-based systems management research. In particular we focus on *role-based* access control and here the *RBAC96* and *OASIS* approaches. The policy framework represented by the *Ponder* language is reviewed in detail, as its notion of *authorisation* and *obligation* policies relates directly to our suggested concepts. A review of *audit controls* complements this chapter.

Chapter 4 outlines our approach to finding a suitable specification language for the identified control principles. This language should be able to define *structural* as well as *behavioral* properties, and should ideally be supported by some means of automated proof or reasoning. The languages we review are the *Object Constraint Language*, the *Z* and *Object-Z* specification languages and *Prolog* logic programming language. We finally decide to use the *Alloy* specification language, a first-order logic specification language whose specifications can be made subject to *automated analysis*. We provide a tutorial on selected aspects of Alloy, and in particular demonstrate a specification technique allowing us to look at the behaviour of a specification over *sequences of states*.

Chapter 5 then uses the insights gained in the first three chapters to define a formal model for expressing control principles. The main components are *policy objects* which can be *authorisations* or *obligations*, where a *review* is a specific type of obligation. *Principals* and the *roles* and *positions* they occupy can be subject to these policies. We distinguish between the *general* and *specific* obligations of a principal. We also specify additional concepts required in the later analysis and exploration of control principles. These include *role* and *supervision* hierarchies, *exclusive roles* and approaches to maintaining a *history* of a principal's actions. This control principle model is structured in the form of separate *modules*.

Chapter 6 begins with a review of existing work on *separation* controls, and then uses our control principle model for the definition and analysis of a selected set of these controls. Based on the notions of *exclusive roles* and *critical authorisation sets* we demonstrate how separation controls may be defined following an incremental approach. We specifically assess the usefulness of Alloy for an automated *analysis of composed separation controls* and the possible *conflicts* when introducing role hierarchies.

Chapter 7 provides a detailed analysis of the *delegation* of policy objects within the context of the defined control principle framework. This sets out with the specification of a general delegation function for policy objects. This function is then used as the basis for the detailed analysis of *delegating authorisations and obligations*, considering that these may have been assigned to a principal *directly* or *indirectly* over roles. In particular, our distinction between general and specific obligations requires an in-depth analysis with respect to their delegation. We also discuss the delegation of roles and its possible effects. *Revocation* complements the delegation of policy objects, and we expand on this in the context of a recent categorisation of revocation schemes. We show how delegation, revocation and separation controls may be further analysed and explored using the Alloy constraint analyser, and discuss the obtained results with respect to the three frameworks reviewed in chapter 3.

Chapter 8 investigates the concepts of *review* and *supervision* and assesses how these may be expressed in the context of the control principle model. A review is an obligation with a previously delegated obligation as its target. *Evidence* serves as an abstraction to the discharge of an obligation. Supervision is the general obligation of a principal to review the obligations of principals in supervised positions. We demonstrate how the Alloy constraint analyser was used to further explore the delegation of obligations under those concepts.

Chapter 9 presents the first part of our case study of the large-scale, role-based access control system of a major European bank. This system defines roles as consisting of *functions* and *positions*. Here we discuss the effects of introducing function and position *hierarchies*. The control principles involved in the *administration* of the system are described before we move to the context of a *local branch* of the bank. We discuss the different types of controls we observed in this branch, and describe the specific process of a *credit application*. The control principle model of chapter 5 is partly based on the insights gained in this chapter.

Chapter 10 presents the second part of our case study. It uses the insights into the domain of a bank branch gained in chapter 9 as the basis for the assessment of the *validity* of our suggested framework and its *applicability* for control principle modelling and analysis.

1.4 Statement of contribution

A framework has been established that defines, integrates, clarifies, compares, categorises, evaluates and explores existing and new control principles. This framework supports the three theses stated in section 1.2.

We review the organisational background of control principles and believe that this thesis:

- helps in the clarification of possible interpretations of organisations and organisational structure, and identifies the notion of control and control principles in the areas of social and management sciences.
- provides a clear comparison and evaluation of three state-of-the-art role- and policy-based frameworks which partially support the expression of control principles.

We present a unified formal representation of control principles and believe that this thesis:

- provides a coherent definition and analysis of a set of separation controls that have been extracted from various, conceptually different, sources.
- provides a clear distinction between delegation as a form of administrative activity and *ad hoc* delegation between principals in non-administrative roles.
- provides a detailed analysis and discussion on revocation schemes.
- provides a novel conceptualisation of the control principle of review, defined as a specific type of obligation with a delegated obligation as its target.
- provides a novel conceptualisation of the control principle of supervision, defined as the general obligation of a principal occupying a position to review the obligations of principals in supervised positions.
- successfully demonstrates that there are relationships between control principles which have not previously received any attention.
- is among the first pieces of work to have extensively used and evaluated the Alloy specification language and subsequently published the achieved results, apart from its inventors at the MIT Software Design Group, headed by Dr. D. Jackson.

We argue that this framework can be used for the analysis and application of control principles in an organisational context and believe that this thesis:

- provides new insights into practical aspects of access control by describing a real-world access control system and the relevant control principles.
- provides an identification of control principles in the context of a credit application process, the modelling of which partially validates our framework.

1.5 Published results

Intermediate and directly relevant results of this thesis have been published in the following order:

1. Schaad, A. and J. Moffett. The Incorporation of Control Principles into Access Control Policies (Extended Abstract only). Invited talk at Hewlett Packard Policy Workshop. 2001. Bristol, UK.
2. Schaad, A., J. Moffett, and J. Jacob. The access control system of a European bank - A case study. 6th ACM Symposium on Access Control Models and Technologies. 2001. Chantilly, VA, USA.
3. Schaad, A. Conflict Detection in a Role-based Delegation Model. 17th IEEE Annual Computer Security Applications Conference. 2001. New Orleans, USA.
4. Schaad, A. and J. Moffett. Delegation of Obligations. 3rd International Workshop on Policies for Distributed Systems and Networks. 2002. Monterey, CA, USA.
5. Schaad, A. and J. Moffett. A Lightweight Approach to Specification and Analysis of Role-based Access Control Extensions. 7th ACM Symposium on Access Control Models and Technologies. 2002. Monterey, CA, USA
6. Schaad, A. and J. Moffett. A Framework for Organisational Control Principles. 18th IEEE Annual Computer Security Applications Conference. 2002. Las Vegas, USA.

Indirectly related and extended collaborative work in the area of role-based access control and enterprise management has been published in:

7. Kern, A., M. Kuhlmann, A. Schaad, and J. Moffett. Observations on the Role life-cycle in the context of Enterprise Security Management. 7th ACM Symposium on Access Control Models and Technologies. 2002. Monterey, CA, USA.
8. Kern, A., A. Schaad, and J. Moffett. An Administration Concept for the Enterprise Role-Based Access Control Model. 8th ACM Symposium on Access Control Models and Technologies. 2003. Como, Italy.

1.6 Chapter summary and conclusion

This chapter has provided an initial motivation for our work and defined three underlying theses in sections 1.1 and 1.2. These will be addressed following an explorative approach, leading to the definition of a framework for organisational control principles.

An initial summary of the structure of this framework and thesis was provided in section 1.3, together with a statement of contribution and listing of published results in sections 1.4 and 1.5.

Chapter 2

Organisations, Control and Control Principles

2.1 Introduction

The overall context of this thesis is the *formal organisation* as opposed to an *ad hoc* organisation. Work is divided among principals in specific roles and positions according to the attainment of a common goal. Relationships between roles, positions and principals emerge. Rules and procedures constrain the behaviour of principals, often in the form of specific workflows. This decentralisation, and the resulting hierarchical structures and workflows across these hierarchies, requires the definition of control goals and their enforcement. Many organisations thus choose to implement explicit systems of internal control such that the goal which motivates the existence of the organisation can be achieved most efficiently. Control principles are the general constraints placed upon an organisation, its structure, processes, activities of principals and supporting information systems.

While it is not possible to describe and discuss the complex characteristics of organisations in full detail, this chapter shall give a brief review of relevant work with respect to formal organisations. The emphasis is to investigate some of the concepts established in management and organisational science required for the later definition of a control principle model. The following issues are discussed in order to establish some of the necessary ground for the rest of this thesis:

1. An initial discussion of the concepts of organisation and organisational structure is provided in section 2.2;
2. A set of basic organisational principles is summarised in section 2.3;
3. Different aspects of organisational structure are discussed in section 2.4;
4. Different types of organisational control and internal control systems are illustrated in sections 2.5 and 2.6.
5. A definition of control principles and their source is part of section 2.7.

2.2 On the definition of (an) organisation

There is no general definition of the term organisation that will satisfy all the different organisational schools of thought such as the scientific management school established by Taylor, Fayol's administrative school, or the bureaucratic school of Weber, all of which and others are compared in [Strati, 2000]. Nevertheless, there are some general observations that may help to understand why this is the case and provide a basic definition that will be sufficient for the context of this thesis.

A commonly established view is to understand organisations as goal attainment systems. Parsons defines organisations as special types of social systems characterised by their "..primacy of orientation to the attainment of a specific goal." [Parsons, 1970]. A similar view is expressed by Elliot, seeing an organisation as "..a co-ordinated body (or system) of individuals (and perhaps machines) arranged to reach some goal or perform some function or service." [Elliot, 1980]. This may be generalised, saying that organisations emerge when there is a shared set of beliefs about a state of affairs to be achieved among people. In order to do so, work is distributed and patterns develop in the relationships between people to coordinate this division of labour [Galbraith, 1977]. This structure, often perceived in the form of hierarchies of authority, is what emerges when organisations are formed, and Pugh links the concept of organisational goal attainment to the concept of structure by saying that "All organisations have to make provision for continuing activities directed towards the achievement of given aims. Regularities in activities such as task allocation, supervision, and co-ordination are developed. Such regularities constitute the organisation's structure, and the fact that these activities can be arranged in various ways means that organisations can have differing structures." [Pugh and Hickson, 1973]. So, similar to there being no general definition of an organisation, the concept of structure will have to be understood differently depending on the context. These observations may now be summarised by defining organisations as being "..composed of people and groups of people; in order to achieve some shared purpose; through a division of labour; integrated by information-based decision processes; continuously through time." [Galbraith, 1977]. We adopt this as our working definition.

Size and complexity of most organisations raise questions about how the activities of people are controlled. More drastically, as Etzioni puts it, "..organisations [..] cannot rely on [..] their participants to internalize their obligations, to carry out their assignments voluntarily." [Etzioni, 1964]. While we support this view, we emphasise that people may also perform some tasks in a way they believe to meet the organisation's objectives, while in fact their behavior actually opposes them. Additional incentives and controls are required to address this. Some controls may be part of the structural design, while others need to be stated explicitly in the form of rules and output controls.

We now investigate some criteria of organisational structures in more detail in section 2.4, followed by a discussion on organisational control and control systems in sections 2.5 and 2.6. However, a list of organisational principles is first presented in the following section 2.3 to emphasise how classic management theory still influences today's design of automated systems.

2.3 The principles of organisation

Since the beginning of the 20th century research has tried to explain the phenomenon of organisations and the underlying principles that determine their existence. According to Urwick [Urwick, 1952] and later adapted in [Mullins, 1999], an organisation is built on ten principles:

1. The principle of the objective
   Every organisation and every part of the organisation must be an expression of the purpose of the undertaking concerned, or it is meaningless and therefore redundant.
2. The principle of specialisation
   The activities of every member of any organised group should be confined, as far as possible, to the performance of a single function.
3. The principle of co-ordination
   The purpose of organising per se, as distinguished from the purpose of the undertaking, is to facilitate co-ordination: unity of effort.
4. The principle of authority
   In every organised group the supreme authority must rest somewhere. There should be a clear line of authority to every individual in the group.
5. The principle of responsibility
   The responsibility of the superior for the acts of the subordinate is absolute.
6. The principle of definition
   The content of each position, both the duties involved, the authority and responsibility contemplated and the relationships with other positions should be clearly defined in writing and published to all concerned.
7. The principle of correspondence
   In every position, the responsibility and the authority should correspond.
8. The principle of span of control
   No person should supervise more than five, or at most, six direct subordinates whose work interlocks.
9. The principle of balance
   It is essential that the various units of an organisation should be kept in balance.
10. The principle of continuity
    Re-organisation is a continuous process: in every undertaking specific provision should be made for it.

Comparing this list to our everyday perception of the organisations we are a part of, it becomes apparent that these principles have not lost much of their validity. We will refer to some of these principles during the course of this thesis to emphasize that the design of automated controls should be accompanied by an understanding of management sciences.

2.4 Organisational structure

In the previous section 2.2, structure has been described as what coordinates the activities of people in organisations. In a more comprehensive definition Merton summarises this by saying that "A formal, rationally organized social structure involves clearly defined patterns of activity in which, ideally, every series of actions is functionally related to the purposes of the organization. In such an organization there is integrated a series of offices, of hierarchized statuses, in which inhere a number of obligations and privileges closely defined by limited and specific rules. Each of these offices contains an area of imputed competence and responsibility." [Merton, 1952]. There are further criteria that can be used to determine dimensions of structure, some of which have already been described by others as principles of an organisation in the previous section 2.3. Child describes six major components of organisational structure [Child, 1988] the most important of which are the grouping together of sections, departments, divisions and larger units; definition of formal reporting relationships, levels of authority and spans of control; the allocation of individual tasks and responsibilities, job specialisation and definition; the delegation of authority and procedures for monitoring and evaluating the use of discretion.

According to the previous definitions, the rest of this section will concentrate on a brief discussion and outline of three key aspects of organisational structure which must of course be understood as being interrelated:

1. The division of work through delegation;
2. Roles and positions;
3. Vertical and horizontal relationships.

2.4.1 Structure through the division of work

Often, the perception of an organisation is determined by the 'organisation chart'. This chart usually reflects the basic differentiation of an organisation, showing how tasks have been divided according to some criteria. This is what may be described as the 'basic structure' of an organisation [Lorsch, 1970]. For example, a frequent criterion according to which this distribution of work may be performed is a specific function such as Engineering or Accounting. Equally often the product or offered service is used. Other criteria could be of a geographical nature, e.g. north and south branches of a bank, or customers to be served, e.g. private and business customers. It is not difficult to see how characteristics such as the customer to be served can have a direct impact on security, e.g. in the case of specifying Chinese Wall policies [Brewer and Nash, 1989] or in the design of role-based systems as we showed in [Kern et al., 2002, Kern et al., 2003]. A detailed analysis of the impact of structural choices to security design and administration is however out of this scope.

The main reason for division of labour is to handle increasing organisational size and complexity. Delegation is the technical means by which division of labour is achieved

[Johnson and Gill, 1993]. This includes the delegation of the needed authority to perform these tasks, and there are several management paradigms according to which delegation may be performed and controlled [Mullins, 1999]. This delegation of tasks and subsequent division of labour directly relates to the dilemma of centralisation and decentralisation. Centralising tasks among a small set of people will have the obvious effect of easier coordination of organisational activities but will inevitably result in bottlenecks. Decentralising certain tasks reduces the individual's workload on the expense of coordination and control efforts.

2.4.2 Structure through roles and positions

Roles and positions have been recognised as a concept to connect the individual member to the organisation's prescriptions and dictates [Salaman, 1980]. A definition which best matches our understanding of roles and positions in this context is made by Pugh, saying that "A role is a set of expectations and behaviours associated with a given position in a social system." [Pugh, 1966]. We make this our working definition for the rest of this thesis.

Within the context of organisations, the term position must be looked at from several perspectives. Positions relate to that part of the job description of a principal where the rank he holds within the organisation is described. His position will place a principal in a network of command and control relationships. On the basis of this rank it is defined what orders and commands he can give and has to obey to, as well as to and from whom these are directed and originate. This authority and duty, as well as further information such as competence expressed through qualifications and levels of experience, are usually part of a job description [Buehner, 1994], the significance of which for access control management has been described in [Roeckle et al., 2000].

Although often seen as identical, roles may be distinct to positions as they can be defined as a set of task-related behaviours required of a principal by his position in an organisation [Lupu and Sloman, 1997]. This behaviour is often referred to as the duty or responsibility of a role, and to fulfill these duties a role also describes the needed authority to do so. This view is also described by Biddle seeing a role as a "..collection of rights and duties.." relating to a position [Biddle, 1979]. We will later provide a context specific discussion on our use of roles and positions in section 5.5. However, we believe that it is not necessary to insist on a generally accepted definition of these terms. The case study we present in chapter 9 supports this view, showing the very specific definition of roles and positions that applies in the context of an organisation with 50,000 employees, which may, however, be of little use in other organisations [Schaad et al., 2001].

While being a useful concept to achieve structure, roles can conflict. The simultaneous occurrence of two or more roles and compliance of a principal with either could make the compliance with the other more difficult and "In the extreme case, compliance with one expectation [..] would exclude completely the possibility of compliance with the other; the two expectations are mutually contradictory." [Katz and Kahn, 1966]. This is in fact an early identification of the concept of mutually exclusive roles, we will later use for the expression of separation controls in chapter 6.

2.4.3 Structure through vertical and horizontal relationships

The previous section identified the position as one of the most basic organisational elements. Depending on the given organisational context, a position may define the competence, authority, duty and responsibility of the principal occupying the position. Linking the positions and creating relationships between them, a distinction may be made between the vertical and horizontal relationships.

Vertical relationships are often termed as line; function; or staff relationships, as, for example, described in [Mintzberg et al., 1998] and [Mullins, 1999]. They will receive their respective meaning in the actual organisational context. These relationships define who interacts with whom and what this interaction entails. Usually this interaction is thought of in terms of superior-subordinate relationships where people give and receive commands. Of course there are always informal relationships which are, however, not considered here.

Horizontal relationships define the workflow structured aspect of organisational relationships [Pugh and Hickson, 1973]. People interact on the basis of a defined set of procedures, sometimes also referred to as workflows or business processes. Such a workflow might be the application for a credit, which is initially recorded by a clerk, then given to a credit evaluation specialist and then to a customer service representative. Such workflows have a direct impact on access control in automated systems, e.g. [Atluri and Huang, 1996] and [Bertino et al., 1997a], but there is no space to discuss general or security related workflow aspects of an organisation in more detail. An excellent introduction with respect to workflows and automated systems is given in [Schael, 1996, Cichocki et al., 1997, Grefen et al., 1999].

2.5 Organisational control

Control is a central organisational function and results out of decentralisation efforts. It is the means by which activities and resources are coordinated and directed towards the achievement of an organisation's goal and implies a degree of monitoring and feedback. Salaman et al. argue that "Control means that members of the organisation have their actions, [..] determined, or influenced, by membership of the organisation." [Salaman and Thompson, 1980].

It is impossible to consider organisational control without considering the nature of power within organisations, how it is distributed and how it originates [Salaman and Thompson, 1980]. The source of authority has been discussed in the classic text of Weber [Weber, 1947]. When there is a high degree of formalisation, then power is delegated in an organisation. The defined rules and procedures will limit what others can do and constrain how authority is delegated [Hickson and McCullough, 1980]. This source of authority and its distribution has also been described in the context of automated systems [Moffett and Sloman, 1988], [Moffett, 1990]. Rueschemeyer further argues that delegation of authority puts continued central control under question and provides a discussion on the dilemma of delegating authority and the retention/maintenance of control [Rueschemeyer, 1986].

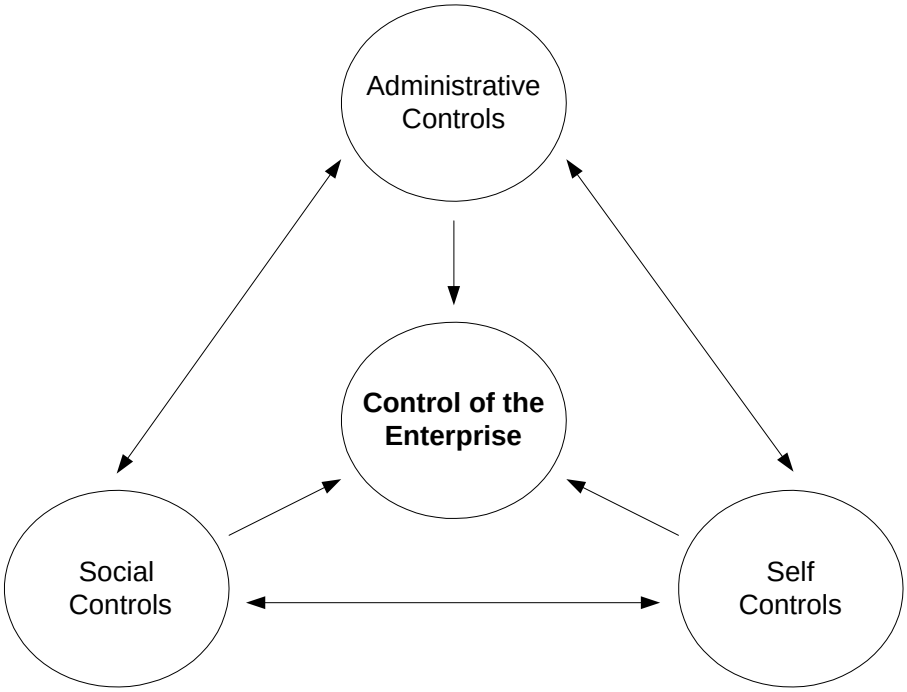

Figure 2.1: Hopwood's three types of control.

Similar conceptual schemes have been developed by Dalton [Dalton and Lawrence, 1971], Hopwood [Hopwood, 1974] and Mintzberg [Mintzberg, 1979], identifying different types of control influence in organisations. The most intuitive in this context is Hopwood's classification, distinguishing between administrative; social; and self controls as shown in figure 2.1. Control of the organisation may then be understood and explained in terms of these types.

However, the focus of this thesis is on administrative controls, which may be defined as "..those mechanisms, techniques and processes that have been consciously and purposefully designed in order to try to control the organizational behaviour(s) of other individuals, groups and organizations." [Johnson and Gill, 1993]. Two means by which administrative control may be achieved is through

- rules and procedures
- output controls

Rules and procedures are the most explicit form of administrative control. One specific context in which such rules become inherently apparent is in the analysis and definition of business processes [Mintzberg, 1979], commonly referred to as workflows. Here tasks can be broken down and be ultimately captured in the form of rules and procedures constraining the range of members' behaviour; increasing the predictability of their actions; and increasing the probability that perceived organisational requirements dominate that behaviour [Johnson and Gill, 1993]. Specifically with the recent trend to establish (partially) automated workflows such as credit applications, insurance claims, material ordering etc., this form of control has gained in importance [Cichocki et al., 1997].

On the other hand when it is not possible (or feasible) to define rules because of the complexity and unpredictability of the tasks that are performed, then output controls need to be established. The everyday accomplishment of tasks is left to the members' judgement and discretion. This requires the ability to define and measure these outputs, as well as taking appropriate corrective or adjusting actions.

2.6 Internal control

Internal control may be defined as the policies, procedures, practices and organisational structures designed to provide reasonable assurance that business objectives will be achieved, and that undesired events will be prevented or detected and corrected. [ISACA, 2000].

Just like organisations have goals describing their primary business objectives, they also have goals with respect to controlling how these objectives are met. These are the control goals of an organisation, which are enforced through a system of internal control. Such a system shall enable them to adhere to external laws and internal regulations, prevent and detect fraud, and continuously enhance the overall quality of the business. The term internal control originated in the early days of modern accounting [Bailey et al., 1983]. It was seen as the natural counterpart to the term external control which describes the auditing of a companies accounts by external, independent auditors. Internal control was enforced through a set of paper-based procedures which should ensure reliable financial reporting and detection of fraud. However, during the 1990's the term internal control had to be redefined, as existing definitions became inappropriate with the advent of integrated information systems [Jajodia, 1997, Moeller, 1997, Strous, 1997].

Internal control is usually enforced following a systems approach. Such an internal control system is based on a constant process of establishment of the appropriate controls; assessment of systems and procedures; evaluation of the results; and adjustment of controls if necessary. It is a system determined by the preventive, detecting and corrective character of its controls [Weber, 1998]. However, establishing an internal control system becomes increasingly more difficult with today's trend to decentralise and the subsequent delegation of authority [Rueschemeyer, 1986, Pugh, 1997]. This leads to the requirement of designing control systems that reflect these delegation and resulting responsibility structures. The administration of an internal control system is thus not the sole responsibility of top-management and auditors but also of the individual business owners throughout the entire organisation.

Subsequently, a new set of control models and frameworks emerged in the 1990's, each addressing the issue of control within a specific domain. Many of these such as the COSO in the US [COSO, 1992], CoCo in Canada [CICA, 1995] and Cadbury in the UK [LSE, 1999] provide a control framework under the background of standard organisational accounting and audit. Others, such as the DIT Security Code of Conduct in the UK, the NIST Security Handbook in the US [NIST, 1999], or the Common Criteria [CommonCriteria, 1999], primarily address the aspect of how to achieve control and evaluate security within IT systems. Frameworks such as COBIT [ISACA, 2000] try to bridge what is perceived as the gap between

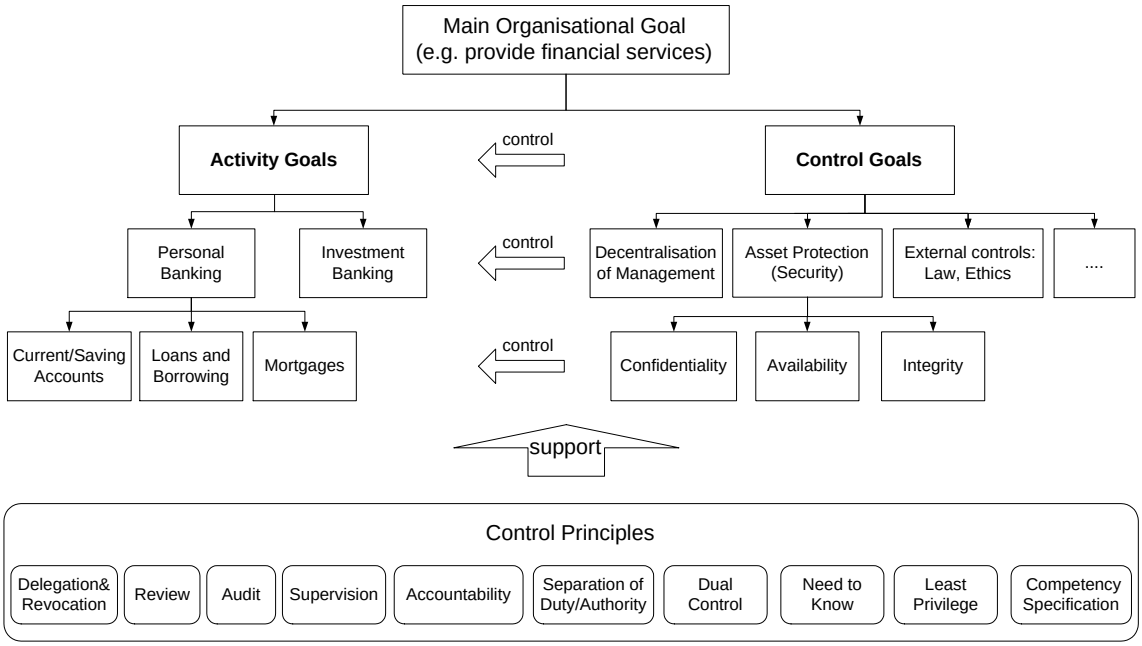

Figure 2.2: Organisational goal hierarchy and control principles.

the above two categories of business and IT control models. Its emphasis is on achieving an IT governance, providing the structure that links IT processes, IT resources and information to enterprise strategies and objectives. A comprehensive summary and comparison of the COBIT, SAC, COSO and SAS 55/78 frameworks is provided in [Colbert and Bowen, 1996].

We think that these models and frameworks may be useful when applied in the context of a specific organisation. They allow organisations to establish a first record of what forms of control have been defined or evolved, and how these may have to be adjusted. However, we did not find any practical use for any of them in the more abstract context of this thesis.

2.7 Control principles

As outlined in the previous section 2.6, an organisation always tries to achieve a top-level goal. Depending on the kind of organisation this might be a goal expressing that, for example, profits need to be made by providing financial services. Such top-level goals are too abstract to be directly achieved and they have to be refined into further sub-goals, which themselves are further refined until they reach an operational level, i.e. they can be executed.

We believe that in this context organisational goals may be divided into two main categories: Activity and control goals. The first category considers those goals that address the main activities of a company, such as personal and investment banking or stock trading. The second category addresses those goals which are established in order to achieve control over these main activities. Typical sub-goals in this category are the decentralisation of management, asset protection and adherence to external ethical, ecological or law-based constraints. The system of internal control enforces these goals. The upper half of figure 2.2 shows such a goal hierarchy where the activity goals are examples of the services offered by Barclays Bank.

Independent of the type of organisation there are common underlying principles used to establish and achieve control over business activities [Cresson Wood, 1990, Swanson and Guttman, 1996, Pfleeger, 1997, Moffett, 1998, Anderson, 2001]. The terms with which these principles are commonly referred to are the:

1. Separation of duty and dual control;
2. Delegation and revocation of authority and obligations;
3. Supervision, review and audit;
4. Specification of accountability and competence;
5. Least privilege and need to know.

In the context of this thesis we refer to control principles as the general constraints placed upon an organisation, its structure, processes, activities and supporting information systems. Control principles emerge from the organisational control goals.

At this stage a more precise definition or categorisation is not desirable, and it is very likely that there are other principles we did not mention, as well as differing interpretations of the same terms. We also do not give any specific recommendations on how to identify control principles. The only requirements for control principle identification and specification are that there should be an underlying formal organisational model, and a control principle expressed in this model should ideally show clear relationships to one or more other control principles, although there may be control principles which are completely orthogonal.

A subset of the control principles listed above will be defined in an appropriate conceptual model in chapter 5. These are the principles of delegation, revocation, separation, supervision and review. All of these show specific relationships to each other. The other principles listed in figure 2.2, although frequently referred to in organisational control manuals, have not been integrated into the suggested control principle framework for varying reasons.

For example, the principle of dual control is a specific case of a separation control and can be easily expressed in our suggested model if required. The principle of need to know is what we believe to be a specific policy dependent on the labelling of principals and objects. If desired this could be easily integrated into our model, but this is not of imminent interest. A principal should always have the least privileges required to perform his duties. Since our model includes a notion of obligations and authorisations this can be modelled and analysed if required, but we do not discuss the principle of least privilege in more detail in this context. The accountability of a principal and specification of his competence seem to directly relate as no principal should be held accountable for what he is not competent enough to do. However, we believe that the specification of competence requires research beyond the assignment of attributes representing competence. What has not yet been fully resolved by current research is the process of how these attributes are methodically determined. Thus we do not further investigate these two principles in this context.

2.8 Chapter summary and conclusion

This chapter has established the ground for some of the discussions in the rest of this thesis. Our main emphasis was on showing the influences of management and organisational science literature to our work. In particular, we discussed:

- that there can be no general definition of the term organisation in section 2.2. In this context we, however, use Galbraith's definition saying that an organisation is "..composed of people and groups of people; in order to achieve some shared purpose; through a division of labour; integrated by information-based decision processes; continuously through time." [Galbraith, 1977];
- that organisational structure may, among others, be achieved through the division of work; roles and positions; and vertical and horizontal relationships on the basis of the definitions given by [Merton, 1952] and [Child, 1988] in section 2.4;
- that control may take many forms in section 2.5. We focus on administrative controls in the form of rules and procedures as well as output controls [Johnson and Gill, 1993];
- that control is enforced in systems of internal control in section 2.6;
- that control principles are general constraints placed upon an organisation, its structure, processes and activities in section 2.7. They are derived from the control goals that support the business activities of an organisation.

We now review selected work in the area of access control and policy-based systems management in chapter 3. This work reflects and supports some of the organisational concepts established in this chapter. This is then followed by the selection and introduction of the Alloy specification language in chapter 4, which is appropriate to define and discuss a selection of control principles as part of chapters 5, 6, 7, 8 and 10.

Chapter 3

Access Control and Policy-based Systems Management

3.1 Introduction

Decentralisation implies the delegation of work and with it the needed authority to be able to perform the arising obligations. In today's organisations this authority is partially expressed in the access rights of a principal within the IT infrastructure used for processing information resources, e.g. access rights at the network-; operating system-; or application-level. Access needs to be controlled as authority may be used inappropriately by principals. As such access control is one of the main mechanisms to support the confidentiality, availability and integrity of an organisation's information assets.

An access control system should correspond to the structure of an organisation. Since roles have been identified as one main aspect of organisational structure, this has established a field known as role-based access control. To enforce further control over principals acting in these roles, explicit rules and policies have to be defined. These do not only consider the access rights of a principal but also his obligations. The expected behaviour of access control systems is asserted by means of a continuous audit which also includes the logging of the activities of principals.

This chapter provides a brief review of the field of access control in section 3.2. This is followed by a summary and discussion of role-based access control in section 3.3, policy-based management of systems in section 3.4 and audit in section 3.5. Three models described in particular are:

1. The RBAC96 role-based access control model family and its extensions in section 3.3.1.
2. The OASIS role-based architecture for secure interoperation of services in distributed systems in section 3.3.2.
3. The Imperial College policy framework, recently summarised by the Ponder language in section 3.4.1.

3.2 Access control

Access control is commonly understood as the mechanisms and processes involved in the mediation of every request to resources and data maintained by a system and determining whether the request should be granted or denied [Samarati and Vimercati, 2001][1]. This mediation must be performed by a trusted reference monitor [Anderson, 1972] and any access control system must be supported by authentication and auditing mechanisms as shown in figure 3.1 [Sandhu, 1996].

A common approach is to distinguish between mandatory and discretionary access control [Pfleeger, 1997]. Mandatory access control is based on the regulations mandated by a central authority and is commonly represented through multilevel security policies based on the classifications of system objects and subjects, e.g. [Bell and La Padula, 1973], [Biba, 1975]. Discretionary access control is based on the identity of subjects, and there are explicit access rules about who may or may not perform certain actions on system resources. Discretionary also means that subjects may pass on their authority to other subjects. This granting and possible revocation is then controlled through specific policies, a summary of which can be found in [Castano et al., 1994].

The access matrix model described in [Lampson, 1971] was the first framework to describe discretionary access control. The columns of the matrix correspond to objects, rows to subjects and entries to the privileges of a subject on an object. The state of this matrix at some moment in time is also referred to as the authorisation state, defining what a subject can do. The HRU model [Harrison et al., 1976] later formalised the rules according to which this matrix evolves. This is done in the form of a set of operations that change the entries of the matrix, as well as operations for adding or removing objects or subjects. This flexibility gives rise to the safety problem, and in general, establishing whether a certain state can be reached, i.e. whether a certain access can be granted, is undecidable. To overcome this problem, two approaches have been taken [Tidswell and Jaeger, 2000]. The access control model may be restricted such that safety can be proven, e.g. [Jones et al., 1976], or the access control model is augmented with expressions describing the safety requirements of any configuration such that access can be verified, e.g. [Bertino et al., 1999a].

Two points have to be considered with respect to the access matrix:

1. Any abstract, higher level representation of access authority, e.g. through roles, can ultimately be displayed in the form of an access control matrix.

2. Due to other system constraints the authorisation state, i.e. what a subject can do, might be a subset of the authority state, i.e. what a subject may do. In other words, although an access right has been granted to a principal by an administrator, the current situation might not allow him to access an object due to, for example, some previously performed access on the same object, e.g. [Brewer and Nash, 1989] or other context dependent criteria, e.g. [Bacon et al., 2002, Covington et al., 2001].

[1] Compare this mediation with the sociological definition of control as the ability to initiate and terminate actions at one's discretion as given by Pfeffer and Salancik [Pfeffer and Salancik, 1997].

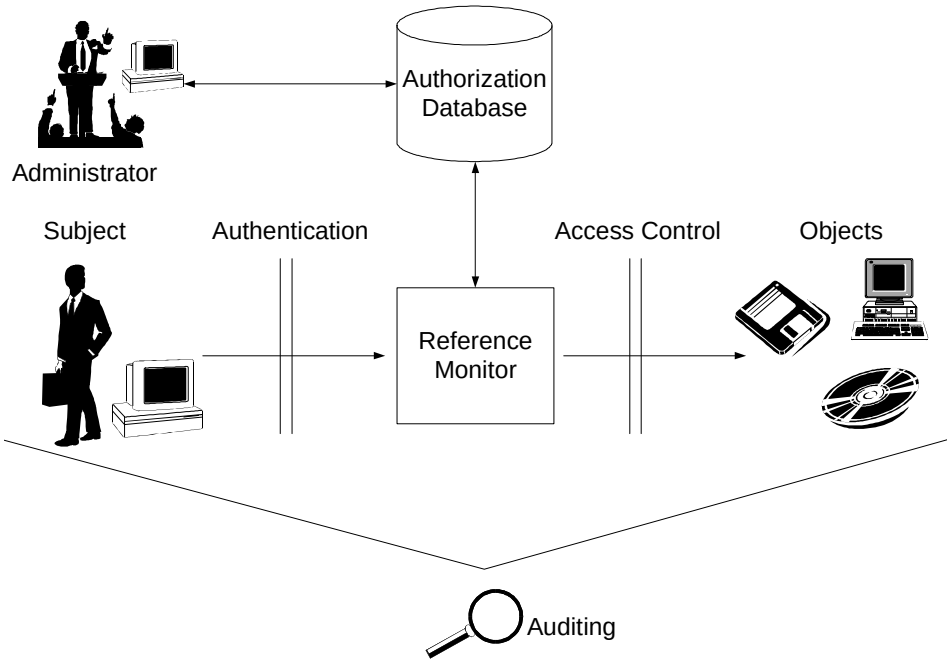

Figure 3.1: Authentication, Access Control and Audit.

For a more complete summary of access control models and policies; vulnerabilities of mandatory and discretionary policies; forms of implementation of the access matrix; granting and revocation of access rights; the safety problem; conflict detection in access control policies and their resolution; information flow analysis; and further current research directions, the introductory textbooks of [Amoroso, 1994, Pfleeger, 1997, Anderson, 2001] and the tutorial given in [Samarati and Vimercati, 2001] are referred to.

While this thesis is not concerned with the actual mediation and enforcement of access requests, it investigates structural aspects of access right representation, i.e. the authority state, as well as the effects of a principal's behaviour on the authorisation state.

In the following sections we summarise the RBAC, OASIS and Ponder frameworks as all three incorporate a notion of roles. However, each does this with a slightly different understanding and model-specific characteristics that will have to be investigated. These models also provide the means for expression of simple rules or policies which determine the authorisation and authority state. Additionally, they all integrate, or are the basis for delegation mechanisms, the nature and meaning of which also needs to be explored.

3.3 Role-based access control

3.3.1 The RBAC96 - NIST standard

Much of the work on role-based access control systems is based within the context of the RBAC96 access control model family [Sandhu et al., 1996], which since then has evolved to a NIST standard [Ferraiolo et al., 2001] allowing for a standardised integration into security products.

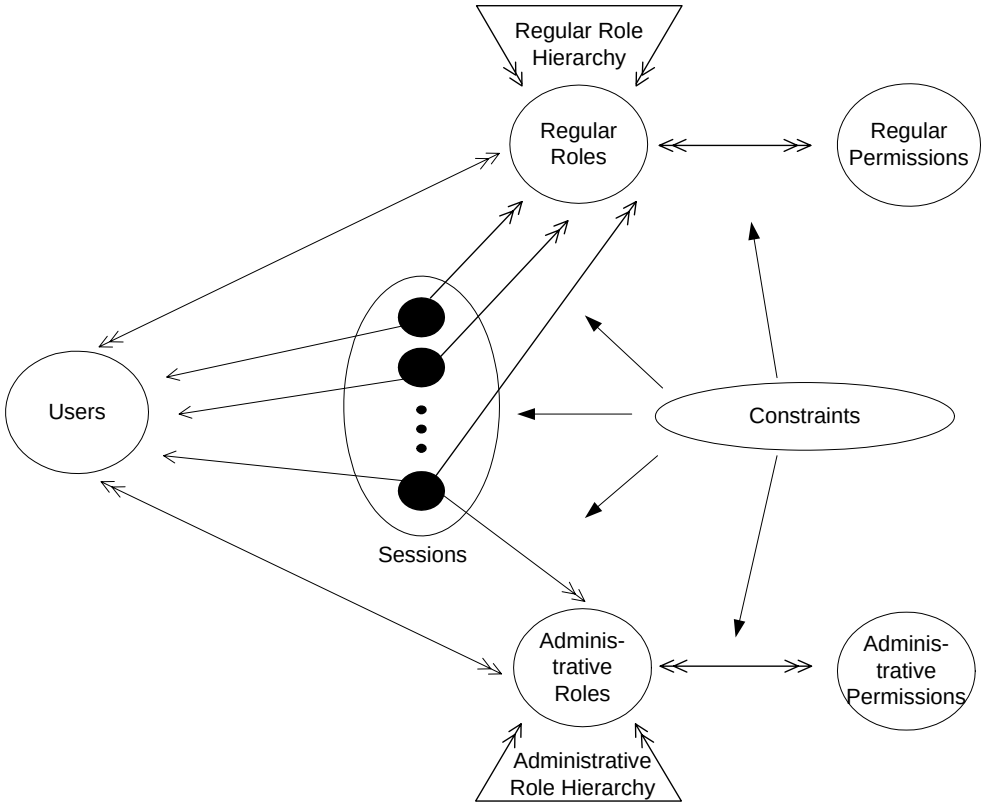

Figure 3.2: RBAC96 and ARBAC97 model families.

In the RBAC96 model family, the central notion is that permissions are associated with roles, and users are made members of appropriate roles, thereby acquiring the roles' permissions. The model family consists of four sub-models, gradually adding functionality to the basic model consisting of user-, role-, permission entities and the relations between them.

RBAC96 is seen as policy neutral and can be used to enforce different types of policies, e.g. mandatory multilevel policies [Sandhu, 1996]. The kind of policy it supports depends on the configuration of the different RBAC model components such as the role hierarchies, constraints, and user/role and role/permission assignments as indicated in the upper half of figure 3.2.

A user is a representative of the real world that can be held responsible for an action. Roles are named job functions within an organisation. A user can be assigned to many roles and a role can be assigned to many users. Sessions are mappings of one user to possibly many roles. A user might open several sessions, e.g. several windows running an application on a workstation, in which he activates a subset of the roles he is member of, thus enforcing the principle of least privilege. Multiple roles are simultaneously activated while a session must always be uniquely related to an individual user. A permission is an approval of a particular mode of access to one or more objects in the system. Role hierarchies are used to represent an organisation's lines of authority and responsibility. Senior roles inherit the associated permissions from a junior role. They can be considered as constraints in so far as that permissions assigned to a junior role must always be assigned to all senior roles. Constraints are used to enforce certain control principles such as the separation of duties.

They are predicates that return a value of "acceptable" or "not acceptable", when being applied to a user or role function or the assignment relations. Common constraints include mutually exclusive roles, cardinality constraints and prerequisite roles. We used selected aspects of RBAC96 for our initial attempts in establishing a control principle framework [Schaad and Moffett, 2001, Schaad, 2001, Schaad and Moffett, 2002a]. The RBAC96 model has also been the basis for our collaborative work on role engineering [Kern et al., 2002].

Based on the RBAC96 model, several extensions have been proposed, where those relevant to the control principle context of this thesis consider aspects of role-based constraint expression and role-based administration and delegation.

Initial attempts to express constraints in the context of the RBAC model are documented in [Chen and Sandhu, 1995]. Later, the RCL2000 language was presented with the specific intention of specifying and analysing role-based constraints [Ahn, 2000]. It is based on the RBAC96 model family and uses sets combined with defined functions for the expression of constraints. Constraints are categorised as prohibition and obligation constraints, defined by *cannot_do* and *must_do* rules. Prohibition constraints are constraints that forbid a RBAC component to do something which is not allowed. Separation of Duty constraints belong to this category. Obligation constraints force a component to do something. Constraints identified in the simulation of lattice-based access control and Chinese Wall policies belong to this category.

The administrative role-based access control model ARBAC97 [Sandhu et al., 1999] expresses the idea of using RBAC to manage RBAC through decentralisation of administrative authority. A distinction is made between regular and administrative roles and permissions as shown in the lower half of figure 3.2. ARBAC97 consists of three sub-models. These describe the decentralised administration through user-role assignment (URA97), permission-role assignment (PRA97) and role-role assignment (RRA97). Two central concepts of ARBAC97 are the *administrative range* and *prerequisite conditions*. They regulate and impose restrictions on the administration of system objects. The administrative range reflects the set of roles over which an administrator has authority. Depending on the context he can assign and revoke users to or from a role, alter role hierarchies, and assign or revoke permissions. In case of a user-role assignment, a prerequisite condition could be used, e.g. to express that any user to be assigned to a role *r1* must already be assigned to another role *r2*. Some technical limitations of this approach have been investigated by using different forms of formal representation [Crampton and Loizou, 2002, Schaad and Moffett, 2002b]. We have also provided a critical evaluation of the ARBAC approach in the context of applying ARBAC in the practical management of Enterprise roles [Kern et al., 2003].

Recently, a framework for role-based delegation models based on RBAC96 has been presented [Barka, 2002]. This framework provides a discussion on how roles may be delegated by users not explicitly acting in administrative roles as in ARBAC97.

We will provide a more in depth and critical review on the RBAC96 model and its extensions in chapters 5, 6 and 7.

3.3.2 The OASIS framework

OASIS is a role-based access control architecture for achieving secure interoperation of services in distributed systems [Hayton et al., 1998]. It is linked to the Cambridge Event Architecture (CEA) [Bacon et al., 2000], allowing for the constant monitoring of system entities. The declared aim of OASIS is to allow autonomous management domains to specify their own access control policies in accordance with an agreed set of service level agreements between domains [Bacon et al., 2002]. Role-based means that privileges are associated with a role rather then being directly related to an individual principal. Services name their client roles and define and enforce role activation and authorisation policies, controlling the user/role and role/privilege mappings respectively. Having activated a role, membership rules define predicates that must remain true for the time of activation, thus maintaining an active security environment. Like RBAC96, OASIS is session based and may require a possible prerequisite initial role such as `authenticated user` on the basis of which further roles may be activated.

There are, however, a number of differences to the RBAC96 model described in the previous section 3.3.1. These differences have been summarised in [Bacon and Moody, 2002]:

- Roles are service-specific; there is no need for a globally centralized administration of role naming and privilege management;
- Roles may be parameterized, as required by applications;
- There is no explicit role hierarchy, hence no inheritance of privileges;
- Roles are activated within sessions by presenting the credentials specified by the policy;
- OASIS provides an active security environment facilitated by session-limited privilege allocation. Conditions checked during role activation can include constraints on the context; roles are deactivated if membership conditions subsequently become false.

The basic architecture of an OASIS service for the definition and activation of roles has been presented in [Bacon and Moody, 2002]. Figure 3.3 has been adopted from this and a possible role activation scenario is described in the following where the item numbers correspond to those in the figure:

1. A principal activates a role by presenting credentials to a service.
2. The service validates the credentials in accordance with existing policies and, on success, issues a role-membership certificate (RMC) at the same time generating a credential record.
3. The RMC may then be used by the principal for further service requests which must meet the constraints of any existing authorisation policies such that
4. access to the service is granted.

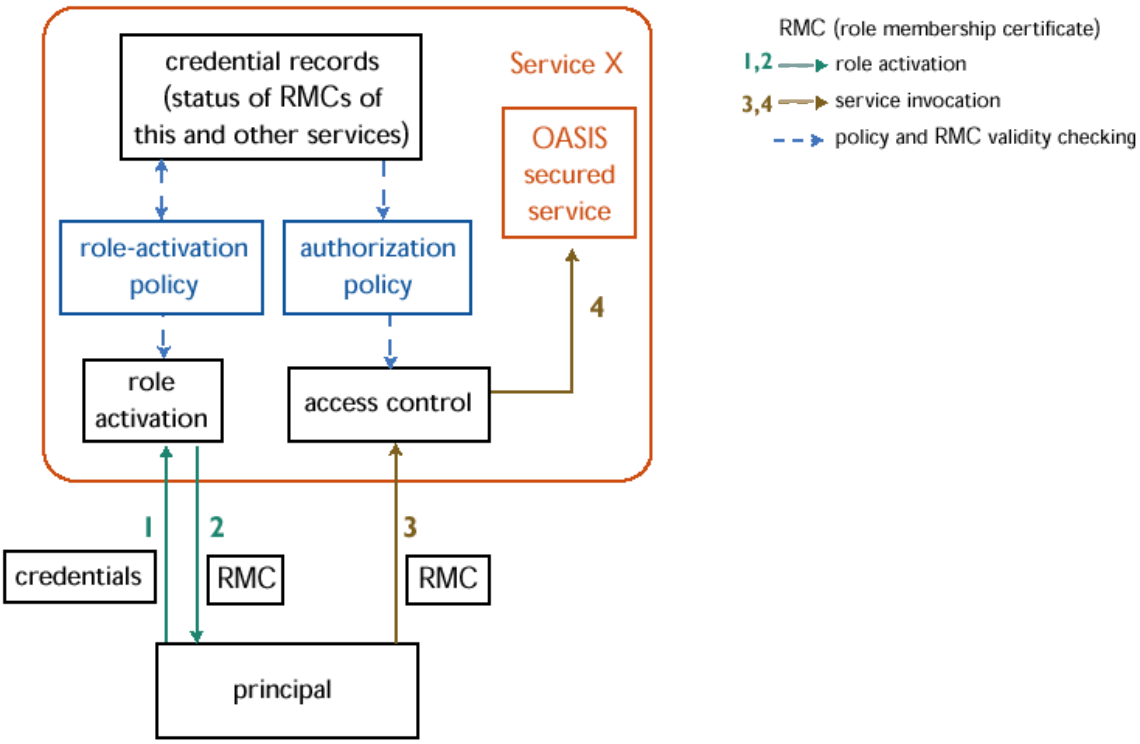

Figure 3.3: OASIS framework architecture.

An extension to the basic OASIS framework is the concept of appointment as a specific case of role delegation which will be discussed in more depth in section 7.8.1.2. It allows users in specific appointer roles to issue appointment certificates without the appointers holding the role themselves. The appointee may then use the certificate to activate roles and the certificate can correspond to credentials such as a required qualification or organisational membership.

OASIS allows for specification of interface and compliance meta-policies to enable communication and compliance in or between distinct administrative domains and their local policies. Meta-policies are defined as sets of rules that describe the management of policies and consist of data types; objects; functions; roles and rules [Belokosztolszki and Moody, 2002]. Compliance meta-policies provide local and external users with information about an access control policy without making its details public. Interface meta-policies describe how the roles of one domain can be used in another. One example for such a compliance policy is that users must have read access to their data and no one except the owner may modify this data. An example for a communication or interface policy is that as part of a service level agreement, the generic role of a local doctor should have access to the records of his patient which are stored within the domain of a hospital.

Again, we will provide a more in depth and critical review of selected aspects of OASIS in the context of chapters 5, 6 and 7.

3.4 Policies for distributed systems

We understand policies as persistent declarative specifications, derived from management goals, defining choices in the behaviour of a system [Moffett and Sloman, 1993]. Policy-based approaches to system management allow the separation of the rules that govern the behaviour of a system from the functionality provided by that system [Sloman, 1994b].

3.4.1 The Ponder language and policy framework

A policy framework has been established at the Imperial College through years-long research into policy-based management of distributed systems. In particular, it is based on the concepts of:

- management domains for structuring systems [Sloman and Twidle, 1994];
- policies which define choices in the behaviour of a system [Sloman, 1994b];
- notations to support policy specification [Marriott et al., 1994], [Damianou et al., 2001];
- and role-based extensions for ease of policy administration [Lupu, 1998].

Domains provide a means of grouping and partitioning objects in systems according to certain criteria. Domains hold a reference to their member objects. The user representation domain (URD) represents the human (login) in a system. Policies are relationships between subject and target objects. Policies apply to domain objects and we can distinguish between the following basic policy types:

- Authorisation policies;
- Obligation policies;
- Delegation policies;
- Refrain policies;
- Information filtering policies;
- Meta-Policies.

These concepts have been implemented in Ponder, a declarative, object-oriented language for specifying security and management policies for the objects in a distributed system. Policies are separated from the managers that interpret them, which allows the behaviour of the management system to be changed without re-coding the managers.

In the following, a brief description of selected, context-relevant, policies defined in Ponder is given. For a fully comprehensive introduction to Ponder and the technical aspects of the policy framework we refer to [Damianou et al., 2001] and [Damianou, 2002][2].

[2]This reference will be cited for the rest of this thesis when referring to any aspect of the Imperial College policy framework.

3.4.1.1 Authorisation, delegation and obligation policies

Authorisation policies specify what activities a subject is permitted or forbidden to do to a set of target objects. They are implemented on the target host by an access control component. Authorisation policies may be of a positive or negative modality. An example for a positive authorisation policy could be that members of a *Clerk* domain may perform the *alter_account()* action on objects in the *Accounting* domain of a company.

```
inst auth+ alter_company_accounts {
   subject     /Clerk;
   target      /Accounting;
   action      alter_account();
}
```

Delegation policies specify which actions subjects are allowed to delegate to others. A delegation policy permits a subject to grant privileges it possesses as a result of an existing authorisation policy to a grantee. As such a delegation policy is a specific authorisation policy declaring the right to delegate. In the following example, a subject defined in the *alter_company_accounts* authorisation policy may delegate the *alter_account()* action to a grantee in the *Trainee* domain.

```
inst deleg+ (alter_company_accounts) deleg_alter_company_accounts {
   grantee     /Trainee;
   target      /Accounting;
   action      alter_account();
}
```

Obligation policies specify what activities a subject must perform with respect to a set of target objects and define the duties of the policy subject. They are triggered by events and are interpreted by a manager agent at the subject. We consider the example of the obligation to process a cheque which partly requires the previous authorisation. When an invoice from a supplier arrives, a clerk `c` has to issue a cheque to that supplier and then alter the respective accounts to indicate that the invoice has been paid for. Here the `->` symbol is part of a simple process description language and defines a sequence of actions.

```
inst oblig process_cheque {
   on          invoice_arrival(supplier);
   subject     c = /Clerk;
   target      /Accounting/Supplier_Accounts;
   do          c.issue_cheque(supplier) -> c.alter_account(supplier);
}
```

Refrain policies are similar to negative authorisation policies but are enforced by the subject and not by the target access controllers as these might not be trusted.

Policies can be defined as types from which instances can be declared. In that case a general authorisation policy type for altering accounts might be declared, which can then be used to create specific instances for altering payable and receivable accounts.

```
type auth+ alter_company_accounts (subject s, target t) {
   action      alter_account();
}

inst auth+ alter_acc_pay = alter_company_accounts(/Clerk,/Accounting/Pay);
inst auth+ alter_acc_rec = alter_company_accounts(/Clerk,/Accounting/Rec);
```

3.4.1.2 Constraints and meta-policies

Ponder further allows for the definition of constraints limiting the applicability of a policy. These constraints are defined using the Object Constraint Language (OCL) [Warmer and Kleppe, 1998] and can be distinguished as:

- Constraints on the attributes of subjects and targets, e.g. do not allow writing to accounts if `subject.status = "Trainee"`;
- Constraints on time, e.g. validity period for an authorisation policy is normal working hours expressed as `time.between("8:00", "18:00")`;
- Constraints based on action or event parameters.

Meta-policies are policies about policies, used to define application specific constraints. A separation of duty policy is an example of such a meta-policy as it may restrict existing authorisation policies depending on the current context.

The following is an example of a meta-policy considering a possible static separation of duty to prevent the same person from being able to issue and approve a cheque by signing it.

```
inst meta process_cheque_separation raises conflict_in_cheque_process(a) {
   [a] = self.policies -> select (p1, p2 |
     p1.subject -> intersection(p2.subject) -> notEmpty              and
     p1.action -> exists (act | act.name = "issue")                  and
     p2.action -> exists (act | act.name = "sign")                   and
     p2.target -> intersection (p1.target) -> oclIsKindOf (cheque))
   a -> notEmpty ;
}
```

The OCL language will be discussed in more depth in section 4.3.1, where its general suitability for expressing constraints and organisational control principles is investigated.

3.4.1.3 Composing policy specifications

To ease policy administration, Ponder provides the means for composing policy specifications through the concepts of groups; roles; role hierarchies; relationships; and management structures.

Groups are used to relate sets of policies with respect to some meaningful attribute. As such a group may, for example, contain policies like authorisation, obligation or meta-policies. Roles are similar to groups in that they provide a semantic grouping of policies with a common subject. This subject is usually a position in the organisation, e.g. a branch manager. Thus, a principal can be assigned to such a position and subsequently acquires the related policies. Such a branch manager role could allow a principal to (de)activate alarms, and it might also require him to check the tills of the branch every afternoon.

```
type role Branch_Manager () {
   inst auth+ Set_Alarm {...}
   inst oblig+ Check_Tills {...}
}
```

Roles can be part of a hierarchy using a specialisation mechanism. Such a role hierarchy could be expressed as *role Branch_Manager() extends Clerk()* where a *Branch_Manager* inherits, adds and possibly overrides the elements of a *Clerk* role.

Relationships express the interaction of roles with each other. For example, at the end of each week a clerk may have to provide a summary of all cheques issued to the senior accountant. Ponder supports such relationships by grouping the policies which define the rights and duties of the roles participating in a relationship have towards each other. Policies alone cannot fully specify these relationships. Interaction protocols defining message content and permitted message sequences are provided. Since multiple managers can be assigned to a role, the role policy activities have to be synchronised by concurrency constraints.

Management structures define a configuration of role instances and relationships relating to an organisational unit. Such a structure could describe a general branch of a bank which is then instantiated for a particular branch in a town.

3.4.1.4 Policy conflict analysis in Ponder

Policies may conflict with each other and the Ponder framework considers the following types of conflict as initially described in [Moffett and Sloman, 1994, Lupu and Sloman, 1999]:

1. Modality conflicts;
2. Application specific conflicts.

Modality conflicts can be summarised as two policies having the same subjects, targets and actions but opposite modalities, e.g. a positive and negative authorisation policy. Application

specific conflicts may be those of separation of duty; self management; multiple managers; resource priority; or conflicts of interest.

Policy conflict analysis may be categorised into static and dynamic analysis. Static analysis refers to the detection of any conflicts or inconsistencies at compile-time, while dynamic analysis loosely refers to detection at run-time.

While Ponder and its toolkit provide facilities for the static analysis of policies and, for example, the detection of modality conflicts, there is no support for dynamic policy analysis. This is because research in the area of dynamic policy analysis is still immature and only recently some progress has been made, e.g. [Dunlop et al., 2002, Bandara et al., 2003].

We believe that our work may indirectly contribute to the field of dynamic policy analysis as it allows for the exploration of the behaviour of principals over sequences of states as we will, for example, show in section 7.7.

3.4.2 Other approaches to policy-based systems management

There are many other approaches to policy-based management of systems. For reasons of space we could, for example, not consider policy specification in the areas of databases and networks [Sibley et al., 1991, Wies, 1994], or by means of logic programming [Jajodia et al., 1997a, Ortalo, 1998]. This reflects the interdisciplinary nature of work on policy, ranging from management sciences to particular areas of computer science such as network and database management, artificial intelligence, access control and others.

This section reviews four policy languages and frameworks in more detail. The reason behind their choice is that the first reviewed language (PDL) provides a different, more abstract approach to policy specification than for example Ponder. This includes a notion of states and histories which is also considered in the second language (SPL), here with the difference of specifically considering obligation policies. The third (LGI) then provides a more general specification language and framework for control and coordination which is based on Prolog. Lastly, the specification language of an enterprise framework (ODP) is briefly discussed.

The Policy Description Language (PDL) [Lobo and Naqvi, 1999] defines policies as principles or strategies for a plan of action, designed to achieve a particular set of goals. PDL is based on the event-condition-action paradigm, so a defined rule is triggered when an event occurs and the condition is satisfied. PDL is used as the basis for defining conflict detection and resolution mechanisms for (history-based) policies [Chomicki and Lobo, 2001]. The conceptual gap to policy languages like Ponder is, however, too wide as to directly apply these results. For example, PDL does not have any corresponding notion of authorisation.

In [Ribeiro et al., 2001b] an access control language called SPL is introduced. This language is a policy-oriented constraint-based language composed of entities, sets, rules and policies. Policies are seen as complex constraints that result from the composition of rules and sets into logical units. A restricted type of obligation is discussed which caters for a) two or more actions mutually obliging each other and b) an obligatory action that is causally dependent on a prior trigger action. To enforce these types of obligations, a monitoring mechanism is

suggested in [Ribeiro et al., 2001a], based on an underlying model of atomic action execution related to the transaction concept. A sequence of actions has to be performed or not, thus allowing for the enforcement of security policies with dependencies on past events, e.g. principal `p1` must do action `a2` if principal `p2` has done action `a1`, or future events, e.g. principal `p1` cannot do action `a1` if principal `p2` will not do action `a2`.

The laws in the law-governed interaction framework (LGI) [Minsky and Ungureanu, 2000], are expressed in a restricted version of Prolog, also following the event-condition-action paradigm. This expressive liberty of Prolog is bought at the cost of the commonly known drawbacks of Prolog such as the specific interpretation of negations and lack of type enforcement (compare section 4.3.3). In general, the focus of LGI is more on aspects of control and coordination mechanisms than actual policy specification. As such LGI introduces the concept of enforced obligations and sanctioning mechanisms.

Policy specification languages and other organisational related concepts have also been investigated in the work of the Open Distributed Processing (ODP) community [Cole et al., 2001]. The reference model for ODP [ITU, 1994] provides an architecture for specifying and implementing distributed systems and is based on a multiple viewpoints approach. The enterprise viewpoint considers the objectives and policies of the enterprise that the system is meant to support. Policies are defined as rules with a particular purpose governing the behaviour of the enterprise. They comprise obligation, permission and prohibition policies. Communities are the main structuring element and consist of principals and resources which are then related through roles, processes and policies. The policy language is based on structured English and predicate logic in the form of the Object-Z notation [Steen and Derrick, 1999].

We deemed none of the above languages or frameworks suitable for our purposes of control principle analysis and exploration as they are either too specific in their purpose, e.g. LGI's focus on coordination, or simply overly complex, e.g. PDL because of its ambition to detect policy conflicts.

3.5 Audit

Access control and more general systems management requires audit facilities [Sandhu, 1996] to, for example, detect and deter any unlawful access or recover from any misuse of authority or other kind of system failure. We have identified audit as a control principle in section 2.7. However, its mission, character and strength may vary depending on the type of organisation [Gallegos et al., 1999], and there is no single definition we can give to it in the context of our control principle framework.

In its most general sense, auditing may be seen as an activity of checking that a system performs its required function and may be performed internally by the organisation or by external auditors. Auditing may also result in recommendations for system improvement. Here, a system can be seen as anything, from the entire organisation as a system, down to an individual computerised information system.

In this context we concentrate on information systems audit, as opposed to, for example, financial audit [Clark and Wilson, 1987]. Even within this narrow context there are many different kinds of information systems audit, some of which are listed below:

- organisational audits assessing how management control over information technology is achieved, e.g. [Gallegos et al., 1999];
- technical information systems audit assessing infrastructure, data centers, or data communication, e.g. [Weber, 1998, Smith, 1999];
- specific application audits, e.g. [Douglas, 1983];
- development and implementation audits at the requirements, specification, design, implementation stages, e.g. [Douglas, 1995, Gallegos et al., 1999];
- standard compliance audits, e.g. ISO 9000;
- system management audits, e.g. ISO 7498-4, addressing fault management; configuration management; accounting management and security management.

In a more general sense, information systems auditing may be seen as a process of collecting and evaluating evidence [Gallegos et al., 1999]. This usually requires the logging, monitoring and creation of audit trails of activities in a system. For example, section 9.8 will describe the case study of a bank where the generation of evidence by logging any administrative actions is obligatory and has to follow the bank's policies on logging and evidence generation.

We can make a distinction between the two main types of logging activities. A full log, e.g. in database systems [Date, 1994] helps to recover from faults. Where a full log is not necessary or impractical [Sandhu, 1990], partial logs give information about certain previously selected system properties, e.g. access attempts of a user or the number of sent packets in a network.

Monitoring actions comprise the activities of generation of information, processing the information and the dissemination/retrieval of information, in order to make decisions and perform control functions. Monitoring is a passive process not changing the behaviour of the monitored system [Sloman, 1994a].

Audit trails provide a record of the computing processes which have been applied to a particular set of source data, showing each stage of their processing and allowing the original data to be reconstituted [Oxford, 1990]. Unlike logs they are usually more complex and interwoven by referring to different kind of logs. Audit trails are seen as a necessary component in the Biba model [Biba, 1975].

It may be argued that an optimal solution would be the case where every single change in a system is recorded. However, this is probably never the case in any real-world information system because of the economic constraints these systems have to work under such as, for example, limited storage and evaluation capacities.

3.6 Chapter summary and conclusion

Decentralisation implies the delegation of work and with it the needed authority which is partially expressed in the access rights of a principal within the IT infrastructure. In this chapter we have reviewed work on models for access control and policy-based systems management. A brief introduction to the field of access control, its problems and current research directions was given in the initial section 3.2. We then discussed role-based access control in section 3.3, where we looked in particular at the RBAC96 and OASIS role-based access control models in sections 3.3.1 and 3.3.2. This was followed by a review of policy-based management of systems in section 3.4 focusing on the Ponder language and policy framework in section 3.4.1.

We looked at these three models and frameworks because in some form or the other they all:

- incorporate a concept of roles to reflect organisational structure;
- allow for the specification of permissions/authorisation policies as well as obligation policies;
- include delegation mechanisms to reflect changes to the relationships between principals, roles and policies;
- provide some means for the definition of rules and policies that may be used to express further controls.

This review was intentionally short as we will provide a more in depth and critical review of selected aspects of these models in chapters 5, 6 and 7. Alternative policy languages and frameworks were reviewed in section 3.4.2.

The reason why we did not use the Ponder language for the definition of a control principle framework lies in our explorative approach. We believe that using Ponder may have been too restrictive. In hindsight this seems to have been the right decision, as we will later show with respect to our novel concepts of review and supervision.

Considering audit controls, our emphasis was on showing that there is no general definition by means of listing some specific types of audit in section 3.5. Audit needs to be supported by logging, monitoring and audit trails. In section 5.10, we describe how we maintain information about the access of an object as well as about the delegation and revocation of policy objects in the context of this thesis. For example, the information about the access history of an object or principal is required by the dynamic separation controls described in section 6.2, and information about delegation activities is required by the revocation controls as described in section 7.5.3.

With this and the previous chapter we have set much of the ground for establishing our model of organisational control principles in chapter 5. Before doing so, we discuss in the following chapter 4 what particular specification language we will use for this purpose.

Chapter 4

The Alloy Specification Language

4.1 Introduction

In general, the choice of a specification language must be driven by the nature of the problem to be modelled [Lamsweerde, 2000a, Jackson and Rinard, 2000, Sommerville, 2001]. If the problem appears to be structural, and the focus is on objects and their relationships, the language of choice will probably differ substantially from a language that might have been selected for the specification of security protocols or concurrent systems.

Chapter 2 has provided an initial outline to control principles in a more general organisational context. This was followed by chapter 3, providing a description of specific models, languages and frameworks for access control and policy-based systems management we deemed to be relevant with respect to control principle expression. Both chapters indicated that we require support for the expression of structural properties, e.g. roles and principals which are members of these roles, as well as behavioral properties, e.g. delegation activities between principals.

This chapter describes our investigation of a selected set of more general specification languages to support and communicate the definition, analysis and exploration of control principles. It consists of two parts.

1. A summary and critical evaluation of selected specification languages is provided in sections 4.2 and 4.3. This includes a documentation of some of our initial attempts to control principle expression as part of our explorative approach. As a result, the Alloy specification language was chosen as the basis for control principle specification, exploration and analysis.

2. A brief tutorial on the Alloy language is given in section 4.4. The specification techniques demonstrated in this tutorial will then be used throughout the rest of this thesis.

4.2 Evaluation criteria

The nature of control principles requires the formal specification of structural and behavioural aspects. The structural properties that will have to be defined are relationships between entities such as roles assigned to principals or groups these entities are a member of. We also have to be able to express changes to these relationships. These behavioural properties are defined by operations that constrain the transition from one state to another.

There are many languages that support the formal specification of such structural and behavioral properties. An exhaustive description would be impossible in this context and has been given elsewhere, e.g. [Balzert, 1996, Clarke and Wing, 1996, Lamsweerde, 2000a]. In the following sections we present a selected set of declarative languages we experimented with. The criteria we informally used for the evaluation are summarised in the following:

- Expressiveness
  Can the language be used to describe the specific structural and behavioural aspects of control principles? To what degree is it tied to a specific methodology or problem domain?

- Explicitness
  Is the language explicit or are there some implicit constraints, e.g. adherence to a specific logic system?

- Uniformity
  Do basic formal concepts remain unchanged or are there specific interpretations, e.g. like the specific interpretation of negation in Prolog?

- Logical Limitations
  Are there any limitations, e.g. is the arity of relations limited to binary?

- Tool support
  Does it support basic syntax and type checking of a specification as well as its further analysis, verification and animation?

4.3 Comparison and evaluation

4.3.1 The Object Constraint Language

The Object Constraint Language (OCL) is an object-oriented specification language designed to support specifications in the Unified Modeling Language (UML). OCL is part of the UML 1.4 standard and was also used for the specification of the UML meta-model. Having been influenced by earlier formal specification languages such as Z, the OCL is based on basic set theory and predicate logic. OCL makes it possible to define constraints within an object-oriented system development process [Warmer and Kleppe, 1998, Clark and Warmer, 2002].

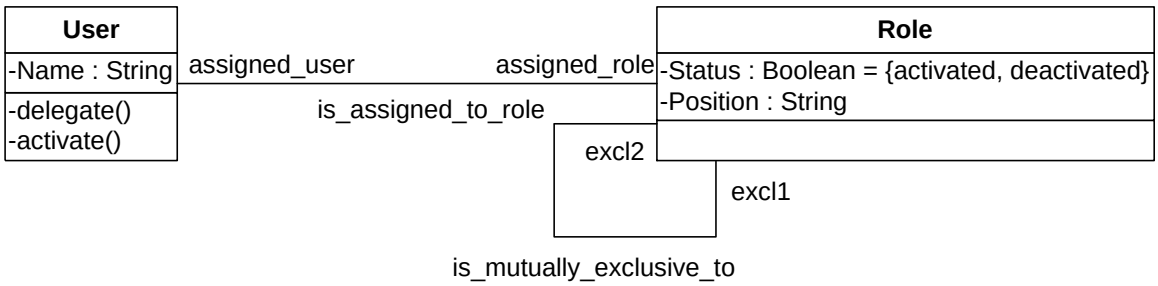

Figure 4.1: UML diagram for OCL specification.

OCL constraints can be specified as invariants; pre- or post-conditions; and guards. An invariant is a restriction on a class that must always hold for the class and any of its instantiated objects. A pre-condition is a restriction that must hold before an operation is executed. A post-condition is a restriction that must hold after an operation was executed. A guard is a restriction on the transition of an object from one state to another. In figure 4.1 we defined a class model with the classes User and Role, and the relationships is_assigned_to_role and is_mutually_excl_to. The following examples of OCL constraints are based on this structure.

The context keyword defines the class in the associated class model we choose to start from. In this case the context class has the name User. The inv, pre or post keyword indicate the type of constraint. Using the self keyword we say that taking an instance of the class User we navigate along the assigned_role association end of the is_assigned_to_role relation. The '–>' operator delivers the set of roles a user is assigned to (It does not mean 'implies'). On this set we may perform a defined operation such as size, returning the number of elements of the set. In this case the size of this set must be less then '10'. Thus every user must be assigned to fewer than 10 roles as expressed in the following OCL constraint.

context User inv: self.assigned_role –> size < '10'

There is tool support for writing OCL specifications. These tools range from simple syntax checkers such as those available at Klasse Objecten (http://www.klasse.nl/ocl); open-source CASE tools such as ArgoUML [Robbins et al., 1998]; and animation tools like USE [Richters and Gogolla, 2000]. There is also work in the area of translating OCL to first-order logic [Beckert et al., 2002] and Prolog [Gray and Schach, 2000].

The OCL only allows for the specification of binary relations, a constraint which did not allow us to naturally express certain concepts in our framework. Apart from this and the fact that OCL specifications are somewhat cumbersome to read, the OCL has one main disadvantage. At the stage of writing this thesis it did not have a fully defined formal semantics and we refer to [Cengarle and Knapp, 2001, Clark and Warmer, 2002] for a discussion on the efforts to date. While there has been an ongoing effort in providing a formal semantics for the OCL over the last years no major advances have been achieved yet to our knowledge. A further discussion on technical limitations of the OCL can be found in [Vaziri and Jackson, 1999] and in recent UML conference proceedings [Bezevin and Muller, 1998, France and Rumpe, 1999, Evans et al., 2000].

Name	Static Separation of Duty
Status	Typechecked: YES Animated: YES
Purp.	Any two roles a user is assigned to must not be exclusive to each other as specified by the 'is_mutually_exclusive_to' association.
Desc.	We use the excl1/excl2 association ends between every pair of roles to determine whether any two roles are mutually exclusive to each other.
OCL	context User inv: not self.assigned_role–>exists(r1,r2:Role| r1.excl1–>includes(r2) and r2.excl2–>includes(r1))

Name	Dynamic Separation of Duty
Status	Typechecked: YES Animated: YES
Purp.	Any two roles a user is assigned to can be exclusive to each other if at least one of them is deactivated.
Desc.	We use the excl1/excl2 association ends between every pair of roles to determine whether any two roles are mutually exclusive to each other. We then check for the role status.
OCL	context User inv: self.assigned_role–>exists(r1,r2:Role| (r1 <> r2) and (r1.status = #activated) and (r2.status = #deactivated) and (r1.excl1–>includes(r2) or r2.excl1–>includes(r1)))

Name	Simple Delegation Constraint
Status	Typechecked: YES Animated: NO (because USE did not support operations)
Purp.	A user must be in possession of a role before he delegates that role. After the delegation he must lose that delegated role.
Desc.	In the precondition we first obtain the set of roles a user is assigned to, only if that set includes the role he is delegates he can execute the delegate() operation. In the postcondition we use the same expression but have to negate it.
OCL	context user::delegate(role_to_be_delegated: Role) *pre*: self.assigned_role–>includes(role_to_be_delegated) *post*: not(self.assigned_role–> includes(role_to_be_delegated))

Figure 4.2: Three examples of control principle specification in OCL.

Experimenting with different OCL compilers we found cases where the same specification behaved differently or would not compile at all. It was for these reasons that we discarded the OCL as a language for control principle specification. We nevertheless outline and document some of our initial approaches to control principle specification with OCL. In this context we only present three of these specifications to illustrate some general OCL properties and how we used the supporting tools. In particular, we first established an object model, a small subset of which is shown in figure 4.1. This was then annotated with appropriate OCL constraints. The definition of the class model was done using the ArgoUML tool

version 0.7, with the OCL specification being based on OCL 1.3 as supported by the type-checker integrated in ArgoUML. The class model and OCL specifications were then manually translated into the USE input language as described in [Richters and Gogolla, 2000] for purposes of animation.

The three OCL specifications shown in figure 4.2 are an example of our initial specification attempts. It can be seen how these make use of the classes and associations as outlined in figure 4.1. We intentionally chose this simple example for reasons of clarity. Since we did not follow this approach, we provide no further explanation on any specific aspects of these specifications.

4.3.2 Z and Object-Z

The formal specification language Z [Spivey, 1992] is based on the mathematical concepts of sets, functions and first-order predicate logic. Types define the entities of interest in the specification of a system and its states. State transitions are described in terms of relationships between inputs and outputs of operations. Invariant predicates constrain such state changes. Schemas are constructs to modularise a specification. There is no room for a further description of Z, and we refer to the tutorials given in [Bowen, 1996, Davies and Woodcock, 1996] as well as the information provided on Z in the Formal Method Archive (http://www.afm.sbu.ac.uk) maintained by Jonathan Bowen.

Object-Z is an extension to the Z notation. Its development was driven by the motivation of integrating formal techniques with the object-orientation paradigm. Like Z, Object-Z has a fully defined syntax and semantics, and formal proof is supported. For an introduction to the Object-Z specification language we refer to [Smith, 2000].

Although, both Z and Object-Z would provide us with a basis for rigorous formal proof supported by tools (e.g. Z/Eves, CadiZ, Formaliser), we found them to have several practical disadvantages. They do not naturally support a more abstract visual modelling language such as the UML. Especially constructs like associations between classes would have to be implemented in a quite cumbersome way. Thus an easy traversal of the class model is initially not possible.

These experiences are based on our initial attempt to generate Z specifications from UML diagrams. For this we used RoZ, which is a plug-in for the Rational Rose CASE tool [Dupuy et al., 1998, Dupuy et al., 2000]. RoZ allows for the expression of UML specifications in the Z (or Object-Z) language. UML diagrams and their annotations are translated into Z specifications and proof obligations can be generated. Theorem provers like Z-EVES can be used to discharge proof obligations. We have briefly evaluated RoZ for translating parts of the UML class diagram in figure 4.1 into the Object-Z language. The resulting output can be seen in figure 4.3, and we again do not go into any particular details of this specification as it shall only illustrate our initial attempts to control principle specification.

We did not choose Z for our purposes, mainly because it is very complex and would have required more time to learn than permitted in this context.

USER

ROLE

UserExt

$User : \mathbb{F}\ USER$

RoleExt

$Role : \mathbb{F}\ ROLE$

is_assigned_to_roleRel

$RoleExt;\ UserExt$
$assigned\_user : USER \pfun \mathbb{F}\ Role$
$assigned\_role : ROLE \pfun \mathbb{F}\ USER$

$\mathrm{dom}\ assigned\_user \subseteq User$
$\bigcup(\mathrm{ran}\ assigned\_user) \subseteq Role$
$assigned\_role = \{x : \bigcup(\mathrm{ran}\ assigned\_user) \bullet$
$x \mapsto \{y : \mathrm{dom}\ assigned\_user \mid x \in assigned\_user(y) \bullet y\}\}$

is_mutually_exclusive_toRel

$RoleExt;\ RoleExt$
$excl1 : Role \pfun \mathbb{F}\ Role$
$excl2 : Role \pfun \mathbb{F}\ Role$

$\mathrm{dom}\ excl1 \subseteq Role$
$\bigcup(\mathrm{ran}\ excl1) \subseteq Role$
$excl2 = \{x : \bigcup(\mathrm{ran}\ excl1) \bullet$
$x \mapsto \{y : \mathrm{dom}\ excl1 \mid x \in excl1(y) \bullet y\}\}$

Figure 4.3: Object-Z output of translation of UML model in figure 4.1 using RoZ.

4.3.3 Prolog

Our third attempt was to adopt a logic programming approach for control principle specification using the Prolog (**Pro**gramming in **log**ic) language. Prolog is a declarative language aimed at describing what is known about a problem in terms of given knowledge about objects, relationships between objects and the desired solution. A Prolog program consists of a set of clauses. Each clause is either a fact or a rule describing how a solution relates to the given facts or how new facts can be inferred from them. A general introduction to Prolog is provided in [Clocksin and Mellish, 1996] and [Bratko, 2000], further surveys over the more general area of logic programming and related area of deductive databases are [Apt, 1999, Liu, 1999, Zaniolo et al., 1997, Ramakrishnan and Ullman, 1993, Gallaire et al., 1984].

Prolog has been used for previous work in the area of policy specification [Moffett, 1990], as well as work in the area of organisational modelling [Fox et al., 1998] and definition of separation controls [Knorr and Weidner, 2001]. In fact, we also had very early success in the definition and analysis of separation controls in the context of role-based systems as documented in [Schaad and Moffett, 2001] and [Schaad, 2001]. We do not review our initial work using Prolog here but refer to section 11.4.3 for a discussion of future work.

One of the drawbacks we found when using Prolog was the lack of type and syntax checking. There are extensions to the language mitigating this problem, but these are often incompatible with the ISO/IEC 13211-1:1995 standard and many compilers. This absence of such restrictions made it hard to focus on the specification of certain properties. Also the way programs are executed by left-to-right and depth-first search may not always be the preferred strategy and makes a search incomplete. Success of automated analysis is thus to a certain degree dependent on the specific structure of a Prolog program. The direct link between implementation and automated analysis also caused other distractions. For example, a series of delegations and revocations could be expressed as updates to a list. Defining the structure of this list and implementing the rules manipulating this list often made us forget about the more fundamental properties underlying the delegation and revocation of objects. Other drawbacks of Prolog include its reasoning being based on Horn clauses, the use of negation as failure and the unification algorithm omitting an occurs check. We refer to [Clocksin and Mellish, 1996] and [Bratko, 2000] for a more detailed discussion. To summarise, we initially chose Alloy instead of Prolog because its formal restrictiveness and purely declarative nature allowed us to concentrate on the abstraction of control principles. Section 11.4.3.2 discusses possible future applications of Prolog for modelling control principles.

4.4 Choosing an executable declarative approach: The Alloy specification language and its analysis facilities

4.4.1 The Alloy language

A recently developed notation, strongly influenced by Z and similar to the OCL notation, is Alloy [Jackson, 2001]. Alloy is an ASCII-based first-order logic modelling language supported by a tool for fully automated syntactic and semantic analysis. For a full introduction to the language syntax and its semantics refer to [Jackson, 2002]. Further application examples are given in [Jackson and Fekete, 2001, Krishnakumar and Sloman, 2001, Mikhailov and Butler, 2002] and possible applications domains include modelling structural properties of object-oriented software, analysis of key management schemes, railway safety, file synchronisation and air-traffic control. Depending on the specific domain, the properties that can be analysed range from reachability and absence of deadlocks, e.g. in a peer-to-peer protocol, to application specific soundness of theorems, e.g. some record is always returned by the lookup function of a naming scheme[1]. We initially assessed Alloy in the context of detecting conflicts in role-based administration [Schaad and Moffett, 2002b].

[1] http://sdg.lcs.mit.edu/alloy/case-studies.html

An Alloy model can consist of the following main components:

- *Basic types* - A basic type represents the set of atoms used for later analysis and is represented by a signature using the `sig` keyword.
- *Relations* - A relation defines a set of tuples of atoms of basic types and is defined within the signature.
- *Facts* - Facts are explicit constraints on the model defined by `fact` keyword.
- *Functions* - Functions either relate system states or define specific constraints and evaluate to true or to false. They can also be used to return other defined values. They are defined by the `fun` keyword.
- *Assertions* - Assertions are statements that are supposed to be true. Counterexamples are generated by the analyser if an assertion does not hold. They are defined by the `assert` keyword.

In the following sections only the elements essential for the specification of control principles are presented. First the basic syntax is introduced in section 4.4.1.1. This is followed by a brief explanation of signatures and relational composition and navigation expressions in section 4.4.1.2. Alloy is then directly compared to Z by on basis of the well-known birthday book example in section 4.4.2. Section 4.4.3 will outline the basic working of the Alloy constraint analyser, while section 4.4.4 will summarise parts of the presented notation and explain the general process of specifying and analysing Alloy models in terms of a more practical example.

4.4.1.1 Standard logical operators and quantifiers

In Alloy, every expression denotes a relation. Relations can be of any arity. Sets of atoms are represented by unary relations and scalars by singleton unary relations. So depending on the type and arity of the given arguments, the operators will yield a (singleton) unary, binary or n-ary relation.

The standard operators are:

```
+  // union
&  // intersection
-  // difference
```

The operators for comparing relations are:

```
=   // equality
in  // membership
```

The standard quantifiers are:

```
all x:  e | F   // universal:  F is true for every x in e
some x:  e | F  // existential:  F is true for some x in e
```

The available logical operators are:

```
!F       // negation:  not F
F && G   // conjunction:  F and G
F || G   // disjunction:  F or G
F => G   // implication:  F implies G
F <=> G  // bi-implication:  F when G
```

In addition, Alloy provides the following quantifiers:

```
no x:  e | F    // F is true for no x in e
sole x:  e | F  // F is true for at most one x in e
one x:  e | F   // F is true for exactly one x in e
```

4.4.1.2 Signatures, relational composition and navigation

A signature declaration introduces a basic type, along with a collection of relations called fields. The declaration `sig S {f:E}` declares a basic type `S` and a relation `f` of type `E`. If some `x` has the type `S`, the expression `x.f` will have the type `E`.

Signatures may be extended by using the `extend` keyword. A signature declared as an extension is a subsignature and creates only a set constant along with a constraint making it a subset of each supersignature listed in the extension clause. The subsignature takes on the type of the supersignature. A field declared in a subsignature is as if declared in the corresponding type signature, with the constraint that the domain of the field is the subsignature.

In Alloy two tuples can be composed or joined if the last atom of the first tuple matches the first atom of the second tuple. The resulting tuple consists of the atoms of the first and second tuple, leaving the matching atom out. When two relations `a` and `b` are joined in `a.b`, the resulting relation is obtained by taking every combination of a tuple in `a` and tuple in `b` and including their join.

The product of two relations `a` and `b` is `a->b` and yields every combination of a tuple from `a` with a tuple from `b` and concatenating them without dropping the intermediate atom. The treatment of scalars as singleton unary relations allows for the 'navigation' over fields of atoms. Given a scalar `a`, the expression `a.b.c` denotes the relation obtained by traversing from `a` over `b` to `c`.

The available relational operators are:

```
.   // The join of two relations
->  // The product of two relations
~   // The transpose of a relation
^   // The transitive closure of a relation
*   // The reflexive transitive closure of a relation
```

<table>
<tr><th>Z specification of a birthday book</th><th>Alloy specification of a birthday book</th></tr>
<tr><td>Basic types of the specification:

$[NAME, DATE]$.

Schema for state space of the system, known is the set of names with birthdays recorded and birthday is a function which, when applied to certain names, gives the birthdays associated with them:

BirthdayBook
$known : \mathbb{P}\, NAME$
$birthday : NAME \pfun DATE$

Operation to add a new birthday, the declaration $\Delta BirthdayBook$ describes a state change. The birthday function is extended to map the new name to the given date:

AddBirthday
$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$
$birthday' = birthday \cup \{name? \mapsto date?\}$</td><td>Basic Alloy signatures:

<pre>sig Name{}
sig Date{}</pre>
A birthday book has two fields: <code>known</code>, a set of names (of persons whose birthdays are known), and <code>birthday</code>, a function from names to dates:

<pre>sig BirthdayBook {
known:set Name,
birthday:Name->!Date
}</pre>
The operation <code>AddBirthday</code> adds an association between a name and a date. The <code>'</code> symbol indicates a state change for the birthday book:

<pre>fun AddBirthday (bb,bb':BirthdayBook,
n:Name,
d:Date){
bb'.birthday = bb.birthday + (n->d)
}</pre></td></tr>
</table>

Table 4.1: Direct comparison of Alloy and Z using the birthday book example [Spivey, 1992].

4.4.2 Comparing Z and Alloy

One of the motivations for the Alloy project[2] was to bring the kind of automation offered by model checkers to Z-style specifications. The Alloy Analyzer is designed for analyzing state machines with operations over complex states. Alloy can be viewed as a subset of Z. However, unlike Z, Alloy is first order, which makes it analyzable. A detailed comparison of Z and Alloy is part of [Jackson, 2001].

Alloy's composition mechanisms are designed to have the flexibility of Z's schema calculus, but are based on different idioms:

- extension by addition of fields, similar to inheritance in an object-oriented language,
- and reuse of formulas by explicit parameterization, similar to functions in a functional programming language.

One standard example used in introductory tutorials for Alloy[3] and Z is that of a birthday book recording people's birthdays. We informally compare this example in table 4.1, but refer to the tutorial in section 4.4.4 for a more context relevant introduction to Alloy.

[2] http://sdg.lcs.mit.edu/alloy
[3] http://sdg.lcs.mit.edu/alloy/samples/birthday.als

4.4.3 The Alloy constraint analyser

The Alloy language is supported by an analysis tool which makes use of standard satisfiability (SAT) solvers such as zChaff [Jackson et al., 2000]. The analyser translates the Alloy specification into a conjunctive normal form formula, that is handed over to the SAT solver, whose solution is in turn translated back to be displayed by the analyser [Jackson, 2000].

There are two ways of using the constraint analyser:

1. Instances of functions can be generated by using the `run` command.
2. Assertions can be tested for counterexamples by using the `check` command.

Running a function will return a model that satisfies the specification. Checking assertions will generate a counterexample when these are found to be false. In both cases the user has to specify a scope that bounds the size of the domains. Models are then generated within that scope. While this makes the problem finite and reducable to a boolean formula, not being able to generate a solution does not mean that no solution exists.

The Alloy analyzer is neither a model checker nor a theorem prover [Jackson, 2002]. Model checkers such as SMV or Promela [Clarke et al., 2000] perform a temporal analysis, comparing a state machine to another machine or temporal logic formula. In contrast, theorem provers such as PVS or HOL are interactive tools for deduction-based automated reasoning [Huth and Ryan, 2000]. When failing to prove a theorem it is often difficult to see whether the theorem is invalid, or whether the proof strategy failed. The augmentation of a theorem proving approach with Alloy has been considered in [Mikhailov and Butler, 2002] to mitigate this.

4.4.4 Specifying and analysing a simple domain model

In [Damianou, 2002], the domain model of Ponder has been given a semantics using Alloy. The drawback of this specification is its limitation to only two states due to the technical restrictions of Alloy at that time. In the following, an outline is given of how this specification could be improved by modelling object behaviour over sequences of states. This example will serve as a practical introduction to the basic concepts of Alloy, at the same time describing some key techniques that will be used throughout the following parts of this thesis. The full specification of the example is part of appendix A.

4.4.4.1 An initial specification

In their most basic definition, domains can be described as objects which contain other objects, similar to a file system on a computer. Thus, two signatures need to be introduced to capture this abstraction as shown in signature 4.1.

Alloy Signature 4.1 *Introducing an* `Object` *and a* `Domain` *signature.*

```
sig Object{}
sig Domain extends Object {
 contains: set Object}
```

The `Domain` signature extends the `Object` signature and introduces a new relation `contains`, relating a domain object to zero or more objects as indicated by the use of the `set` keyword.

Constraints may be specified on these signatures and their relations, in this case for example, that the containment relationship may not form a cycle. Here the `^` symbol in fact 4.1 defines a transitive closure operator.

Alloy Fact 4.1 *No cycles in the* `contains` *relationship.*

```
fact {all o : Object | o !in o.^contains}
```

To generate a model which satisfies the specification, an empty function `generate_somestate` is defined and executed with a search scope of, for example, up to seven instances for each signature as shown in function 4.1.

Alloy Function 4.1 *An empty function to generate an arbitrary model.*

```
fun generate_somestate (){}
run generate_somestate for 7
```

Of course, more complex functions or assertions may be specified to check for the presence or absence of some properties. Alloy supports an incremental specification approach and allows for the mitigation of any contradictions, inconsistencies or ambiguities at an early stage in the specification. The graphical feedback of the analyser, as for example shown in figure 4.4, grants an additional insight into the properties of the specification.

4.4.4.2 Introducing a notion of states

We now define functions which reflect how objects are added or removed from a domain.What is needed is a way of observing objects and the relationships they participate in over several states. States must thus maintain information about their objects and relations between them separately.

Alloy does not provide this in terms of a built-in state mechanism. It is therefore necessary to:

1. Model states as objects themselves, a technique referred to as objectification of state [Jackson, 2001];

2. Specify a set of frame conditions for operations causing state transitions.

State	Domain	Object
State_1	Object_0	Object_1
State_1	Object_1	Object_2
State_2	Object_0	Object_1
State_2	Object_1	Object_2
State_2	Object_0	Object_3
...	...	...

Table 4.2: State-based representation of the `contains` relationship.

We thus define a state signature `sig State` and the previous binary `contains` relation defined in the `Domain` signature 4.1 is changed to a ternary relation with the new signature `State` as its first tuple.

Alloy Signature 4.2 *Introducing a* `State` *signature.*

```
sig State {
 contains: Domain -> Object}
```

Any n-ary relationship can be transformed into a state relationship in a similar manner by taking the relationship name and adding it to the `State` signature together with the respective tuples. However, we note that this approach is computationally more expensive, and it should be considered carefully which signature in a specification requires to be transformed.

So a possible `contains` relationship for two states might look like table 4.2 if expressed in tabular form. Projecting over each state allows for the easy (graphical) examination of any state changes (See also corresponding figure 4.4).

4.4.4.3 Simple domain management operations

The two most elementary operations are those of adding and removing an object from a domain. The add operation can be specified in the following function 4.2:

Alloy Function 4.2 *Adding an object to a domain.*

```
fun add_object (disj s, s' : State,
                disj d1,o1 : Object){
   (d1->o1 !in s.contains) &&
   (s'.contains = s.contains + d1->o1)}
```

Here, `o1` represents the object to be added to a domain `d1` in the after state `s'`. This relationship did not exist in the before state which is expressed as `d1->o1 !in s.contains`. The frame condition `s'.contains = s.contains + d1->o1` ensures that every other relation remains the same and that the object is added. The keyword `disj` indicates that any two objects are disjoint. It can be argued that the precondition `d1->o1 !in s.contains` is not needed and whether the `&&` (and) symbol should be replaced by an implication. This

depends on the later analysis context. Note further that `s` and `s'` have no defined meaning in Alloy unlike, for example, `s'` would have in a notation like Z [Spivey, 1992].

The specification now provides all the information needed to observe the behaviour of this add operation by running it within a search scope of four objects but only two states, using the command `run add_object for 4 but 2 State`. One possible result of this operation can now be (graphically) observed in the transition from the first to the second state in figure 4.4, where an object Object_3 is added to a domain object Object_0.

In a similar manner the operation of removing an object can be defined in the following function 4.3:

Alloy Function 4.3 *Removing an object from a domain.*

```
fun rem_object (disj s, s' : State,
                disj d1,o1 : Object){
   d1->o1 in s.contains &&
   (s'.contains = s.contains - d1->o1)}
```

As a third operation the moving of an object between two domains can be specified as follows in function 4.4:

Alloy Function 4.4 *Moving an object from a domain.*

```
fun move_object (disj s, s' : State,
                 disj o1,d1,d2 : Object){
   (d1->o1 in s.contains && d2->o1 !in s.contains) &&
   (s'.contains = s.contains - d1->o1
                             + d2->o1)}
```

However, the above operation may not be required since the elementary operations of removing and adding objects can be composed to model the moving of an object between management domains.

This can be seen in assertion 4.1, stating that the result of such a composition of `rem_object`() and `add_object()` over states `s`,`s'` and `s''` should be identical to the result of the explicit `move_object()` operation from state `s` to `s''`.

Alloy Assertion 4.1 *Assert that removing and adding an object from and to a domain has the same effect as moving it in between the two domains.*

```
assert move_works {all disj s,s',s'' : State |
                   all disj o1,d1,d2: Object |
   rem_object(s, s', o1, d1) &&
   add_object(s', s'', o1, d2) =>
       move_object(s, s'', o1, d1, d2)}
```

As it was expected, checking this assertion using the `check` command will not generate any counterexample in the defined search space.

4.4.4.4 Defining a sequence of states

The previous section showed how state changing operations can be defined and executed. It is desirable to be able to execute one or more functions with respect to sequences of states with an arbitrary length. More specifically, this will be of importance for the analysis and exploration of behavioral aspects of control principles in the later chapters of this thesis.

A sequence of states is reflected in the following signature `State_Sequence` (line `1.`). Such a sequence is defined by a `first` and a `last` state (line `2.`), ordered through the relation `next` (line `3.`). The `!->!` part of the relation expresses that this is a one to one relation, making the sequence a total order.

Two explicit constraints are specified as part of the `State_Sequence` signature. Within a sequence of such operations all states must be reachable from the first state (line `6.`) Secondly, all states apart from the last (line `7.`) must be related via one or more operations such as adding or (re)moving an object (lines `8.-15.`).

State Sequence 4.1 *A sequence of domain operations expressed as an explicit signature*

```
1.  sig State_Sequence {
2.    disj first, last : State,
3.    next: (State - last) !->! (State - first)
4.  }
5.   {//Constraints on the signature
6.    all s : State | s in first.*next
7.    all states : State - last | some s = states | some s' = states.next |
8.     (some obj : Object | some d1,d2 : Domain |
9.         add_object(s, s', obj, d1)       //add an object
10.        ||                               //or
11.        rem_object(s, s', obj, d1)       //remove an object
12.        ||                               //or
13.        move_object(s, s', obj, d1, d2)  //move an object
14.    )
15.  }
```

Now models can be generated that satisfy this specification within the defined search space. One of the many possible state sequence models we chose here consists of four states, where each transition is achieved through the adding, removal or move of an object. Apart from some textual output of the analyser in the form of a tree structure, this model may be graphically represented and is displayed in figure 4.4. There we show the unprojected sequence where the ternary relation `next` is represented by hyperarcs. Below the same sequence and the explicit states can be seen as a result of projecting over the State_Sequence signature. Projecting again over each state shows the changes that took place. Based on an initial state State_1, we can observe how Object_3 is added to domain Object_0. Then Object_4 is added to domain Object_1 in State_2, whereas in State_3 the Object_3 is moved from the domain Object_0 to the domain Object_1, resulting in the final state State_4.

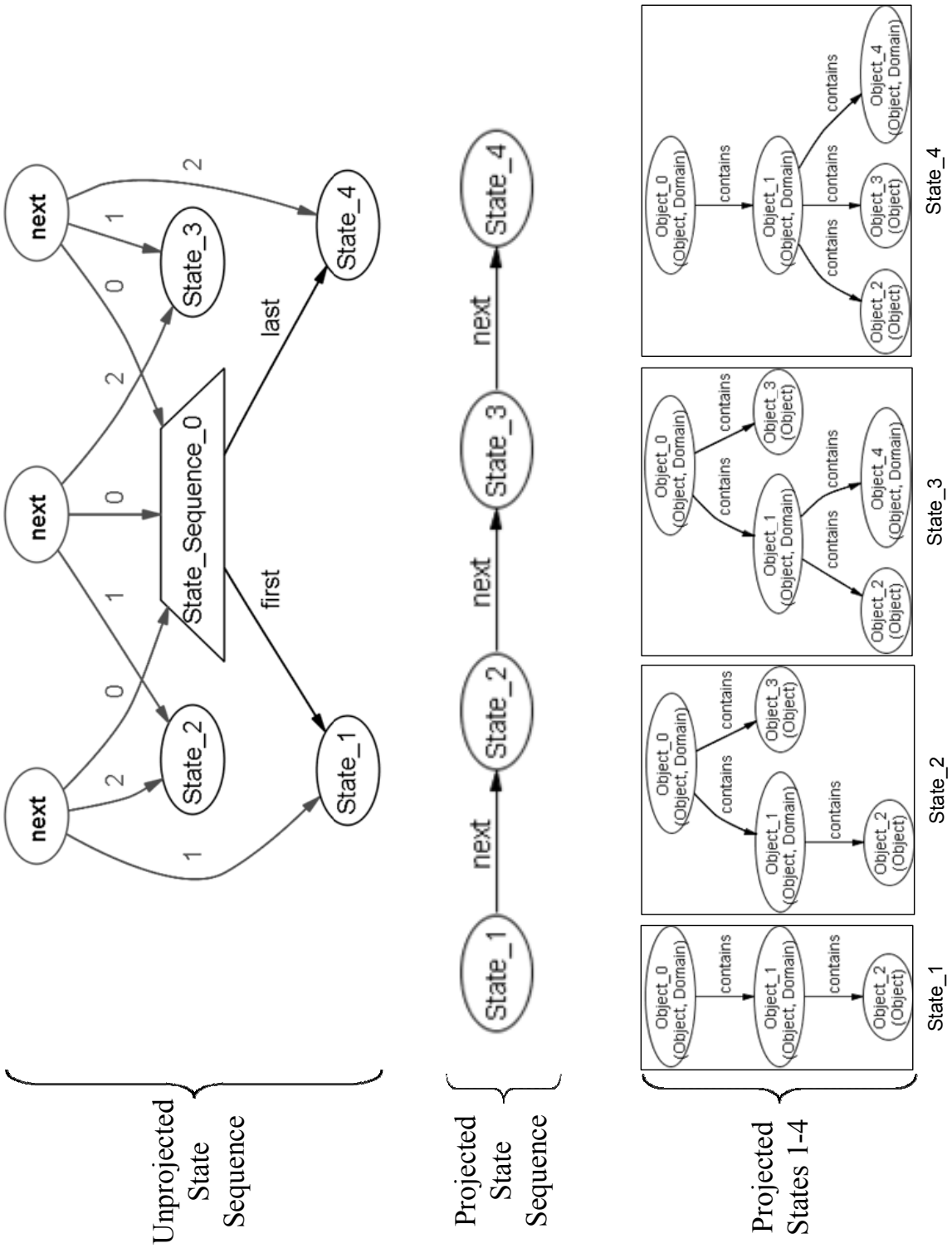

Figure 4.4: (Un)projected state sequence and the states resulting out of some domain manipulating functions.

4.4.5 General modelling approaches in Alloy

There are two conceptually distinct ways of specifying a model in Alloy. One way is to define a basic set of types and use the Alloy analyser to create models in the form of relations between instances of these types with respect to the defined facts, functions and assertions. This is what we have done in the previous section 4.4.4, and what we will do in terms of defining our control principle model in chapter 5.

A different style of specification is to define signatures such as `sig Clerk` and then explicitly enumerate clerks such as `Alice`, `Bob` or `Bill` by extending them from the `Clerk`. Using the `static` keyword, we indicate that a signature contains exactly one atom [Jackson, 2002]. In the same way other specific objects such as invoices or roles may be enumerated. Facts may then be used to explicitly define the relationships of these objects to each other, e.g. to specify that `Alice` only handles `Private_Customer` accounts. This is by no means a novel specification approach, but the analysis facilities of Alloy make it more explicit. Our case study presented in chapter 10 will adopt this specification style.

4.5 Chapter summary and conclusion

In this chapter we reviewed a set of selected languages deemed suitable for the specification of the structural and behavioral properties of control principles. This review was guided by an informal set of evaluation criteria defined in section 4.2. The investigated languages included the Object Constraint Language (OCL) in section 4.3.1; the Z specification language and its object-oriented extension in section 4.3.2; and the Prolog language for logic programming in section 4.3.3. It is clear that this is by no means an exhaustive list of suitable candidates, but we see it as sufficient within the given constraints.

As a fourth, and ultimately chosen candidate we investigated the only recently published Alloy language. Alloy is a first-order logic modelling language with tool support for fully automated syntactic and semantic analysis. We provided an introductory tutorial on the language. This was an extension of the domain model initially presented in [Damianou, 2002], allowing for the expression of sequences of states through a technique referred to as objectification of state.

As already indicated in section 3.6 it can be claimed that the Ponder language as reviewed in section 3.4.1 may also have been a suitable candidate for our purposes. However, we chose Alloy instead of Ponder, because we did not want to be restricted by Ponder's concepts. This seems to have been the right decision and we show in section 7.8.2.2 that the concept of delegating obligations we will introduce later cannot be modelled in Ponder at this stage. Additionally, Alloy's analysis facilities support our explorative approach.

On the basis of the discussion provided in the previous two chapters 2 and 3, the demonstrated specification techniques will now be applied for the definition of control principles in the following chapters 5, 6, 7, 8 and 10.

Chapter 5

A Model for Organisational Control Principles

5.1 Introduction

The definition, analysis and exploration of the structural and behavioral properties of control principles requires an underlying conceptual basis.

On the basis of the discussions in the previous chapters 2, 3 and 4, this chapter defines a model for control principles using the Alloy language. In particular, this includes:

1. A general overview of the model and the corresponding Alloy signatures representing the basic model elements and relations between them in section 5.2;
2. A detailed discussion of selected components and extensions to this initial overview with a focus on:
   - the modelling of authorisations and obligations in section 5.3 focusing on the distinction between general and specific obligations in section 5.3.3;
   - the concept of review obligations, review actions and evidence in section 5.4;
   - the distinction between roles and positions in section 5.5;
   - the definition of role hierarchies and exclusive roles in sections 5.6 and 5.7;
   - the definition of actions in section 5.8;
   - the definition of a history mechanism in section 5.10.
3. A description of the modular architecture of the specification, and listing of frequently used functions in section 5.11.

Instead of a simple listing of model-specific assumptions and formal statements, this chapter provides motivations for each design decision and follows an exploratory approach by including brief discussions of other possible conceptualisations where this seems appropriate.

5.2 The conceptual model

5.2.1 An initial overview

The structure of the conceptual model we use as the basis for the specification, analysis and exploration of control principles is displayed in figure 5.1. This representation gives only a first overview of the most basic elements and relationships and must not be mistaken for the entire model. Nevertheless, this graphical approach provides us with a better means of communicating the specification than pure text and maps uniquely to parts of the textual Alloy specification. The boxes correspond to Alloy signatures and grey lines with an empty arrow indicate object extension as supported by the Alloy `extends` keyword. Black lines with a filled arrow declare the relations between the objects and correspond to the relations in a signature. The direction of the arrow indicates the order of the signatures taking part in the relation. This initial model is then further refined at different layers of abstraction in the course of this chapter.

A key notion is that all but the `Action` signature in the model are (in)directly extended from a general `Object` signature. While this is bought at the expense of higher computational costs with respect to the later analysis, as well as a reduced level of type checking, it provides for:

1. The integration into existing object-oriented policy frameworks, e.g. [Damianou, 2002];
2. A higher level of abstraction within the specification reflected by:
   - constraints that are specified only once for a general signature and;
   - functions that may be applied to any extension of a signature.
3. Easier establishment of relationships between signatures.

`Objects` can be members of `Groups`. A group is itself an object and may thus also be a member of some other group. A `Principal` is an object representing a human user or automated component in the system. A `Policy Object` is an abstract representation of a rule determining the behaviour of principals in the system. A policy object is either an `Authorisation` or an `Obligation` and can have `subjects` and `targets` it applies to. Policy objects may be related to a principal either directly or through a `Role` he is a `member` of, since policy objects may have principals or roles as their subject. Policy objects `define` a set of `Actions`. In the case of obligations these are the actions that have to be performed and in case of authorisations the allowed actions. Execution of an action `creates Evidence` which is `specified` by an obligation such that it can be investigated whether the obligation was satisfactorily met. A `Review` is a specific kind of obligation and results out of the previous delegation of an obligation. `Review Actions` are a specific kind of action and evidence is `reviewed by` them. Two role specific relations allow for the formation of `role hierarchies` and the definition of mutually `exclusive` roles. A `Position` is a specific kind of a role with some associated, context-dependent, attributes. Positions can form supervision hierarchies over the `supervises` relation.

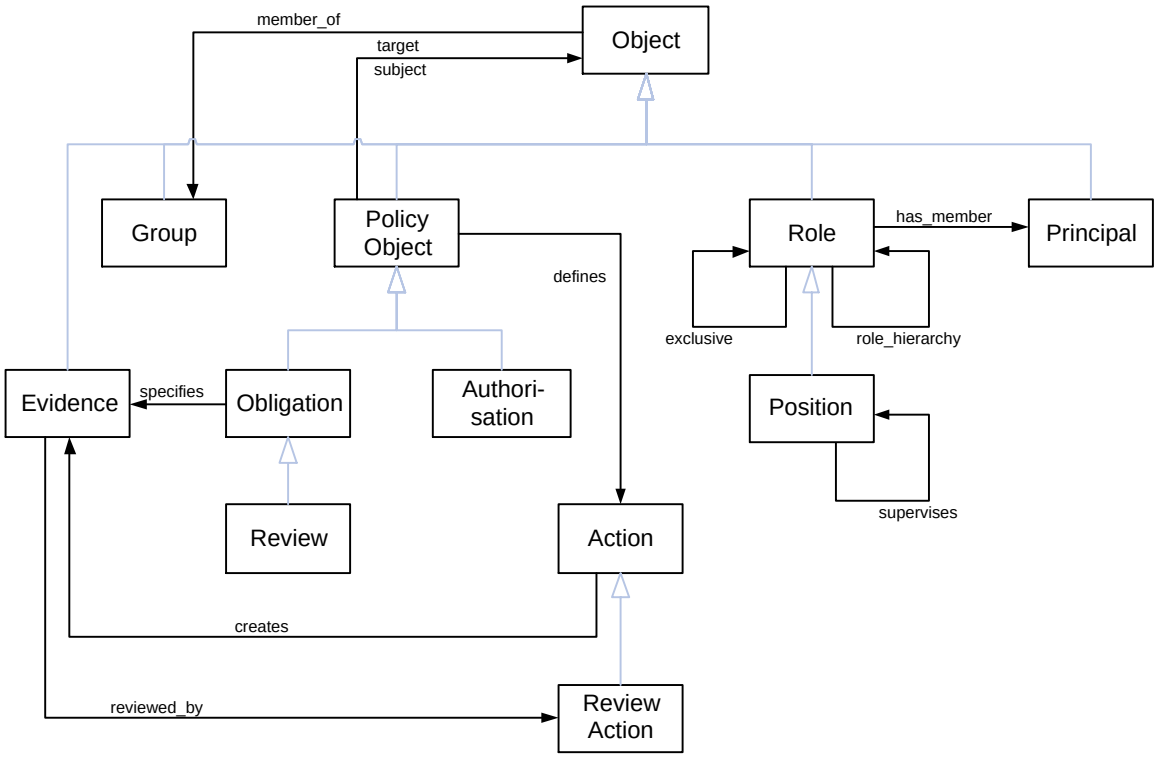

Figure 5.1: The conceptual control principle model.

5.2.2 The basic Alloy specification

A part of the model presented in figure 5.1 is captured in the following Alloy specification 5.1, showing some signatures and their relations. The keyword `sig` introduces a new type which can be extended using the `extend` keyword. To partition objects into distinct sets the keyword `disj` is applied. For the full specification, which includes some specific relations not shown in this initial model, we refer to appendix B.1.

Alloy Specification 5.1 *Example of some basic signatures and their relations in the control principle model.*

```
sig Object{
  member_of: set Group}
disj sig PolicyObject extends Object{
  target: set Object,
  subject: set Object,
  defines: set Action}
disj sig Obligation extends PolicyObject{
  specifies: set Evidence}
disj sig Role extends Object{}{
  has_member: set Principal,
  exclusive: set Role,
  role_hierarchy: set Role}
...
```

5.2.3 Objectification of state

This representation does not allow for the modelling of distinct states. Thus it needs to be transformed to cater for this requirement by means of objectifying state. We have described in the previous section 4.4.4 how to do this and only a part of the resulting `State` signature 5.1 is shown to illustrate this transformation.

Alloy Signature 5.1 *State signature.*

```
sig State{
 s_member_of: Object -> Group,
 s_target: PolicyObject -> Object,
 s_subject: PolicyObject -> Object,
 ...
}
```

For the rest of this thesis this representation will be used unless indicated otherwise. The leading `s_` indicates that a relation is a state-based relation. The effect is that facts and functions must now be expressed by quantifying or enumerating over states. So instead of defining that a group `g` is not its own member by specifying that `g` is not in the set of objects delivered by `g.member_of`, the redefinition is that a group `g` is not in the set of its members in a state `s` as delivered by `g.(s.s_member_of)`. The following fact 5.1 illustrates this.

Alloy Fact 5.1 *Stateless and state-based specification of an irreflexive group membership.*

```
fact {all g: Group |
 g !in g.member_of
}

fact {all s: State | all g: Group |
 g !in g.(s.s_member_of)
}
```

5.3 Modelling authorisations and obligations

Within our control principle model, policy objects are either authorisations or obligations. Principals, or the roles principals are a member of, may be subject to these policy objects. In other words, a principal is related to a set of policy objects over the roles he holds or on the basis of a direct assignment. The target of a policy object defines the objects that the actions of the policy are executed against.

In this context these subjects and targets are, however, expressed through explicit relations, and not in the form of domain expressions as in Ponder since there is no formally defined domain model in this thesis.

5.3.1 Authorisations

Authorisations state what a principal is permitted to do on the basis of using the actions defined by the authorisation. In this thesis only positive authorisations are considered since policy objects do not have any explicit modality. A general closed world policy is assumed to hold. This means that a principal can only execute the actions defined by the authorisations that he holds on some target objects. Authorisations can be shared between principals through roles or on the basis of direct assignments.

The actual meaning and granularity of authorisations is application-dependent. We consider the context of a business process for opening new customer accounts in a bank. A possible authorisation for a clerk might be `Auth_Open_Accounts`. Such an authorisation defines a set of actions such as `create_new_account` and `do_initial_credit_check` that describe what the opening of an account may comprise. How these actions may be modelled is shown in section 5.8.

In this thesis authorisations are what we understand as being the representation of authority at the enterprise level. They can only be fully understood within the context of the workflows and obligations derived from the goals of an organisation. The actions defined by an authorisation are situated at the application level, while the operations of an action are specific to the middleware and operating system.

5.3.2 Obligations

Obligation policies are an abstraction for defining the actions that must be performed by a principal on some target object when some specified event occurs. While this definition reflects our understanding of obligations, it requires a more detailed discussion on the requirements this raises with respect to the Alloy specification.

To begin with, this specification is mainly concerned with structural properties and the possibilities to model dynamic behavior are limited to the state sequence approach introduced in section 4.4.4. This means that there is no event architecture as in, for example OASIS [Bacon et al., 2002], that would allow us to explicitly model triggering events. This, and the current representation of obligation policies does at this stage not allow us to clearly represent:

- what it means for a principal to hold an obligation;
- how obligations relate to roles.

5.3.2.1 Processing invoices: A motivating example

We consider a general obligation policy which specifies that clerks have to process customer orders for money transfers. The defined event on which the obligation arises might in this case be the arrival of an order in the clerk's inbox. When this event occurs, the clerk now has the specific obligation to process this order.

This situation may be represented in the form of the following two abstract Ponder-style obligation policies, where the *specific_order_obligation_1* of *clerk_A* for *order_1* is an instance of the *general_order_obligation*. This example assumes that the specific *clerk_A* is in the set of *Clerks* and the specific *order_1* is in the set of *Orders*.

```
oblig general_order_obligation {
 on        order_arrival();
 subject   /Clerks;
 target    /Orders;
 do        process_order();
}

oblig specific_order_obligation_1 {
 subject   clerk_A;
 target    order_1;
 do        process_order();
}
```

On the basis of this observation it seems natural to us to adopt the notion of general obligations which have specific obligations as their instances.

5.3.2.2 Obligations and roles

The problem is that the control principle model we have developed so far is primarily a structural model, using roles as a convenient administrative shorthand over which to relate principals and policy objects. It does at this stage not allow us to describe situations such as the previous order processing example. Additionally, it is not yet clear how principals are related to obligations when roles are involved.

If a principal is member of a role, then he has the authorisations of that role at his discretion. Since several principals may be a member of the same role, this means that the same authorisation applies to several principals. This does not raise any conceptual difficulties. However, in the case of obligations this relationship requires further clarification as there initially seem to be two contradicting requirements. On the one hand it is desirable to specify an obligation that applies to several principals and roles appear to be the ideal structural means for doing so. On the other hand an obligation should be clearly related to one principal only, such that a) it can be assessed who can be held to account at any time and that b) the same actions are not performed twice. This is of even more importance when considering the delegation of obligations.

5.3.2.3 General and specific obligations

The problems described in the previous section can be resolved on the basis of the general assumption of this model that a distinction must be made between general and specific

obligations. This means that principals may have the same general obligation through a common role, but the specific obligation instances of this general obligation must be directly related to exactly one principal. The sharing of specific obligations between principals is therefore excluded. We will describe in section 7.4.3 how this concept elegantly supports the delegation of obligations.

We are not explicitly concerned about the context such general obligations are defined in. This would require a notion of workflows and events, an explicit specification of which, although indispensable within any real world situation, does not directly contribute to the goals of our thesis.

To summarise, the following requirements and assumptions have been identified and discussed in this section:

1. A distinction between general and specific obligation policies needs to be made.
2. General obligations may be shared between roles or principals, but a specific obligation must always be related uniquely to a principal.
3. The assumption is that specific obligations have been created based on some general obligations. However, since there is no explicit architecture to model triggering events, this creation is outside the scope of this model.

It now needs to be investigated how these requirements can be expressed in terms of the specification developed so far, what extensions have to be defined, and whether it makes sense to actually do this at the current level of representation.

5.3.3 Representing general and specific obligations in Alloy

We consider the previously given example of a clerk and his obligation to process incoming orders in section 5.3.2.1. Several possibilities of how to abstract and specify such a situation have been explored through partial Alloy specifications.

In the simplest case, a reflexive relation `has_instance` could have been defined for the `Obligation` signature. This had the disadvantage of having to implicitly express the distinction between general and specific obligations in facts and thus hiding them. For example, a general obligation may be defined as an obligation that is not an instance of some other obligation, while a specific obligation always is an instance of exactly one such general obligation. Although partially valid, this is not very helpful in communicating the specification.

A second possibility would have been to define two distinct signatures for a general and specific obligation and extend them from the obligation signature. Although legitimate, this was not done due to the general design principle of keeping the specification as simple as possible and thus achieving a better performance in the later automated analysis. Additionally, this would have caused more explicit facts to be specified.

The third and finally considered conceptual representation was to define an explicit signature `ObligationInstance` related to the `Obligation` signature through a `has_instance` relation.

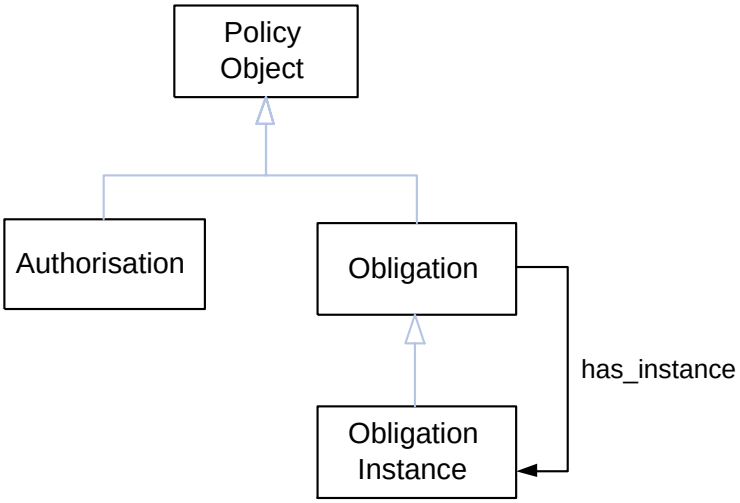

Figure 5.2: Modelling obligation instances in Alloy.

While this adds another degree of complexity to the model, it now explicitly expresses the distinction between general and specific obligations. What is currently lost here, however, is that such specific obligations are also obligations, i.e. the facts specified for policy objects and general obligations should also apply for them. This is easy to resolve by extending the specific obligation instance from the general obligation. The extension also explains why such specific obligation instances must not be confused with the object-oriented notion of class instances. In this context general obligations and their instances are distinct signatures with distinct properties. This conceptualisation is now recorded in the following new signatures and the amendment to the existing `State` in specification 5.2, with the corresponding graphical representation being displayed in figure 5.2.

Alloy Specification 5.2 `ObligationInstance` *and extended* `State` *signatures.*

```
disj sig ObligationInstance extends Obligation{}
disj sig State {
 ...
 s_has_instance: Obligation -> ObligationInstance
}
```

From now on when talking about general obligations we refer to the `Obligation` signature, while specific obligations refer to the `ObligationInstance` signature.

There are now several further constraints that need to be specified in order to clarify this proposed extension and satisfy the requirements elicited in the previous section. Some of these consider basic structural properties such as an obligation not having itself as an instance. Such constraints are not described any further here, and only those we see as specific to the control semantics of the model are discussed in the following. We refer to appendix B.1 for a complete specification.

It has been said that obligations should be carried out by one principal. This requirement must now be refined to distinguish between general obligations and obligation instances.

An obligation instance must always relate to exactly one principal: Shared obligations are excluded. This first constraint is an example for a cardinality constraint that cannot be specified as part of the `PolicyObject` signature since it does not apply to authorisations.

Alloy Fact 5.2 *An obligation instance must always relate to exactly one principal*

```
fact {all s : State | all obl : ObligationInstance |
   one (obl.(s.s_subject) & Principal)
}
```

We further define that a specific obligation must be the instance of one general obligation.

Alloy Fact 5.3 *An obligation instance has always one general obligation.*

```
fact {all s : State | all o : ObligationInstance |
   one ((s.s_has_instance).o & (Obligation - ObligationInstance))
}
```

It must also be the case that if a principal has a specific obligation, then this must be the instance of a general obligation he has through one of his roles or is a direct subject of.

Alloy Fact 5.4 *Every specific obligation a principal holds must be an instance of a general obligation he is a subject of through one of his roles or directly.*

```
fact {all s: State | all disj p1: Principal | all o: ObligationInstance |
   some o & (s.s_subject).p1 =>
     (s.s_has_instance).o in (s.s_subject).p1  ||
     (s.s_has_instance).o in (s.s_subject).((s.s_has_member).p1)
}
```

Finally, we define that a general obligation may only be directly assigned to a principal or to one of his roles but not to both. The reasons for this are mainly related to the performance of our later analysis, but we also think that any dual assignment does not make sense within an organisational context. Compare this with our later discussion in section 7.4.2 on a similar constraint in the context of delegating authorisations.

Alloy Fact 5.5 *A general obligation can only have a principal or one of his roles as a subject, but not both.*

```
fact {all s : State | all p1 : Principal |
      all o : Obligation - ObligationInstance |
         some p1 & o.(s.s_subject) =>
           no o & (s.s_subject).((s.s_has_member).p1)
}
```

Some of the above and other not further specified constraints such as a specific obligation always having a general obligation, could have been specified in terms of cardinality constraints inside the relations of a signature. It was decided to not do this but always explicitly state them as facts for reasons of clarity and sometimes undesired side-effects on other signatures due to the use of the signature extension mechanism.

5.4 Review and evidence

When an obligation is delegated, it may be made subject to a review obligation. The specific reasons and organisational motivations for this will be discussed in more detail in section 8.2, and this section only concentrates on the basic structural properties required to support such delegation activities. These are fully defined in appendices B.1 and B.4.

A review is defined as a specific type of obligation by using the object extension keyword for the `Review` signature as indicated in figure 5.1. It has a previously delegated obligation as its target through the `target` relation of the `PolicyObject` it is extended from. `Evidence` determines what the later discharge of such a delegated obligation has to produce to convince the delegator that the obligation has indeed been performed. At this level, evidence serves as an abstraction for what eventually has to be produced, but not that it has been produced. The later would require a notion of discharging and enforcing obligations (compare, for example, [Minsky and Ungureanu, 2000]), which is not part of this framework. Evidence is reviewed by the specific actions of the review that has the obligation specifying the evidence as its target. This is reflected in the design decision to extend specific review actions from a general action as shown in figure 5.1. Additional constraints not further described here ensure that a review action may thus only be defined by a review obligation and vice versa.

The natural question to ask is how this concept of a review integrates with the definitions made in the previous section 5.3.3, that consider the distinction between general and specific obligations. Alloy does not initially assume that an extended signature is disjoint from the signature it is extended from. This is used as a convenient way of capturing that there may also be general and specific review obligations. Not having defined a review to be disjoint in the initial specification 5.1, a review may thus assume the type of an `Obligation` or `ObligationInstance`. The only requirement that needs to be explicitly stated in fact 5.6 is that a review instance can only have a general obligation of the type review and vice versa.

Alloy Fact 5.6 *An obligation instance is a review if and only if the general obligation it is an instance of is a review.*

```
fact {all s : State | all o : ObligationInstance |
 o in Review <=> (s.s_has_instance).o in Review
}
```

What are the effects of these assumptions? As it will be made clear in section 8.2.2, it must have been defined earlier how a review is performed. 'Earlier' in this case means that at the time a general obligation is assigned, the corresponding general review is assigned in parallel if delegation and review have to be supported. Thus, when an obligation instance is delegated, a review instance is created on the basis of the corresponding general review obligation. This instance now defines what review actions have to be performed on some evidence. As a result, the review may generate some evidence as well.

The control principles of delegation and review will be discussed in detail in chapters 7 and 8. Specifically section 8.2.3 will further clarify the constraints informally discussed here.

5.5 Roles and positions

So far, roles have only been given meaning in the form of a structuring mechanism to relate principals and policy objects. In fact, in the current context, there is no further meaning that should be given to them. The discussions in, for example, [Sandhu et al., 1996, Lupu, 1998, Schaad et al., 2001] support this by showing that any additional semantics are application dependent. Nevertheless, the interpretation of roles as opposed to positions in this context needs to be clarified as this has implications for the later definition of controls.

We believe roles to have a functional character, partially describing what a principal can or must do. The control principle model captures this through the assignment of authorisation and obligation policy objects. A principal may be member of several roles and this assignment is not static but may change according to the natural organisational changes taking place.

There is, however, a requirement to capture the permanent representation of a principal in an organisation which is commonly referred to as the position of principal [Buehner, 1994]. This position assignment does usually not change unless a principal joins or leaves the company, or is promoted. Such positions have with them a fixed set of authorisations and obligations which characterise the position. Positions are part of an organisational command and control hierarchy expressed in the form of supervision relationships. This is why we see positions as a specific kind of role in this context. It is clear that this perception does not fully capture real-world organisational complexity, but it will allow for the later expression of certain controls. There are organisations who employ similar constructs for access control purposes, defining roles as a combination of functions and positions [Schaad et al., 2001]. Positions may have some specific attributes which characterise them. These attributes can be anything which makes sense within the given organisational context. A few examples might be attributes indicating:

- A geographical location, e.g. North and South;
- A numerical value, e.g. the restriction to only handle accounts under £10,000;
- A qualifying string representing a legal construct such as power of procuration.

In the context of this conceptual model it is not possible to enumerate all eventual attributes. As such some kind of abstraction is required which then needs to be given a meaningful semantics in an organisational context. A signature `Attribute` is introduced to cater for this and the general `Object` is related to such an attribute via the relation `has_attribute`. We imagine, for example, the position of a `Sales Manager`. This could be assigned to an attribute `Region South`. Any principal occupying that position must also be assigned with that attribute. This situation corresponds directly to the first item in the above list. It is clear that this abstraction is very crude, but it will suffice our purpose and we have defined an example in fact 10.7 where we use such attributes to control delegation activities. Using a fully fledged object-oriented specification language like the OCL would avoid the explicit construction of object attributes as we did, but it does not have the advantages Alloy provides.

5.6 Role hierarchies

Role hierarchies [Sandhu et al., 1996] or role graphs [Nyanchama and Osborn, 1999] have been identified as one possible means to reflect parts of the organisational structure. They may yield benefits with respect to a system's administration, but they may also be a source of conflict with other systems policies [Moffett and Lupu, 1999, Schaad and Moffett, 2001]. Thus, it is not surprising that some communities advocate their use [Ferraiolo et al., 2001] whilst others do not [Yao et al., 2001]. Yet, this discussion is outside the scope of this thesis.

There are, nevertheless, some interesting properties and problems that justify the integration of a basic form of role hierarchies into the control principle model. First of all, they are important when talking about organisational structure. They must, however, by no means be mistaken to always map exactly to that structure, as we discussed in both [Kern et al., 2002] and [Kern et al., 2003]. Secondly, role hierarchies have been identified as a possible source of conflict with respect to separation controls [Kuhn, 1997]. Lastly, the general suitability of Alloy for the specification and later analysis of such hierarchies may be demonstrated.

It has also been shown by us [Kern et al., 2002] and others [Awischus, 1997, Moffett and Lupu, 1999], that there may be situations where it is desirable to let the same role participate in different hierarchies with possibly different semantics and relationships. These semantics and relationships are application dependent, and in this context only very restricted assumptions may be made considering role inheritance and exclusive roles in such distinct hierarchies.

For the rest of this section the discussion will concentrate on issues related to the integration of role hierarchies in the control principle model which includes:

- the minimum requirements for the specification of role hierarchies in this context;
- the possibility of a role being part of several distinct hierarchies;
- the semantics of a role hierarchy with respect to
  - the types of policy objects defined in this framework;
  - the general inheritance of policy objects and activation of roles.

5.6.1 Specifying a single role hierarchy

In this context, role hierarchies are represented through the relation `role_hierarchy`. The general intuition is that roles senior to other roles inherit from these junior roles.

Several constraints may be specified on such a hierarchy, the most important being the absence of any cycles, defined in the following fact 5.7.

Alloy Fact 5.7 *Cycle free role hierarchy.*

```
fact {all s : State | all r : Role |
 r !in r.^(s.s_role_hierarchy)}
```

Other constraints may be added to require such a hierarchy to be partially or even totally ordered. It may also be desirable to constrain the width and depth of a hierarchy to model organisational concepts such as span of control, e.g. compare [Mullins, 1999]. Such constraints have been defined in appendix B.1. Alternatively, the appropriate templates as provided by the Alloy distribution, e.g. for defining a partial or total order, may be used [Jackson, 2001].

5.6.2 Specifying multiple role hierarchies

There are situations where the same role may participate in distinct role hierarchies. We have initially identified this in [Schaad and Moffett, 2002a, Kern et al., 2002] and know of at least one real world example where this is required. However, to our knowledge this has not been explicitly discussed in the existing literature. Matrices [Mullins, 1999], a structure frequently found in today's organisations motivate such multiple role hierarchies. Here a principal may, for example, work for a department at the same time being allocated to a project whose members span across several departments. Consequently, he may occupy the same role which, however, is part of different role hierarchies. The advantages and problems of this form of organisation have been amply discussed, e.g. [Knight, 1977] and space and time do not permit a detailed discussion of the underlying organisational motivations. Nevertheless, we briefly show in the following how multiple role hierarchies may be defined in Alloy.

A single signature `RoleGraph` abstracts a set of roles related over the `role_hierarchy` relation. Similar to earlier parts of the specification, this signature and its relations may be specified on their own or as part of a state.

Alloy Specification 5.3 `RoleGraph` *signature and extended* `State`*.*

```
sig RoleGraph{
 role_hierarchy: Role -> Role,
 exclusive: Role -> Role}

sig State{
 ...
 s_role_hierarchy: RoleGraph -> Role -> Role,
 s_exclusive: RoleGraph -> Role -> Role}
```

Since we do not declare this `RoleGraph` as `static`, there may be more than one role graph, and a role may be a member of several such graphs. The same constraints as discussed in the previous section can be defined for these. However, we then have to additionally quantify over all role graphs, e.g. to ensure acyclic hierarchies as shown in fact 5.8.

Alloy Fact 5.8 *All role hierarchies are cycle free.*

```
fact {all s : State | all rg : RoleGraph | all r : Role |
 r !in r.^(rg.(s.s_role_hierarchy))
}
```

The above `RoleGraph` signature also includes an `exclusive` relation needed for the definition of mutually exclusive roles as described in the following section 5.7. Here we note that some roles may be exclusive to each other in one hierarchy but not in or across some other.

5.6.3 The semantics of role hierarchies

In this context we introduce two possible semantics that may be given to role hierarchies. These consider the direct inheritance of policy objects and the selective activation of inherited roles. This follows what has been described in [Sandhu, 1998], distinguishing between permission inheritance (usage) and activation hierarchies, catering for situations where the activation hierarchy should extend the inheritance hierarchy.

As initially discussed, the control principle model describes the assignment of general policy objects to roles. Since policy objects can be authorisations or obligations, we need to discuss the treatment of these with respect to the two semantics.

5.6.3.1 Policy object inheritance

One interpretation of a role hierarchy is that senior roles inherit the policy objects of junior roles. This means that if a principal `p1` occupies a role `r1` which is senior to some other role `r2` as defined in the relationship `r2 in r1.role_hierarchy`, then `p1` has `r2`'s policy objects at his discretion.

In the case of authorisations, no conceptual difficulties arise as this may be compared to the inheritance of permissions in the RBAC96 model [Sandhu et al., 1996]. There is no global constraint that needs to be defined. Instead, functions that evaluate which authorisations or roles a user holds such as `all_inherited_roles()` defined in function 6.13 have to consider possible hierarchies. A more specific example for this is given in section 6.3.4.2.

The question that arises is whether this inheritance should also be the case for obligations, and what the actual meaning of inheriting obligations is? As it was discussed in section 5.3.2, an obligation should be uniquely assigned to a principal, and we made the distinction between general and specific obligations to be able to assign obligations to roles, at the same time demanding the unique assignment for specific obligations. It seems natural to treat these general obligations like authorisations with respect to role inheritance, as both should ideally propagate together. If obligations were not inherited, then a principal may end up with more authority then actually needed with respect to the general obligations of his occupied roles. Thus, a general obligation is inherited between roles like an authorisation.

Any principal `p1` occupying a role `r1` senior to some other role `r2`, where `r2` is subject to a general obligation `gob_1` may then be subject to an instance `iob_1`, where `iob_1 in gob_1.has_instance` holds. We also have to note that this now requires an alteration to the requirement enforced in fact 5.4 that does currently not take role hierarchies into consideration. To summarise, every specific obligation a principal holds must be an instance of a general obligation he is either directly subject to, or over a role, or via role inheritance.

5.6.3.2 Role activation

The second possible meaning of a role hierarchy is that principals in senior roles may activate roles junior to them. This means that if a role `r2` is junior to a role `r1` expressed as `r2 in r1.role_hierarchy`, then any member of role `r1` must be considered as being a legitimate member of `r2` and is allowed to activate `r2`. Since we have not defined any constraint on the activation of roles yet, we do this in fact 5.9, taking a possible role hierarchy into consideration. A principal may thus be allowed to activate a role if he is either a direct member of the role or occupies a role that is senior to the role to be activated. This activation is abstracted through a relation `s_has_active`.

Alloy Fact 5.9 *Extended role activation constraint.*

```
fact {all s : State | all r2 : Role | all p : Principal |
      r2 in p.(s.s_has_active) =>
        p in r2.(s.s_has_member) ||
        (some r1: Role | (p in r1.(s.s_has_member) &&
                          r2 in r1.^(s.s_role_hierarchy)))
}
```

There are no major conceptual issues considering the distinction between authorisation and obligation policy objects with respect to the activation of inherited roles. It must only be clear that if a role is not activated by a principal, then he should have no specific obligations which are instances of a general obligation of that role.

5.7 Exclusive roles

Mutually exclusive roles are one concept for enforcing separation controls as later discussed in section 6.2. Their definition is application dependent and is based on an evaluation of the criticality of the authorisations defined by the roles with respect to the organisational context. This means that for some organisational reasons, pairs of roles were defined as being incompatible. An example might be the two roles of a programmer and configuration manager in a software company. One principal should not hold both such roles as the result might be a possible corruption of the code repository [Schaad and Moffett, 2001]. Several constraints must hold on this exclusive relationship [Kuhn, 1997], [Simon and Zurko, 1997].

In terms of the Alloy specification, the constraints that are enforced are that no role should be exclusive to itself, a constraint that is obvious to the human specifier but not to the automated analysis facilities of Alloy.

Alloy Fact 5.10 *A role is not exclusive to itself*

```
fact {all s : State | all r : Role |
 r !in r.(s.s_exclusive)
}
```

Equally important is the definition of the term 'mutual'. It should always be the case that if a role `r1` is exclusive to a role `r2` then `r2` is also exclusive to `r1`.

Alloy Fact 5.11 *Mutual exclusion of two roles*

```
fact {all s : State | all disj r1, r2 : Role |
 r1->r2 in (s.s_exclusive) => r2->r1 in (s.s_exclusive)
}
```

A constraint often demanded for maintaining separation rules in a role hierarchy is that two exclusive roles should have no common senior role [Kuhn, 1997]. This is done by expressing that if two roles `r1` and `r2` are exclusive, then it must be the case that there is no third role `r3` which is part of the intersection (`&`) of `r1`'s and `r2`'s senior roles (i.e. the transpose `~` of the transitive closure `^`).

Alloy Fact 5.12 *Exclusive roles must have no common senior.*

```
fact {all s : State | all disj r1, r2, r3 : Role |
 r1->r2 in (s.s_exclusive) =>
  r3 !in (r1.~^(s.s_role_hierarchy) & r2.~^(s.s_role_hierarchy))
}
```

We will expand on this latter constraint in the context of our discussion in section 6.3.

5.8 Actions

In the context of this model an action is defined following the notion of an object-oriented method. It consists of an operation that can be performed on an object. This object must be identical to the object specified in the target of the policy object defining the action, such that executing the action can succeed. So an action can be formally defined as consisting of a tuple `Object -> Operation` as described in the following signature.

Alloy Signature 5.2 *Action signature.*

```
sig Action {
 consists_of: Object -> Operation
}
```

The object is any available and uniquely identifiable object in the system environment. The operation is any operation that has been specified as being executable in a meaningful way against the object. However, the granularity of the operation is dependent on the application context [Nyanchama and Osborn, 1999]. This view is similar to the RBAC notion of permissions as uninterpreted symbols, where each system defines operations and objects according to its level of abstraction [Sandhu et al., 1996]. For example, an operating system uses granular operations such as *read()* and *write()* on objects such as files and directories.

A database will have defined operations such as *select()* or *append()* on tables, while specific banking middleware, as studied in the context of [Schaad et al., 2001], may have defined operations like *alter_account_data()* on customer account objects.

However, for our later purpose of automated analysis this static view does not suffice as it does not show the state changing nature of actions. Where behavioral aspects are to be observed in our later automated exploration, we represent actions using Alloy functions. For example, the action of reading the account of a customer, which may be part of a more general authorisation to provide customer service, may be represented in the following function `read_account` 5.1. This function takes as its input the before and after state as well as objects of a defined type, in this case a principal and an account. The body of the function remains empty as we are not interested in its behavior. We will show how to apply this approach in the case study in section 10.2.

Alloy Function 5.1 *Using functions to represent actions: Reading an account.*

```
fun read_account (disj s,s': State, p: Principal, ac: Account) {
  ... //application specific content
 }
```

Unfortunately, such a function `read_account` cannot be referred to as an object over a relation, and we will discuss this in more detail in section 10.2.1.

To summarise, authorisations represent authority at the enterprise level, their actions are the specific implementations of this authority in the context of automated systems. Operations are the most fine-grained element, and remain hidden to anybody not involved in the implementation of actions. It will have to be decided in the relevant analysis context what level of granularity is required for representing the authority of a principal.

5.9 Alternative representation of the `subject` relationship

We have said that the policy objects in this model follow, in a very restricted form, the syntax and semantics of policies in the Ponder language as reviewed in section 3.4.1. A policy has subjects it applies to and targets it is enforced upon.

However, sometimes it is not natural to think in terms of a policy object and its subjects. There are situations where it is more natural to talk about a principal holding a policy or a policy being assigned to a role, modelled by the corresponding relations `s_holds_policy` and `s_assigned_to_role`. This is, however, just a different perspective and thus, the `s_subject`, `s_holds_policy` and `s_assigned_to_role` relationships have to correspond. The following fact 5.13 enforces this correspondence and expresses that if some policy is assigned to a role, then this role should be subject to the policy. In a similar manner, fact 5.14 defines that if a principal holds a policy, then the policy has the principal as its subject. The use of the biimplication in both facts ensures that no other objects apart from roles and principals can be the subject of a policy. This last requirement would have, of course, to be lifted if complex structures such as a policy having some other policy as its subject were to be analysed.

Alloy Fact 5.13 *A policy is assigned to a role iff the policy has the role as a subject.*

```
fact {all s : State | all pol : PolicyObject | all rol : Role |
 (pol -> rol) in s.s_assigned_to_role <=> (pol -> rol) in s.s_subject
}
```

Alloy Fact 5.14 *A principal holds a policy iff the policy has the principal as a subject.*

```
fact {all s : State | all pol : PolicyObject | all prin : Principal |
 (prin -> pol) in s.s_holds_policy <=> (pol -> prin) in s.s_subject
}
```

There are also situations where for reasons of better performance of the automated analysis it may be desirable not to use the `s_subject` relation at all, but only work with `s_holds_policy` and `s_assigned_to_role` relationships instead. The reason for this is our design decision of using a general `Object` and the application of Alloy's extension mechanism. We will discuss this later in section 11.4.3.

5.10 Maintaining a history

There are several situations where it is desirable to maintain a history about changing relationships between entities over states. For example, when a principal accesses an object deemed to be of some importance such as a specific account, then this access should be recorded. One reason for this is that specific controls in the system may be dependent on this information as we will discuss in chapter 6. In a more general sense the means for auditability must be provided as discussed in section 3.5. In the real world this would be done by maintaining various forms of logs [Pfleeger, 1997, Weber, 1998]. However, it is not feasible to record every change to the system state [Sandhu, 1990], and obviously it is a trade off to decide what information needs to be logged, ideally based on an existing security risk analysis and assessment. This is outside the scope of this thesis. The forms of history that have to be maintained for the later analysis of control principles are:

1. A history of accessed objects;
2. A history of delegated objects;
3. A history of revoked objects.

This is a subset of a similar list reported in the more general context of accounting audit trails in [Weber, 1998].

Why is it exactly this information that needs to be maintained? One main aspect of this thesis is the delegation of policy objects, and there are situations where it cannot be inferred from the information given in a current state, i.e. the existing objects and their relations, how this state was actually achieved. However, such past events may then in turn determine what transitions to the next state should be allowed. For example, the later definition of

separation controls requires information about the previous access, delegation and revocation activities of a principal.

There are several valid forms of how to maintain such a history, and we illustrate our explorative approach by comparing the two conceptualisations we experimented with.

5.10.1 Maintaining a specific history through specific relations

History may be defined as a single signature 5.3 which then specifies the relations which are important to keep track of.

Alloy Signature 5.3 *A history signature consisting of three defined relations to record a principals access of an object, and the delegation and revocation of an object between two principals.*

```
sig History {
 accessed_object:  Principal -> Object -> Role -> Authorisation,
 delegated_object: Principal -> Principal -> Role -> PolicyObject,
 revoked_object: Principal -> Principal -> Role -> PolicyObject
}
```

This signature may then be related to a state via a relation `s_history` where the `option` keyword requires a state to have one or no history, i.e. something or nothing happened.

Alloy Signature 5.4 *Integrating a history to a state.*

```
sig State{
 ...
 s_history: option History}
```

However, although this is theoretically a valid approach, it is not practical in terms of the intended automated analysis. The more fields a relation has, the larger is the search space that needs to be computed with respect to the involved signatures and the defined search scope. One way of mitigating this may be to define constraints that help to reduce the search space for the state signature and history relation. For example, it should always be the case that the history of a state has one or no entry, as defined by using the `option` keyword. This is because a transition from one state to the other is based on some state-changing function that does or does not need to be recorded. In many cases this property could also be ensured by the frame conditions of the state changing functions, but when generating arbitrary models it helps to reduce the number of any unwanted models. A second example for such a constraint may be to define that if there are entries in the `accessed_object` relation then there are no entries for the `delegated_object` and `revoked_object` relations.

Although feasible for analysing smaller examples, the approach suggested here proved to be incompatible with our intended automated analysis of larger examples involving several states as described in the later section 10.2. We thus looked at how to maintain a specific history through a specific signature.

5.10.2 Maintaining a specific history through a specific signature

The second and chosen approach to maintain a history in the context of this thesis addresses the computational problem outlined in the previous section by changing the columns of a relation into individual fields of a signature. Consider the following signature 5.5:

Alloy Signature 5.5 *Maintaining a delegation history through a specific signature.*

```
sig DelegationHistory{
  delegating_principal : Principal,
  receiving_principal : Principal,
  based_on_role : option Role,
  delegated_policy : PolicyObject
}
```

We now do not maintain the information about which principal delegated which policy object in the form of an explicit relation, but in an explicit signature with several binary relations. We can do this because we know that, for example, in the context of the delegation of a policy, exactly one principal delegates exactly one policy object to exactly one other Principal in between two states. This cardinality is indicated by the absence of the `set` keyword. The delegating principal may delegate on the basis of a direct assignment or over a role as indicated by using the `option` keyword. Specifically this latter point could not be resolved in a relation `Principal -> Principal -> Role -> PolicyObject` as there is no kind of *null* value in Alloy that would allow us to express that no role but a direct assignment was used for the delegation. A `RevocationHistory` and an `AccessHistory` signature have been defined in a similar way in appendix B.1.

Having defined three history preserving signatures raises the question of how to relate these to a state. We do this as shown in the following signature 5.6, declaring a distinct relation that links each history to a state.

Alloy Signature 5.6 *Integrating a specific history to the* `State` *signature.*

```
sig State{
  ...
  s_access_history: option AccessHistory,
  s_delegation_history: option DelegationHistory,
  s_revocation_history : option RevocationHistory
}
```

Using the navigation expressions we can then elegantly access the required fields. For example, in order to obtain the authorisation used by a principal to access an object in some state `s` we may write `(s.s_access_history).used_authorisation`.

With respect to the later automated analysis, the approach outlined in this section and expressed in signatures 5.4 and 5.5 proved to have significant computational as well as conceptual advantages over the approach in the previous section 5.10.1.

5.10.3 Constraints on the history

The definition of a history mechanism requires the introduction of some constraints to allow for a meaningful later analysis.

To begin with we already said that a state may have one or no history. This has been indicated by using the `option` cardinality in signature 5.4, but we must also express that we demand only one kind of history for a state. This is defined in the following fact 5.15, which says that if a state has, for example, a delegation history then there has been no access or revocation.

Alloy Fact 5.15 *Only one kind of history for a state.*

```
fact {all disj s : State | some s.s_delegation_history =>
                                  no s.s_revocation_history &&
                                  no s.s_access_history
                           some s.s_revocation_history => ...
}
```

Additionally, a history must not be shared between states although the actual entries may be identical, expressed as fact 5.16.

Alloy Fact 5.16 *No shared kind of history for any two states.*

```
fact {all disj s1, s2 : State |
      no s1.s_revocation_history & s2.s_revocation_history &&
      no s1.s_delegation_history & s2.s_delegation_history &&
      no s1.s_access_history & s2.s_access_history
}
```

Certain functions define the conditions under which a state change may happen and how a history may be changed accordingly. For example, a function `access_object()` requires a principal to have the authorisation to access some object. There are, however, situations in our later analysis where we would like to look at some properties without explicitly defining such constraints as part of functions like the above. We may then define such a constraint in the form of the following fact 5.17. This expresses that if some state has a history for a principal accessing an object, then the used authorisation must have been in the set of policies available to the principal at that time.

Alloy Fact 5.17 *A principal must have the authorisation to access some object and subsequently change the history of a state.*

```
fact {all s : State |  all p : Principal | all s : State | all o : Object |
 some p & s.s_access_history.accessing_principal &&
 some o & s.s_access_history.accessed_object =>
  some (s.s_access_history.used_authorisation &
        all_available_auth_policies(p, s))
}
```

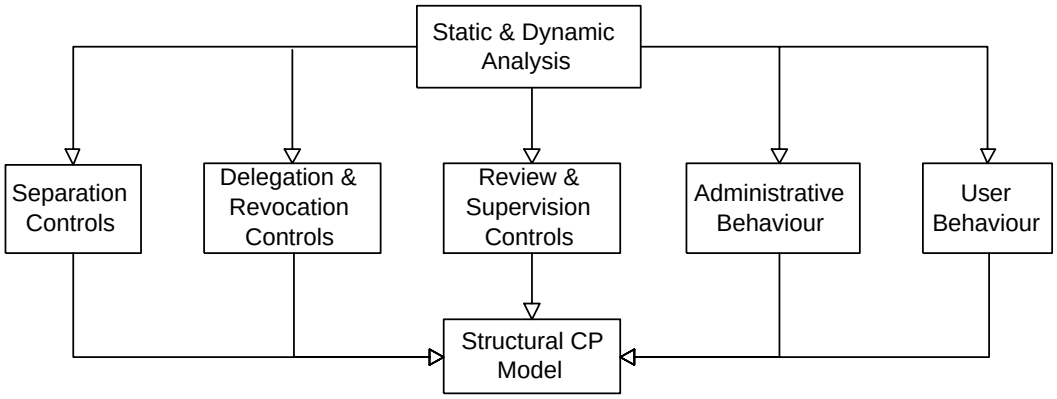

Figure 5.3: Modular structure of specification.

5.11 Additional design considerations

5.11.1 Modular specification structure

Following good software engineering principles it is desirable to structure the specifications underlying the proposed framework. Alloy's concept of modules allows for the separation into separate, partially dependent specifications as shown in figure 5.3. The arrows indicate a dependency relationship between the modules, defined by the `open` keyword in each such module. For example, the module describing delegation and revocation controls requires the signatures and relations defined in the structural control principle module. Most of the signatures and constraints in this chapter are part of the structural control principle module and explicit references to the appendices have been provided throughout.

The purpose of each module can be summarised as follows with the underlying basis being the conceptual model described in the previous sections 5.2 - 5.10:

- The control principle structure module contains the basic signatures, relations and facts as shown in figure 5.1. This module is defined in appendix B.1.
- The separation controls as discussed in chapter 6 are specified in a separate module, defined in appendix B.2.
- The delegation and revocation controls as discussed in chapter 7 are specified in a separate module, defined in appendix B.3.
- The review and supervision controls as discussed in chapter 8 are specified in a separate module, defined in appendix B.4.
- Administrative activities, such as the alteration of the role hierarchy, are part of the administrative behaviour module. This is defined in appendix B.5.
- User behaviour such as accessing an object is modelled in the user behaviour module. This is defined in appendix B.6.
- The static and dynamic analysis module defines a set of assertions about a state and sequences of states. These are defined in appendix B.7.

5.11.2 Specifying recurring functions

We have briefly mentioned in section 4.4.1 that apart from acting as constraints, functions may also be used to return a value in the form of a set of some objects. Thus, we describe a collection of functions needed at various points in the framework. These functions mainly consider the roles and policy objects held by a principal in some or several states, as well as his previous access or delegation activities. Thus, this collection can be summarised informally as consisting of the following two groups:

- Functions evaluating structural aspects, i.e. assignments of objects such as the authorisations available to a principal through his exclusive roles.
- Functions evaluating behavioral aspects, i.e. forms of history such as the objects accessed by a principal up to a given state using a specific role.

The following two sections give more detailed examples of some of the defined functions used frequently in the context of this framework. A complete listing is provided in appendix B.1.

5.11.2.1 Assignment evaluation

In the context of the later discussion of separation controls we often require the exclusive roles a principal is a direct member of. This may be expressed in the following function 5.2 `all_available_exclusive_roles()` which returns all exclusive roles for a principal. The expression `set Role` indicates the expected return type as delivered by the `result` expression in the body of the function. We refer to [Jackson, 2001] for a more detailed discussion.

Alloy Function 5.2 *Return all exclusive roles a principal is a direct member of.*

```
fun all_available_exclusive_roles(p:Principal,s:State):set Role{
 result = {role : Role |
           some p & role.(s.s_has_member) &&
           some role.(s.s_exclusive)}
}
```

Certain functions may deliver subsets of the objects returned by some other functions and the above function may then be used as part of the following function `all_excl_authorisations` 5.3 to return all the authorisations acquired over exclusive roles.

Alloy Function 5.3 *Return all authorisations acquired over exclusive roles.*

```
fun all_excl_authorisations(p:Principal,s:State):set Authorisation{
 result = {auth : Authorisation |
           auth in (s.s_subject).(all_available_exclusive_roles(p,s))}
}
```

Where possible, such a composition has been checked by means of assertions. For example, in the simplest case we may assert that we expect one function to deliver a subset of the other.

To simply obtain all the direct and role-based authorisations of a principal, the following function 5.4 has been defined.

Alloy Function 5.4 *Return all direct and role-based authorisations for a principal.*

```
fun all_available_auth_policies (p:Principal, s:State): set Authorisation{
 result = {auth : Authorisation |
   (auth in (s.s_subject).((s.s_has_member).p)) ||   //over a role or
   (auth in (s.s_subject).p)}                        //direct
}
```

This does not include any evaluation of role hierarchies, and we show in section 6.3.4.2 and function 6.13 how this may be done.

5.11.2.2 History evaluation

When looking at sequences of states we need to be able to assess what happened up to a certain point. For example, we may require information about the objects a principal has accessed so far through a specific role. This is captured in the following function 5.5.

Alloy Function 5.5 *Return set of objects a principal accessed so far through a role.*

```
fun return_set_of_accessed_objects_through_a_role_so_far
 (p: Principal, r: Role, cstate: State): set Object{
       result = {o : Object | some s : State |
                 s in states_so_far(cstate) &&
                 p in s.s_access_history.accessing_principal &&
                 o in s.s_access_history.accessed_object &&
                 r in s.s_access_history.used_role}
}
```

This function makes use of the function `states_so_far()` which returns all states before some defined current state `cstate`.

The following two functions 5.6 and 5.7 will also be needed later, e.g. in section 6.2. They deliver the set of all authorisations used by a principal up to a current state and the set of all authorisations used by a principal on some object. The fact that we consider a current state `cstate` in the first function, while we implicitly quantify over all states in the second function, shall emphasize that such functions may have to be adjusted, depending on the context they are used in.

Alloy Function 5.6 *Return the set of all authorisations a principal used so far (as represented by a current state* `cstate`*).*

```
fun return_set_of_all_authorisations_used
 (p : Principal, cstate : State) : set Authorisation {
  result = {auth : Authorisation | some s: State | some o : Object |
              s in states_so_far(cstate) &&
              auth in s.s_access_history.used_authorisation &&
              p in s.s_access_history.accessing_principal &&
              o in s.s_access_history.accessed_object}
}
```

Alloy Function 5.7 *Return the set of all authorisations a principal used on an object.*

```
fun return_set_of_authorisations_used_on_object
 (p:Principal,o:Object):set Authorisation {
  result = {auth : Authorisation | some s : State |
              auth in s.s_access_history.used_authorisation &&
              p in s.s_access_history.accessing_principal &&
              o in s.s_access_history.accessed_object}
}
```

5.12 Evaluation and related work

The model presented in this chapter shows several similarities to existing work, most importantly to the three frameworks and models summarised in chapter 3, which we have referred to throughout. In the following, a further evaluation of the presented control principle model is given with respect to these frameworks. This is, however, only done for some specific characteristics of each model and is not intended as a complete comparison.

As in the RBAC96 model reviewed in section 3.3, the central concept of the control principle model is the role as a structuring mechanism to relate principals and policy objects. RBAC, however, does only consider authorisations and not obligations. In addition, RBAC excludes the direct assignment of policies to a principal, while the control principle model explicitly defines such a relationship between a principal and a policy object in terms of the `subject` relationship. RBAC96 only incorporates roles and does not distinguish between roles and positions as a specific type of role. This distinction is made in the control principle model context and it has been discussed that the relation between a principal and a position is based on qualifying attributes. In RBAC96, principals activate roles through sessions. This is not part of the control principle model but has been replaced through a direct relation `s_has_active` representing role activation in a state. In the context of the RBAC96 model, role hierarchies have been identified as a possible mechanism to support administration. The control principle model also has a very simple hierarchy mechanism, which allows for the participation of a single role in several distinct hierarchies. This has not been explicitly

considered by the RBAC96 model or any other related role-based models as far as we are aware of.

The similarities and differences between the Ponder framework and RBAC96 have been discussed in detail in [Lupu, 1998] and as such many of the points made in the previous section could be equally discussed when comparing Ponder and the control principle model. One of the most important characteristics is that the control principle model implicitly assumes the semantics of Ponder's authorisation and obligation policy objects. However, Ponder does not have a concept of reviewing delegated obligations through explicit review obligations and evidence as we will show in section 7.8.2.2. While the control principle model and the Ponder framework have certain commonalities, explicit in our adoption of the `Authorisation` and `Obligation` policy types and `subject` and `target` relations, they differ in their underlying motivation. The Ponder framework defines a complete approach to policy-based systems management for real-world systems such as Windows 2000. This resulted in the Ponder policy specification language, an accompanying policy specification workbench and enforcement mechanisms [Damianou, 2002]. The initial motivation for the control principle model was to abstract from the actual enforcement and definition of concrete policies to provide a basis for specification and subsequent automated analysis and exploration of structural and behavioral properties. There are some means by which this can be done in Ponder, namely conflict analysis on the basis of policy modalities. This cannot be done in the context of the control principle model as there are not modalities. We have experimented with integrating a basic form of modalities into the control principle model, but as this was not very efficient and did not yield any further results with respect to the insights documented in [Moffett and Sloman, 1994] and [Lupu, 1998] we dropped this approach.

OASIS [Bacon et al., 2002] is in a way more similar to RBAC96 although emphasis was put on the formal justification and integration into the existing event architecture. The differences to RBAC have been summarised in section 3.3.2. Like RBAC96, OASIS only specifies authorisations, we, however, also consider obligations. Roles in OASIS are parameterised. For example, a role for a primary care doctor might be parameterised with both the doctor's identifier and that of a patient. We have not made any specific provision for such parameters in our model as this would be difficult to realise in Alloy. OASIS is a rule-based framework. The activation and use of roles and authorisations in OASIS is explicitly controlled by role activation, role membership and authorisation rules. For example, a possible role activation policy for the above primary care doctor is that the patient must be registered with the doctor. We have not explicitly modelled such rules as our emphasis is on modelling structure and behaviour. One specific aspect of OASIS is that it explicitly excludes role hierarchies, we in contrast explicitly considered various forms of such hierarchies. The OASIS notion of appointment as a form of delegating authority will be discussed in detail in the context of delegation controls in the later section 7.8.1.2. Here we only note that in OASIS appointment is performed on the basis of application specific attributes similar to those we discussed in the context of defining positions in section 5.5.

Further comparisons and evaluations of our work with respect to these three frameworks will be provided in the appropriate sections throughout the following chapters.

5.13 Chapter summary and conclusion

This chapter presented the basic structural properties and assumptions underlying the conceptual control principle model. This included:

1. A general overview of the model and the corresponding Alloy signatures representing the basic model elements and relations between them in section 5.2;
2. A detailed discussion of selected components and extensions to this initial overview with a focus on:
   - the modelling of authorisations and obligations in section 5.3 focusing on the distinction between general and specific obligations in section 5.3.3;
   - the concept of review obligations, review actions and evidence in section 5.4;
   - the distinction between roles and positions in section 5.5;
   - the definition of role hierarchies and exclusive roles in sections 5.6 and 5.7;
   - the definition of actions in section 5.8;
   - the definition of alternative representations of the `subject` relation in section 5.9;
   - the definition of a history mechanism in section 5.10.
3. A description of the modular architecture of the specification, and listing of frequently used functions in section 5.11;
4. A comparison and evaluation of related work in section 5.12.

This model now forms the foundation for a more detailed discussion on how to specify and analyse separation controls in chapter 6, explore delegation and revocation controls in chapter 7, and define review and supervision controls in chapter 8. In addition, the techniques and engineering decisions we made here will later be reflected in the second part of our case study in chapter 10.

The constraints defined in the course of this chapter are only a subset of the constraints defined in the modules of appendix B. The following chapters will introduce additional constraints and discuss changes to some of the facts and functions defined here. This emphasises that, depending on the context of any analysis, the appropriate constraints provided by our model must be chosen and, if necessary, adjusted.

Chapter 6

Separation Controls

Separation controls are probably the so far best understood type of control principle as it is indicated in the variety of existing work, specifically in the areas of role-based access control and policy-based systems management. Yet, hardly any work seems to address separation within the context of other control principles like those we are going to present in the following chapters.

However, before can we analyse and explore separation controls and their relationships, we need to investigate the existing work in more detail, eventually choosing suitable taxonomies. These will be represented by a set of separation controls defined in terms of our control principle model. Thus, the main points addressed in this chapter are:

1. A discussion on the organisational motivations for defining and enforcing separation controls in section 6.1.1;
2. An initial review of existing work on separation controls, followed by a more specific discussion of separation taxonomies in the context of role-based models in sections 6.1.2 and 6.1.3;
3. The definition, clarification and exploration of a selected set of separation controls in the context of our control principle model in section 6.2.

This is followed by a demonstration of how to use Alloy's analysis facilities for

1. Asserting the validity of individual and composed separation controls in sections 6.3.1 and 6.3.2;
2. Asserting the validity of the degree of shared authorisations in section 6.3.3;
3. Exploring separation controls in the presence of role hierarchies in section 6.3.4.

A discussion and evaluation of other related work is part of section 6.4.

6.1 Separation controls as an internal control principle

6.1.1 Organisational motivation for defining separation controls

Separation controls are probably the so far best understood control principle, as indicated by the variety of existing work. Specifically research in the areas of role-based access control, e.g. [Ferraiolo and Kuhn, 1992, Chen and Sandhu, 1995, Sandhu et al., 1996] and distributed systems management, e.g. [Belokosztolszki and Moody, 2002, Damianou, 2002] has led to the definition of taxonomies and frameworks, that will be reviewed in the course of this section. Although the origins of this principle cannot be clearly identified, it is obvious that the development of organisational theory, e.g. [Pugh, 1990, Mullins, 1999], and internal control and accountancy frameworks, e.g. [COSO, 1992, BIS, 1998] helped in their definition and possible ways of implementation. The two main application areas are the prevention of fraud due to the misuse of powers and the preservation of integrity.

One classic example when talking about separation controls is that of preventing fraud committed by the purchasing officer in a company [Sandhu, 1988]. If he could perform all the necessary steps of creating and authorising an order, recording the arrival of the item, recording the arrival of the invoice and finally authorising the payment, it would be easy for him to place an order with a fictitious company he owns, record a non-existing arrival, pay to the company, and add the non-existing goods to the books. Only the end-of-the-year inventory would reveal the discrepancy between the books and the physical stock. One way of enforcing a separation control in this context would be to not let a principal have all the necessary authorisations for each required step in this process. A more relaxed variation may be to not allow him to perform all the steps on his own. This is sometimes referred to as a dual control since two or more people are needed for the execution of a critical process. Another example of separation as an integrity preserving control could be in the context of a software company [Schaad and Moffett, 2001]. Here a programmer may be required to not test his own code. Likewise, when considering the previous cheque processing example, a single programmer should not be involved in the programming of all the necessary transactions of an accounting application, since he might install some hidden backdoors [Anderson, 2001].

Often, the term "separation of duties" is used in the context of examples like the above. Although commonly used, it does in many cases not really reflect its actual interpretation. This is because in its most general definition the separation of duties may be best described as a means of preventing (in)advertent error and fraud through the general or context-dependent limitation of a principal's authority. Thus, "separation of authority" may be a more precise term. We are not aware of any work that clearly defines the concept of duties, at the same time linking it to a principal's authority. Bearing this in mind, we will nevertheless use this initial term in the following review where other work does so as well.

We now provide a more specific review over the development of separation controls in information systems. This will reveal that there are various ways of specifying separation controls at different conceptual levels. The actual choice will be clearly dependent on the organisational context.

6.1.2 Initial work on separation controls

The control principle of separation has been applied in the commercial and military area long before it found its way into computer security. By partitioning tasks and assigning the sub-tasks to different entities, they are required to cooperate, thus reducing the risk of (in)advertent error or fraud. Separation of duties (initially referred to as "Separation of privilege") as a design principle for the protection of information in computer systems, was first referred to by Saltzer and Schroeder [Saltzer and Schroeder, 1975], themselves pointing to undocumented observations made by Roger Needham.

Clark and Wilson [Clark and Wilson, 1987] then introduced the separation of duties as one of the mechanisms to control fraud and error and ensure the external consistency of the data objects. Separation of duties is seen as a fundamental principle of commercial data integrity control. They later describe two distinct types of separation of duty called the static and dynamic separation of duty [Clark and Wilson, 1988], that essentially consider whether a separation constraint is based on information about one or several system states. These two types of separation of duties have since then been used as the primary means for categorising all existing variations of separation of duties.

Transaction control expressions are a notation for the description and implementation of static and dynamic separation of duty controls [Sandhu, 1988]. An information object is associated with the transaction control expression. The execution of operations on the object causes each transaction control expression to be converted into history. A partial history for transient objects (which have a short lifetime, such as a cheque) and complete history for persistent objects (with a long lifetime such as an account) is maintained. Persistent objects cannot be modified directly by a transaction. Modification of persistent objects is only possible through the side-effects of operations on transient objects. One possible separation of duty is enforced by the rule that for transient objects different transactions must be executed by distinct users. This approach is later extended in [Sandhu, 1990], discussing the definition of more refined dynamic separation controls, the presence of role hierarchies and the possibility of substituting the attribution for a principal and a task.

Later criticism by Nash and Poland addresses the fact that Sandhu did initially not consider the possibility of overlapping assignments of roles to transactions [Nash and Poland, 1990]. They further discuss the object-based separation of duty where every transaction against an object must be performed by a different user. Using Sandhu's transaction control expressions they show how to maintain a history of object access.

Parallel to this, named protection domains (NPD) are presented as a different approach to expressing separation of duty controls [Baldwin, 1990], [Giuri, 1995]. Instead of grouping individuals, privileges are grouped in a NPD. Such privileges might be all the operations needed to run the accounts-receivable portion of a general-ledger application. Only one NPD can be activated at a time, avoiding conflicting privileges or their misuse and making it possible to express separation of duties. It is further suggested to group individuals on the basis of the tasks they perform and not according to organisational hierarchies, an approach which may be seen as a precursor to current role-based approaches.

6.1.3 Separation controls in the context of role-based systems

It was soon realised that organisational roles could be used as a suitable concept for expressing more elaborate and fine-grained separation controls. Thus, it is not surprising that work on the separation of duties received particular attention with the progress made in role-based access control [Sandhu et al., 1996]. One main reason for this is the established concept of mutually exclusive roles as initially discussed in section 5.7. These are any roles that for some organisational reason have been declared to be exclusive, which may then have an effect on their relation to principals and assignment with policies.

In [Simon and Zurko, 1997] the separation of duties is defined as a multi-person control policy requiring that two or more different people are responsible for the completion of a task. This is done by spreading authority and responsibility for an action or task over multiple people. However, choosing a role concept similar to RBAC96 as the basis for the definition of a set of separation of duty controls, they effectively consider authority only. The underlying mechanism they use for implementing separation controls is the mutual exclusion of roles. Constraining the role membership, role activation and role use, leads to two main dimensions along which separation controls can be specified. These are referred to as static separation of duties (strong exclusion) and dynamic separation of duties (weak exclusion), where a further distinction is made between several kinds of dynamic separation. Two of these dynamic controls are the object-based and operational separation. However, we believe that this definition is not fully appropriate as we show and clarify later in sections 6.2.2 and 6.2.3. The identified separation controls are listed informally below. This will serve as the basis for our later specification of separation controls.

1. Static Separation of Duties

   (a) *(Simple) Static Separation of Duties*
   A principal may not be a member of any two exclusive roles.

2. Dynamic Separation of Duties

   (a) *(Simple) Dynamic Separation of Duties*
   A principal may be a member of any two exclusive roles but must not activate them at the same time.

   (b) *Object-based Separation of Duties*
   A principal may be a member of any two exclusive roles and may also activate them at the same time, but he must not act upon the same object through both.

   (c) *Operational Separation of Duties*
   A principal may be a member of some exclusive roles as long as the set of authorisations acquired over these roles does not cover an entire workflow.

   (d) *History-based Separation of Duties*
   A principal may be a member of some exclusive roles and the complete set of authorisations acquired over these roles may cover an entire workflow, but a principal must not use all authorisations on the same object(s).

These observed controls are further formalised in [Gligor et al., 1998]. Here, separation controls or policies are seen in the form of conjunctions of constituent properties of system states and state transitions. A distinction is made between three types of general policy properties called access-attribute (e.g. user-group membership invariants); access-authorisation (e.g. evaluation of access queries); and access-management (e.g. granting of access right). These properties show dependencies, and any omission or incorrect modification of dependent properties may result in ineffective separation policies. Based on these observations eleven types of separation of duty controls are identified. These are formalised and extended variations of those listed in [Simon and Zurko, 1997] and earlier work reported in [Clark and Wilson, 1987, Nash and Poland, 1990, Ferraiolo et al., 1995]. It is further shown how these relate to each other and may be composed if, for example, static or dynamic separation of duties is enforced. Unfortunately, it is not further shown how these separation controls relate to the general policy properties of the underlying role-based system. More specifically, there is no description how any change to one of the system properties might render a separation control useless. We also believe that the suggested controls are overly restrictive. For example, the demanded properties of the suggested 1-step Strict-Static Separation of Duty constraint are that a) any two distinct roles in the context of an application do not have common members; b) any two distinct roles are not authorised to perform operations on the same object; and that c) any of those roles is allowed to perform at most one operation on the objects associated to the application. While this constraint is needed in the context of the given proof of the composability of the suggested separation controls, we do not see any useful area of application. In fact, the benefit gained from using roles might be questioned with respect to item c).

Parallel to this work, a more formal framework for implementing separation of duty controls is presented in [Kuhn, 1997]. As in [Simon and Zurko, 1997] the chosen underlying mechanism is the mutual exclusion of roles. However, the discussion is taken further by analysing relationships between different types of separation controls; properties of mutual exclusion of roles; and constraints on possible role hierarchies. The two main dimensions are again the static (authorisation-time) and dynamic (run-time) separation. Additionally, the degree to which privileges can be shared by exclusive roles is defined, and the complete and partial exclusion are the two extremes to which this can lead. Here only the basic taxonomy is listed below and more precise definitions will be given in section 6.2.5.

1. Authorisation-time/Static Separation of Duties
2. Run-time/Dynamic Separation of Duties
3. Degree of shared privileges between exclusive roles/other roles
   (a) Disjoint/Disjoint (complete)
   (b) Disjoint/Shared
   (c) Shared/Disjoint
   (d) Shared/Shared (partial)

Combining the static and dynamic separation with the complete and partial exclusion of shared privileges, the following four combinations of separation controls are obtained (Note that Kuhn actually uses the terms authorisation-time complete, run-time complete etc.):

1. Static Complete (`S/C`)
2. Static Partial (`S/P`)
3. Dynamic Complete (`D/C`)
4. Dynamic Partial (`D/P`)

These combinations then show the following relationships (where `=>` is the Alloy symbol for implication): `S/C => S/P => D/P` and `S/C => D/C => D/P`. The corresponding Alloy specifications and assertions are part of appendices B.2 and B.7.

An additional constraint on mutually exclusive roles is that any two such roles must not have a common upper bound in a partially ordered role hierarchy [Kuhn, 1997]. We have defined this earlier in fact 5.12. This also forbids the existence of a 'root' role containing all other system roles, which may be compared with the activation hierarchies described in [Sandhu et al., 1996] and the role graph approach outlined in [Nyanchama and Osborn, 1999].

6.2 Specification of separation controls in Alloy

Based on the reviews provided in the previous section, the following sections 6.2.1 - 6.2.4 will use the control principle model established in chapter 5 to define parts of the taxonomies initially suggested in [Simon and Zurko, 1997] and later refined in [Gligor et al., 1998]. These separation controls will be based on the notion of mutually exclusive roles, that were discussed in section 5.7. Additionally, the degree of shared authorisations and combination with static and dynamic separation controls will be specified in section 6.2.5 as suggested in [Kuhn, 1997].

6.2.1 Strict and relaxed role-based separation

To preserve a strict, or sometimes called static, separation, any two exclusive roles must not have a principal as a common member. This is expressed in function 6.1. If two disjoint roles `r1` and `r2` are exclusive, then the intersection of the sets of principals being members of these roles must be empty in all states `s`.

Alloy Function 6.1 *Strict role-based separation.*

```
fun strict_role_based_separation(){all s : State |
                                    all disj r1, r2: Role |
   r1->r2 in (s.s_exclusive) =>
      no r1.(s.s_has_member) & r2.(s.s_has_member)
}
```

The way we specified this strict separation control is representative for all the following definitions of separation controls. Using an Alloy function instead of a fact allows us to only use the separation control when actually needed. Facts would be valid throughout an entire specification and may thus be overly restrictive. Secondly, functions may or may not require any input values. For example, we may quantify over all the required signatures as in function 6.1, or demand for the current state to be provided as in function 6.4.

In a similar manner, a more relaxed separation control is defined in the following function 6.2, allowing for the assignment of exclusive role pairs to a principal but preventing their simultaneous activation, here abstracted by using the `has_active` relation. Two disjoint roles `r1` and `r2` may not be activated by the same principal in any state `s`.

Alloy Function 6.2 *Relaxed role-based separation.*

```
fun relaxed_role_based_separation(){all s : State |
                                    all disj r1, r2 : Role |
  r1->r2 in (s.s_exclusive) =>
    no (s.s_has_active).r1 & (s.s_has_active).r2}
```

This is often referred to as the dynamic separation of duty, e.g. [Clark and Wilson, 1987], because it can only be evaluated at the run-time of a system. However, the term dynamic separation of duty may be misleading in the context of our model as we only look at the properties of a single state at a time. For that reason we decided to name it a relaxed role-based separation property. In RBAC96, the concept of a session is used to relate principals and active roles and a principal can have several sessions active at the same time [Sandhu et al., 1996]. This is not needed here, as we had no demand for an analysis involving several sessions in this context. If desired, we believe that this could be implemented without difficulty.

6.2.2 Object-based separation controls

As the name implies, object-based separation controls impose constraints on the authorisations of a principal with respect to the objects in a system. We believe that this may be done on a static or dynamic basis. This clarifies the slightly misleading interpretation of [Simon and Zurko, 1997], who always classify object and operational separation as dynamic. Static object-based separation as defined in function 6.3 means that if a principal `prin` is a member of any two exclusive roles `r1` and `r2`, then the authorisations of these two roles must not have the same target object.

Alloy Function 6.3 *Static object-based separation.*

```
fun static_obj_separation() {all s : State | all disj r1, r2: Role |
                              all prin : Principal |
  r1->r2 in (s.s_exclusive) =>
    no ((Authorisation & (s.s_subject).r1)).(s.s_target) &
       ((Authorisation & (s.s_subject).r2)).(s.s_target)}
```

As a more relaxed form of this control, the dynamic object-based separation defined in function 6.4 allows a principal to be a member of two exclusive roles where the assigned authorisations overlap in their target objects. However, such a principal must not access the same object through both these roles. At this stage we require some information about the access history of a principal as defined in section 5.10.2 by function 5.5 `return_set_of_accessed_objects_through_a_role_so_far()`. Thus, if two roles `r1` and `r2` are exclusive in a state `s`, and a principal `prin` has accessed an object `o` through a role `r1`, then he must not access the same object through role `r2`.

Alloy Function 6.4 *Dynamic object-based separation.*

```
fun dynamic_obj_separation(cstate: State) {all prin : Principal |
                                           all disj r1,r2 : Role |
                                           all o : Object |
  some r1 -> r2 & (s.s_exclusive) &&
  some prin & r1.(s.s_has_member) &&
  some prin & r2.(s.s_has_member) =>
   no return_set_of_accessed_objects_through_a_role_so_far(prin,r1,cstate) &
      return_set_of_accessed_objects_through_a_role_so_far(prin,r2,cstate)}
```

We do not check for any overlap of targets here, but define dynamic object-based separation in more general terms on the basis of a mutual exclusion relationship only. The fact that this control is dynamic is implied in the definition of the `return_set_of_accessed_objects_through_a_role_so_far()` function, where we consider all states up to the current state explicitly. This current state `cstate` must be provided when calling the `dynamic_obj_separation()` function.

6.2.3 Operational separation controls

Operational separation controls are constraints on the authorisations of a principal with respect to the steps in a critical workflow such as that of issuing a cheque as described in section 6.1.1.

Ideally, this would require the presence of a workflow model. Although we see this as the next step our work may lead to as later discussed in section 11.4.1, it is not the immediate aim to define such a model in this context. Nevertheless, operational separation controls are important constraints that need to be expressed. Other approaches to specifying separation controls such as [Gligor et al., 1998] have used the concept of execution plans as an abstraction. We intend to abstract a workflow in a similar fashion. Thus, we defined a set of critical authorisations in the explicit signature 6.1 `Critical_Authorisation_Set`, that defines a set of `critical` authorisations dependent on the context.

Alloy Signature 6.1 *Critical authorisation set signature.*

```
sig Critical_Authorisation_Set {
   critical: set Authorisation}
```

Again we distinguish between static and dynamic operational controls as done in [Gligor et al., 1998], refining and clarifying the initial definition of [Simon and Zurko, 1997] in section 6.1.3.

The static operational separation function 6.5 defines that a critical set of authorisations `c_set` must not be a subset of the authorisations available to a principal `prin`. Here, we use the function 5.3 `all_excl_authorisations()` to return all authorisations currently available to a principal through his exclusive roles. This is, however, only one of many possible other definitions. Whatever the context of the situation demands, a different function may be used, for example, one considering all of a principal's (inherited) authorisations, and not only those which are exclusive.

Alloy Function 6.5 *Static operational separation control.*

```
fun static_op_separation () {all s : State |
                             all prin : Principal |
                             all c_set : Critical_Authorisation_Set |
  c_set.critical !in all_excl_authorisations(prin, s)}
```

The dynamic operational separation function 6.6 defines that a critical set of authorisations `c_set` may be a subset of the authorisations available to a principal `prin`, but he must not use all of them. The function 5.6 `return_set_of_all_authorisations_used()` returns the set of authorisations used up to the current state `c_state`.

Alloy Function 6.6 *Dynamic operational separation control.*

```
fun dynamic_op_separation (cstate: State) {
  all prin : Principal |
  all c_set : Critical_Authorisation_Set |
     c_set.critical !in return_set_of_all_authorisations_used(prin, cstate)}
```

6.2.4 History-based separation controls

As object-based and operational separation controls might be too strict in some cases, a history-based separation allows a principal to use all the authorisations of his roles, even if this covers a critical set, but he must not do this with respect to the same object(s). This implies that:

1. an object was either accessed more than once but not using all authorisations; or
2. all critical policies have been used but not on the same object.

These requirements can be captured in the following Alloy function 6.7:

Alloy Function 6.7 *History-based separation control.*

```
fun hist_separation () {all prin : Principal |
                        all c_set : Critical_Authorisation_Set |
                        all o : Object |
  c_set.critical !in return_set_of_authorisations_used_on_object(prin,o)
}
```

The function 5.7 `return_set_of_authorisations_used_on_object()` returns the set of authorisations a principal has used on some object. Here, we consider all authorisations available to a principal, and a variation may be to only consider the authorisations a principal holds through his exclusive roles.

6.2.5 Degree of shared authorisations

Authorisation policy objects may be shared by roles to different degrees. This is referred to as another dimension of separation controls [Kuhn, 1997]. In this context, four types of shared authorisation policies can be distinguished as defined in functions 6.8-6.11. These types show dependencies and specifying the possible degrees of shared policies as functions allows for their composition which we assert later in section 6.3.3.

Alloy Function 6.8 *Shared/Shared (Partial) - Authorisations can be shared by exclusive roles and other non-exclusive roles as long as each exclusive role is assigned with at least one authorisation not available to the other.*

```
fun ss (s : State, disj r1, r2: Role) {
 r1->r2 in (s.s_exclusive) =>
   some (((s.s_subject).r1 & Authorisation) -
         ((s.s_subject).r2 & Authorisation)) &&
   some (((s.s_subject).r2 & Authorisation) -
         ((s.s_subject).r1 & Authorisation))}
```

Alloy Function 6.9 *Shared/Disjoint - Authorisation policies can be shared by exclusive roles as long as each exclusive role is assigned with at least one authorisation not available to the other, but any authorisation assigned to an exclusive role must not be assigned to other non-exclusive roles.*

```
fun sd (s:State, disj r1, r2: Role) {
 r1->r2 in (s.s_exclusive) =>
   ss (s,r1,r2) &&
   no ((((s.s_subject).r1) & Authorisation).(s.s_subject) - r1 - r2) &&
   no ((((s.s_subject).r2) & Authorisation).(s.s_subject) - r1 - r2)}
```

Alloy Function 6.10 *Disjoint/Shared - Exclusive roles must not share any authorisation policy, but an authorisation policy assigned to an exclusive role may be assigned to other non-exclusive roles.*

```
fun ds (s:State, disj r1, r2: Role){
 r1->r2 in (s.s_exclusive) =>
   ss(s,r1,r2) && // this is required in analysis to avoid empty models
   no ((s.s_subject).r1 & Authorisation) &
      ((s.s_subject).r2 & Authorisation)}
```

Alloy Function 6.11 *Disjoint/Disjoint (Complete) - Exclusive roles must not share any authorisations, and an authorisation assigned to an exclusive role cannot be assigned to any other non-exclusive roles.*

```
fun dd (s:State, disj r1, r2: Role) {
 r1->r2 in (s.s_exclusive) =>
   ds(s, r1,r2) &&
   sd(s, r1,r2)}
```

These four degrees may then be combined with the strict and relaxed separation controls 6.1 and 6.2, following the four combinations suggested by [Kuhn, 1997] reviewed and summarised at the end of section 6.1.3. We only give an example for a strict/complete combination in the following function 6.12, and refer to appendix B.2 for the remaining three combinations.

Alloy Function 6.12 *Static complete - (Strict complete in our terminology)*

```
fun strict_complete() { all s : State | all disj r1, r2 : Role |
      r1->r2 in (s.s_exclusive) =>
       strict_role_based_separation() && dd(s,r1,r2)}
```

6.3 Analysis of separation controls

In this section we demonstrate how the validity of the defined separation controls may be asserted by means of automated analysis. Here only some examples are provided, and the appendix B.7 provides additional material. We begin by showing how individual separation controls may be analysed. These are all examples of very simple reformulations of expected properties, checked by using Alloy's `assert` mechanism. This is followed by the composition and assertion of separation controls. We then close this section with an analysis of separation controls in the presence of role hierarchies.

6.3.1 Individual separation controls

Consider the strict role based separation control defined in function 6.1. We may check the validity of this control by defining an assertion 6.1. This expresses that if two roles `r1` and `r2` are mutually exclusive, and principal `p1` is a member of `r1` then, as a consequence, `p1` may

not be a member of r2 if the strict separation property is required to hold. When checked, this assertion will not generate any counterexamples.

Alloy Assertion 6.1 *Assert strict role based separation control.*

```
assert strict_role_based_separation_assert {all p1:Principal| all s:State|
                                            all disj r1, r2: Role |
 some r1 -> r2 & (s.s_exclusive) &&
 some p1 & r1.(s.s_has_member) &&
 strict_role_based_separation() => no p1 & r2.(s.s_has_member)}
```

We may further define an assertion for the relaxed separation control defined in function 6.2, in that case checking for the activation of the exclusive roles through the has_active relation. So if two roles r1 and r2 are mutually exclusive, then a principal p1 may not have both of them active at the same time under the relaxed separation control. This is almost identical in its structure to the previous assertion 6.1 and is thus not shown here for reasons of space. However, we note that the role activation rule defined in fact 5.9 is required here.

For checking the dynamic object-based separation control defined in function 6.4, we make use of the function return_set_of_accessed_objects_through_a_role(), which, when given a principal and a role, returns the set of objects accessed by that principal in a sequence of states. We may then say that if two roles r1 and r2 are mutually exclusive, and the dynamic object-based separation property defined in function 6.4 holds, then no object may have been accessed through both these roles by principal p1. In other words, the intersection of the two sets of objects accessed through each role by the principal must be empty. This is expressed in the following assertion 6.2.

Alloy Assertion 6.2 *Assert dynamic object-based separation.*

```
assert dynamic_obj_sod_assert1{all p1 : Principal | all s : State |
                               all disj r1, r2 : Role |
 some r1 -> r2 & (s.s_exclusive) && dynamic_obj_separation(s) =>
   no (return_set_of_accessed_objects_through_a_role(p1, r1) &
       return_set_of_accessed_objects_through_a_role(p1, r2)}
```

In fact, a variation of this assertion initially generated a counterexample. Without going into the explicit details, the underlying reason was that we had implicitly assumed that if two roles are exclusive in one state, then they are exclusive in all states. In the counterexample we could, however, observe that the two roles r1 and r2 were exclusive in one but not the other state. This had to be addressed in terms of an explicit framing constraint excluding any changes to the s_exclusive relation over a sequence of states.

At this stage we would like to draw attention to the fact that within assertions such as the above we use expressions like some r1 -> r2 & (s.s_exclusive), although an expression r1 -> r2 in (s.s_exclusive) may be perceived as identical in its meaning. This is not always the case since every expression in Alloy is defined as a relation. Thus, there is

a possibility of `r1 -> r2` being empty, which in turn may lead to the generation of a counterexample in the form of an empty model when using the membership operator `in`.

As the last example we choose to state an assertion about the static operational separation control defined in function 6.5. This makes use of an abstracted set of critical authorisations defined in signature 6.1. Three authorisations `a1`, `a2`, `a3` are defined as critical in a set `c_set`. A principal `p1`, who holds `a1` and `a2` must not hold the third authorisation `a3` if the static operational separation is assumed to hold.

Alloy Assertion 6.3 *Assert static operational separation.*

```
assert static_op_separation_assert{all s : State | all p1 : Principal |
                                   all c_set : Critical_Authorisation_Set |
                                   all disj a1, a2, a3 : Authorisation |
  a1 + a2 + a3 = c_set.critical &&
  some a1 & all_available_auth_policies(p1, s) &&
  some a2 & all_available_auth_policies(p1, s) &&
  static_op_separation() => no a3 & all_available_auth_policies(p1, s)}
```

A possible assertion for validating our history-based separation is similar in its structure as shown in appendix B.7, and is thus not shown here for reasons of space.

6.3.2 Composed separation controls

In a similar manner we may state assertions about composed separation controls. For example, an initial assertion is that if the strict separation control defined in function 6.1 holds for a principal, then this implies that the relaxed separation control defined in function 6.2 holds as well. This may be expressed as follows:

Alloy Assertion 6.4 *Assert that if strict separation holds, relaxed separation holds*

```
assert strict_separation_implies_relaxed_separation {
 strict_role_based_separation() => relaxed_role_based_separation()}
```

Other examples for such a composition may be to assert that if the static object-based separation holds, then the dynamic object-based separation holds as well. The specification of the corresponding assertion is similar to the above assertion 6.4 and is documented in appendix B.7.

The taxonomy suggested in [Gligor et al., 1998] defines a set of eleven separation controls which can be composed as we discussed in 6.1.3. Specifically this taxonomy seems to have been driven by the idea of demonstrating composition rather than possible practical applications of the suggested controls. Not all of the separation controls suggested by us can be composed in such a way as they substantially differ. For example, some controls are based on exclusive roles whilst others use the notion of a critical set of authorisations. We thus only sought to apply such composition of controls where this makes sense, at the same time demonstrating the suitability of Alloy for this.

6.3.3 Analysing the degree of shared authorisations

A different kind of separation controls, based on the degree of shared authorisations, has been specified in section 6.2.5, following the approach of [Kuhn, 1997]. Certain composability criteria with respect to these established properties had been defined by Kuhn, and we have specified and checked these in terms of our control principle model as documented in appendices B.2 and B.7. An example is given in the following assertion 6.5. The Disjoint/Disjoint property defined in function 6.11 stated that two exclusive roles do not share authorisation policies, and policies assigned to an exclusive role are not assigned to any other non-exclusive roles. The Disjoint/Shared property defined in function 6.10 stated that exclusive roles must not share authorisation policies, but that authorisation policies assigned to an exclusive role may be assigned to other non-exclusive roles. Thus we may say that if function 6.11 `dd()` is assumed to hold, the weaker function 6.10 `ds()` must equally hold.

Alloy Assertion 6.5 *If a Disjoint/Disjoint property for shared authorisations holds then the Disjoint/Shared property holds as well.*

```
assert implication_ddds {all s : State | all disj r1, r2 : Role |
 dd(s, r1, r2) => ds(s, r1, r2)}
```

6.3.4 Separation controls and role hierarchies

The control principle model incorporates a notion of role hierarchies, defined in section 5.6, that we have not taken into consideration until now. In fact, as we will describe in the course of this section, the separation controls as we have specified them may be subject to conflict with the properties of a role hierarchy. We have described a similar situation in the context of an administrative role-based access control model in [Schaad and Moffett, 2002b].

The problematic issue of role hierarchies has been recognised, e.g. in [Sandhu et al., 1996, Moffett and Lupu, 1999], and as a consequence role-based models such as OASIS do not support them at all [Bacon et al., 2002]. Other approaches to integrated definitions of role hierarchies and separation controls like [Nyanchama and Osborn, 1999] and [Kuhn, 1997], define strict global constraints like exclusive roles having no common senior as expressed in fact 5.12. Thus, if role hierarchies and separation controls are required in a system, then these are usually formally specified to support proof. We are not aware of any work that describes an automated analysis approach to support such proof as we do.

Whether or not the presence of an inheritance relationship between roles will conflict with separation controls depends on the particular application context. In this context we may only demonstrate how to validate a specification. The definition of conflict detection and resolution mechanisms in a working system is outside this scope, although we have done some initial work on defining separation rules and role hierarchies in Prolog [Schaad, 2001], coupling these with a relational database system [Schaad and Moffett, 2001].

Our separation controls as defined in chapter 6 do not explicitly consider the presence of role hierarchies for reasons of performance. In the following two sections we will outline the

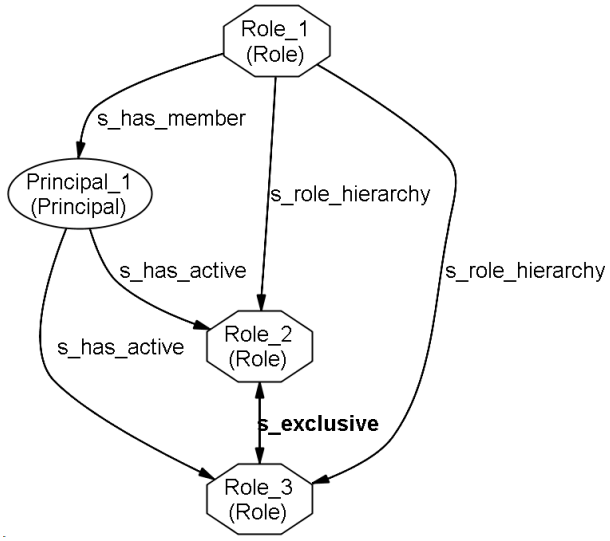

Figure 6.1: Activation role hierarchy.

conflicts to which this may lead, how Alloy assisted us in finding these, and what changes may have to be made to the separation controls. These two sections reflect the two semantics we may give to role hierarchies as discussed in section 5.6.

6.3.4.1 Activation role hierarchies conflicts

One possible case is that a role activation hierarchy may conflict with the relaxed role-based separation control defined in function 6.2. As discussed in section 5.6, one of the possible semantics we may give to a role hierarchy is that if a role `r2` is junior to a role `r1` then any member of role `r1` is considered to be a legitimate (although not direct) member of `r2`. This would then allow a principal to activate any such inherited role as defined in fact 5.9.

If now two roles `r2` and `r3` are exclusive to each other and junior to a role `r1`, then any principal being a member of `r1` would have to be considered to be an indirect member of `r2` and `r3`. This, depending on the organisational context, may be considered as a violation of the relaxed separation constraint when the principal activates `r2` and `r3`.

We became aware of this type of conflict when checking the strict separation control through the following assertion 6.6. If two roles `r1` and `r2` are exclusive, and the strict separation property holds, then these roles cannot be activated by a single principal.

Alloy Assertion 6.6 *Role hierarchy and strict role-based separation conflict.*

```
assert role_based_separation_assert_variation {all s:State|all p1:Principal|
                                               all disj r1, r2 : Role |
  r1 -> r2 in (s.s_exclusive) && strict_role_based_separation() =>
     no (s.s_has_active).r2 & (s.s_has_active).r1}
```

A counterexample was, however, generated, represented graphically in figure 6.1. Here we can see that Principal_1 can activate two exclusive roles Role_2 and Role_3 since both are junior to role Role_1 the principal is a member of.

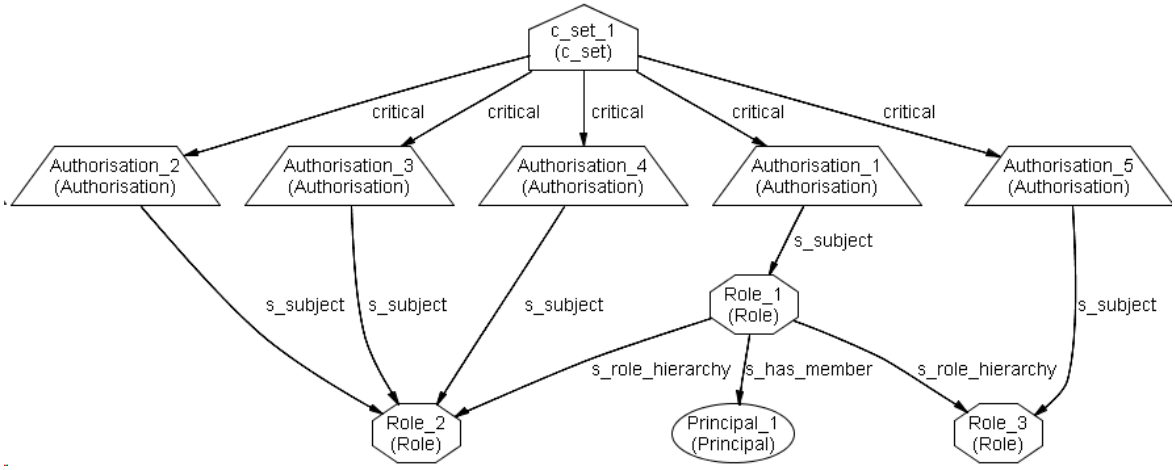

Figure 6.2: Inheritance role hierarchy.

6.3.4.2 Authorisation inheritance role hierarchy conflicts

As described in section 5.6.3.1, the second possible type of semantics we may choose to give to a role hierarchy is that of a senior role inheriting the authorisations of its junior roles. With respect to this definition, one possible conflict we detected during the analysis was that of an authorisation inheritance role hierarchy with a static operational separation of duty.

As defined in function 6.5, a principal should not be in possession of all the authorisations defined in a critical authorisation set. This static operational separation constraint uses function 5.4 `all_available_auth_policies()`. This function does, however, not consider any inherited roles and will thus only return the authorisations directly related to a role. It can be seen in figure 6.2 how principal Principal_1 is a member of role Role_1. This membership may then allow him to use all the authorisations defined in the critical set c_set_1 due to Role_1 being senior to Role_2 and Role_3.

Instead of listing the specific assertion this possible conflict was identified with (which is part of appendix B.7), we define a function 6.13 that takes role hierarchies into consideration. This yields all inherited roles for a principal and may be used for defining a separation property that holds in the presence of role hierarchies.

Alloy Function 6.13 *Return all inherited roles for a principal*

```
fun all_inherited_roles(p:Principal,s:State):set Role{
  result = {r : Role | r in ((s.s_has_member).p).^(s.s_role_hierarchy)}}
```

The static operational separation constraint may then have to be updated depending on the specific context of analysis.

6.4 Other related work

Five general types of conflict of interest are discussed in the context of the role-graph model presented in [Nyanchama and Osborn, 1999]. The model is based on sets of users, roles and

privileges. Roles, unlike in other models such as RBAC96, are only used to group privileges. Further distinctions include the grouping of users and assignment of such groups to roles, as well as an implication relationship between privileges. Senior roles inherit the permissions assigned to roles junior to them in the hierarchy. There are two explicit roles, MIN and MAX, representing the upper and lower bound of the graph respectively. Authorisation management is based on changes to the User/Group; User/Role; Group/Role; Role/Role; Role/Permission; and Permission/Permission relations. The types of conflicts that are discussed are User/User; User/Group; Group/Group; User/Role; Role/Role; Role/Permission; and Permission/Permission conflicts, with an in depth discussion on the latter three. Using the notion of conflicting roles, it is demonstrated how basic separation controls can be enforced within the constraints imposed by the role graph. We have experimented with this approach but did not find it in any way more expressive than the separation controls we finally chose.

The RSL99 language for the expression and analysis of separation properties is presented in [Ahn, 2000]. It is part of a framework for specifying separation of duty and conflict of interest policies in the context of the RBAC96 model. Separation of duties is understood as the partitioning of tasks and associated privileges so that the cooperation of multiple users is required. This is done by controlling membership in, activation, and use of roles as well as permission assignments. RSL99 is based on user, role and permission conflict sets. The separation properties that may be expressed in RSL99 are largely based on the frameworks of [Simon and Zurko, 1997] and [Gligor et al., 1998], that we partially described in the previous section. Although the language is based on a first-order logic, it is not flexible enough to express those properties of our control principle framework not related to separation. In fact, RSL99 does neither cater for the expression of any constructs that could not be expressed in other existing first-order logic languages, nor does it identify any new types of separation controls or relationships. Additionally, other more expressive authorisation languages, which can also express separation properties have been presented earlier, e.g. [Jajodia et al., 1997a].

In [Knorr and Weidner, 2001] separation of duty constraints are defined in the context of a petri-net based workflow model. The Prolog logic programming language is used for the expression of the model, the separation constraints and the analysis. In fact, there is a strong resemblance to our initial work presented in [Schaad and Moffett, 2001]. The main shortcomings of these approaches are that they both specify separation constraints in the context of specific examples and not in a more general form.

In the context of policy-based systems management, separation controls are part of a wider set of meta-policies, which are policies about policies [Moffett and Sloman, 1994, Lupu, 1998]. In the Ponder policy specification language, reviewed in section 3.4.1, separation policies are seen as application specific constraints which may be expressed in OCL [Damianou et al., 2001]. Examples of separation policies are given [Damianou, 2002], but no comprehensive overview is provided. The notion of meta-policies has also received attention in the context of distributed role-based access control systems like OASIS [Bacon et al., 2002, Belokosztolszki and Moody, 2002] which was reviewed in section 3.3.2. Examples for simple separation policies are given, but no comprehensive overview is provided.

6.5 Summary and conclusion

In this chapter we have discussed separation controls and investigated how some well-known separation controls may be expressed in the context of our proposed conceptual framework and formal model. This included:

1. A discussion on the organisational motivations for separation controls in section 6.1.1.
2. An initial review of existing work on separation controls in section 6.1.2 which led to the identification of suitable taxonomies for defining separation controls;
3. The definition of a selected set of separation controls in the context of our control principle model in section 6.2, which we believe to have further clarified the taxonomy provided by [Simon and Zurko, 1997];
4. A demonstration of the automated analysis facilities provided by Alloy for checking individual and composed separation controls as well as an exploration of the presence of different kinds of role hierarchies in section 6.3.

It is clear that the set of separation controls defined in this context is not complete with respect to all the possibly existing variations of this property. For example, we only consider sets of exclusive roles and not of exclusive principals or permissions as, for example, [Ahn and Sandhu, 1999, Nyanchama and Osborn, 1999] do. This should, however, not be difficult and strongly resembles the notion of exclusive roles. We did also not aim to fully specify the taxonomy defined in [Gligor et al., 1998] that, to our knowledge, is the only work formally specifying and proving the relationships between its 11 identified separation properties. Furthermore, we do not make any formal claims with respect to the preservation of safety properties as defined [Kuhn, 1997].

If any of the above was required, then the specification of the necessary signatures, facts, functions and assertions in Alloy should be of no major difficulty, since all of the existing work on separation properties we reviewed uses some kind of first-order predicate logic. However, we believe that this would not have provided any new contributions or insights to the field. The separation controls presented in this chapter are what we perceive to be the most realistic and useful separation controls in the context of real organisations.

Several of the separation controls we introduced in this chapter are based on the concept of mutually exclusive roles. We believe that an interesting and not yet sufficiently covered area of research is the engineering of exclusive roles. Some progress has been made in the general area of role engineering, e.g. [Coyne, 1995, Epstein and Sandhu, 1999, Roeckle et al., 2000, Neumann and Strembeck, 2002], but there is still no methodical approach considering aspects of organisational structure and the needed mappings to systems and applications enforcing separation controls. Although we did some initial research into this area [Kern et al., 2002] it is outside the scope of this thesis. Nevertheless, we see the engineering of roles as essential to provide realistic and effective separation controls.

Chapter 7

Delegation and Revocation of Policy Objects

7.1 Introduction

This chapter discusses the delegation and revocation of policy objects in the context of the control principle model presented in chapter 5. This includes:

- A discussion of the organisational motivations for delegation in section 7.2, which makes the important conceptual distinction between *ad hoc* delegation and administrative delegation further discussed in section 7.3;
- The identification of a general delegation function and discussion of the possible assignments of delegating and receiving principals to a delegated object in section 7.4.1;
- A discussion of the properties underlying the delegation of different types of policy objects with respect to this general delegation function in sections 7.4.2 and 7.4.3;
- The delegation of roles as a convenient administrative shorthand in section 7.4.4;
- A discussion and definition of the properties underlying the revocation of different types of policy objects in section 7.5, based on already existing revocation schemes.

We then analyse and explore delegation, revocation and separation controls in more detail. This includes:

- Examples of assertions that validate the defined delegation and revocation controls in section 7.6;
- Examples of state sequences that show how a separation control may conflict with delegation and revocation controls in section 7.7.

At the end of this chapter we provide a comparison with existing work on role and policy-based delegation and revocation in section 7.8. This includes a detailed discussion on the requirements for delegating obligations in Ponder in section 7.8.2.2.

7.2 Organisational motivation for delegation

Before discussing any technical aspects of delegation in more depth we have to define what we mean by delegation and what the organisational motivations for delegation are. As pointed out in chapter 2, delegation may be understood in several ways.

In a more general definition, delegation may be used as a term for describing how duties and the required authority propagate through an organisation, usually in terms of the refinement of a high-level organisational goal into manageable policies which eventually lead to the execution of some task [Muller, 1981, Moffett, 1990]. This is often referred to as decentralisation or Management by Delegation [Mintzberg, 1979]. In this general sense, delegation considers the passing of policy objects from one principal to another with respect to the performance of some activity and resulting attainment of some common organisational goal [Blau and Scott, 1962]. Delegation as a means for the distribution of work thus creates structure as we have already discussed in section 2.4.1.

However, often the term delegation is also used to describe how a principal passes some specific policy object on to some other principal, because the current structure does not allow the achievement of a goal one or both of these principals have [Pugh, 1997]. The reason why there is no supporting structure may be that either some goal could not be foreseen, or that principals are explicitly provided with a greater flexibility of how to achieve their goals [Mullins, 1999]. If such delegation activities occur frequently, have a regular pattern or principals delegate some object indefinitely, then this indicates that the current organisational structure and procedures do not reflect the goals of the involved principals. This initially temporary and *ad hoc* delegation must now become part of the regular administrative delegation activities shaping the formal organisational structure.

Depending on the type of organisation there may be different factors motivating such general administrative delegation or *ad hoc* delegation between specific principals. Some examples are listed in the following:

1. Lack of resources - A principal does not have the resources sufficient to achieve a goal. Examples for such resources could be a lack of time or equipment.

2. Competence - A principal is not sufficiently competent to achieve a goal. He has to delegate parts or all of the associated policies.

3. Specialisation - A principal might be sufficiently competent to achieve a goal, but it is more efficient to delegate to principals in specialist positions, such that the achievement, for example, takes less time.

4. Organisational policies - Goals may conflict and specific organisational policies may require a principal to delegate.

These examples show that we can have either organisational (Items 1.-3.) or policy-based (Item 4.) factors that may cause delegation.

7.3 Administration vs. *ad hoc* delegation

As indicated in the previous section, this thesis is based on a general distinction between two interpretations of delegation which need to be clarified. These are:

1. administrative delegation (administration) and
2. *ad hoc* delegation (delegation)

A valid question is what exactly distinguishes administration and *ad hoc* delegation. This distinction is often not made clear, e.g. [Zhang et al., 2001]. Both cause some sort of policy object assignment to be changed (although administration may also comprise activities such as setting a maximum number of roles to be activated), where administration has a high degree of similarity, regularity and repeatability, and conversely *ad hoc* delegation has a low degree of these. We argue that delegation may be seen as distinct from administration. Three characteristics can be used to support this distinction, and we will refer to these throughout the following sections, specifically when evaluating related work in section 7.8. These are:

- the representation of the authority to delegate;
- the specific relation of a principal to an object;
- the duration of this relation.

With respect to the first point we observe that in automated systems, dedicated roles reflect specific administrative authority and obligations in the form of their assigned policies. These roles are then occupied by principals referred to as 'administrators' which exercise these policies with respect to the management of the system's objects, e.g. [Sandhu et al., 1999]. The functionality of these administrative roles is usually distinct from the activities that determine the purpose of the organisation, which means that a principal in such an administrative 'staff' role is not involved in any of the main 'line' business activities. There are, however, situations where the organisational structure may require a principal to hold both, administrative and non-administrative roles. One real-world example we know of is a bank branch where an employee may perform certain administrative tasks such as creating new user accounts for that branch, although most of his time is dedicated to serving the customers of the branch. A second real-world example is that of a company where at some stage a few thousand employees were allowed to perform tasks such as resetting passwords. Although this form of decentralised administration proved to be robust with respect to the authentication of principals it was later abolished. Unfortunately, we do not know any more specific details about the environment in both examples. With respect to these examples we further observed that there may also be roles not explicitly marked as administrative, that may nevertheless comprise policies which may be perceived as being administrative. It is these roles and their occupants that the term delegation refers to in this context.

As a second point, we also consider the relationship between principals and the objects that are administered or delegated to distinguish between administration and delegation. Usually,

an administrator is not directly related to the objects he administers. For example, if he assigns an authorisation to a principal, he should not have that authority himself or be allowed to assign it to himself [Moffett, 1990]. The opposite must be the case for a principal delegating an object. For example, the delegation of an authorisation requires the delegating principal to be assigned with that authorisation.

The third point considers the duration of a delegation, i.e. the duration of changes to relations affected by delegation activities. While this cannot be measured in this context, a valid generalisation is that administrative changes should be longer lasting and more durable than changes based on delegation. The reason is that administration considers long term changes to the structure with respect to the general goals of the organisation, while delegation is usually performed with respect to the achievement of a single activity or goal.

The following section sets out how policy objects are delegated between principals in the context of the control principle model. We do not consider administrative forms of delegation such as the assignment of principals, roles and policy objects, and refer to appendix B.5 for some examples of such activities.

7.4 Delegation of policy objects

The *ad hoc* delegation of policy objects requires us to distinguish between the delegation of authorisation and obligation policies. Specifically the delegation of obligation policies has to adhere to some distinct rules that need to be discussed in detail when discussing general and specific obligations. Since our understanding of delegation is based on the three properties stated in the previous section 7.3, it now needs to be evaluated how far our control principle model supports these and what other assumptions have to be made.

To begin with, we have discussed that any delegation activity requires the authority to delegate. In classic access control models such as HRU [Harrison et al., 1976], or database systems such as System R [Griffiths and Wade, 1976], this authority is defined by a simple boolean flag attached to an authorisation. More elaborate frameworks such as Ponder [Damianou et al., 2001] have specific delegation policies specifying the authority to delegate as described in section 3.4. The underlying assumption in the context of our framework is that if a principal delegates, then he does this on the basis of some given authority. This authority is, however, not explicitly modelled in the context of our control principle model, because we are at this stage only interested in the structural aspects of delegating policy objects. However, we believe that if desired, the control principle model could be easily extended and would provide a suitable basis for modelling such authority as we showed in [Schaad and Moffett, 2002b].

Secondly, we have stated that a principal must be in some way assigned to the policy object he intends to delegate, which in this context implies a direct or role-based assignment. Delegation also implies possible changes to these assignments, not only for the delegating principal but also for the receiving principal.

Thirdly, the duration of a delegation cannot be modelled in the context of this framework. This would require more elaborate temporal models such as [Bertino et al., 2001] probably in combination with workflow systems such as those described in, for example, [Bertino et al., 1999a]. However, this context does not allow any further examination and workflows have been identified as future work in section 11.4.

Before we discuss and define specific delegation functions, we investigate some general properties that are common to the delegation of both types of policy objects in this framework.

7.4.1 General delegation properties

What is the most general delegation function that can be defined in the context of this framework? Such a function must take the relationship between the delegated object and the delegating and receiving principal into consideration.

Thus, when delegating a policy object, a distinction must be made between:

1. A policy object held directly by the delegating principal, i.e. a policy object `pol` which has the delegating principal `p1` as its subject. This may be expressed as:

   `p1 in pol.(s.s_subject).`

2. A policy object held through one of the roles of the delegating principal, i.e. a policy object `pol` which has a role the delegating principal `p1` is a member of, as its subject. This may be expressed as:

   `p1 in pol.(s.s_subject).(s.s_has_member).`

The object to be delegated must be related to the delegating principal in one or both of the above ways, while there is currently no specific assumption about the delegatee already holding the object. After the delegation the delegatee should be related to the policy object, although he already might have been related to the object before. The delegating principal might lose or retain the object. If specific assumptions are made about either case, then these will have to be explicitly stated in the relevant context.

These initial observations already give rise to a complex scenario space determined by the (non)membership of a delegated policy object `pol` in the participating principal's set of available policies. See table 7.1, that describes the possible assignments of `pol`. There, `p1` refers to the delegating principal while `p2` is the delegatee and `s` and `s'` refer to the states before and after the delegation respectively. When comparing the upper two cells, the missing fourth entry in the upper left cell reflects that the delegating principal `p1` should be assigned to the policy object he intends to delegate. With respect to the possible states for `p2`, the lower right cell shows only two possible entries, since the receiving principal should be directly assigned to the delegated object after the delegation. However, it is not the case that we may get from one valid assignment in state `s` to any other in state `s'`, as the two might not be compatible with respect to some of the previously described implicit (conceptual) or explicit constraints. Therefore, these scenarios must be complemented by a

I. Assignments in state `s` for principal `p1`	II. Assignments in state `s'` for principal `p1`
Case 1: p1 in pol.(s.s_subject) && p1 !in pol.(s.s_subject).(s.s_has_member) Case 2: p1 !in pol.(s.s_subject) && p1 in pol.(s.s_subject).(s.s_has_member) Case 3: p1 in pol.(s.s_subject) && p1 in pol.(s.s_subject).(s.s_has_member)	Case 1: p1 in pol.(s'.s_subject) && p1 in pol.(s'.s_subject).(s'.s_has_member) Case 2: p1 !in pol.(s'.s_subject) && p1 in pol.(s'.s_subject).(s'.s_has_member) Case 3: p1 in pol.(s'.s_subject) && p1 !in pol.(s'.s_subject).(s'.s_has_member) Case 4: p1 !in pol.(s'.s_subject) && p1 !in pol.(s'.s_subject).(s'.s_has_member)

III. Assignments in state `s` for principal `p2`	IV. Assignments in state `s'` for principal `p2`
Case 1: p2 in pol.(s.s_subject) && p2 !in pol.(s.s_subject).(s.s_has_member) Case 2: p2 !in pol.(s.s_subject) && p2 in pol.(s.s_subject).(s.s_has_member) Case 3: p2 in pol.(s.s_subject) && p2 !in pol.(s.s_subject).(s.s_has_member) Case 4: p2 !in pol.(s.s_subject) && p2 !in pol.(s.s_subject).(s.s_has_member)	Case 1: p2 in pol.(s'.s_subject) && p2 in pol.(s'.s_subject).(s'.s_has_member) Case 2: p2 in pol.(s'.s_subject) && p2 !in pol.(s'.s_subject).(s'.s_has_member)

Table 7.1: Possible assignments for principals and objects before and after delegation of a general policy object.

function which provides the mapping between the identified possible assignments of policy objects and principals involved in the delegation. This function 7.1 consists of three basic cases.

If the policy to be delegated is assigned directly to principal `p1` before the delegation, then this leads to a change in the `s_subject` relation between states `s` and `s'`, where principal `p2` is now related to the object, and principal `p1` either loses or retains it after the delegation.

The second case covers a situation where the policy object to be delegated is assigned to a role of principal `p1`. While `p2` will be subject to `pol` after the delegation, there is no further explicit assumption about any change to `p1`'s relations.

The last case specifies that if the object to be delegated is neither held directly by `p1` nor indirectly through a role, then there are no changes with respect to the `s.s_subject` relation. A frame condition for maintaining the other relations of the framework complements this general delegation function and is fully specified in appendix B.3.

These three cases are sufficient to cover what has been summarised as valid assignments between principals and a policy object to be delegated in table 7.1. This general delegation function only considers the assignments of the policy `pol` to principal `p1` and remains silent about the possibility of principal `p2` already being the subject of `pol` before the delegation. More specifically, `p2` may be subject to `pol` directly or through a role.

Alloy Function 7.1 *A general delegation function for delegating a policy object* `pol` *from principal* `p1` *to* `p2` *between states* `s` *and* s'.

```
fun general_delegation(disj s,s': State, disj p1,p2: Principal,
                       pol : PolicyObject){
 //Case 1: If principal p1 is directly subject to pol in state s,
 //then p2 will be subject to pol in s' while p1 may (not) lose pol.
   (p1 in pol.(s.s_subject) =>
     (s'.s_subject = s.s_subject + pol -> p2 ||
      s'.s_subject = s.s_subject + pol -> p2
                                 - pol -> p1))          &&
 //Case2: If pol is in one of p1's roles then p2 must be subject to
 //pol in s' but p1 retains pol.
   (p1 in pol.(s.s_subject).(s.s_has_member) =>
      s'.s_subject = s.s_subject + pol -> p2)           &&
 //Case 3: If p1 does not hold pol in state s then nothing changes in s'
 //with respect to the s_subject relation.
   ((p1 !in pol.(s.s_subject) &&
     p1 !in pol.(s.s_subject).(s.s_has_member) =>
      s'.s_subject = s.s_subject))                      &&
 //Frame Condition
   general_delegation_frame(s,s')
}
```

With respect to the first case, the direct assignment of `p2` to `pol` may have been present at the time of starting the system, or was established through a delegation by some other principal during system run-time. Our notion of states and the supporting history mechanism defined in section 5.10.2, allows us to maintain the appropriate information.

However, it is questionable whether the delegation of a policy object to a principal who already holds it makes always sense in the particular organisational context. Since at least in the case of authorisations several practical examples support such multiple delegation [Hagstrom et al., 2001], we do not enforce any particular constraint here to rule this out. With respect to the later revocation of policy objects this will, however, cause a higher degree of complexity as discussed in section 7.5.3. There may also be the case of a principal delegating the same object more than once to the same principal. If there is a revocation of the object in between those delegations than this is valid, but if not, then such a second delegation should be ruled out, and we have defined a constraint supporting this in appendix B.3.

So far, this basic delegation function does not consider the specific type of policy object to be delegated. While this allowed us to examine some general delegation properties in this section, it requires us to refine the function with respect to the specific type of delegated policy object and possible organisational circumstances of the delegation in the following three sections 7.4.2, 7.4.3 and 7.4.4.

7.4.2 Delegating authorisations

When explicitly delegating an authorisation policy object, several possible assignments of this policy object can be observed with respect to the two involved principals before and after the delegation. These assignments are a subset of the possible combinations of assignments described in the previous section 7.4.1 and table 7.1. Accordingly, this section discusses how we determined this subset and the assumptions underlying its definition. In particular, the following two (in)direct assignments of an authorisation policy to a principal drive this discussion:

1. We observe that the delegated authorisation may be held by the delegating principal `p1` directly and through a role before the delegation (See entry `I.3.` in table 7.1).
2. We observe that the delegated authorisation may be held by the delegating principal `p1` only through a role before the delegation (See entry `I.2.` in table 7.1).

With respect to the first point, the simultaneous assignment of a policy object to a principal and a role needs to be questioned with respect to authorisations. To begin with, we did not encounter any situation where such a dual assignment was of any specific interest with respect to the analysis of control principles. Also, we argue that if such a dual assignment occurs, than this is likely to be the result of an immature role finding process and understanding of the authorisation structure. For example, we consider a principal holding an authorisation both directly and through one of his roles. If now the role is used as the basis for delegation, then the current function `general_delegation()` could result in the direct assignment being removed for the principal. Vice versa, if the direct assignment is used as the basis for delegation and the principal should lose the authorisation, then he would still have it through his role. This might conflict with some higher-level organisational policies regulating the delegation of authorisations, but we do not suggest any conflict resolution mechanisms in this context.

It is for these reasons that we decided to define a constraint on the assignment of authorisation policy objects. This fact 7.1 effectively rules any dual assignment out and thus reduces the possible states as initially described in table 7.1.

Alloy Fact 7.1 *If an authorisation* `auth` *is assigned to the role* `r` *of a principal* `p` *then it must not be directly assigned to the principal.*

```
fact {all s : State | all auth : Authorisation |
      all r : Role | all p : Principal |
       (some auth.(s.s_subject) & r) &&
        p in r.(s.s_has_member) =>
           no auth.(s.s_subject) & p
}
```

However, we believe that if such a constraint may be deemed as too strict in a specific context, the history mechanism defined in section 5.10.2 is flexible enough to resolve any possible

I. Assignments in state s for principal p1	II. Assignments in state s' for principal p1
Case 1: p1 !in auth.(s.s_subject) && p1 in auth.(s.s_subject).(s.s_has_member) Case 2: p1 in auth.(s.s_subject) && p1 !in auth.(s.s_subject).(s.s_has_member)	Case 1: p1 !in auth.(s'.s_subject) && p1 in auth.(s'.s_subject).(s'.s_has_member) Case 2: p1 in auth.(s'.s_subject) && p1 !in auth.(s'.s_subject).(s'.s_has_member) Case 3: p1 !in auth.(s'.s_subject) && p1 !in auth.(s'.s_subject).(s'.s_has_member)

III. Assignments in state s for principal p2	IV. Assignments in state s' for principal p2
Case 1: p2 !in auth.(s.s_subject) && p2 in auth.(s.s_subject).(s.s_has_member) Case 2: p2 in auth.(s.s_subject) && p2 !in auth.(s.s_subject).(s.s_has_member) Case 3: p2 !in auth.(s.s_subject) && p2 !in auth.(s.s_subject).(s.s_has_member)	Case 1: p2 in auth.(s'.s_subject) && p2 !in auth.(s'.s_subject).(s'.s_has_member)

Table 7.2: Possible assignments of a delegated authorisation auth for principals p1 and p2 as defined by function 7.2 and fact 7.1.

ambiguities considering a delegation, i.e. we would always know whether an authorisation was delegated on the basis of a role or direct assignment.

So far we have described the delegation of an authorisation, only implicitly considering the possibility of the delegating subject retaining or losing this type of policy object. The retainment of a delegated authorisation does not raise any conceptual problems and neither does the loss of a delegated authorisation considering any direct assignment between the authorisation and the delegating principal. However, a problem arises, if the authorisation was delegated on the basis of role membership. It cannot be simply removed from the role as this would propagate to all other possible occupants of that role, or may imply a change to the s_has_member relation which is considered to be outside the scope of delegation. Solutions such as the direct assignment of a negative authorisation [Bertino et al., 1997b], or creation of a new role with all but the delegated authorisation and the subsequent re-assignments, are not feasible within the context of this framework. Thus, this possibility is not modelled here, and if a role membership was the basis for the delegation of an authorisation, then the authorisation must remain with the role.

The following delegation function for authorisations 7.2 captures this. In combination with fact 7.1 and the previously defined general_delegation() function 7.1, it determines the possible assignments of such a delegated authorisation for the involved principals as shown in table 7.2. It also defines how to update the delegation history when delegating an authorisation, distinguishing between a role-based or a direct assignment to the delegating principal. Furthermore, it excludes the possibility that the delegating principal does not hold the object to be delegated, as this may lead to the generation of empty models.

Alloy Function 7.2 *Delegation of an authorisation* `auth` *from a principal* `p1` *to* `p2` *in between states* `s` *and* $\mathtt{s}'$.

```
fun delegate_auth(disj s,s': State, disj p1,p2: Principal,
                  auth: Authorisation){
 //To avoid empty models we define as a precondition, that
 //authorisation auth is held by p1 directly or through a role
   (auth in (s.s_subject).p1 ||
   (auth in (s.s_subject).((s.s_has_member).p1))) &&
 //Delegate using the general delegation function 7.1
   general_delegation(s,s',p1,p2,auth) &&
 //Delegation history update which distinguishes between direct
 //and role-based delegation
   (s.s_delegation_history).delegating_principal = p1 &&
   (s.s_delegation_history).receiving_principal = p2 &&
   (s.s_delegation_history).delegated_policy_object = auth &&
   (some r: Role | (auth in (s.s_subject).p1 =>  //If deleg. was direct
                     no (s.s_delegation_history).based_on_role) &&
                   (auth in (s.s_subject).r &&
                    p1 in r.(s.s_has_member) =>  //If a role was used
                     (s.s_delegation_history).based_on_role = r)) &&
 //Frames for other relationships - Fully defined in appendix B.3
   individual_auth_delegation_frame(s,s')}
```

An example of the organisational context this delegation function may be used in is given in section 10.4.1 of our case study.

7.4.3 Delegating obligations

It now needs to be explored whether the delegation of obligations can be treated in a similar manner as the delegation of authorisations has been in the previous section. In particular, we have to assess whether the general delegation function 7.1 is still valid with respect to our abstraction of general and specific obligation policy objects in section 5.3.3, and what other restrictions may apply. This requires to informally recall some of the previously specified constraints on obligations, more specifically that:

- A specific obligation instance must always be held by exactly one principal as defined in fact 5.2;
- The specific obligation instance of a principal must correspond to a general obligation he holds directly or through a role as defined in fact 5.4.
- A general obligation may be shared between two principals, either directly, through roles or both, but it may not be held by a principal directly and through a role as defined in fact 5.5.

I. Assignments in state s for principal p1	II. Assignments in state s' for principal p1
Case 1: p1 in iob.(s.s_subject) && p1 !in iob.(s.s_subject).(s.s_has_member)	Case 1: p1 !in iob.(s'.s_subject) && p1 !in iob.(s'.s_subject).(s'.s_has_member)

III. Assignments in state s for principal p2	IV. Assignments in state s' for principal p2
Case 1: p2 !in iob.(s.s_subject) && p2 !in iob.(s.s_subject).(s.s_has_member)	Case 1: p2 in iob.(s'.s_subject) && p2 !in iob.(s'.s_subject).(s'.s_has_member)

Table 7.3: Possible assignment of a delegated obligation instance iob for principals p1 and p2 as defined by function 7.3 and explicit facts 5.2, 5.4, 5.5.

Accordingly, if a principal decides to delegate an obligation, then this may refer to a general or specific obligation. We argue that we can define a single delegation function 7.3, that can be used to delegate both types of obligation.

As in the case of delegating authorisations, this function is based on the earlier defined general delegation function 7.1. To facilitate later analysis and exploration of control principles, the delegate_obligation() function defines as a precondition that the obligation obl to be delegated must be held directly or over a role by the delegating principal p1. We then use function general_delegation() for the actual delegation, followed by an update to the delegation history and the definition of the appropriate framing conditions.

7.4.3.1 Delegating specific obligations

While so far no explicit assumptions had to be made about the status of the receiving principal with respect to the delegated object, the delegation of a specific obligation now requires the receiving principal to already hold the corresponding general obligation. In other words, a specific obligation can only be delegated between principals with the same general obligation. A principal should not be delegated a task he is not meant to do.

While this clearly restricts the scenarios identified in table 7.1, it does not have to be defined explicitly in the delegation function 7.3 as it is implied in the previously defined fact 5.4.

The cardinality constraint 5.2 for the delegated obligation instance and a principal further restricts the possible delegation scenarios. Either the delegating principal holds the obligation or not and so does or does not the receiving principal. Thus, any meaningful delegation is restricted to the cases I.1/II.4 for principal p1, and III.4/IV.2 for p2 from table 7.1 as summarised in table 7.3. This has the side effect that the problematic discussion about whether and how the delegating principal should retain or lose the policy object he delegates does not arise with respect to specific obligations.

The delegation of specific obligations raises concerns with respect to how control over the delegation process may be achieved. This will be discussed in detail in section 8.2 introducing the concept of review.

Alloy Function 7.3 *Delegation of an obligation* `obl` *from a principal* `p1` *to* `p2` *in between states* `s` *and* `s'`.

```
fun delegate_obligation (disj p1,p2: Principal,
                         disj s,s' : State,
                         obl : Obligation) {
 //Precondition, principal p1 or his role must be subject to obl
   (obl in (s.s_subject).p1 ||
   (obl in (s.s_subject).((s.s_has_member).p1))) &&
 //Delegation
   general_delegation(s,s',p1,p2,obl) &&
 //Update of delegation history, distinguishing between direct or
 //role-based delegation
   (s.s_delegation_history).delegating_principal = p1 &&
   (s.s_delegation_history).receiving_principal = p2 &&
   (s.s_delegation_history).delegated_policy_object = obl &&
    (some r : Role | (obl in (s.s_subject).p1 => //If it was direct
                         no (s.s_delegation_history).based_on_role) &&
                     (obl in (s.s_subject).r &&
                     p1 in r.(s.s_has_member) => //If a role was used
                     (s.s_delegation_history).based_on_role = r)) &&
 //Frames for other relationships - Fully defined in appendix B.3
   individual_obl_delegation_frame(s,s')
}
```

7.4.3.2 Delegating general obligations

For delegating a general obligation we may also use the previously defined delegation function 7.3. There are, however, some additional constraints that need to be considered. More specifically, these constraints concern whether the general obligation is assigned to a role of the delegating principal or directly to him, and what should be done with any currently existing specific obligations for that general obligation.

If role membership is the basis for the delegation, then we may observe that the delegating principal must retain the general delegation. The reasons for this are identical to what was discussed in section 7.4.2 considering the delegation of authorisations. In case of the obligation being delegated on the basis of a direct assignment, the delegating principal may also choose to drop the obligation if there are no existing instances. Again this issue and its implications are mostly identical to previous discussions and table 7.4 reflects this.

However, since there may be existing instances for a general obligation, it needs to be decided what to do with these after the delegation. A principal may choose to delegate all, some or none of any existing instances together with the general obligation. This is again a decision that will have to be made according to the current organisational context and the motivation underlying the delegation. To summarise this discussion:

I. Assignments in state s for principal p1	II. Assignments in state s' for principal p1
Case 1: p1 !in gob.(s.s_subject) && p1 in gob.(s.s_subject).(s.s_has_member) Case 2: p1 in gob.(s.s_subject) && p1 !in gob.(s.s_subject).(s.s_has_member)	Case 1: p1 !in gob.(s'.s_subject) && p1 in gob.(s'.s_subject).(s'.s_has_member) Case 2: p1 in gob.(s'.s_subject) && p1 !in gob.(s'.s_subject).(s'.s_has_member) Case 3: p1 !in gob.(s'.s_subject) && p1 !in gob.(s'.s_subject).(s'.s_has_member)

III. Assignments in state s for principal p2	IV. Assignments in state s' for principal p2
Case 1: p2 !in gob.(s.s_subject) && p2 in gob.(s.s_subject).(s.s_has_member) Case 2: p2 in gob.(s.s_subject) && p2 !in gob.(s.s_subject).(s.s_has_member) Case 3: p2 !in gob.(s.s_subject) && p2 !in gob.(s.s_subject).(s.s_has_member)	Case 1: p2 in gob.(s'.s_subject) && p2 !in gob.(s'.s_subject).(s'.s_has_member)

Table 7.4: Possible assignment of a delegated general obligation gob for principals p1 and p2 as defined by function 7.3 and explicit facts 5.2, 5.4, 5.5.

1. If a delegated general obligation gob has some instances with the delegating principal p1 as its subject then it may be the case that:
   (a) the delegating principal p1 retains gob and the receiving principal p2 obtains it, at the same time all, some or none of the existing instances are delegated or that;
   (b) the delegating principal p1 loses gob and p2 obtains it, at the same time all the existing instances are delegated.
2. If a delegated general obligation gob has no instances, then it may be the case that the delegating principal p1 retains or loses gob, while the receiving principal p2 obtains it

Our initial attempt to model the delegation of a general obligation was to cover all the above cases in a single delegation function `delegate_general_obligation()`. Although we succeeded, this function has several conceptual disadvantages. To begin with, we could not use the `general_delegation()` function 7.1 to support such a delegation of a general obligation. The reason for this was that the `general_delegation()` function only changes the `s_subject` relation with respect to one type of policy object at a time. However, in the case of delegating a general obligation, we have to consider two types of objects, the general obligation and any possibly existing instances for it. Secondly, our history mechanism is designed with the intention to only record the delegation of one policy object at a time. A delegation function for a general obligation would have to be purely declarative, allowing for several assignments to change between two states.

It is for this reason that we do not further consider a function for specifically delegating general obligations in the context of this section, but define it in appendix B.3. If required,

this may then be used to assert the validity of a step by step delegation of a general obligation and any possible instances using function 7.3, by means of comparing the resulting states.

7.4.4 Delegating policy objects through delegating roles

We discussed in section 2.4 that roles are a natural means of reflecting organisational structure. Recently, the delegation of roles between principals has received attention, e.g. [Zhang et al., 2001, Barka, 2002]. We argue that such a delegation of roles is identical in its results to a stepwise delegation of the assigned policy objects through the two delegation functions 7.2 and 7.3.

This form of delegation implies that principals may delegate entire roles between them which may be convenient as all policy objects assigned to the role are delegated together. This helps to maintain the policy configuration consistent, considering that the policies for a role should have been subject to careful specification reflecting the obligations and authorisations for that role. The delegation of roles thus indirectly implies a delegation of policy objects.

A practical example would be the case of an employee going on holidays leaving his work and subsequently delegating his role to a colleague for the duration of his absence. When considering organisations with 50,000 or more employees then such seemingly trivial scenarios have been identified as a major challenge with respect to system administration [Schaad et al., 2001]. The delegation of roles might be a way to lift this burden of an administrator. It is clear that the delegation of entire roles is a very delicate matter considering systems security, and the decision whether or not to support this may be dependent on the specific kind of organisation and other forms of control such as audit. Here we can only look at some of the technical implications of this form of delegation.

In the context of this framework, the delegation of roles is achieved through a change to the `s_has_member` relation, that indicates which roles a principal is a member of. Similar to the delegation of policy objects, the delegating principal may choose to drop or retain the role he delegates, which depends on the context and reason for delegation (see section 7.2). Additionally, some history must be established for later audit purposes together with a set of framing conditions. However, such an operation has side effects with respect to the assigned policy objects and the possibility of retaining or losing the role.

The delegation of a role that has only authorisations assigned to it does not present any conceptual difficulties. Of course the delegation of such a role needs to be monitored with respect to other controls such as separation, but we cannot further consider this here.

In the case of delegating a role with assigned general obligations, the role delegation operation will be constrained by the facts defined in section 5.3.3 and listed informally in the beginning of the previous section 7.4.3. Two situations may be observed. In the first case there are no current specific obligation instances for the delegating principal at the time of delegation. An example for this might be where an employee goes on holidays and finishes all his work before delegating his role to a colleague for the duration of his absence. In the second case, there are some specific obligations at the time of delegation as the current situation might require

the delegating employee to leave, without having been able to finish his work. Both cases require us to be more specific about delegating roles with general obligations, and it must also be decided what to do with any currently existing obligation instances the delegating principal holds. If there are no obligation instances for the delegating principal and the role to be delegated at the time of delegation, then the delegation of a role through a change to the `s_has_member` is sufficient. However, if there are instances that are a part of a delegated role's obligations then the `s_subject` relation has to be changed accordingly. This means that the delegatee now replaces the delegating principal as the subject for any instance of a general obligation delegated together with the role. We discussed this delegation of a general obligation in more detail in the previous section 7.4.3.2.

Since we do not have an imminent interest in the further analysis and exploration of delegating roles we only sketch a possible delegation function 7.4 for delegating a role. This function captures the previously discussed properties, more specifically the change to the `s_has_member` relation. The function would in fact be sufficient and as discussed above, the delegation function 7.3 may then be used to delegate any obligations on a step by step basis. We decided to emphasise the distinction between such a stepwise delegation and a purely declarative approach by showing how the `s_subject` relation changes with respect to all existing obligation instances for the delegating and receiving principal. This role delegation function may then be used to assert a series of stepwise delegations of all obligation instances.

Alloy Function 7.4 *Delegation of a role* `r` *from a principal* `p1` *to* `p2`*, supporting the delegation of* `p1`*'s existing obligation instances* `iob` *for all general obligations* `gob` *assigned to role* `r`*.*

```
fun delegate_role (disj s,s': State,
                   disj p1, p2: Principal, disj r: Role){
 //Precondition, principal must be member of role to be delegated
   some ((r -> p1) & (s.s_has_member))  &&
 //Delegation of the role, with principal retaining or losing it
   (s'.s_has_member = s.s_has_member  + (r -> p2) ||
    s'.s_has_member = s.s_has_member  + (r -> p2)
                                     - (r -> p1)) &&
 //Change the subject relationships for all obligation instances with
 //respect to delegating/receiving principals. In this case all obligation
 //instances for p1 through the general obligation are delegated to p2.
 //Note that we do not support any partial delegation of instances here.
   all iob : ObligationInstance |
   all gob : Obligation - ObligationInstance |
    gob -> r in s.s_assigned_to_role &&
    iob -> p1 in s.s_subject &&
    gob -> iob in s.s_has_instance =>
     s'.s_subject = s.s_subject - (iob -> p1)
                                + (iob -> p2) &&
 //For frames and updates to History refer to appendix B.3}
```

7.5 Revocation of policy objects

The delegation of policy objects must be complemented by their revocation. However, specifying revocation controls may be very complex as, for example, demonstrated in the work of [Griffiths and Wade, 1976, Jonscher, 1998, Bertino et al., 1997b], addressing revocation of permissions in the context of database systems. A unifying framework for revocation has only been proposed recently [Hagstrom et al., 2001]. This is, however, limited to the revocation of permissions directly assigned to a principal and does not include a notion of roles. This section explores how far this framework can be applied for the revocation of policy objects in the context of our control principle model.

In general, revocation of an object is based on its previous delegation and thus requires the following pieces of information [Samarati and Vimercati, 2001]:

1. The principals involved in previous delegation(s);
2. The time of previous delegation(s);
3. The object subject to previous delegation(s)

Our conceptual model provides this information through the history signatures defined in section 5.10, and may thus support the various forms of revocation as described in the revocation framework of [Hagstrom et al., 2001]. In this framework different revocation schemes for delegated access rights are classified against the dimensions of *resilience*, *propagation* and *dominance*. Since resilience is based on negative permissions, we do not consider this here, as there is no corresponding concept for the policy objects in our model. The remaining two dimensions may be informally summarised as follows:

1. Propagation distinguishes whether the decision to revoke affects
   - only the principal directly subject to a revocation (local); or
   - also those principals the principal subject to the revocation may have further delegated the object to be revoked to (global).
2. Dominance addresses conflicts that may arise when a principal subject to a revocation has also been delegated the same object from other principals. If such other delegations are independent of the revoker then this is outside the scope of revocation. If, however, such other delegations have been performed by principals who, at some earlier stage, received the object to be revoked via a delegation path stemming from the revoker, then the revoking principal may
   - only revoke with respect to his delegation (weak);
   - revoke all such other delegations that stem from him (strong).

Based on these two dimensions, we established 4 different revocation schemes which, due to the absence of resilience, are a subset of those described by [Hagstrom et al., 2001]. These are summarised in table 7.5.

No	Propagation	Dominance	Name
1	No	No	Weak local revocation
2	No	Yes	Strong local revocation
3	Yes	No	Weak global revocation
4	Yes	Yes	Strong global revocation

Table 7.5: Revocation schemes.

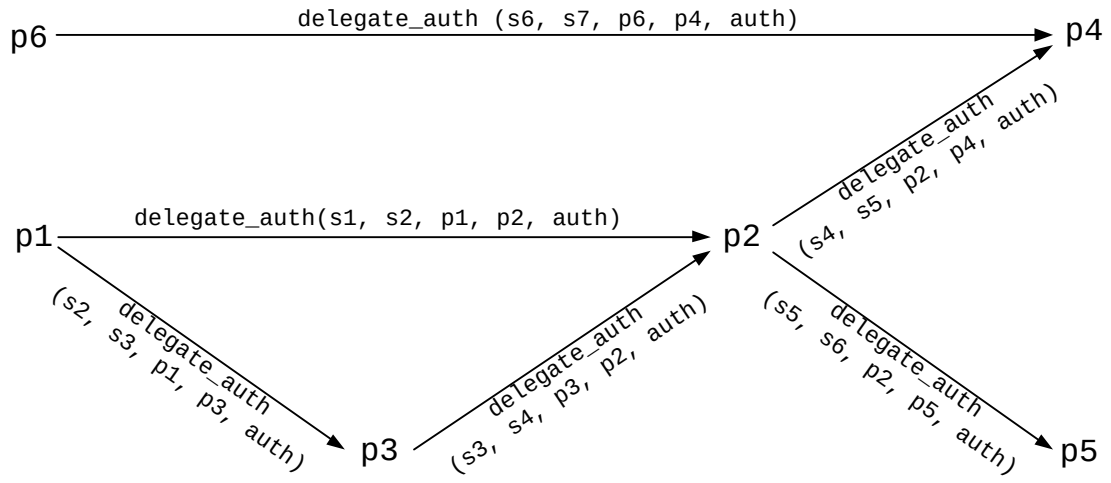

Figure 7.1: A delegation scenario for delegating an authorisation `auth`.

We will now investigate how far these schemes can be expressed and integrated with respect to our control principle model and the specific types and characteristics of policy objects. The following two sections will thus discuss the revocation of delegated policy objects along the lines of the above revocation schemes.

7.5.1 Revoking delegated authorisations

Since our authorisations are similar to the notion of permissions in [Hagstrom et al., 2001], we will describe the four revocation schemes in terms of a possible delegation scenario. We use function 7.2 `delegate_auth()` to state that a principal `p1` delegated an authorisation `auth` to a principal `p2` in state `s1`. Similarly, `auth` was delegated by `p1` to `p3` in state `s2`; by `p3` to `p2` in state `s3`; by `p2` to `p4` in state `s4`; by `p2` to `p5` in state `s5`; and finally by `p6` to `p4` in state `s6`. This is summarised in the graph displayed in figure 7.1 where the nodes stand for the principals, and the arcs are labelled with the respective delegation activity. We assume that in this example the principals always retain the authorisation they delegate.

A weak local revocation of an authorisation is the simplest case as it does not propagate or dominate any other delegations of the authorisation. For example, if principal `p2` revokes `auth` from principal `p5`, then `p5` will not hold `auth` anymore. If, however, `p1` revokes `auth` from principal `p2`, then `p2` will continue to hold `auth` due to the delegation of `auth` by `p3` in state `s3`.

A strong local revocation will address this later scenario, and if p1 strongly revokes auth from principal p2 locally, then p2 will not continue to hold auth, however, p4 and p5 will. The strong revoke by p1 will only result in p2 losing auth completely, because p3 had been delegated auth by p1 and then delegated it to p1. All delegations of auth to p2 stem from p1. If, for example, p2 strongly revokes auth from p4, then p4 will still hold auth because p2 has no influence on the delegation of auth by p6.

The weak global revocation addresses the revocation of policies which have been delegated more than once through a cascading revocation. Thus, if p1 globally revokes auth from principal p2 then this will result in p5 losing auth, but p2 and p4 will still hold auth due to the delegation of auth by p3 and p6 in s3 and s6 respectively.

Letting p1 revoke auth from principal p2 strongly and globally, auth will then not be held by p2 and p5 anymore, but p4 will still hold it due to the individual delegation by p6.

7.5.2 Revoking delegated obligations

The delegation of obligations should be complemented by revocation mechanisms as well, and we investigate in the following whether the previously identified four revocation schemes can also be applied in this context. Since we may delegate general and specific obligations as discussed in section 7.4.3, their revocation must also be discussed separately.

7.5.2.1 Revocation of specific obligations

In fact 5.2 we required an obligation instance to be assigned to exactly one principal at any time. It is for this reason that many of the problems we described for the revocation of authorisations cannot occur. We illustrate this with respect to the two applicable dimensions of revocation:

1. Dominance does not apply as it is not possible for a principal to have been delegated the same obligation instance from different sources.

2. Propagation may apply as a principal is able to delegate a delegated obligation. However, when he delegates he may not retain this obligation.

With respect to the second point, we consider the example of a principal p1 having delegated an obligation instance iob to a principal p2 who in turn delegated it to a principal p3. Should principal p1 now be able to revoke iob directly from p3 or only from p2? We believe that a principal should only be able to revoke a delegated obligation from the principal he delegated it to. The order of revocation corresponds to the way the obligation was initially delegated. There may be organisations where a direct revocation of iob from p3 by p1 may be desirable, e.g. a coercive organisation with distinct command structures, where decisions may have to be made rapidly like a hospital or military organisation [Mullins, 1999]. We do, however, not believe that any further argumentation would contribute to the overall goal of the thesis.

7.5.2.2 Revocation of general obligations

We have argued in section 7.4.3 that the delegation of general obligations can be treated almost identically to the delegation of authorisations. Thus, the underlying question here is whether this also applies to the revocation of general obligations. To clarify this, we again look at the revocation of general obligations with respect to the two dimensions of revocation considered in this context:

1. Dominance applies since a general obligation may be held by several principals. These may independently delegate this obligation, perhaps to the same principal at different times.

2. Propagation applies since a general obligation may be delegated several times between principals.

Considering the first item, the question is whether the revocation of a multiply delegated obligation may override other delegations or not. This has been referred to as strong and weak revocation respectively in section 7.5. We believe that this issue can be addressed like the strong and weak revocation of authorisations described in section 7.5.1, as long as the constraints defined in facts 5.4 and 5.5 hold.

The second item demands to distinguish between local and global revocation. The latter possibly causes a series of cascading revocations if a general obligation has been delegated several times. Again, this can be theoretically treated as in the case of revoking authorisations, but the following points must be considered.

The delegation of a general obligation may have been followed by the delegation or creation of some instances of that general obligation. This may influence the revocation of a general obligation, because of the constraint that a principal may only hold an obligation instance if he has the corresponding obligation, as expressed in fact 5.4. So if a general obligation is revoked, then this should result in the revocation of any existing delegated instances for the principal subject to a revocation.

A different situation may, however, be where an instance has been created on the basis of a delegated general obligation. The question is whether the revoking principal should be able to revoke an obligation instance he did never hold and subsequently never delegated. One approach may be to demand that the principal holding such instances must first discharge these before a delegated general obligation can be revoked. Another option may be to allow for the delegation of such an instance back to the revoking principal by the principal the general obligation is revoked from.

Whatever the decision, we believe that these points present no technical difficulties with respect to the actual revocation activity. A more detailed discussion on organisational aspects of revocation and resolution of any conflicts is, however, outside this scope.

7.5.3 Defining revocation mechanisms for the CP model

Several procedural revocation algorithms exist, e.g. [Griffiths and Wade, 1976, Fagin, 1978, Jonscher, 1998]. However, the definition of a revocation mechanism in a declarative way is a non-trivial task and the only work we are aware of is [Bertino et al., 1997b], which is unfortunately strictly tied to their specific authorisation model, and may also be difficult to understand without the possibility of tool supported analysis.

As pointed out at earlier stages of this thesis, one main underlying design principle of our model is that a declarative specification of delegation and revocation operations should reflect their possible procedural counterparts. This means that they only cause a change to the `s_subject` relation with respect to the actual objects involved in the delegation and revocation. The strong local and the weak and strong global functions may however also cause changes to the `s_subject` relation with respect to other objects not explicitly defined when calling a revocation function, since this only describes what the resulting state should look like. We thus argue in the following, that the strong local, and weak and strong global revocation can all be modelled in terms of a series of weak local revocations.

For this reason we only formally outline the function `weak_local_revocation()` in section 7.9, and provide a full definition in appendix B.3. The three remaining types of revocation are discussed less formally, as they may be understood as a series of weak local revocations.

7.5.3.1 Weak local revocation

Before considering the weak local revocation of a policy object in more detail, we informally recall some possible delegation scenarios that may have an effect on the behaviour of a weak revocation.

1. A policy object may have been delegated by a principal on the basis of a direct or role-based assignment.
2. The principal a policy object was delegated to may have already been assigned with the object.
    - either because he was assigned with the object at the time of system setup or;
    - because of a prior (and not yet revoked) delegation from some other principal.

Depending on the situation, a weak local revocation may behave differently with respect to the changes in the `s_subject` relation. For example, we consider that a principal `p1` delegates an authorisation policy object `auth` he directly holds to a principal `p2` in state `s1`. As a result of this delegation `p1` loses `auth`. Principal `p2` is also delegated `auth` by some other principal `p` in state `s2`. In state `s3` principal `p1` revokes `auth` from `p2`. Because of the delegation by `p` in state `s2`, `p2` must not lose `auth`. Principal `p1` must be assigned with `auth` again and the revocation of `auth` from `p2` by `p1` must be recorded by an update to the revocation history of state `s3`. A variation of this scenario may be that principal `p1`

delegated auth in state s1 on the basis of a role he is a member of. This would then mean that when he revokes auth from p2 in s3, the s_subject relation would not change at all. This is because of the delegation by p in state s2 and the fact that the initial delegation by p1 in s1 was based on a role, demanding the retainment of auth by p1. Nevertheless, the revocation would still have to be recorded by an update to the revocation history.

It would not be helpful to provide an exhaustive list of all such possible delegation scenarios here, and the two above examples only reflect in parts the complexity of a weak local revocation within our framework. There are, however, four properties that need to be evaluated. These concern whether there were multiple delegations of a policy object by and to the same principal; whether a role was used for delegation; whether there were multiple independent delegations; and whether the receiving principal was already subject to the delegated policy object before any delegations. The following four functions 7.5-7.8 express these properties. These are then be composed in the revocation function 7.9 to cover all possible scenarios we encountered during our exploration of revocation properties. Space does not permit a detailed description, but this can be found in appendix B.3.

Alloy Function 7.5 *The precondition of revocation. This function evaluates true if a revoking principal* p1 *delegated the policy object* pol *to a principal* p2 *in some state before the current state* cstate *and did not revoke* pol *from* p2 *between that delegation and* cstate*.*

```
fun revocation_precondition (cstate: State, disj p1, p2: Principal,
                             pol: PolicyObject){...}
```

Alloy Function 7.6 *A role was used for the delegation. This function evaluates true if a principal* p1 *delegated the policy object* pol *to* p2 *in a state before the current state* cstate *on basis of a role-based assignment to* pol*.*

```
fun role_was_used_for_del_of_pol (cstate: State, disj p1, p2: Principal,
                                  pol: PolicyObject){...}
```

Alloy Function 7.7 *The object to be revoked was delegated by some other principal. This function evaluates true if some principal* p *other than* p1 *delegated the policy object* pol *to principal* p2 *and did not revoke it before the current state* cstate*.*

```
fun pol_was_delegated_by_other_p (cstate: State, disj p1, p2: Principal,
                                  pol: PolicyObject){...}
```

Alloy Function 7.8 *The principal* p2 *a policy object* pol *is to be revoked from may have held* pol *even before any prior delegation by the revoking principal* p1*. This function evaluates true if principal* p2 *held* pol *directly in the first state of a state sequence.*

```
fun rev_p_held_pol_initially (cstate: State, disj p1, p2: Principal,
                              pol: PolicyObject){...}
```

Alloy Function 7.9 *Weak local revocation function composed of functions 7.5-7.8.*

```
fun weak_local_revoke (disj s1, s2: State, disj p1, p2: Principal,
                       pol: PolicyObject){
 //Precondition: p1 must have delegated pol to p2 before s1
   revocation_precondition(s1, p1, p2, pol) &&
 //Case I: No role was used for this initial delegation
   (!role_was_used_for_del_of_pol(s1, p1, p2, pol) =>
   //Case I.1: No other delegations occurred
     (!pol_was_delegated_by_other_p(s1, p1, p2, pol) =>
       //Case I.1.a: p2 did hold pol initially
         (rev_p_held_pol_initially(s1, p1, p2, pol)    =>
            update_rev_history(s1, p1, p2, pol) &&
            s2.s_subject = s1.s_subject + pol -> p1) &&
       //Case I.1.b: p2 did not hold pol initially
         (!rev_p_held_pol_initially(s1, p1, p2, pol)   =>
            update_rev_history(s1, p1, p2, pol) &&
            s2.s_subject = s1.s_subject + pol -> p1
                                        - pol -> p2)) &&
   //Case I.2: Some other delegation occurred
     (pol_was_delegated_by_other_p(s1, p1, p2, pol)  =>
        update_rev_history(s1, p1, p2, pol) &&
        s2.s_subject = s1.s_subject + pol -> p1)) &&
   ...
 //The following second part contains the same cases
 //if a role was used for the initial delegation.
}
```

These four functions are now composed to define the `weak_local_revoke` function 7.9. For reasons of space we only show the first half of this function above.

The revoking principal `p1` must have delegated a policy object `pol` to `p2` for any revocation to succeed (`Precondition`). If no role was used for the delegation (`Case I`) and the delegation was performed on a direct assignment instead, then we check whether there were no delegations of `pol` to `p2` by other principals (`Case I.1`). This is sufficient as a principal must not delegate the same object twice without an intermediate revocation as defined in the precondition. If `Case I.1` holds, then we check whether principal `p2` held `pol` initially or not (`Case I.1.a` and `I.1.b`) and update the revocation history and `s_subject` relation accordingly. If there were delegations by other principals (`Case I.2`) then we do not need to check for any initial assignments and just update the revocation history and `s_subject` relation. The second part of the function checks for the case of a role having been used for the delegation and is not shown here as it is similar in its structure to the first part. The full function and sequence of delegations and revocations we used to test and validate this weak local revocation with can be found in appendix B.3.

7.5.3.2 Strong local revocation

A strong local revocation would be almost identical in its specification. As in the function `weak_local_revoke()` we would have to check whether a principal was delegated the policy object to be revoked from some other principal. If this is true, like in the example in figure 7.1, and all delegations of an authorisation policy object `auth` to a principal `p2` stem from principal `p1` requesting the revocation, then `p2` will not be subject to `auth` anymore. On the other hand a strong local revoke of `auth` from `p4` by `p2` would have no effect on `p4`'s assignment to `auth` due to the previous delegation of `auth` to `p4` by `p6`.

These scenarios emphasize again the need for not only keeping track of delegated but revoked policy objects as well. Due to the underlying assumption of our model that only one revocation may happen at a time, such a strong local revocation function cannot be used directly, since it may change several relationships which we cannot keep track of. Nevertheless it may be used to assert that a certain sequence of weak local revocations would suffice for the definition of a strong local revocation. With respect to example 7.1 this would mean that a sequence of weak local revocations of `auth` from `p2` by `p1` and `p3` should be equal to a single strong local revocation of `auth` from `p2` by `p1`.

7.5.3.3 Weak and strong global revocation

Alloy has only recently started to support recursion, an indispensable mechanism to support global revocation as we described it in the previous section 7.5.1. At the time of writing our specification there was no available documentation or examples for the use of recursion. Nevertheless, we feel that at least an outline of how to define global revocation must be provided. The constraint analyser could provide us with a reasonable level of assurance about the working of a global revocation function we defined in appendix B.8.

Weak and strong global revocations are similar in their effects to their local counterparts, however, they also consider any possible further delegations of the object to be revoked by the principal this object is revoked from. We have described this in the previous section 7.5.1 and only want to point out some specific issues that need to be considered when defining such a global revocation.

Since recursion is required to provide for a global revocation, this means that the weak and strong global revocations functions consist of two parts. In the first part we check whether the object `pol` had been initially delegated by the revoking principal `p1` to `p2` as previously outlined. In the second part we then need to check whether there was any further delegation of `pol` to some other principal `p`. If this is so the global revocation function calls itself, now with `p2` being the revoking principal and `p` being the principal `pol` is revoked from. An example of such a recursive revocation is provided in appendix B.8.

A series of weak local revocations may achieve the same result as weak and strong global revocations, but we did not investigate this any further considering formal proof, as there was not immediate need in the context of this thesis.

7.6 Analysis of delegation and revocation controls

To analyse and assert the expected behaviour of the delegation and revocation functions, we place them within state sequences that already define some initial assignments or assignments we expect to be true at some stage during the sequence. For example, we may want to explicitly observe that an authorisation `auth` was initially delegated on the basis of a role from a principal `p1` to a principal `p2` and further over principal `p3` to `p4`. This may be expressed in state sequence 7.1 shown in the following.

State Sequence 7.1 *Authorisation* `auth` *is delegated from principal* `p1` *over principals* `p2` *and* `p3` *to* `p4`*.*

```
fun delegation_state_sequence(){some disj s1,s2,s3,s4 : State |
                                some disj p1, p2, p3, p4 : Principal |
                                some auth : Authorisation |
 //Some assumptions about our states
   State_Sequence.first = s1 &&
   s1 -> s2 + s2 -> s3 in State_Sequence.next &&
 //Some initialisations - May have to be adjusted depending on analysis
 //Here auth is assigned to a role of p1 and to no other principal in s1
   (auth -> p1) !in s1.s_subject &&
   auth in all_available_policies  (p1, s1) &&
   auth !in all_available_policies (p2, s1) &&
   auth !in all_available_policies (p3, s1) &&
   auth !in all_available_policies (p4, s1) &&
 //The actual sequence of delegations
   delegate_auth(s1,s2,p1,p2,auth) &&
   delegate_auth(s2,s3,p2,p3,auth) &&
   delegate_auth(s3,s4,p3,p4,auth)
}
```

Running a function expressing such a sequence of states allows us to generate models confirming our stated assumptions. However, in some cases this approach also revealed some unexpected behaviour and the gained insights found their way into our observations in the previous sections 7.4 and 7.5.

There are further simple but effective assertions we can make about the delegation and revocation of authorisations. For example, we consider that a principal `p1` delegates an authorisation `auth` to a principal `p2` in state `s1`. He then revokes the same authorisation from `p2` in state `s2`. This implies that in state `s3` the assignment of policy objects is the same as in state `s1`, expressed as `s1.s_subject = s3.s_subject`. This assertion can be expressed in the following state sequence 7.2, similar in its structure to the previous sequence 7.1. As a side effect this assertion also checks the validity of the frame conditions supporting the delegation and revocation function.

State Sequence 7.2 *Asserting the delegation of an authorisation* `auth` *by a principal* `p1` *and its immediate revocation.*

```
assert delegation_revocation_sequence{all disj s1,s2,s3 : State |
                                      all disj p1, p2, p3, p4 : Principal |
                                      all auth : Authorisation |
  //Some assumptions about our states
    State_Sequence.first = s1 &&
    s1 -> s2 + s2 -> s3 in State_Sequence.next &&
  //Some initialisations - Here that p1 holds auth in s1 and p2 does not
  //but also, for example, that roles were involved in the delegation etc.
    auth in all_available_policies  (p1, s1) &&
    auth !in all_available_policies (p2, s1) &&
  //The actual assertion of the sequence of delegations
    delegate_auth(s1,s2,p1,p2,auth) &&
    weak_local_revoke(s2,s3,p1,p2,auth) =>
      s1.s_subject = s3.s_subject
}
```

We used sequences and assertions like the above to support our findings considering the delegation and revocation of policy objects. They are all very similar in their structure and further examples can be found throughout the appendix B.3.

7.7 Exploring separation, delegation and revocation controls

State sequences like those described in the previous section 7.6 are very explicit about what is supposed to happen. Usually the specifier already has a very clear idea about what properties he is interested in. If there are no such properties, then it would be desirable to follow a more explorative approach. This is specifically the case when looking at possible relationships between the separation controls described in chapter 6 and the delegation and revocation controls discussed in this chapter.

For example, it is interesting to be able to ask questions such as: "*Based on some initial assignments, can a static operational separation always be maintained in a fixed sequence of states, where the functions* `access_object`(), `delegate_auth`() *and* `weak_local_revoke`() *define the state transitions?*". Such a question may then be expressed in a state sequence such as the following 7.3. There, a fixed sequence of states is defined, where any state change has to satisfy some defined functions. Pre- and post-conditions may further constrain this sequence, demanding that some properties hold at the beginning, the end or at some state within a sequence of states. This is in fact an approach similar to the one we outlined in our initial example of domain manipulations in section 4.4.4.

There are some relationships to the area of model checking [Clarke et al., 2000], for example, the temporal patterns reported in [Dwyer et al., 1998, Dwyer et al., 1999]. However, we have not yet explored these in more detail.

State Sequence 7.3 *A general analysis approach using state sequences*

```
fun some_state_sequences(){some disj s1,s2,s3,s4,s5,s6: State |
                           some disj s,s': State - State_Sequence.last |
                           some obj: Object | some disj p1, p2: Principal |
                           some auth: Authorisation |
 //The state sequence
   s1->s2 + s2->s3 + s3->s4 + s4->s5 + s5->s6 in State_Sequence.next &&
   some s -> s' & State_Sequence.next &&
 //General constraints
   static_op_separation() && //static operational separation is maintained
 //Preconditions
   --e.g. no p2 & auth.(s1.s_subject) &&
 //Postconditions
   --e.g. some p2 & auth.(s6.s_subject) &&
 //Functions that may lead from any state s to s'
   (access_object(s,s',p1,auth,obj) ||   //access of an object or
    delegate_auth(s,s',p1,p2,auth)  ||   //delegation of an object or
    weak_local_revoke(s,s',p1,p2,auth))} //weak local revocation
```

We have obtained some useful initial results through this approach, which we then investigated in detail by means of more explicit state sequences. The most interesting cases we found were that of static object-based and operational separation controls preventing the delegation of an authorisation policy, and the delegation and revocation of an authorisation policy to circumvent a static operational separation control. We discuss these two examples in the following paragraphs.

A static operational separation control may prevent delegation activities from succeeding. If a principal delegates an authorisation which is part of a set of defined critical authorisations, and if the receiving principal is already in possession of all but this one authorisation, then this delegation may fail. This has been modelled in a state sequence which can be found in appendix B.7. In a similar manner, a static object-based separation control may conflict with a principal's intention to delegate since the receiving principal may already be in possession of some conflicting authorisations with the same target object as the authorisation to be delegated. An existing static object-based separation control may, however, forbid the receiving principal to be in possession of two such authorisations.

Of more interest is how the delegation and revocation of an authorisation policy may circumvent a static operational separation control. More specifically, the ability of a principal to delegate and revoke policy objects may allow him to circumvent separation controls as he can actively manipulate his set of available policies. We consider a possible variation of the static operational separation control, initially defined in function 6.5, in the following function 7.10. This takes the access attempts of a principal into consideration. Here static operational separation means that a principal `p1` may only access an object `o` between two states `s1` and `s2`, if a critical set of authorisation policies is not a subset of his currently

available policies. In other words, we do not explicitly check what authorisations have been available to him in the past.

Alloy Function 7.10 *A variation of the static operational separation control defined in function 6.5. This variant explicitly considers the access attempts of a principal.*

```
fun static_op_separation_variant(){all disj s1, s2 : State |
                                   all c_set: Critical_Authorisation_Set |
                                   all p1 : Principal | all o : Object |
                                   all auth : Authorisation |
  s1 -> s2 in State_Sequence.next &&
  access_object (s1,s2, p1, auth, o) =>
    c_set.critical !in all_available_auth_policies(p1,s1)}
```

We consider further that two authorisations `auth1` and `auth2` make up a critical authorisation set `c_set.critical` and are also available to a principal `p1`. Considering the above variation of the static operational separation, this alone would not cause a conflict. However, `p1` may delegate `auth2` to some principal `p2`. He then accesses some object `o` using `auth1`, revokes `auth2` from `p2`, delegates `auth1` to `p2` and then accesses `o` using `auth2`. It is clear how this may circumvent the intended functionality of the static operational separation 7.10. The following state sequence 7.4 supports this and the full specification is part of appendix B.7.

State Sequence 7.4 *Circumventing a static operational separation.*

```
fun circumvent_static_operational_separation(){
  //Quantifications and initialisations
    ...not shown for reasons of space...
  //Static operational separation must hold
    static_op_separation_variant() &&
  //p1 delegates auth2 to p2
    delegate_auth(s1,s2,p1,p2,auth2) &&
  //p1 accesses o using auth1
    access_object(s2,s3,p1,auth1,o) &&
  //p1 revokes auth2 from p2
    weak_local_revoke(s3,s4,p1,p2,auth2) &&
  //p1 delegates auth1 to p2
    delegate_auth(s4,s5,p1,p2,auth2) &&
  //p1 accesses o using auth2
    access_object(s5,s6,p1,auth2,o)}
```

In contrast to the static operational separation control defined in function 6.5, the variant used here allows a principal to be in possession of all critical policies of some critical set, as long as he does not attempt to perform an access. This is the reason why the state sequence described above would not yield a model if the static operational separation defined in function 6.5 were used instead of the variation 7.10. In fact, it was this exploration that made us detect such a circumvention and change the static operational constraint accordingly.

7.8 Related work

Within automated systems the basic mechanisms and problems of delegating and revoking policies have received attention as early as in the HRU model in the form of propagating access rights at the discretion of a subject [Harrison et al., 1976], or in the form of database grant/revoke mechanisms [Griffiths and Wade, 1976]. More advanced models such as the typed access matrix [Sandhu, 1992], or the extended authorisation models of [Bertino et al., 1997a, Bertino et al., 1999b], that provide low-level mechanisms for delegation at the operating system or middleware level, have also treated this issue and a related discussion was part of section 3.2. Thus, the following sections provide a summary and evaluation of more specific work on the delegation and revocation controls along the lines of the RBAC96, OASIS and PONDER frameworks (see sections 3.3 and 3.4), and the distinction between administrative and *ad hoc* delegation in section 7.3.

7.8.1 Delegation and revocation in role-based models

7.8.1.1 Delegation and revocation in RBAC96-style systems

In RBAC-style systems, we can distinguish between two main approaches to administration and delegation, using either distinct administrative roles to administer role-based systems, or allowing for delegation based on the discretion of principals acting through 'regular' roles with delegation authority.

Administrative role-based access control (ARBAC) models are based on the idea of using roles to administer roles [Sandhu et al., 1999, Sandhu and Munawer, 1999, Oh and Sandhu, 2002]. They are all based on the explicit distinction between an administrative and a regular role. Administrative roles have authority over regular, and possibly other administrative roles. In the following, we use the ARBAC97-URA97 model, initially described in section 3.3.1, as the basis for our discussion. This model defines how user-role relationships may be managed. The roles that an administrative role has authority over are defined through the concept of an administrative range. To be eligible for role membership users have to fulfill certain prerequisite conditions such as already being a member of some general employee role. The authority to control these user-role assignments is captured in a relation $can\_assign \subseteq AR \times CR \times 2^R$. For example, the expression can_assign (ar_x, rr_y,{rr_a, rr_b, rr_c}), would state that a member of the administrative role ar_x can assign a user, who is currently a member of regular role rr_y, to be a member of regular roles rr_a, rr_b or rr_c. Likewise, a revocation relation $can\_revoke \subseteq AR \times 2^R$ controls the revocation of a user's membership from any regular role in the powerset 2^R through an administrative role from AR. However, no detailed description of further criteria for administrative delegation as we provided them is given. In a similar manner, the authority to change the role-permission (PRA97 submodel) and role-role (RRA97 submodel) relations are defined. We provided a more exhaustive comparison and discussion on some of the disadvantages of this approach in [Kern et al., 2003].

Approaches like the role-based delegation model RDM2000 do not explicitly distinguish between administrative and regular roles. They consider user-to-user delegation as the process of a user delegating his role to another user [Zhang et al., 2001]. RDM2000 is also based on RBAC96, reviewed in section 3.3.1, but distinguishes between original and delegated user-role assignments. The authority to delegate is defined in a relation $can\_delegate \subseteq R \times CR \times N$, with R, CR and N defining the sets of roles, prerequisite conditions and maximum delegation depth respectively. A user may either delegate one of his original roles or a role he received through a previous delegation. Several forms of revocation based on a relation *can_revoke* are introduced, but are only discussed very informally.

The role-based delegation model RBDM0 described in [Barka, 2002] also uses this notion of original and delegated roles, and the authority to delegate is captured in a relation $can\_delegate \subseteq R \times R$, so a user in an original role r_1 may delegate this role to another user in a role r_2. Note that here this second role would actually be a precondition when thinking in terms of the ARBAC97 or RDM2000 model. Additionally, RBDM0 presents a set of characteristics along the lines of which the delegation of roles may be categorised.

Within the context of our distinction between administrative delegation and *ad hoc* delegation as outlined in the initial section 7.3, we may summarise and compare the ARBAC97, RDM2000 and RBDM0 approaches as follows:

1. We distinguished that an administrator is usually not related to the objects he administers while *ad hoc* delegation requires a relation between the delegating principal and the object to be delegated. None of the three models makes this explicit distinction, which initially leads to confusion about the terms administration and delegation. However, the way they regulate administration and delegation follows our distinction. ARBAC97 separates administrators from regular roles that they may assign to users by means of distinct role hierarchies. There seems to be no explicit constraint that an administrator cannot administer any regular or even administrative roles he may hold himself, but this may be easily expressed if required. In RDM2000 and RBDM0 a user may only delegate a role of which he is actually a member which also satisfies our definition.

2. We further discussed the authority to administer or delegate. All three models explicitly state this authority in the form of an authorisation relation. This again indicates that any of these activities cannot occur totally at random but must have been identified during the setup and evolution of the supporting system.

3. Finally, we considered the duration of administered or delegated relationships. This is of course not quantifiable in this context, but there are indications that support our definition. ARBAC97 does not make any specific assumptions about the duration of the relationships between, for example, users and roles. Both RDM2000 and RBDM0, however, state timing constraints as a criterion for revocation. Again, this point would actually require empirical validation, but nevertheless appears to be a useful informal criterion to distinguish between administration and delegation.

7.8.1.2 Appointment in OASIS

The basic OASIS model has been summarised in section 3.3.2. One specific extension to OASIS is the notion of appointment [Bacon et al., 2002, Yao et al., 2001]. Appointment extends and abstracts role delegation and an appointment certificate does not directly convey privileges to its holder, but allows him to activate roles with assigned privileges instead. Such an appointer role encompasses the authority to appoint, a concept related to ARBAC's administrative roles (see section 7.8.1.1). The holder of an appointer role is not necessarily delegating in the sense of our definition made in section 7.3, as he does not have to be entitled to the privileges conferred by the appointment certificates. Delegation is defined as the process of a principal transferring certain privileges to an agent to perform tasks on his behalf [Bacon et al., 2002]. Appointment is compared to the role-based delegation of [Barka, 2002], discussed in the previous section, and it has been observed that a partially delegated role is in fact a new role sharing an overloaded name, defining a subset of the privileges of the delegating role [Bacon et al., 2002]. A grantor can also only delegate the privileges that he possesses. As such the delegation of roles can be seen as a special case of appointment in which a user holding some role may appoint some other user to activate the same role.

OASIS is the only model which considers the nature of an appointment and thus distinguishes between task assignment and qualification-based appointment. In task assignment the role of an appointer corresponds to the responsible job entity such as a project manager role. Such task assignment is thought of as being transient, i.e. for the duration of some task that needs to be performed. In a qualification-based appointment the appointment certificate is not issued for a single purpose but reflects long-lived and persistent qualifications such as those obtained through passing the examination of a certified accountant. This is similar to the distinction we made in section 7.3 considering time as a possible indicator for administration or delegation. Lastly, it may be worth noticing that there is no explicit notion of retaining or losing a role that is delegated. Also OASIS does not allow for the delegation of individual privileges as any privileges derive solely from the roles activated on the basis of an appointment.

OASIS claims to support rapid and selective revocation. This is done by invalidating the certificate issued on an appointment. When an appointment is made, the appointer is issued not only with an appointment certificate but also with a revocation certificate that is specific to that appointment. In summary, four possible ways of revoking an appointment are listed in [Bacon et al., 2002]. These include revocation by the appointer; by anyone in the appointer role which was used for the appointment; by resignation of the appointee; or by rule-based system revocation (e.g. expiry time or fulfillment of tasks).

To our knowledge OASIS does not claim to support any form of cascading appointment and no explicit discussion on more complex revocation scenarios as we described them in section 7.5 is given. Nevertheless, OASIS does not seem to exclude the appointment of the same principal by multiple other principals. Considering revocation this would require at least a notion of weak and strong revocation. This is not provided in OASIS as far as we know.

7.8.2 Delegation in policy-based models

The concept of authority was recognized in [Moffett and Sloman, 1988], identifying it as one out of several access control policy components and understanding it as the legitimate power to make policy decisions. The focus is on the delegation of authority in accordance with the policies of the management of an organization. In particular, the issue of the source of authority; the question of ownership over resources; and the impacts of policy decisions are discussed. However, this discussion is restricted to an organisational level, and a detailed description or suggested method for how authority and ownership actually find their way into a computer system is not given. A distinction between having access rights and being allowed to pass those rights on is made. This general decoupling of the assignment of policies from having polices is what has been identified by us in section 7.3 as one criterion supporting the distinction between administrative delegation and *ad hoc* delegation. Identifying and making the authority structure of an organisation explicit acts as policy enforcing, since any policy decision without the necessary authority is not recognised as binding by the involved entities. A model based on Prolog is presented to assess the consequences of delegating policies and to validate security policies. This model incorporates the organisational authority structure; the resource authority structure; and the general mechanism of delegating authority within the defined hierarchies. This distinction between organisational and resource authority structure is important, but no detailed indication of how the one maps onto the other is given.

In [Sloman and Moffett, 1991] the application of role domains to permit a flexible but controlled delegation of authority is discussed. The delegation of authority is based on the negotiation between managers and is not administered from a central point. In addition, the paper discusses the subtle differences between the tasks of the owner, manager and security administrators. The concept of (role) domains is used to represent management structures. Domains name groups of objects to which a common policy can be applied. Role domains extend that concept in order to define the range and nature of the managerial authority of the members of a domain. They have scope attributes used to identify the objects over which the domain members have managerial authority. This work has greatly influenced the Ponder framework, reviewed in section 3.4, and as such the specific aspects of delegating authority (and obligations) in Ponder are discussed in the following two sections.

7.8.2.1 The delegation of policies in Ponder

Ponder [Damianou, 2002], defines delegation policies which specify the authority to delegate for a subject. A detailed example of the structure of such a delegation policy has been given in section 3.4, which may also include constraints such as the number of possible cascading delegations. However, such a delegation policy does not control the actual delegation [Damianou et al., 2001]. A delegation policy merely permits subjects to grant privileges they possess, due to an existing authorisation policy, to grantees. It thus states the authority to delegate and its actions comprise a *delegate()* method. When this method is executed by the delegating principal, a separate authorisation policy is created with the receiving principal as the subject.

Pre-condition	Delegation of Authority	Delegation of Obligation
Policies to be delegated	`auth+` (2.)	`oblig` (1.)
Authority to delegate	`deleg+` (3.)	not explicitly supported (4.)
Obligation to delegate	not explicitly supported (6.)	not explicitly supported (5.)

Table 7.6: Ponder coverage matrix.

With respect to the evaluation criteria of section 7.3, we observe that the authority to delegate is explicitly defined in a delegation policy. There are, however, no specific assumptions about the delegating principal retaining or losing the delegated authorisation. The fact that a delegation policy contains a `valid` clause suggests that delegation is not permanent, but limited to a specific period or other temporal constraint.

There is no explicit work in the Ponder context addressing the revocation of delegated policies as this is believed to be a more application specific issue [Damianou, 2002]. Also, Ponder only supports the delegation of authorisations, and it is natural to ask how far the delegation of obligations could be supported. The requirements for this are discussed and evaluated in the following section 7.8.2.2.

7.8.2.2 Supporting the delegation of obligation policies in Ponder

While Ponder explicitly supports the authority to delegate authorisation policies, it does not make any assumptions about:

- the obligation of a principal to delegate policies;
- the specific properties of delegating obligations.

The following scenario serves as an example to explain and discuss these issues. This has been initially presented in [Schaad and Moffett, 2002c]. We consider a principal `A` delegating the obligation `print_report` to a principal `B`. This requires the specification of the following policies, where each item represents an entry in table 7.6.

1. The actual obligation that principal `A` intends to delegate.

```
oblig print_report {
  subject A;
  target  report;
  do      print();
}
```

2. The authorisation which allows principal `A` to discharge his printing obligation.

```
auth+ print_report {
  subject A;
  target  report;
  action  print();
}
```

3. A policy which authorizes `A` to create an authorisation policy for `B` to print report. This is a Ponder delegation policy and must refer to policy 2. `auth+ print_report` when the delegate method is executed.

```
deleg+ (print_report) deleg_print_report {
  grantee B;
  subject A;
  action print();
}
```

4. An authorisation policy for `A` to create the obligation to print for `B`. We believe this authorisation policy 4 to be similar in its structure to policy 3, and the policy object `print_report` of the type `oblig` would be part of the header. However, this form of policy does not seem to be explicitly supported by Ponder.

5. An obligation policy to perform the delegation of the obligation. This is because any delegation activity should be driven by an obligation reflecting the need for this delegation. There is no specific policy of this kind defined in Ponder and we do not discuss the feasibility of this any further.

6. An obligation policy to perform the delegation of authority. This is because any authorisation should ideally have a corresponding obligation. There is no specific policy of this kind defined in Ponder and we do not discuss the feasibility of this any further.

If any of the above policies are missing, delegation cannot take place. The extent to which these are supported by Ponder is summarized in table 7.6. The numbers indicate how this corresponds to the given example of delegating an obligation. We have not assessed the practicality of writing a full Ponder specification and the OCL constraints needed to achieve the effect of items 4., 5. and 6. any further. However, although Ponder does not explicitly support the authority to delegate obligations, or the obligation to delegate authority or obligations, it seems likely that extensions to its policy object meta-model could support these functions.

7.8.3 Delegation of obligations and responsibilities

The delegation of obligations within distributed systems is informally discussed in [Cole et al., 2001]. A distinction is made between the transfer of an obligation; sharing; splitting; and outsourcing of an obligation. Only the outsourcing of an obligation can be considered as a delegation within our context, while the other three activities refer to general obligation management that mainly considers questions of obligation refinement. We specifically ruled out the sharing of an obligation instance in fact 5.2, but allow for the sharing of a general obligation as discussed in section 5.3.3. The suggested splitting of obligations might be represented in our approach by a re-assignment of the relevant actions, but this has not been investigated any further.

The delegation of obligations and responsibility has also been described in the context of the ORDIT methodology for requirements engineering [Strens and Dobson, 1993, Dobson, 1993]. A distinction is made between consequential and causal responsibility. Consequential responsibility is a ternary relationship between two agents and a state of affairs, and causal responsibility as a binary relation between an agent and an event or a state of affairs. While an agent is responsible for something, an obligation expresses that he has to (not) do something. A binding between an obligation and a responsibility is created through a state of affairs they are associated with. The obligation maintains or changes the state of affairs for which the responsibility is held. Obligations can be delegated and the new obligation holder becomes responsible to the original obligation holder for discharging the obligation. This leads to a distinction between functional obligations that define what an agent must do with respect to a state of affairs, while structural obligations arise out of the delegation of obligations.

Both lines of work do, however, give no further technical explanation of how such a delegation may be supported and reasoned about. Additionally, the concepts of obligation, responsibility, accountability etc., are used outside any existing formal framework or taxonomy and no precise definitions are given.

7.9 Chapter summary and conclusion

In this chapter we have investigated the delegation and revocation of policy objects within the context of our suggested control principle model.

Initially, we looked at the organisational motivations that may drive such delegation activities in section 7.2. We realised some ambiguities in the current literature between delegation as an administrative activity and *ad hoc* delegation at the discretion of users in non-administrative roles. To mitigate this situation, we presented three characteristics which we then used for the rest of this chapter to clearly state what we mean by administrative delegation and *ad hoc* delegation in section 7.3. These criteria were also used in the later evaluation of related work and comprised:

- the representation of the authority to delegate;
- the specific relation of a principal to an object;
- the duration of this relation.

We then defined in section 7.4.1 what we believe to be the most basic form of a delegation function. This function was used to support our discussion on the delegation of authorisations in section 7.4.2, as well as the delegation of general and specific obligation policies in section 7.4.3. Specifically the fact that policy objects may be assigned to a principal directly or on the basis of a role led to the discovery of some interesting properties. The insights gained in this discussion were then used to clarify the meaning of delegating policy objects through delegating roles in section 7.4.4.

In section 7.5 we looked at the revocation of delegated policy objects. For this we assessed the revocation framework presented in [Hagstrom et al., 2001] and its notion of weak and strong, local and global revocation. We then used this framework to informally discuss the revocation of authorisations in section 7.5.1, and general and specific obligations in section 7.5.2. We investigated in section 7.5.3 how the exploration of revocation properties for policy objects may be further supported by using Alloy. This resulted in the definition of a weak local revocation function. We argued that this function is sufficient to support the strong local and weak and strong global revocation of policy objects if required.

We provided an outline of how we checked and asserted the expected behaviour of the delegation and revocation functions in section 7.6. This was followed by the exploration of relationships between separation, delegation and revocation controls in section 7.7. In particular, we gave evidence that separation controls may prevent delegation activities, and demonstrated that the use of delegation and revocation controls may lead to a circumvention of separation controls. However, it can be claimed that the violations and conflicts that were described are too hypothetical and could be avoided by simple precedence rules, e.g. strict separation must always be maintained. This is of course true, but the consequence of this may be that the organisation's business is obstructed by excessive control. The alternative approach is setting static organisational controls at a level that may allow violations, and supplementing them with dynamic controls and post hoc audits to maintain the required degree of control in the organisation. This is, however, a hard problem. To begin with, the question of the required or optimum degree of control has been investigated even before automated information systems found their way into organisations, e.g. [Weber, 1947, Fayol, 1949, Urwick, 1952, Dalton and Lawrence, 1971, Salaman and Thompson, 1980]. Secondly, it has only been recently that progress has been made in the area of policy conflict detection and resolution, e.g. [Jajodia et al., 1997b, Lupu, 1998, Chomicki et al., 2000, Schaad, 2001, Dunlop et al., 2002, Bandara et al., 2003]. Space does not permit a detailed discussion.

Finally, we critically evaluated some of the existing work on delegation and revocation in section 7.8. This review was mainly performed along the three frameworks initially reviewed in sections 3.3 and 3.4, and the three criteria introduced in section 7.3. We also discussed in section 7.8.2.2 what requirements the delegation of obligations would raise with respect to the Ponder language.

With respect to the overall context of this thesis we have established a conceptual model of control principles in chapter 5 together with a set of separation, delegation and revocation controls in chapters 6 and 7. We now focus on how control may be achieved over the delegation of specific and general obligations through the concepts of review and supervision in the following chapter 8.

Chapter 8

Review and Supervision

8.1 Introduction

In the previous chapter we have described the delegation of policy objects. We investigated the delegation of obligation policies which required us to make a distinction between:

- the delegation of specific obligation instances and;
- the delegation of general obligations.

These two forms of delegating obligations raise concerns about how control may be retained over a delegated obligation by the delegating principal. In this chapter we describe and discuss the two control principles of review and supervision to address this issue. This includes:

- A discussion on the general motivation for achieving control over delegated obligations in section 8.2.1;
- An initial example and formal description of the review concept as the obligation of a principal to examine the results of a previously delegated obligation in sections 8.2.2-8.2.5;
- An exploration of delegation and review controls, focusing on the delegation of delegated obligations, and the delegation of the review of a delegated obligation in section 8.2.6;
- A discussion on the general motivation for establishing supervision hierarchies between positions in section 8.3.1;
- An initial example and formal description of the supervision concept as the general obligation of a principal in a position to review the obligations of principals in subordinate positions in sections 8.3.2 and 8.3.3.

The concepts established in this chapter, although commonly used in our daily vocabulary, have to our knowledge not been treated at this level of detail in the existing literature.

8.2 Review

8.2.1 The concept of review and its organisational motivations

Obligations are continuously created, delegated, revoked or discharged according to the overall goals of an organisation and the general principle of distributing work. Ideally, there should never be any uncertainty about who currently holds an obligation, whether somebody has discharged his obligations, the effect of such a discharge, and who has to ultimately ensure that the tasks of an obligation are performed. For this reason it is necessary to hold to account persons who delegate obligations. This also includes the ability to trace back any delegated obligations to the initial delegator. In order for them to be able to give an account of the obligation that they have delegated, they must review it. We propose that this may be done by creating a review policy referring to the delegated obligation. The holder of such a review policy has then to make sure that the obligation he delegated has been carried out satisfactorily.

A review does not act as a direct enforcement mechanisms for the delegated obligation, but as a post-hoc control. In that context, review may been seen as a detective mechanism. If the review fails because the delegated obligation has not been discharged at all or satisfactorily, this may trigger corrective measures to be taken by the reviewer himself or some other principal with the required authority.

However, in a more general sociological context, the awareness of the presence of a review may already have an effect on the behaviour of the human principal to whom the obligation to be reviewed has been delegated. In that context, review may be seen as a preemptive control, trying to prevent the current obligation holder from not discharging his obligation. Additionally, the delegating principal is prevented from delegating without being held accountable. Both parties, the delegator and the delegatee, thus know about the effects of delegating and accepting a delegated obligation. Although mainly a form of control, review may also help to reduce uncertainty about how to discharge the delegated obligation. Since the principal to whom the obligation has been delegated should know about the source of this delegation, he may call upon the initial obligation holder to provide him with the necessary information to allow for a successful discharge. However, we have to leave these sociological and organisational theory related issues of the review concept aside.

In the context of this thesis, review is understood as an obligation referring to a previously delegated obligation which has to examine the results of the discharge of this delegated obligation. We recall our previous distinction between procedural and output controls as the components of administrative control in section 2.5. We may then argue that review conforms to both these definitions of control. On the one hand the creation of a review is part of the delegation procedure for certain obligations, while on the other hand this review then controls the output of the discharge of a delegated obligation through examination.

Before describing the specific details of delegating obligations in the context of our control principle model, we illustrate the concepts we have established so far in an informal motivating example in the following section 8.2.2.

8.2.2 Reviewing delegated obligations - A motivating example

A review policy is the obligation of a principal to investigate the state of affairs of an obligation he delegated to another principal. In other words, a review is an obligation on an obligation, where some defined review actions provide the application specific information on how to perform the review. The following example illustrates this.

We can imagine an obligation policy for Jon to prepare the quarterly sales report to have the following, Ponder-like, structure:

```
oblig  Prepare_Sales_Report_1stQ {
 on          01/05/01;
 subject     Jon;
 target      Sales_Database;
 do          collect_departmental_reports(),
             consolidate_reports(),
             access_report_generator();
}
```

This says that on the 1^{st} of May Jon must prepare the report by performing the required actions with the sales database being the target. Current business requires Jon to delegate this obligation to Clara. We assume that Clara is equally qualified and authorised to do this.

In his role as the delegator, Jon now has to review that Clara carried this task out satisfactorily before the deadline. This is done by defining some evidence that has to be generated by Clara when discharging the delegated obligation, as well as the review actions that are used to review this evidence. Here, this evidence may be the actual `Sales_Report_1stQ` that is generated, and in this context we assume that there are some defined tuples to relate this evidence to the actions of the review.

```
reviewed_by (Sales_Report_1stQ, view_report())
reviewed_by (Sales_Report_1stQ, calculate_checksums())
```

Together with this information the review obligation for Jon looks like the following:

```
review oblig Prepare_Sales_Report_1stQ{
 on          01/05/01;
 subject     Jon;
 target      Obligation "Prepare_Sales_Report_1stQ";
 do          view_report(),
             calculate_checksums();
}
```

Jon can now discharge his review obligation by viewing and calculating the checksums in the sales report Clara generated when discharging the obligation Jon had delegated to her.

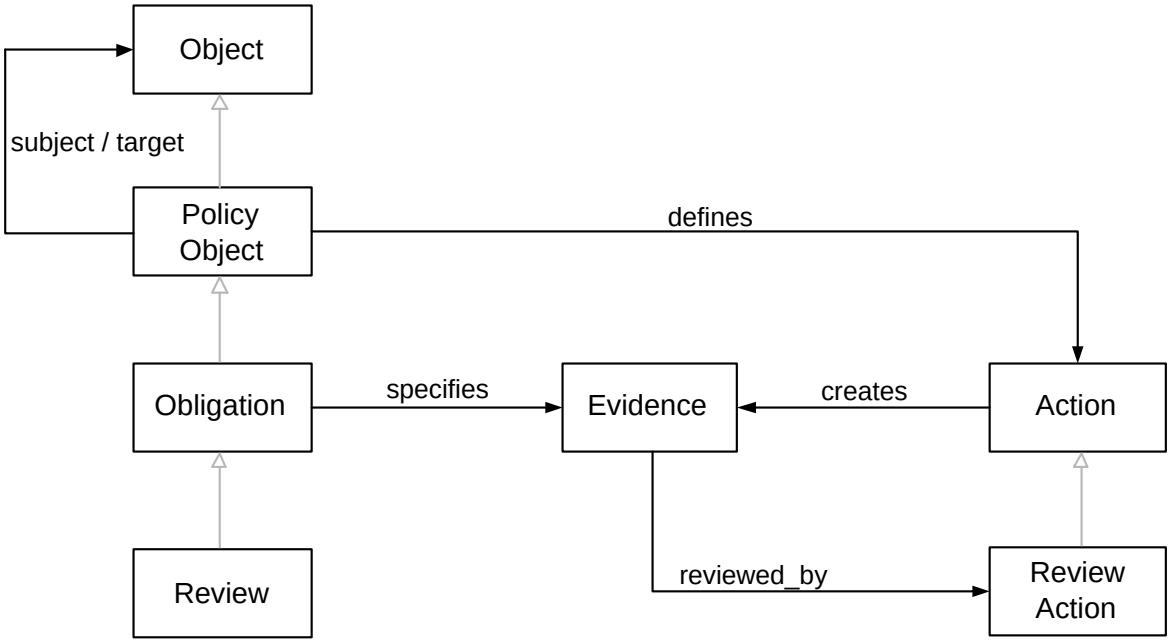

Figure 8.1: Review specific conceptual model.

8.2.3 Expressing review controls in the context of the CP model

In the context of the control principle framework, the activity of review describes a post-hoc control that aims at controlling delegated obligations. Several issues had to be taken into consideration for expressing review controls in the context of the control principle model. Figure 8.1 supports the following discussion on these issues and serves as a graphical representation of parts of the conceptual model initially described in figure 5.1. The facts and functions to be introduced are also part of appendices B.1 and B.4.

A review policy is created as the result of delegating an obligation and is a specific type of obligation itself. This specialisation relationship between an obligation and a review obligation has been modelled using Alloy's extension mechanism as described in section 5.1. Any further specific obligation types found to be useful could be integrated in a similar fashion.

We enforce in the following fact 8.1, that a review obligation may only have other obligations as its target. This might be a too restrictive constraint in real-world situations, but will facilitate and support later analysis in the context of this thesis.

Alloy Fact 8.1 *A review object may only have an obligation as its target.*

```
fact {all s: State | all r: Review |
  r.(s.s_target) in Obligation}
```

Being policy objects, ordinary obligations and review obligations both define actions, however a review obligation may only define review actions, while an obligation which is not a review should only define non-review actions. Again, in real world scenarios the corresponding fact 8.2 might be too restrictive, but it is nevertheless expressed here for reasons of later analysis.

Alloy Fact 8.2 *A review object only defines review actions, and an obligation which is not a review only defines actions which are not review actions.*

```
fact {all s: State | all r: Review |
  r.(s.s_defines) in ReviewAction}

fact {all s: State | all o: Obligation - Review |
  o.(s.s_defines) in (Action - ReviewAction)}
```

Obligations specify evidence which is created when an obligation is discharged. We initially described this in section 5.4. This evidence may be an artefact created alongside the manipulation of the delegated obligation's target object, or simply the target object itself. For example, in the first case we consider a receipt which is issued on the basis of some transaction. Likewise, a form of evidence may be obtained by querying the current balance of an account object which has been the target of a delegated obligation. However, in the context of this model we abstract evidence as an Alloy signature.

It must be the case that any evidence is reviewed by the actions defined in the review that has the obligation specifying the evidence as its target. This correspondence ensures that evidence can actually be reviewed by the reviewing subject. Note that no strict equality is enforced in the supporting fact 8.3, i.e. a review may define actions that support the review of some different evidence.

Alloy Fact 8.3 *Evidence is reviewed by the actions that are defined by the review that has the obligation specifying the evidence as its target.*

```
fact {all s: State | all o: Obligation |
      all e: Evidence | all a: Action |
 a in e.(s.s_reviewed_by) =>
  a in ((s.s_target).((s.s_specifies).e)).(s.s_defines)}
```

It follows from this that if some review has an obligation as a target, then its review actions must actually review the obligations evidence. This now establishes the strict correspondence and facts 8.4 and 8.3 could be expressed in the form of one single constraint. However, this separate definition allowed for the expression and investigation of several other scenarios that are not further described here.

Alloy Fact 8.4 *If some review has a target then the review's actions must review the targets evidence.*

```
fact {all s : State |
      all disj o1, o2 : Obligation |
 o1 in Review &&
 o2 in o1.(s.s_target) =>
  o2.(s.s_specifies) in (s.s_reviewed_by).(o1.(s.s_defines))}
```

Additionally, the relationship between the evidence and the actions creating the evidence needs to be constrained. These actions could of course also be review actions as we showed in section 5.2.1. It should be the case that if some evidence is declared to be created by some actions, then these actions should correspond to the actual obligation specifying the evidence. Again, it is a matter of choice and current context, whether a strict equality is enforced as done in the supporting fact 8.5, or whether only some of the obligation's actions may be needed to create the evidence.

Alloy Fact 8.5 *If some evidence is created by some action then this action must be defined by the obligation specifying the evidence.*

```
fact {all s : State | all e : Evidence |
  (s.s_creates).e = ((s.s_specifies).e).(s.s_defines)}
```

This relationship between the evidence of a delegated obligation policy, and the actions in the review policy that has been created in its place, is application-dependent. This implies that these relationships have either been negotiated by the involved principals at the time of delegation, or have been defined by an administrator at some prior stage. Within this context we do not explicitly define either case, as we are primarily interested in the structural aspects of delegating obligations.

8.2.4 A delegation function defining review controls

In the previous chapter 7.4 we have discussed how a policy object is generally delegated between two subjects, and that there is a difference between delegating authorisations and obligations. We now investigate the definition of a delegation function for obligations that supports the concept of review. However, we first present a generalised description of delegating an obligation, which is then made target to a review obligation. We do not yet make a distinction between delegating specific and general obligations. This allows us to analyse some of the basic review properties first.

In some initial state `s`, a principal `p1` is assigned to the obligation `obl` to be delegated either directly or through his role. After the delegation in state `s'`, `p2` is now subject to `obl`, while `p1` is now subject to the new review obligation `rev'` which has `obl` as its target. This can be captured in the following function 8.1 `general_delegation_with_review()`. Here it can be observed how the `s_subject` and `s_target` relations are updated accordingly to these observations, additionally distinguishing between a direct or role-based delegation. Some evidence `e` must result out of this delegation. A function `new_review(s, rev')` ensures that the object `rev'` did not exist in state `s`. This is complemented by a framing condition and an update to the history of the initial state `s`, distinguishing between a direct or role-based delegation. The components of this delegation function are mostly identical to parts of the earlier defined delegation functions 7.1 and 7.3. However, the presence of the review `rev'` obligation requires additional changes to the `subject` and `target` relationships. This does not allow us to reuse these earlier defined delegation functions. We refer to section 10.5.1 in our case study for a possible organisational context of such a delegation.

Alloy Function 8.1 *A general delegation function for obligations that integrates a concept of review. Principal* `p1` *delegates obligation* `obl` *to principal* `p2`*. As a result, the review obligation* `rev′` *is created which has* `obl` *as its* `target` *and* `p1` *as its* `subject`*.*

```
fun general_delegation_with_review (disj s,s' : State,
                                    disj p1, p2 : Principal,
                                    disj obl : Obligation,
                                    disj rev' : Review) {
 //p1 holds the obligation to be delegated
   (obl in (s.s_subject).p1 ||
   (obl in (s.s_subject).((s.s_has_member).p1))) &&
 //Delegation and assignment of review
 //Case 1: If principal p1 holds obl directly in state s, then p2 will
 //be subject to obl in s' while p1 may or may not lose obl.
   (p1 in obl.(s.s_subject) =>
     (s'.s_subject = s.s_subject + obl -> p2
                                 + rev' -> p1 ||
      s'.s_subject = s.s_subject + obl -> p2
                                 - obl -> p1
                                 + rev' -> p1))          &&
 //Case2: If obl is in one of p1's roles, then p2 must be subject to obl
 //in s' but p1 retains obl.
   (obl in (s.s_subject).((s.s_has_member).p1) =>
      s'.s_subject = s.s_subject + obl -> p2
                                 + rev' -> p1)           &&
 //The target relationship needs to be updated for the
 //new review obligation
   s'.s_target = s.s_target + rev' -> obl &&
 //There is some evidence after delegation
   some e : Evidence | e in obl.(s'.s_specifies) &&
                       no obl.(s.s_specifies) &&
 //The review object did not exist in previous states + frame condition
   new_review(s, rev') &&
   delegate_obl_with_review_frame(s,s',p1, p2, obl, rev') &&
 //Update of the delegation history
   (s.s_delegation_history).delegating_principal = p1 &&
   (s.s_delegation_history).receiving_principal = p2 &&
   (s.s_delegation_history).delegated_policy_object = obl &&
   (some r : Role | (obl in (s.s_subject).p1 =>  //No role was used
                       no (s.s_delegation_history).based_on_role) &&
                    (obl in (s.s_subject).r &&
                     p1 in r.(s.s_has_member) => //A role was used
                       (s.s_delegation_history).based_on_role = r))
}
```

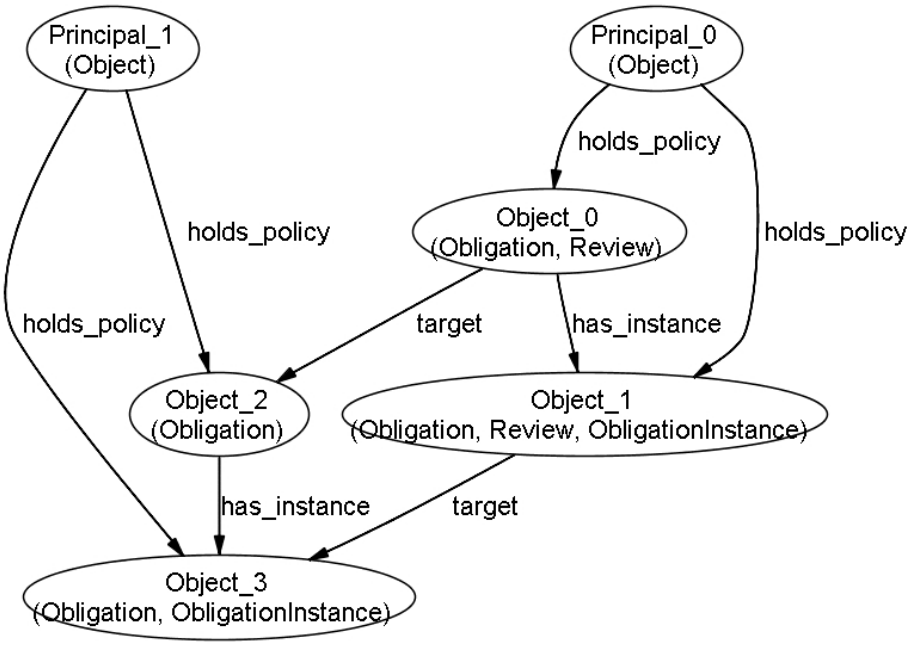

Figure 8.2: A model of general and specific review instances and their targets.

8.2.5 General and specific review obligations

We have explored the basic concepts of reviewing delegated obligations in the previous sections. Now it needs to be investigated how these results fit into our abstraction of general and specific obligations we introduced in section 5.3.3 to resolve the problem of assigning roles and obligations. However, before doing so we must first assess how far a review obligation itself has to adhere to this abstraction.

Since a review is an obligation, the distinction between general and specific obligations must also apply to it. The implication is that any specific review obligation has a general review obligation of which it is an instance. This further means that a principal must have a general obligation to review for any of his particular review instances. What should be the relationship between this general review obligation and some delegated obligation instance that is to be reviewed? There is no direct relationship. However, the general review obligation should have the general obligation, of which any obligation to be reviewed is an instance, as its target. A review instance can only have some other obligation instance as a target, if the same is true for their respective general obligations. This is captured in fact 8.6.

Alloy Fact 8.6 *The general obligation of a review instance should have the general obligation of the delegated obligation as its target.*

```
fact {all s: State |
      all rev: Review & ObligationInstance |
      all o: ObligationInstance |
   o in rev.(s.s_target) =>
     (s.s_has_instance).o in ((s.s_has_instance).rev).(s.s_target)
}
```

This implies that this general review obligation was already held by the delegating principal before the delegation, or was created at the time of delegation. The organisational circumstances for either case or outside the scope of this discussion.

This discussion is supported by figure 8.2 which is part of a model generated when using the delegation function 8.1 for the exploration of these concepts. Here a principal (Principal_0) has delegated an obligation instance (Object_3) to another principal (Principal_1). The delegating principal has the general obligation to review (Object_0) which now has a review obligation (Object_1) as its instance as a result of the delegation. This review instance has the delegated obligation instance as its target, corresponding to the target relationship between the two general obligations. For visualisation purposes we use the `holds_policy` instead of the `subject` relation here, the correspondence has been defined in fact 5.14.

Now we can discuss how delegated general and specific obligations may be made subject to review in more detail.

8.2.5.1 Reviewing delegated obligation instances

We have discussed the delegation of a specific obligation in section 7.4.3.1. The most important supporting constraint was that both principals must hold the same general obligation with respect to the delegated policy object. In combination with fact 8.6, the delegation of obligation instances and according generation of a review obligation instance for the delegating principal is straightforward to specify. In fact, we may use the previously introduced delegation operation 8.1 to do this.

Table 8.1 summarises the possible assignments of principals in the context of delegating and reviewing obligation instances. Here, `iob` refers to the obligation instance that is delegated between principal `p1` and `p2`, while `gob` is the general obligation `iob` is an instance of. The general review obligation `grev` and its instance `irev` complement this.

We can observe that for the delegating principal `p1` one of the four cases in I must be true. He must always be directly assigned to obligation instance `iob` that is to be delegated. We recall fact 5.5, that excludes a general obligation to be held through a role and directly at the same time. Thus, the corresponding general obligation `gob` must be held either directly or through a role. The general obligation `grev` for the review instance that is to be created must also be held either directly or through a role. After the delegation, principal `p1` loses `iob`, and may be assigned to the other involved policy objects as defined in II. For reasons of clarity we explicitly state that he may not be assigned to `iob` anymore through the expression `p1 !in iob.(s'.s_subject)`. Instead, he must now directly hold the review obligation instance `irev`. As in the previous state `p1` must also hold both general obligations `grev` and `gob` through a role or directly. For principal `p2` the following may be observed before the delegation in III. The receiving principal `p2` may not be assigned to the obligation instance `iob`. Again, we chose to state this directly although this is implied in fact 5.2. Additionally, he should be assigned to `gob` for the delegation to succeed, although we did not explicitly state this anywhere. After the delegation in IV, `p2` holds `iob` and the general obligation `gob`.

I. Assignments in state s for principal p1	II. Assignments in state s' for principal p1
Case 1: p1 in iob.(s.s_subject) && p1 in gob.(s.s_subject) && p1 in grev.(s.s_subject) Case 2: p1 in iob.(s.s_subject) && p1 in gob.(s.s_subject).(s.s_has_member) && p1 in grev.(s.s_subject) Case 3: p1 in iob.(s.s_subject) && p1 in gob.(s.s_subject) && p1 in grev.(s.s_subject).(s.s_has_member) Case 4: p1 in iob.(s.s_subject) && p1 in gob.(s.s_subject).(s.s_has_member) && p1 in grev.(s.s_subject).(s.s_has_member)	Case 1: p1 !in iob.(s'.s_subject) && p1 in gob.(s'.s_subject) && p1 in grev.(s'.s_subject) && p1 in irev.(s'.s_subject) Case 2: p1 !in iob.(s'.s_subject) && p1 in gob.(s'.s_subject).(s'.s_has_member) && p1 in grev.(s'.s_subject) && p1 in irev.(s'.s_subject) Case 3: p1 !in iob.(s'.s_subject) && p1 in gob.(s'.s_subject) && p1 in grev.(s'.s_subject).(s'.s_has_member) && p1 in irev.(s'.s_subject) Case 4: p1 !in iob.(s'.s_subject) && p1 in gob.(s'.s_subject).(s'.s_has_member) && p1 in grev.(s'.s_subject).(s'.s_has_member) && p1 in irev.(s'.s_subject)

III. Assignments in state s for principal p2	IV. Assignments in state s' for principal p2
Case 1: p2 !in iob.(s.s_subject) && p2 in gob.(s.s_subject) && Case 2: p2 !in iob.(s.s_subject) && p2 in gob.(s.s_subject).(s.s_has_member)	Case 1: p2 in iob.(s'.s_subject) && p2 in gob.(s'.s_subject) && Case 2: p2 in iob.(s'.s_subject) && p2 in gob.(s'.s_subject).(s'.s_has_member)

Table 8.1: Possible assignments for principals and objects before and after delegation of an individual obligation instance under a review obligation, as defined in function 8.1.

8.2.5.2 Reviewing delegated general obligations

Considering the delegation of general obligations, we recall the discussion in section 7.4.3.2. There we distinguished between delegating a general obligation with and without any current instances for the delegating principal, as well as whether a principal delegates an obligation on the basis of a direct or role-based assignment.

With respect to the first case, we now have to decide what to do with any existing instances. The obvious solution is to make every such instance subject to review. This would correspond to a stepwise delegation for all these existing instances as defined in 7.4.3.2.

However, the subsequent question is what should happen to any new instances after the delegation. Should they also be made subject to review by the principal who delegated the general obligation? We believe that this should be the case as the delegating principal must still be held to account for the effects of his decision to delegate a general obligation, not only with respect to existing obligations, but also any future obligations that may arise on the basis of this delegation of a general obligation. The concept of supervision introduced in the following section 8.3 addresses this issue.

Considering the possible assignments of the principal after delegating a general obligation, there are no further new issues that need to be discussed. The main topic of concern was about the delegating principal losing the general obligation he delegates. In case the general obligation was assigned to him directly, he may be de-assigned from that obligation without any further side-effects on other principals. However, as discussed in sections 7.4.2 and 7.4.3.2, in the case of being assigned to the obligation over a role, the de-assignment from that role would have undesired side-effects. This de-assignment is not modelled here, since we believe that this issue requires more detailed research and is outside the scope of delegation in this context.

The delegation of general obligations under review controls does not present major technical challenges and can also be described in terms of the introduced function 8.1. However, it is well worth to be aware of the organisational context the delegation of general obligations must be placed. In fact, this form of delegation represents one of the main principles of (an) organisation: The distribution of work. This raises concerns with respect to more general supervision structures that emerge and will be discussed in the following section 8.3.

8.2.6 Exploring delegation and review controls

A review is a specific type of obligation, resulting out of the delegation of an obligation. In the context of this definition two interesting delegation scenarios may be observed:

1. The delegation of a delegated obligation.
2. The delegation of the review for a delegated obligation.

Technically, both can be handled by the same delegation function 8.1, since a review is an obligation. However, from the organisational perspective a more detailed discussion is required and will be provided in the following two sections.

Unfortunately, Alloy's labelling mechanism is not intuitive and in most cases the variables of the specification are just the reverse of the labels in the generated models. We take the liberty to change either where this has no further side-effects apart from avoiding confusion.

8.2.6.1 Delegating a delegated obligation

In the case of delegating a delegated obligation we instructed the analyser to look for models which obey the following sequence of operations 8.1. This has been documented in appendix B.7.

This sequence begins with a some assumptions about the order of the defined states. This is followed by a set of initialising constraints, which will have to be adjusted depending on the context of analysis. For example, in this context we define that there have been no previous delegations, and thus there are no review obligations in the first state of the sequence. Additionally, the delegating principal `p1` should hold the obligation `obl` he intends to delegate.

State Sequence 8.1 *Sequence of delegating an obligation from principal* p3 *over* p2 *to* p1.

```
fun obligation_delegation_chain1 (){some disj s1,s2,s3 : State |
                                    some disj p1, p2, p3 : Principal |
                                    some obl : Obligation |
                                    some disj rev2, rev1 : Review |
//Some assumptions about our states
  State_Sequence.first = s1 &&
  s1 -> s2 + s2 -> s3 in State_Sequence.next &&
//Some initialisations - Must be adjusted depending on analysis
  no State_Sequence.first.s_target &&                //No delegations
  no State_Sequence.first.s_specifies &&             //so far and thus no
  no State_Sequence.first.s_reviewed_by &&           //specific evidence
  no State_Sequence.first.s_creates &&               //or any reviews
  no State_Sequence.first.s_defines & Review->Action &&
  no Review.((State_Sequence.first).s_subject) &&
  (obl -> p1) in (State_Sequence.first).s_subject && //p1 has obl
//The actual sequence of delegations (1)
  -- Principal p3 delegates obligation obl to p2 -> rev2 is created
  general_delegation_with_review (s1, s2, p3, p2, obl, rev2) &&
  -- Principal p2 then delegates obligation obl to p1 -> rev1 is created
  general_delegation_with_review (s2, s3, p2, p1, obl, rev1)
}
```

Constraining a sequence of states to follow this function results in obligation `obl` being delegated from principal `p3` to `p2` in between states `s1` and `s2`. As a result the review obligation `rev2` is created for `p3`. Then `p2` delegates `obl` to `p1` in between states `s2` and `s3` as a result of which review `rev1` is created for `p2`. A model which satisfies these constraints can be seen in figure 8.3, which has been composed out of three models generated by the constraint analyser for each state.

We can see how obligation `obl` (represented by Object_3) is delegated twice. The review obligations `rev2` (Object_2) and `rev1` (Object_1) are created accordingly, first for `p3` (Principal_3) in state `s2` and then for `p2` (Principal_2) in state `s3`.

There are now some observations we can make with respect to the relation between the required evidence for the delegated obligation and the two generated review obligations.

In state `s2`, Evidence_3 is the evidence specified by the delegated obligation. As expected, the actions defined by the delegated obligation create this evidence. The evidence is then reviewed by the specific review action Action_3 of the generated review Object_2. We can then observe that after the second delegation step in state `s3`, the second review Object_1 defines the same review action as the first review. The reason for this is intuitive. Both reviews have the same obligation as their target and accordingly review the same evidence. This raises issues with respect to the discharge of the delegated obligation and the effects on the related review obligations which we can, however, not explore further at this stage.

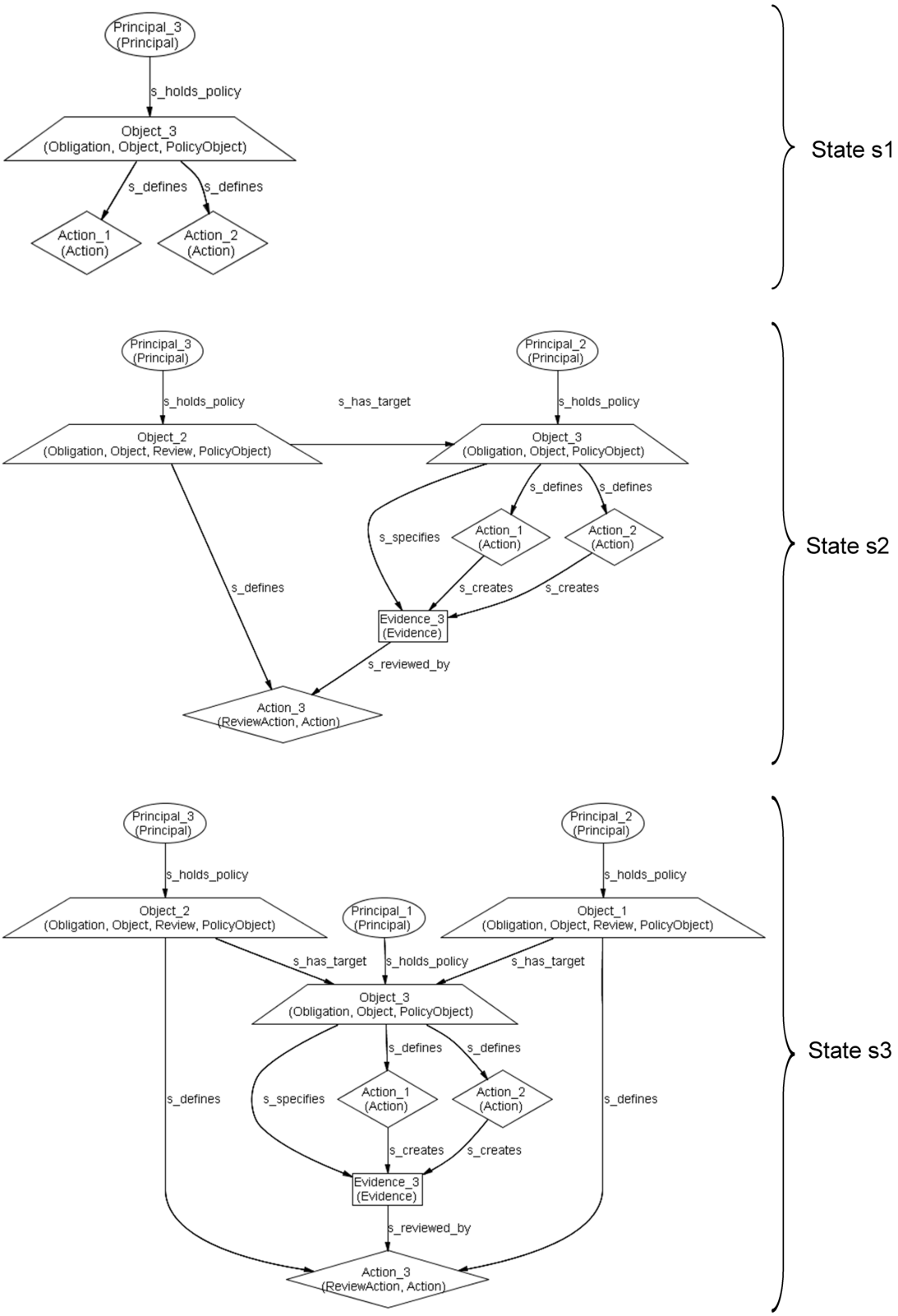

Figure 8.3: Delegation of the same obligation between three principals over three states.

8.2.6.2 Delegating the review for a delegated obligation

In the case of delegating the review for a delegated obligation we instructed the analyser to look for models which obey the following sequence of operations 8.2. This is part of appendix B.7. Again, this sequence begins with some assumptions about the order of the defined states and other context specific initialisations, followed by the actual delegation functions.

State Sequence 8.2 *Delegating the review of a delegated obligation.*

```
fun obligation_delegation_chain2 (){some s1, s2, s3: State |
                                    some disj p1, p2, p3 : Principal |
                                    some obl : Obligation |
                                    some disj rev2, rev1 : Review |
//Some assumptions about our states
  State_Sequence.first = s1 &&
  s1 -> s2 + s2 -> s3 in State_Sequence.next &&
//Some initialisations - Must be adjusted depending on analysis
  no State_Sequence.first.s_target &&                //No delegations
  no State_Sequence.first.s_specifies &&             //so far and thus no
  no State_Sequence.first.s_reviewed_by &&           //specific evidence
  no State_Sequence.first.s_creates &&               //or any reviews
  no State_Sequence.first.s_defines & Review->Action &&
  no Review.((State_Sequence.first).s_subject) &&
  (obl -> p1) in (State_Sequence.first).s_subject && //p1 has obl
 //The actual sequence of delegations (2)
   -- Principal p3 delegates obligation obl to p2
   delegate_obl_with_review (s1, s2, p3, p2, obl, rev2) &&
   -- Principal p3 then delegates review rev2 to p1
   delegate_obl_with_review (s2, s3, p3, p1, rev2, rev1)
}
```

Constraining a sequence of states to follow these functions results in obligation `obl` being delegated from principal `p3` to `p2` in between states `s1` and `s2`. As a result the review obligation `rev2` is created for `p3`. So far, this is identical to the delegation in the previous section 8.2.6.1. Then `p3` delegates his review `rev2` to `p1` in between state `s2` and `s3`, as a result of which review `rev1` is created for `p3`.

A model which satisfies these constraints can be seen in figure 8.4. Here we only show the model for the resulting state `s3`, since the models for states `s1` and `s2` are almost identical to those in figure 8.3. The following observations can be made with respect to this second delegation step and the resulting state `s3`. The initially delegated obligation `obl` (Object_3) is now the target of the review `rev2` (Object_2), which in turn is target of `rev1` (Object_1), as a result of the delegation of `rev2` from principal `p3` to `p1` in state `s2`.

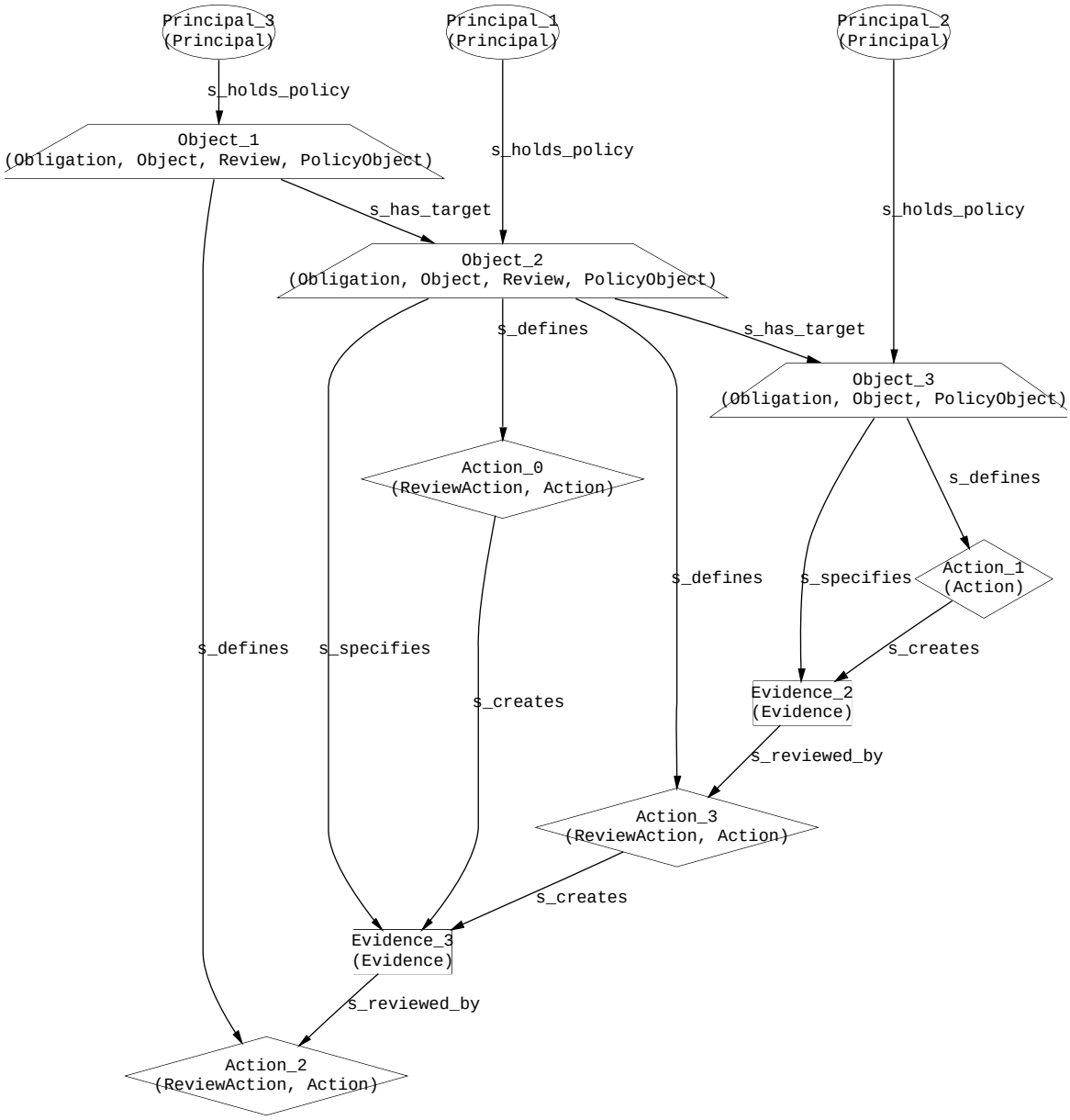

Figure 8.4: Delegating the review for a delegated obligation.

What is interesting to observe is that this delegated initial review `rev2` (Object_2) must now specify some evidence Evidence_3, which is reviewed by the defined review actions of review `rev1` (Object_1). It can also be seen that there is no further evidence object for this last review.

We explored several other delegation scenarios, by either altering the sequence of delegations, the delegating principals and delegated objects, or the pre- or post constraints. However, apart from the fact that principals may start delegating an obligation in a cyclic fashion, no other additional insights were gained that would be directly relevant to the context of this thesis.

8.3 Supervision

8.3.1 The concept of supervision and its organisational motivations

In the context of the formal organisation as discussed in section 2, obligations are delegated in order to facilitate the distribution of work. In the previous sections 7.4.3, 8.2.5.1 and 8.2.5.2 we discussed the delegation of obligations and the supporting concept of review, distinguishing between:

- the delegation of specific obligation instances and;
- the delegation of general obligations.

The first kind of delegation is what we considered as an *ad hoc* form of delegation, allowing individual principals to distribute obligations more efficiently. In this section we investigate the delegation of general obligations as a management activity with the aim of creating a more permanent form of organisational structure through the distribution of work. We believe that the concept of supervision is a control principle that supports this form of delegation.

The distribution of work through delegation mechanisms requires the observation and direction of the execution of any delegated task. We recall the observations we made in section 7.4.3.2 on the delegation of a general obligation and the consequences this may have. There we discussed that the principal who delegated a general obligation should still be held accountable for his delegation, not only with respect to any existing obligation instances that may have consequently been delegated, but also any possible future obligation instances that may arise for a principal on the basis of this delegation. We propose to capture this accountability for a delegated general obligation explicitly in the form of a supervision control.

While the previous discussions about delegation controls have left any form of organisational structure aside, this form of control now requires information provided by the positions principals occupy and their relationship to each other. In section 5.5, we defined positions as a more permanent representation of the obligations and authority of a principal in an organisation. We believe that a supervision control is expressed through a specific supervision relation between positions, that reflects their participation in an organisational command and control hierarchy.

We define supervision as the general obligation of a principal occupying a position to review the obligations of principals in supervised positions. This supervision relationship is the result of some prior delegation of general obligations.

Supervision is initially a relationship between positions. However, since positions are often uniquely occupied by a principal, we may sometimes also say that some principal supervises some other principal(s). If a position is occupied by several principals, then we must refer to other organisational criteria in order to resolve any ambiguities. This will be discussed in more detail in section 8.3.3. However, we first provide a more detailed motivating example for supervision as a control on delegation activities.

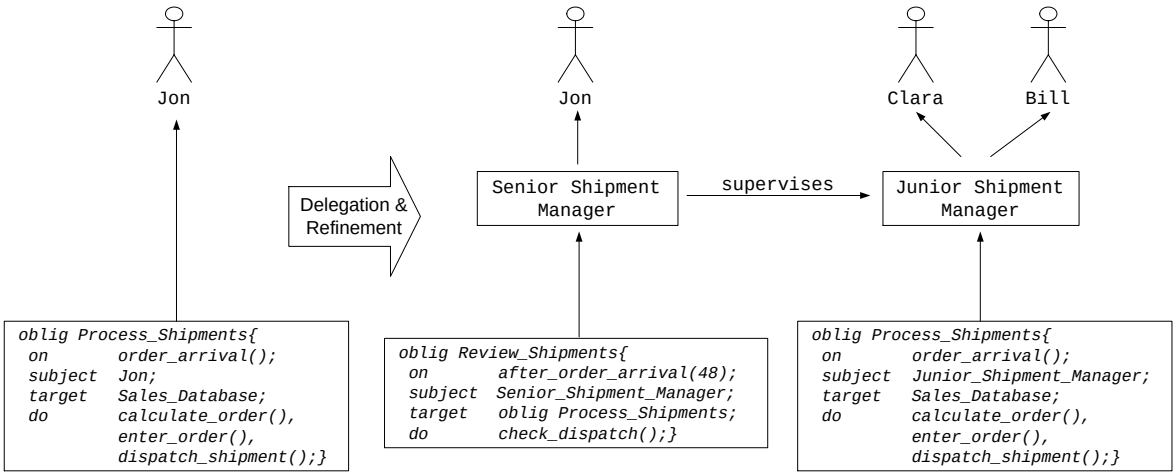

Figure 8.5: Delegation, refinement and supervision.

8.3.2 Supervising delegated obligations - A motivating example

The concept of supervision builds upon our earlier established concept of review. Some of the underlying organisational motivations for establishing this kind of control are thus similar to those for a review we have discussed in section 8.2.

The supervision relation between positions has little meaning by itself, unless there are some supporting review obligations. We illustrate this in the following example, supported by figure 8.5.

We consider a company in which a principal Jon processes outgoing shipments. The company grows and with it the amount of shipments. Soon, Jon is not able to handle this task anymore. Two new employees Clara and Bill are hired. Jon now delegates his obligation to process shipments to these two new employees. More precisely, positions are created to handle the growth of the organisation, and Jon in his new position as a `Senior_Shipment_Manager` delegates the general obligation to process shipments to the position `Junior_Shipment_Manager` occupied by the two new employees. With the continuing expansion of the company there will be further delegations and refinements of such obligations, for example, there may be shipment manager positions for regions or key account customers.

The `Senior_Shipment_Manager` position supervises the `Junior_Shipment_Manager` position. This means that through his position, Jon has an obligation to review that Bill and Clara process shipments correctly. In this case Jon might have to review the dispatch of every shipment 48 hours after the initial order.

We believe that this review on the basis of a supervision relation shows some further characteristics. To begin with, a supervisor like Jon will not necessarily review every shipment because this is not economic. Instead, a selective review will usually be performed. For example, Jon only reviews shipments with a certain value; shipments to a certain key customer; or shipments that show some other kind of exception. Secondly, with the delegation of the general shipment obligation and possible assignment to a new position, Jon may have to give up this obligation and the required authority. This is, however, context-dependent. Thirdly,

it depends very much on the maturity of the organisation and the delegation process in how much detail the review obligations for a supervision are defined. There may be situations where it is left at the discretion of the principal in a supervising position to decide what the appropriate means of review is. On the other hand the criteria for how to perform a review may be very strict. Fourthly, the review that needs to be performed by Jon may define the same review actions that Bill will have to perform if he decides to delegate an individual shipment obligation to Clara.

8.3.3 Expressing supervision controls in the context of the CP model

Having clarified our understanding of supervision, we now explore how this concept may be supported more formally in Alloy. We defined supervision as the general obligation of a principal occupying a position to review the obligations of principals in supervised positions. This supervision relationship is the result of some prior delegation of general obligations. The `supervises` relationship expresses this supervision. This relationship alone is, however, not enough to satisfy our definition of supervision as the general obligation to review.

8.3.3.1 Constraints on the supervision relation

We believe that there must be an explicit set of constraints which, in combination with the information delivered by the supervision relationship, supports this definition. We explore some of these in the following and appendix B.4.

To begin with we may specify that if a position `pos2` is supervised by a `pos1`, then every general obligation `obl`, that has `pos2` as a subject, must be target of some general review obligation `rev` which has `pos1` as its subject. The following Alloy constraint 8.7 supports this.

Alloy Fact 8.7 *If a position* `pos2` *is supervised by a position* `pos1`*, then all obligations* `obl` *which have* `pos2` *as a subject, must be the target of some review obligation* `rev` *which has* `pos1` *as its subject.*

```
fact {all disj pos1, pos2: Position | all o: Obligation | all s: State |
  pos2 in pos1.(s.s_supervises) &&
  pos2 in o.(s.s_subject) =>
     some rev : Review | rev -> o in (s.s_target) &&
                         pos1 in rev.(s.s_subject)}
```

This interpretation of supervision may, however, be too strict since the use of the universal quantifier demands that every obligation of a supervised position must be reviewed. There may be coercive organisational forms that may require this, but nowadays organisations usually do not demand such a strict form of supervision for economic reasons [Johnson and Gill, 1993, Mullins, 1999].

Another point considers whether supervision should be restricted to supervised positions and any assigned obligations, or to other roles as well. For example, in case of the previous

example it may be desirable to demand that not only obligations with pos2 as a subject are reviewed, but also the obligations that principals which are members of pos2 have through their roles. This can be expressed in the following fact 8.8 and our later case study will show a possible demand for such a constraint in section 10.5.2.

Alloy Fact 8.8 *If a position* pos2 *is supervised by a position* pos1 *then all obligations* obl *which have* pos2*, or a role occupied by a principal in* pos2*, as their subject, must be the target of a review obligation* rev *which has* pos1 *as its subject.*

```
fact {all disj pos1, pos2: Position | all o: Obligation | all s: State |
  pos2 in pos1.(s.s_supervises) &&
  (pos2 in o.(s.s_subject) ||
  ((s.s_has_member).(pos2.(s.s_has_member)) & Role) in o.(s.s_subject)) =>
     some rev : Review | rev -> o in (s.s_target) &&
                         pos1 in rev.(s.s_subject)}
```

8.3.3.2 Providing evidence in supervision relationships

The concept of evidence allows us to treat the obligations of a supervisor in a similar manner as other review obligations. If the subordinate provides the evidence specified in his obligations, the supervisor can review it in order to generate the evidence that he discharged his supervision obligation. This evidence might in turn be reviewed by his next superior etc., up the supervision management chain. The inevitable question of the end of such a supervision chain is dependent on the context and structure of the organisation, but it is clear that there must always be an entity (e.g. Board of Directors) that does not have to generate evidence for any further review. This is similar to the discussions on the source of authority provided in [Moffett and Sloman, 1988].

Review on the basis of a supervision relationship occurs frequently. We thus believe that for economic reasons the review of evidence on the basis of a supervision relation should incur less overheads than the review of evidence that needs to be provided as a result of an *ad hoc* delegation.

8.3.3.3 Resolving ambiguities in the supervision hierarchy

The previous discussions raised questions with respect to the form such a supervision hierarchy should take. As a minimum requirement we enforce that there may be no cycles in the supervises relation. This is straightforward and may be defined as in constraint 5.7, checking for the absence of cycles in role hierarchies. A further requirement may be that the supervision hierarchy follows a tree structure. We do not enforce this here, since we believe that there are organisational forms that require a position to be supervised by more than one other position. One example is that of a matrix organisation [Knight, 1977, Mullins, 1999]. This would mean that only some of the obligations of a supervised positions are reviewed by one of the several supervising positions. In addition, a position is likely to be occupied by several principals.

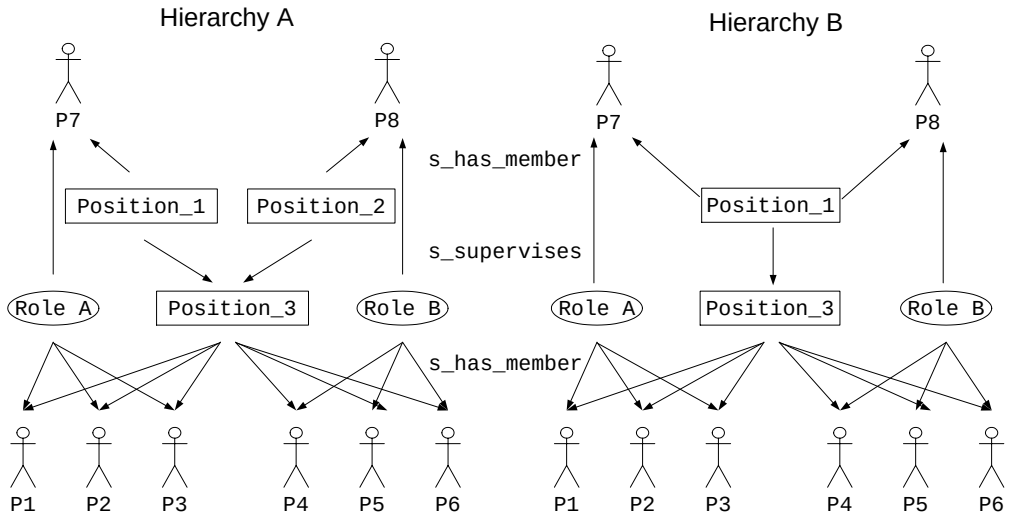

Figure 8.6: Example for supervision hierarchies.

These missing constraints on the supervision hierarchy and cardinality of supervision membership may, however, lead to ambiguities and possible conflicts. In the first case of a position being supervised by two or more other positions, it may not be clear which of the obligations of the supervised position are reviewed by which obligation of the several possible supervising positions. Our supervision constraint defined in fact 8.7 would not be sufficient to clarify this. Likewise, if two or more principals occupy a supervising or supervised position, it is not initially clear which obligation is reviewed by which other obligation, and which principal finally performs the review of any instances. To summarise, we do not enforce any cardinality constraints on the position membership, and some additional organisational attributes must be considered to resolve any possible conflicts. We use the two examples of supervision hierarchies given in figure 8.6 to support our discussion.

In Hierarchy A, the `Position_3` is supervised by `Position_1` and `Position_2`. These two positions have principal `P7` and `P8` as their member respectively. Principals `P1` to `P6` are members of this supervised position `Position_3`. However, this information alone would not be sufficient to determine what obligations of `Position_3` are reviewed by what review obligations of `Position_2` and `Position_1`. One possibility to resolve this is to require that a general obligation of `Position_3` is reviewed by a general obligation of a position which supervises `Position_3`, where a principal occupying that supervising position is member of the same role as any member of the supervised `Position_3`. Thus the obligation instances of principals `P1` to `P3` are effectively supervised by `P7` and the obligation instances of principals `P4` to `P6` by `P8`.

In Hierarchy B, the `Position_3` is supervised by `Position_1`. This position has principals `P7` and `P8` as its member. Again, this information on its own is not sufficient to determine which principal ultimately performs the review of any obligation instance. This may be resolved as in the previous example, and we require that any obligation instances a principal member of the supervised position `Position_3` has, must be reviewed by a principal in a supervising position and the same role. Of course, if a principal now occupies both roles we would have to consider some different approach.

Both the above cases are only two possible examples, and solutions to resolve any ambiguities will have to be specified according to context relevant criteria like those described in section 5.5. However, it is clear that without considering additional information, here in the form of roles, supervision conflicts may arise. Of course it would be possible to explicitly enumerate any general or instance specific review relationships. For example, in Hierarchy A we may explicitly specify that the one half of the possible obligations of `Position_3` are reviewed by corresponding review obligations of `Position_1`, while the other half are reviewed by corresponding review obligations of `Position_2`. Likewise, we may specify in the context of Hierarchy B that any obligation instances of principals `P1` to `P3` are reviewed by review instances of principal `P7`, while obligation instances of principals `P4` to `P7` are reviewed by review instances of principal `P8`. These may, however, be too heavy-handed approaches, questioning the benefits of structuring mechanisms like roles and positions. The above examples are not specified in a more formal way here. This will be done in the context of our later case study in section 10.5.2, where similar ambiguities in the supervision relation arise and are resolved like in the two example hierarchies in figure 8.6.

To summarise, supervision is a relationship between positions. Principals occupying positions take on the review obligations that arise out of supervision relationships. There may be ambiguities in supervision relationships and thus between review and reviewed obligations and the principals subject to those obligations. Application-specific constraints may have to be defined to resolve these.

8.4 Chapter summary and conclusion

In the course of this chapter we have described how control may be retained over delegated obligations through the novel concepts of review and supervision. This included:

- A discussion on the general motivation for achieving control over delegated obligations in section 8.2.1;
- An initial example and formal description of the review concept as the obligation of a principal to examine the results of a previously delegated obligation in sections 8.2.2-8.2.5;
- An exploration of delegation and review controls, focusing on the delegation of delegated obligations, and the delegation of the review of a delegated obligation in section 8.2.6.
- A discussion on the general motivation for establishing supervision hierarchies between positions in section 8.3.1;
- An initial example and formal description of the supervision concept as the general obligation of a principal to review the obligations of principals in subordinate positions in sections 8.3.2 and 8.3.3.

More specifically, the established concepts of review and supervision are directly based on our distinction between:

1. The delegation of specific obligation instances as described in sections 7.4.3.1 and 8.2.5.1;

2. The delegation of general obligations as described in sections 7.4.3.2 and 8.2.5.2.

This first kind of delegation is what we considered as an *ad hoc* form of delegation, allowing individual principals to distribute obligation instances more efficiently. A review policy is created as the result of delegating such an obligation. This review is a specific type of obligation with the delegated obligation as its target. The concept of evidence, as initially described in section 5.4, determines what a later discharge of such a delegated obligation has to produce to convince the delegator that the obligation has indeed been performed. Evidence serves as an abstraction for what has to be eventually produced but not that it has been produced. However, we believe that the concept of review has some limitations. It must have been specified in advance how a review is performed. As such delegation cannot occur randomly. This is, however, expensive and creates overheads. Often it is also not possible to predict the situations that require delegation.

The second kind of delegation is a management activity with the aim of creating a more permanent form of organisational structure through the distribution of work. The distribution of work through delegation mechanisms requires to observe and direct the execution of any delegated task. Supervision is defined as the general obligation of a principal occupying a position to review the obligations of principals in supervised positions, assuming that at some prior stage these reviewed obligations have been delegated between those positions. This concept also has limitations. Often it is not economic to demand the review of all the activities of all supervised positions. We have discussed this but concluded that a more advanced form of selective review is outside this scope. Since we do not enforce any specific constraints on the form of supervision hierarchies, any ambiguities have to be resolved depending on the context.

The analysis and exploration of review and supervision controls also indicated the limitations of a declarative specification approach on the basis of Alloy. We will expand on this in the context of our conclusion in section 11.4.3.1.

We have established a conceptual model of control principles in chapter 5, together with a set of separation, delegation and revocation controls in chapters 6 and 7. We described how control may be achieved over the delegation of specific and general obligations through the concepts of review and supervision in this chapter and can now validate, compare and apply our results in the context of a case study described in the following two chapters 9 and 10.

Chapter 9

Case Study - Part I

9.1 Introduction

Little has been done to identify and describe existing role-based access control systems within large organisations and the applied control principles. This chapter represents the first part of our case study, and describes and evaluates the access control system and controls of a major European Bank as initially published in [Schaad et al., 2001].

We have restructured this early description and added some new material. This chapter is now split into two main parts. The first part considers the general working of the access control system. In particular it provides a discussion on:

- The basic structure and conceptual design of the system in sections 9.2 to 9.4;
- The system-specific definition of roles consisting of functions and official positions, and evaluation of the number of roles in comparison to the system user population in section 9.5;
- The perceived weaknesses of the Bank's system in section 9.6;
- How the system compares to the RBAC96 models, focusing on role inheritance with respect to function and position hierarchies in section 9.7;

We will then discuss some of the observed controls in the administration of the system in section 9.8. This is followed by the description of controls in the context of a branch of the bank in section 9.9, specifically with respect to a credit application process in section 9.9.2. This process shows some interesting properties, as it mixes human controls and automated controls implemented in the logic of the supporting applications.

9.2 The FUB access control system of Dresdner Bank

The particular case study that we present in this chapter was carried out in co-operation with Dresdner Bank, a major European bank with 50,659 employees and 1,459 branches worldwide [Schaad et al., 2001]. The main business, with about 6.5 million private customers and 1,000 branches, is situated in Germany, the rest in Europe and overseas. Recently, the Bank merged with the Allianz Insurance, forming one of Europe's biggest all-finance corporations.

The Bank uses a variety of different computing applications to support its business, many of which have their origin in the mainframe world, but also more recently deployed client-server based systems. This strong degree of decentralisation through branches provides a major challenge to the general administration of the supporting IT infrastructure and access control in particular.

Before 1990, most of the host-based applications used the local access control file administered at the relevant host for the determination of access rights. For each employee, the access rights had to be administered manually at the individual application level. This caused enormous overheads as a result of the growing number of employees working with these applications. Additionally, the maintenance of several application-level security files for each user was an error-prone process and could not be justified within the general security policy framework. In order to improve this situation a system, called the FUB (= **Fu**nktionale **B**erechtigung), was developed by the Bank as no suitable commercial solutions were available at that time. In this system, access rights are given to the individual user according to a combination of his job function and official position within the organisation. The direct assignment of access rights to the individual user at the application level was discontinued.

The FUB is an example of an enterprise-wide role-based access control system. Applications cannot make access control decisions on their own. They grant access to data on the basis of a centrally provided security profile. Over 60 applications within the bank make use of this system. These cover a wide area of organisational functions such as private customer instruments at the local branch, credit data checks, automated signature approval or the administration of Unix accounts. An application is launched by a user who first identifies and authenticates himself to it. Initially the application has no knowledge of any relevant access permissions the user might possess. It queries the FUB about the security profile of the current user in order to obtain this information.

Since 1990, and thus much earlier than most of the published role-based access control discussion, the FUB system has hosted roles and delivered access rights for usage within other applications that run under various environments such as UNIX derivatives (SINIX/AIX) or WINDOWS NT/2000. On average 42,000 security profiles are distributed by the FUB per day. The time needed by the system to determine one individual security profile is approximately 85 ms. The system's availability rate is 99% per year, achieved through a physical and logical decentralisation of the involved hard- and software.

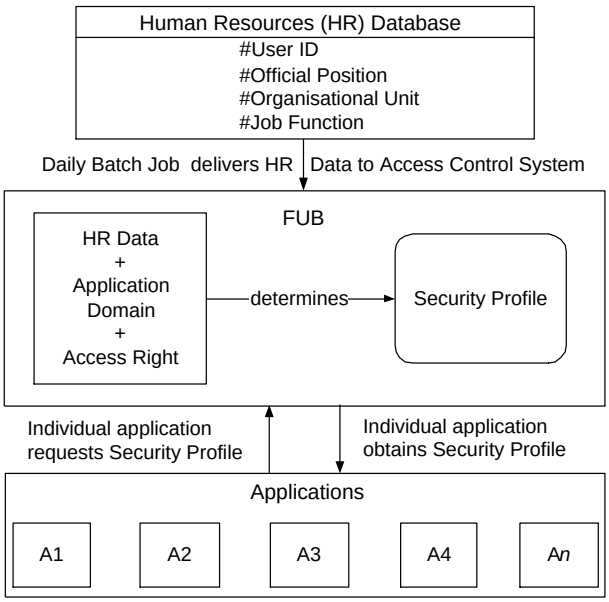

Figure 9.1: The basic FUB structure.

9.3 The basic structure of the system

The FUB is a role-based access control system. Roles are defined as a combination of the

- official position and
- job function of a user

Typical official positions could be that of the ordinary Clerk, Group Manager or Regional Manager. Functions represent the user's daily duties such as being a financial analyst, share technician or internal software engineer. Additionally, the organisational unit to which a user belongs is used as an access control criterion for certain applications. All these data are defined and maintained in the human resources database. A batch job runs between the human resources system and the FUB every night. Thus, the access control system has a very accurate image of the current organisational status and existing roles. This is in fact common practice in other commercial access control systems, e.g. [Awischus, 1997].

Within the FUB the data delivered by the human resources database are linked to applications. Together with application specific access rights, these are the basis for the security profile of a user. When a user starts an application the FUB delivers this security profile that tells the application which individual access rights the user possesses. Figure 9.1 shows the basic architecture of the system and its interfaces to the human resources database and individual applications.

Employees belong to an organisational unit. Ideally each employee is only assigned to one role. However, in special circumstances an employee can be given up to four roles (e.g. in the case of illness of a colleague). Several applications can be accessed through a role. Each application has a set of access rights assigned to it. A simplified underlying data model is shown in Figure 9.2.

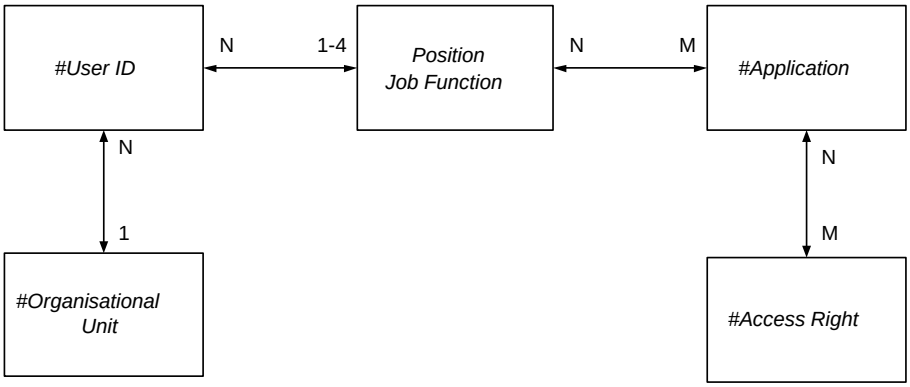

Figure 9.2: The basic FUB data model.

9.4 An application example

To make the above technical description more concrete we provide a scenario that reflects the daily business at a high-street branch of Dresdner Bank. Figure 9.3 supports this example in the form of a sequence diagram.

An existing bank client wishes to discuss his personal savings situation with the branch's financial advisor. The advisor and the client go to a meeting room which contains a Personal Computer. The advisor identifies and authenticates himself to the machine using a smartcard and his password. He launches an application that allows him to enter the records of his client which are stored on a central server.

When the application is launched it issues a request to the FUB, querying which rights the advisor has within the application domain. The application request contains the personnel number, which was obtained during the identification and authentication process. Also the application identifier is submitted to obtain the relevant authorisation profile for the application. Once the FUB has used these data to deliver the security profile, the application knows which access rights are assigned to the role of the user and allows him to execute his access rights accordingly. In this particular case information about the relevant organisational unit to which the advisor belongs (here, the branch) will prohibit him from accessing account data outside his branch. His access rights are confined within the organisational domain of the branch, a simple example of least privilege.

One weakness of the current FUB system is, as we show in figure 9.2, that a user can be assigned to more than one role. This is required when a colleague becomes ill or is on holiday, but also in more permanent cases where a clerk works in a Branch A in the morning and a Branch B in the afternoon. In models like RBAC96 [Sandhu et al., 1996] a user must choose which role to activate for a session. However, the session concept does not exist in the FUB. So when a user logs onto a system he has all the access rights of all the roles to which he is assigned. This creates problems with respect to the principle of Least Privilege and possible separation controls. Careful administration and monitoring of user/role assignments is needed to prevent security violations and conflicts. The Bank is aware of this problem, however, in this case business requirements dominate over security considerations.

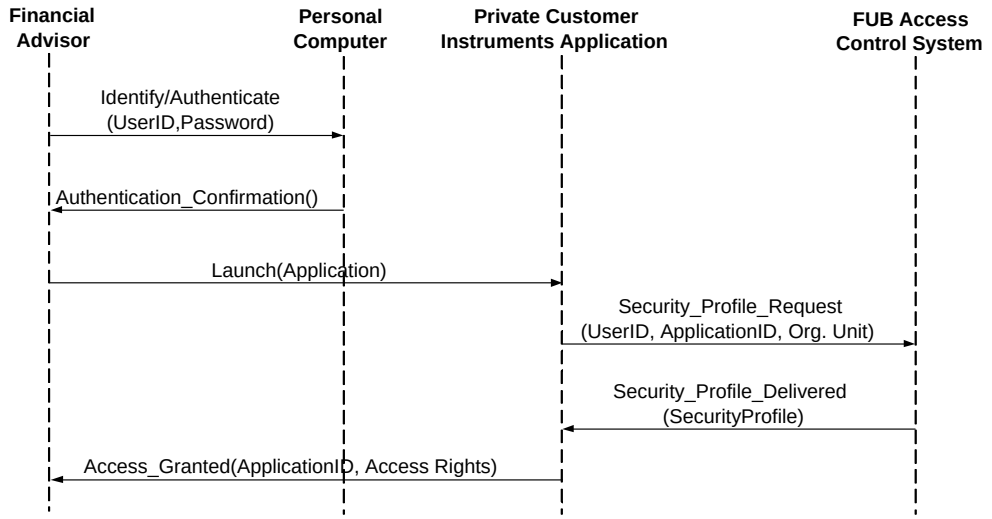

Figure 9.3: FUB authorisation example.

9.5 Roles in the FUB system

A role in the FUB system is defined using the official position within the organisational hierarchy and a description of the job function. These data are delivered by the human resources system. From now on we will refer to a role using the construct *function/Official Position*. We will use lower case for functions and capitalisation for positions. An example of this would be *financial analyst/Group Manager*, indicating that somebody has the function of being an analyst and holds the official position of a Group Manager. Theoretically, the total number of roles would be the product of every official position and every function. However, the actual number of roles is a subset of this, as certain possible roles such as *secretary/Member of the Board* do not occur in reality.

Within the Bank there are 65 official positions that can range from an ordinary Clerk in a branch, through the Branch Manager, to a Member of the Board. These participate in groups of partial orders expressing command and control structures. Hierarchies arise with further organisational indicators such as cost centers, departments or project groups.

These positions are combined with 368 different job functions provided by the human resources database. Although there would be a possible set of 23,920 roles, the number of roles that are currently in use is about 1300. As the access control system is constantly updated it is subject to changes occurring when functions and positions are created, and more importantly, deleted.

Each night human resources data about employees, their function, rank and organisational unit are transferred into the FUB. However, not every single employee can actually be seen as an active user of the FUB system (e.g. cleaning staff, catering, internal postal services etc.). Thus, there are only about 40,000 users in the FUB.

These figures match an oral estimate that was given at the ACM RBAC2000 Workshop, Berlin, suggesting that the number of roles in a role-based system is approximately 3-4% of the user population. With 40,000 FUB users and 1300 Roles, we obtain a role/user ratio of

approximately 3.2%, supporting this estimate. However, this distribution is not uniform. It would be interesting to analyse the role/user ratio according to the position hierarchy, but we do not have the information that is needed. Our later findings in the context of other organisations partially support these figures [Kern et al., 2002], but also show that such a ratio is only meaningful considering the specific definition of a role in the context of a given organisational structure. For example, we reported an insurance company with 17,000 users and 120 roles; a bank with 30,000 users (where a user has on average 10 roles) and 400 roles; and another bank with 31,000 users and 450 roles (where 1 role is assigned to 70 users on average). Organisations and their definitions and use of roles are too diverse for a common user/role ratio. However, within a specific application domain such a figure may be useful.

Another issue in the administration of the system is that it has also to provide access control services to users who cannot be considered as permanent staff. This group of users includes third party consultants, temporary staff and freelancers. They work for the Bank during projects with a length varying from several weeks to years. Many of them come for the duration of a project, leave, and often come back shortly after the project has finished, working for another, sometimes related project. Ideally, this group of users should always get a new account when starting work and their accounts should be deleted when leaving the project. However, this creates overheads. Thus, information about this group is not held in the Human Resources database, but is locally administered by the FUB staff. User accounts are created once and are usually kept (de-activated) when a consultant leaves. We have made similar observations in the context of access control administration within other commercial security management products [Kern et al., 2003].

The overheads occurring with the administration of this temporary staff should not be underestimated as this user group, containing hundreds of users at a time, is subject to constant changes. However, we did not take this issue into consideration when providing the above figures. The role/user ratio only applies to full-time staff.

9.6 System weaknesses and development goals

In the above description of the system we already noted certain weaknesses of the system. Dresdner Bank is aware of desirable improvements that are addressed in the current development of a new access control system. Apart from more general issues, such as interface design and business specific software engineering aspects, suggested improvements concern:

- Access right allocation
  In the current system access rights can only be allocated to the combination of function/Position and organisational unit. Further possibilities of allocating access rights, especially on a per user basis, do not exist. This would be a feature that violates the principle of separation of users from access rights by means of roles. However, in the case of certain access rights (e.g. set of access rights representing the power of attorney for an employee) it is desirable to assign these directly to the user.

- Grouping employees
  In the current system, employees can only be grouped according to the combination of function/Position and organisational unit. It is not possible to group employees according to other criteria and assign group-specific access rights. A grouping mechanism will provide ease of administration as only the group needs to be assigned with a role and not each individual.

- Grouping access rights
  The current system does not allow for the grouping of access rights which naturally belong together. An example is the grouping of access right 101 for *create account*, and 102 for *delete account* into a single group G1 for *account manipulation*. These grouped rights will provide an easier assignment of access rights to applications.

- Covering/Standing-in regulations
  In the current system regulations for the covering of one employee by another (e.g. because of holidays or illness) do not exist. It would be desirable to only partially delegate application-specific access rights. Another problem considers the ability of one person to stand in for two others at the same time. This might violate separation controls

- Competencies and Constraints
  The current system does not allow the specification of competencies and constraints in the security profile (e.g. authorisation to sign contracts up to Euro 100,000 only). There are, however, procedural controls in place that achieve a similar effect.

Some of the higher-level development goals consider the mapping of the access control system to the existing organisational structure, since the Bank has to face a continuous organisational change and needs to provide for a flexible support of its business strategies:

- Continuous organisational change
  Organisations of the size of Dresdner Bank are subject to a constant change process in their structural and functional organisation. This is due to a continuous orientation of the Bank's business to the market needs. Also unforeseen acquisitions or mergers can cause a major organisational change. The current access control system does not provide the flexibility to meet these changes without a major administrative effort. It would be interesting to observe the impacts of the recent merger of Dresdner Bank with the Allianz Insurance Group on the access control system. However, we do not have any information on this.

- Flexible support of business strategies
  Strategies in the private customer business require that an employee can be related to a certain organisational structure. This could be in the form of branches or cost centers but also more abstract structures such as enterprise-wide working groups, task forces or projects. The access control system must be able to reflect these structures.

9.7 The FUB system and the RBAC model

It is interesting to observe how far the Bank's access control system can be compared with other models of role-based access control. One candidate for comparison is the RBAC96 model, described in more detail in section 3.3.1. It is well-defined, easy to understand and most importantly, it can be configured to support various access control policies, according to the specific need. In addition, it forms the basis for a NIST standard, and we have recently seen and presented other case studies where real-world access control systems are compared to RBAC96 and its derived models [Marshall, 2002, Kern et al., 2003].

The two main issues we consider in the following two sections are the access right inheritance through a role hierarchy and the grouping of users.

9.7.1 Access right inheritance through a role hierarchy

The RBAC96 model has the concept of access right inheritance through a partially ordered role hierarchy, where superior roles inherit all the access rights of their inferiors. When looking at figure 9.2, we can see that there is no role inheritance structure of this kind in the FUB. The Bank's access control system does not offer any inheritance features. Should this be changed and what are the impacts of such a change?

In the RBAC96 model "Role" is an atomic concept, defined as a named job function within the organization [Sandhu et al., 1996], compare section 3.3.1. Its natural counterpart in the FUB system is the FUB Role, which consists of *both* function and position; see examples in table 9.1. The partial ordering of the FUB role can therefore be defined upon either or both of function and position. We discuss the two possible orderings below.

1. Hierarchy of Official Positions
   There are strict partial orders in the organisation for official positions (denoted by the $>$ symbol). This has little meaning by itself, but when combined with function it is often reflected by a real hierarchy of actual power. For example, the *financial analyst/Group Manager* role (Role B) has more power than the *financial analyst/Clerk* role (Role A). Table 9.2 shows that B has as many or more access rights in the Money Market Instruments, Derivatives Trading and Interest Instrument applications and access rights to Private Customer Instruments. On the other hand there is, as we might expect, no similar hierarchical relationship of power between *office banking/Group Manager* and *financial analyst/Clerk* because they work in different functional areas. Therefore we could define a role hierarchy in which one role is superior to another if its position is superior and their functions are identical. Using the "." symbol as a selector this may be expressed as:

   $$Role(x) > Role(y) \Leftrightarrow Role(x).Position > Role(y).Position \wedge Role(x).Function = Role(y).Function$$

   In the example above, given that *Group Manager* $>$ *Clerk*, we could economise on access right definition and table 9.2 could be rewritten as table 9.3.

Role	Function	Official Position
A	financial analyst	Clerk
B	financial analyst	Group Manager
C	financial analyst	Head of Division
...	...	...
X	share technician	Clerk
Y	support e-commerce	Junior
Z	office banking	Head of Division

Table 9.1: FUB roles consist of functions and Positions.

Role	Application	Access Right
A	Money Market Instruments	1,2,3,4
	Derivatives Trading	1,2,3,7,10,12
	Interest Instruments	1,4,8,12,14,16
B	Money Market Instruments	1,2,3,4,7
	Derivatives Trading	1,2,3,7,10,12,14
	Interest Instruments	1,4,8,12,14,16
	Private Customer Instruments	1,2,4,7
...	...	...

Table 9.2: Roles, applications and access rights.

Role	Application	Access Right
A	Money Market Instruments	1,2,3,4
	Derivatives Trading	1,2,3,7,10,12
	Interest Instruments	1,4,8,12,14,16
B	Money Market Instruments	7
	Derivatives Trading	14
	Private Customer Instruments	1,2,4,7
...	...	...

Table 9.3: Rewritten table 9.2, assuming that role B inherits access rights from role A.

2. Hierarchy of function

   There could also be a partial ordering of functions. An example for the two functions *inspector* and *finance accountant* would be: *inspector* > *finance accountant*. This is an "is a" hierarchy, meaning that in order to carry out the functions of an inspector, one has to be a finance accountant and needs all the access rights of one. It is always true that an inspector is a finance accountant, because otherwise he would not be competent to perform the function. Regardless of official position it might therefore be a good idea to generate a role hierarchy based on functions, so that superior functions inherit access rights from their inferiors. This alternative role hierarchy would be defined as having one role superior to another if its function is superior, without regard to position. This may be expressed more formally as:

   $Role(x) > Role(y) \Leftrightarrow Role(x).Function > Role(y).Function$

Discussing the pros and cons of using inheritance structures in the Bank as outlined above, there are several issues to consider:

- Choice of role hierarchy
  For practical reasons the Bank should only make one choice for the role hierarchy, although the RBAC96 model allows multiple hierarchies. We have noted two candidates above: A position hierarchy with matching functions and a function hierarchy. It would be necessary to study the organisation's access control structure in detail, paying particular attention to the gained simplification in each case, and to the other issues which we discuss immediately below.

- Compatibility of role hierarchy with other organisational needs
  It has been recognised in [Sandhu et al., 1996] that it is not possible to simply pick up an existing organisational hierarchy and import it into a RBAC system as conflicts might arise. We have given an example above in which the use of the position hierarchy with matching function for our role hierarchy would simplify the access rights table for a *Group Manager*. However, we do not know whether this would also be appropriate for *Head of Division*. Perhaps access right inheritance would give him more rights than he needs to carry out his job, thus violating the principle of Least Privilege. In order to deal with problems of this kind, it may be necessary to create a hierarchy which is compatible with, but not identical to the original hierarchy, using the concept of private roles as suggested in the RBAC96 model.

- Fine grained access control
  If we institute access right inheritance, and subsequently it turns out that there is an access right which is needed for the *financial analyst/Clerk* but not for the *financial analyst/Group Manager* role, then we need to deconstruct, wholly or partially, this portion of the role hierarchy. It is not possible to solve the problem by giving a negative right to the superior position, because negative rights are inherited downwards.

- Separation controls
  Access right inheritance through role inheritance may conflict with separation controls. It was pointed out in [Moffett, 1998] how this can happen. We extended this later in the context of a software company [Schaad and Moffett, 2001, Schaad, 2001]. It is not desirable to let a project manager inherit from a senior programmer the right to read the repository. The project manager does not necessarily have the technical knowledge and might consider shipping bad code if he had access to it before it is released by the senior programmer. We can be confident in the example of table 9.2 that the Bank has avoided any separation conflicts, because it has strict internal control procedures. If access right inheritance were to be used, the Bank would have to re-examine the rights table in detail, in order to ensure that no conflicts of this kind were introduced. This is likely to restrict severely the extent to which inheritance can be integrated.

Considering the advantages and disadvantages we cannot reach any conclusion about these particular cases, as we have neither the knowledge nor the authority to do so. In favour of

the introduction of a role hierarchy for the FUB is the economy of access rights that this would bring. We have however pointed out a number of factors that would reduce these advantages, and they would need to be examined in detail before reaching a decision.

9.7.2 Grouping

As mentioned in section 9.6, the Bank considers introducing a mechanism for the grouping of employees. A grouping mechanism will provide ease of administration as only the group needs to be assigned to a role and not each individual. The RBAC96 model does not allow for the assignment of groups to roles. It explicitly states that the assignment of users to roles is a relationship between (individual) users and roles. However, other approaches to role-based access control like [Nyanchama and Osborn, 1999] have recognised the value of using the group as a basis for the definition of roles. The concept of domains as summarised in [Damianou, 2002] can also be used to provide a mechanism for grouping users and the assignment of authorisation policies to these groups.

9.8 Separation controls in the FUB administration

We have provided an initial description and discussion on selected aspects of the FUB access control system in the previous section 9.2. We will now continue with a discussion and description of the separation controls that were identified in the administration of the access control system.

The actual definition of roles, and the assignment of users and access rights to the role, is performed in different departments within the organisation, listed in the following and summarised in figure 9.4. This is a kind of operational separation control achieved through the delegation of administrative authority. The assignment of roles, users, access rights and applications can only be performed in the following departments:

- Human Resources Department
  Role Definition and *User/Role Assignment.*
- Application Administration
  Access Right Definition and *Application/Access Right Assignment.*
- FUB Administration
  Role/Application Assignment.

Users and roles are initially created in the human resources department. A user is given a unique number that serves as his user identification number. Also roles are defined there, maintaining and combining functions and official positions. Users are then assigned to roles here. This makes sense and reflects the close relation between role-based access control and other areas such as human resources and organisation.

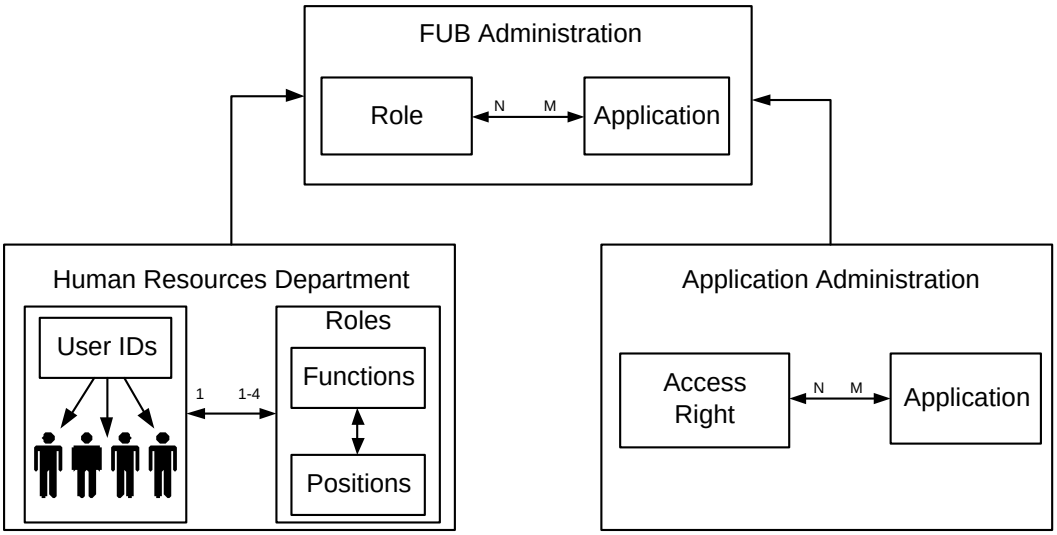

Figure 9.4: Decentralised access control administration

The assignment of access rights to an application is done for each individual application through the application administrator. This takes the form of assigning a set of numbers, representing specific access rights, to an application identifier (e.g. *PKI = Private Customer Instruments*). The semantics of these numbers are only known to the application administrator (e.g. *203 = Create new account in the PKI application*). The benefit of this is that it remains unknown to the later FUB administrator responsible for the role/application assignment what access right a specific number represents within the application domain. In addition, the application administration process is subject to the principle of Dual Control. One person can alter data, whereas a second person has to confirm these data.

The FUB administrator can only perform the role/application assignment. He does this in consultation with the application administrators who will give him information about which role should have which access rights in the context of the application. It can be seen in figure 9.5 how the FUB administrators enters the blinded access rights to the applications, but does not know what these access rights mean in the application context.

This shows the strong level of separation in the entire administration process. In addition, the generation of evidence by logging any administrative actions is obligatory and has to follow the Bank's policy on logging and evidence generation.

However, some of these control principles are clearly broken when it comes to the administration of freelancers or consultants as outlined in section 9.5. This group is locally administered by the FUB staff. No information about this group comes from the HR department. The consultants are given an identifier by the FUB staff and applications and access rights are directly assigned to the identifier when the consultant works on a project.

The task of revoking role and access right assignments is done very elegantly in the Bank's system. When a user leaves the company or moves within the organisation, then the human resources department will ensure that any role assignments cease to exist for this user or are changed accordingly. Since this information is updated on a 24 hour basis, the access control system has a very accurate image of the user population. However, this tight cou-

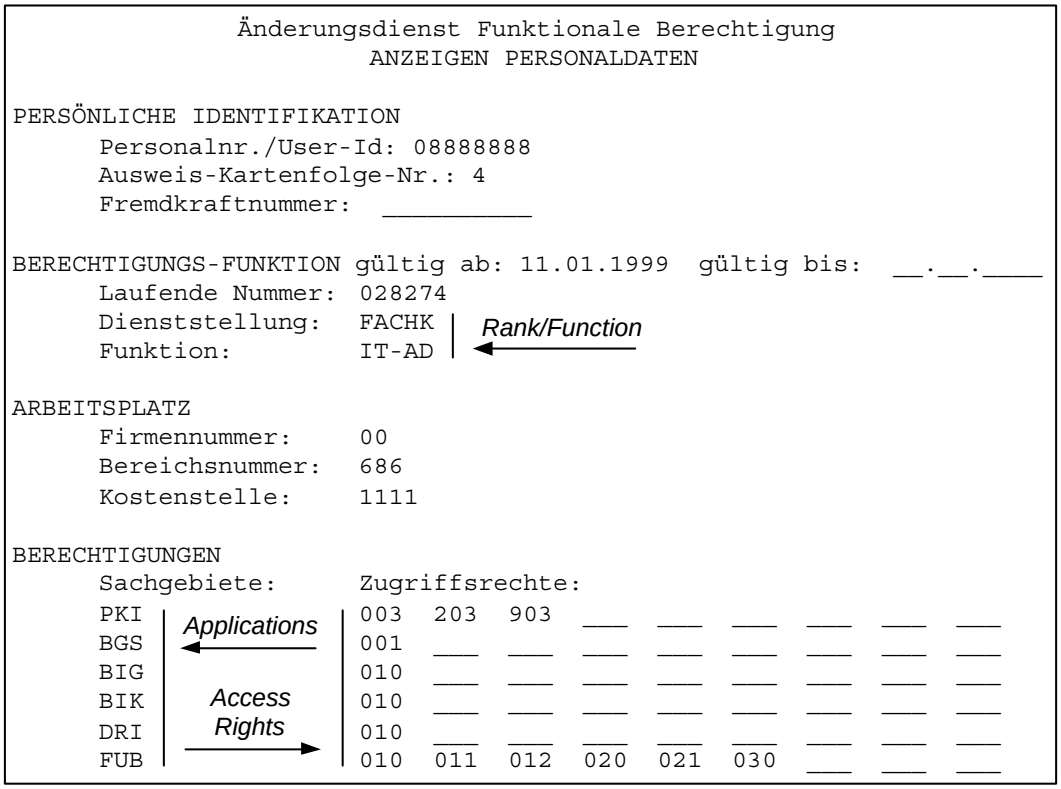

Figure 9.5: FUB screenshot.

pling bears certain dangers, because deleting a user from the human resources database will subsequently delete all his work when he ceases to exist in the system. Additional controls and organisational measures exist to prevent valuable work from being lost.

9.9 Control principles in the use of the access control system

Having described the administration of the access control system and the involved separation controls, this section provides a description of some of the applied control principles in the context of a local branch of the bank. A credit application process is used as an illustrative example. We have informally spoken with staff at a particular branch but did not apply any specific interview method or record any of these conversations. Additional insights into the application domain of a bank branch were obtained from our observations made in [Kern et al., 2002], as well as those reported in [Schael and Zeller, 1993, Agostini et al., 1994, Ortalo, 1998, Agostini and De Michelis, 2000, Anderson, 2001, Perwaiz and Sommerville, 2001].

9.9.1 Background information

The particular branch of the bank is a medium-sized branch with respect to its annual turnover and amount of customers. This means that the branch provides services for some thousand individual customers, as well as for companies with several dozen to hundreds of employees. The monetary assets that are involved easily exceed a few million Euro and

control over business activities such as share trading, credit or mortgage management is a stringent requirement.

About 30 Employees work in the branch. Apart from the general *Clerk* there are specific roles that may be informally described as *Head of Branch*, *Private Customer Advisor*, *Business Customer Advisor*, *Mortgage Advisor*. We do not, at this stage, explicitly consider any specific function and position assignments.

Different kinds of control are enforced at different conceptual levels. For instance, we found simple integrity checks encoded in the application logic and user interface; controls enforced through a specific workflow sequence and separation of tasks; controls in the form of the assignment of an employee to a specific group and role; as well as post-hoc controls through the internal audit department. We have briefly discussed such forms of control in sections 2.5 and 2.6, as well as throughout the thesis. Before we consider the credit application process in more detail, we will give some general examples of the kinds and levels of control that we observed.

- Controls enforced through application logic and interface design
  One example we have seen is that of a stock trading application. Here a financial advisor cannot buy or sell stocks without appropriate funds. The application will refuse to execute the order. This may sound obvious, but there are in fact specific trading techniques where this control might not be desirable.

  For example, depending on the seniority of the advisor, he may buy shares for a customer although the customer does not have the appropriate funds at that current moment. This is possible because the advisor knows that, for example, the interest of the customer's government bonds will be due the next day. The advisor may thus give the customer a 24 hour credit. Specific dual controls are in place to cater for such situations.

  We also know that this share trading application is able to analyse the trading activities of an advisor. If suspicious patterns emerge, the application may automatically notify a different advisor for reasons of dual control.

- Controls enforced through organisational structure and workflow design
  Apart from controls such as two required signatures or other forms of dual control, various forms of separation controls and Least Privilege principles are applied. One example is that of an advisor only having the authority to view accounts of the customers of the branch he works for.

  A different kind of operational separation is that an advisor selling a packet of shares for a customer may not transfer an equivalent amount of money to any other account than the current account of the customer. Otherwise he might sell shares and transfer the money to an account he owns himself. Any transfer to other accounts needs to be done by a different person. In this particular case we observed that this person was in fact located outside the branch in the regional headquarters.

- Controls enforced through internal audit and supervision
  Post-hoc controls in the form of internal audit are probably the most common form of control, as they do not obstruct or delay any specific business activity. For example, the bank will monitor if shares are traded for an employee of the bank in order to detect any insider trading.

  A different example is that the head of the branch is automatically notified of customers not paying back a credit. If he does not acknowledge that these 'foul' credits were brought to his attention, the next superior will be notified, and the case will receive a higher escalation number until it is resolved.

The general rule of thumb is that the higher the sum at stake the more controls are enforced. However, it is not always possible to estimate the value of assets (e.g. commercial reputation) in monetary terms [Pfleeger, 1997], which is one of the fundamental problems in general security and risk management.

9.9.2 A credit application process

A standard service provided by a bank is that of offering credits to its customers. This may take various forms such as extending the overdraft limit on a current account; providing mortgages for buying a house; or simply offering a fixed sum of money the customer may use at his discretion. Depending on the specific kind of credit, the application process will differ in the principals involved and data that need to be considered. In fact, the specific type of credit requested will have a direct effect on the involved controls. In the following we provide a simple example of a customer applying for a credit.

A customer applies for a sum of 10,000 Euro. Together with his advisor he will fill out a credit application, usually in an electronic form. Apart from his personal details, this form will require information about his financial situation, e.g. current salary, any offered securities or other assets. The application will then gather information about the customer's credit history using an external credit database. Interestingly, German law requires the customer's consent for this in the form of a signature. Advisors may not obtain this information at random, an example of an external, law-enforced, control. The application will then use the data that were obtained to evaluate whether the customer should be granted a credit and under which conditions this should be done. If the decision is positive, then a contract is generated which becomes legally binding with the customer's signature. In this case, control and credit approval is solely enforced through the application logic.

There are, however, situations which require other forms of control. We consider the following two examples. In the first example, the customer is rejected the credit of 10,000 Euro by the application. The advisor may, however, still provide the credit since there are circumstances on the customer's side which, in accordance with the organisational regulations, satisfy its approval. The second example may be that the application approved the credit, but the advisor agrees to lower the interest rate by a certain percentage. In both cases the control described in the following paragraph is enforced.

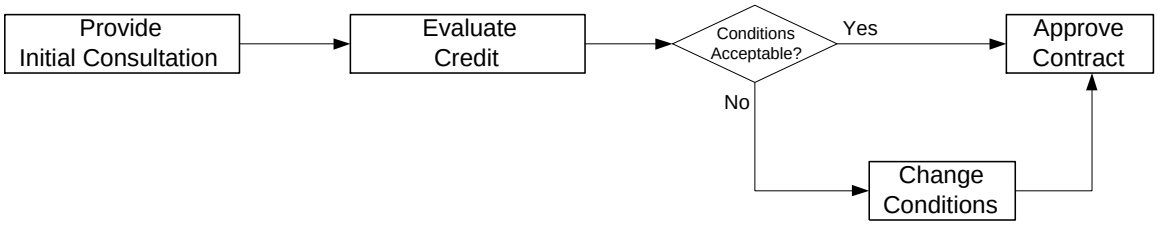

Figure 9.6: Basic credit application process in DIN 66001 notation.

Any transactions which show deviations from normal business practice will be brought to the attention of the advisor's superior. This is done in the form of a new entry in the superior's monitoring application. This monitoring application will keep a general list of transactions that require his attention or approval. In fact, this application will provide detailed information about the involved employees, kind of transaction and priority of the transaction. The superior may approve the transaction through a mouse click. Since he has been fully authenticated to the machine and application, this approval is binding on him.

We have no knowledge of the immediate effects of his approval or rejection. However, it is likely that certain transactions will only commit with his approval, while other transactions will commit without his approval and some kind of correction is performed in case of the superior's rejection. Speaking to staff at the branch, they asserted that a high level of informal trust is part of a superior's and subordinate's relationship, which is in fact perceived as part of the corporate culture. In the case of the credit application scenario this means that the advisor will only agree to provide a credit under conditions he knows his superior will approve. Likewise, a superior will usually rely on the integrity of his subordinate's decisions.

This system of approval is, however, not always based on an existing superior-subordinate relationship. Certain transactions only require approval from a peer, a basic dual control. We could not obtain information how far this approval and monitoring system is linked to the notion of roles as we described it in the previous section 9.5. Also, there is no single hierarchical structure for this branch, but several hierarchies exist for different purposes. For example, the head of the branch is the general superior for the branch staff considering general disciplinary measures. This does, however, not necessarily allow him to give specific orders to the senior private customer advisor. This advisor is subject to the directives of a superior in the Bank's regional headquarters with respect to his daily duties as an advisor. We have discussed this issue in the more general context of automated enterprise administration in [Kern et al., 2003].

9.10 Chapter summary and conclusion

This chapter represented the first part of our case study, and described and evaluated the access control system and controls of a major European Bank. We gratefully acknowledge the help of Mr. Horlacher, Mr. Petersen-Nickelsen and staff at Dresdner Bank Frankfurt, as well as Mr. Breiler at Dresdner Bank Hameln in making this case study possible.

The chapter consisted of two main parts. The first part provided a case study of the FUB role-based access control system of Dresdner Bank. In particular, the following points were discussed:

- The basic structure and conceptual design of the system in sections 9.2 to 9.4;
- The system-specific definition of roles consisting of functions and official positions, and evaluation of the number of roles in comparison to the system user population in section 9.5;
- The perceived weaknesses of the Bank's system in section 9.6;
- A comparison of the systems to the RBAC96 models, focusing on role inheritance with respect to function and position hierarchies in section 9.7;

In the second part we then discussed some of the observed controls in the administration of the system in section 9.8. This was followed by the description of controls in the context of a branch of the bank in section 9.9, specifically with respect to a credit application process in section 9.9.2. This process is partially automated through a credit evaluation application which will recommend contracts. However, a financial advisor may alter proposed contracts with the effect of the application notifying the advisor's supervisor of the change and requesting his approval.

As it can be seen in figure 1.1, describing the structure of our thesis, the observations made in this context influenced the definition of our control principle model in chapter 5. In particular, the distinction between *functions* and *Positions* influenced our design decision to distinguish between `Roles` and `Positions` in our model as described in section 5.5.

These observations of the branch of the Bank and the credit application process, now form the basis for our final chapter 10, which evaluates and validates our control principle model and the controls defined in chapters 5 to 8.

Chapter 10

Case Study - Part II

10.1 Introduction

The work presented in the previous chapters raises questions considering the validity and applicability of our suggested control principle framework. In particular, the following points need to be clarified in the following two sections:

1. How far does our control principle model and the concepts established in the past chapters reflect the observations we made in sections 9.8 and 9.9 in the first part of our case study?

2. What are the benefits and limitations of using Alloy for practical control principle specification and analysis?

We will use the example of the credit application process as described in section 9.9.2 to provide an answer to the above questions. However, we emphasise that this evaluation is based on an abstraction and sometimes hypothetical extension of this process. Nevertheless, the following discussion should be sufficient to answer the above questions and support the validity of our suggested framework in the context of this thesis. It will also help to summarise and clarify some of the concepts we established.

We begin with the identification and modelling of the objects that are involved and their relationships in section 10.2, which will also include the definition of exclusive roles, critical authorisation sets and role hierarchies, as well as a discussion on alternative approaches to assigning authority. This is followed by a discussion on possible separation controls in section 10.3, and how these may be applied in an incremental fashion depending on the required degree of control. We then look at the delegation of policy objects in section 10.4, what possible conflicts this may cause and the definition of additional controls on delegation activities. This will lead to a discussion on review and supervision controls in section 10.5.

10.2 Modelling the application process and organisational context in Alloy

We model an abstraction of the credit application process described in section 9.9.2, and the involved objects and their relationships in Alloy. This is done by explicitly enumerating objects using the `static` keyword, which indicates that a signature contains exactly one atom [Jackson, 2002]. We have briefly referred to this specification approach in section 4.4.5. The full specification can be found in appendix C.

The identification and definition of policy objects as well as their possible structuring by means of roles must follow a thorough engineering approach. A more formal definition of this approach is, however, outside the scope of this thesis, although we have at some stage looked at the engineering of roles in the context of enterprise security management [Kern et al., 2002].

We believe that the 'business process' is the underlying concept needed for any kind of successful policy-based systems or security management. We identified this as one main component of organisational structure in section 2.4.3. The discipline of business process analysis as part of the more general business process (re)engineering area is too diverse to be discussed here [Woehe and Doering, 1996, Mullins, 1999]. However, the close relationship to software engineering is evident, e.g. [Jacobson, 1995, Georgakopoulos and Hornick, 1995, Anton, 1996, Janssen et al., 1999].

Once such a process and the involved principals and objects have been identified and analysed in a given organisational context, obligations can be derived from it. In our context this process has been abstracted and described in terms of figure 9.6. Here each step in the process corresponds to an obligation. Once the obligations have been defined, we then derive the authorisations required to fulfill these obligations.

Specifically with respect to our framework and the distinction between general and specific obligations, we can say that the identification of the underlying business process is the basis for creating the general obligations of a principal, while the actual running of the process creates the specific obligations instances.

10.2.1 Identifying the involved objects

The following principals `A` to `I` as well as the `Credit_Application` participate in the credit application process. These are extended from the `Principal` signature which is extended from the `Object` signature as defined in section 5.2.1.

Alloy Signature 10.1 *Principals involved in the credit application process.*

```
static disj sig Principal_A, Principal_B, Principal_C,
                Principal_D, Principal_E, Principal_F,
                Principal_G, Principal_H, Principal_I,
                Credit_Application extends Principal{}
```

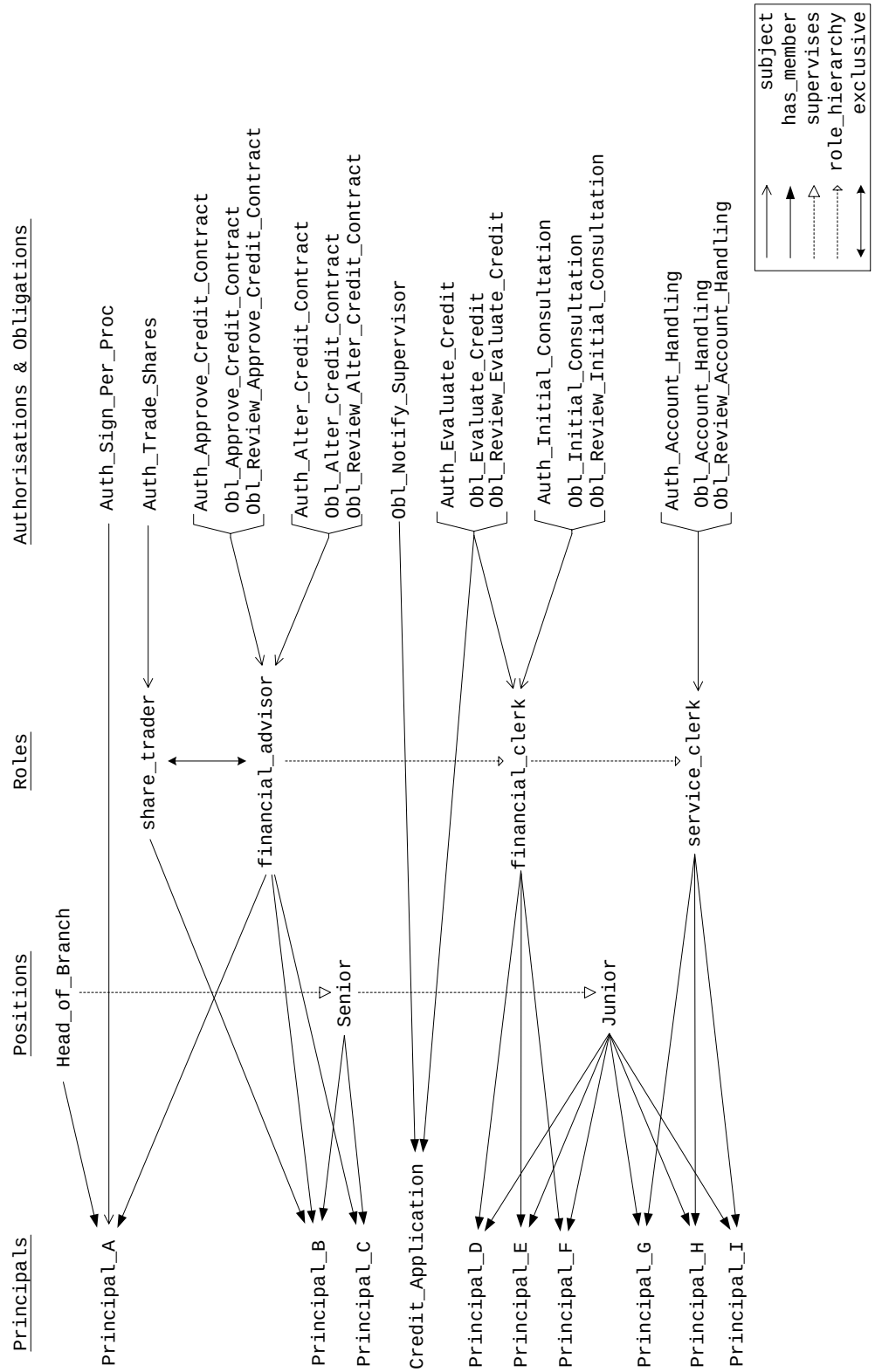

Figure 10.1: Assignments between the objects involved in the credit application process.

We defined the other objects required in the modelling of the credit application process in a similar manner. We do not provide the full signatures here for reasons of space but introduce these informally in the following with the support of figure 10.1. There the relations between the previously identified objects are depicted by different styles of lines. The `subject` relation for policy objects is a solid line with an open arrow, while the `has_member` relation for roles and positions is a solid line with a closed and filled arrow. The `supervises` and `role_hierarchy` relations are represented by dotted lines with a large closed empty arrow and a small closed empty arrow respectively. The `exclusive` relation has the form of a solid line with two filled arrows.

The credit application process consists of four stages as indicated in the previous figure 9.6. This is, however, not formally modelled in this case study, since our framework does not provide any explicit notion of workflow or process models. The process is defined by the involved obligations and required authorisations for each stage. The credit application begins with an initial consultation, followed by the evaluation of all the required information. We assume that for simplicity this evaluation always succeeds in this context, i.e. the application always proposes a contract. As described in the previous section 9.9.2, there is the option of a principal altering the conditions suggested by the credit evaluation application before the final approval.

We now identify the obligations that are part of the credit application process. These are the four obligations `Obl_Initial_Consultation`, `Obl_Evaluate_Credit`, `Obl_Alter_Credit_Contract` and `Obl_Approve_Credit_Contract`. The triggering event for these obligations would be the initial request of the customer for a credit. Additionally, there is the obligation `Obl_Account_Handling`. Also, each obligation has a corresponding review obligation to indicate how it may be reviewed as a result of an *ad hoc* delegation or a general supervision. For example, the obligation `Obl_Evaluate_Credit` is reviewed by the obligation `Review_Obl_Evaluate_Credit`.

Naturally, authorisations should match the defined obligations, and ideally there should be a strict correspondence to the credit application process as described in figure 9.6. These are the authorisations `Auth_Initial_Consultation`, `Auth_Evaluate_Credit`, `Auth_Alter_Credit_Contract`, `Auth_Approve_Credit_Contract` and `Auth_Account_Handling` which should be self-explanatory. There are also the two authorisations `Auth_Trade_Shares` and `Auth_Sign_Per_Proc`, not directly related to the actual credit application process but required for our later exploration of controls. We do not require the definition of matching obligations for the latter two here, since they do not contribute to any examples we shall give.

The authorisations now define a set of actions as discussed in section 5.8, which are, however, not part of figure 10.1. We chose to represent these actions in the form of functions that apart from defining input values do not show any behaviour, because this level of detail is irrelevant to the discussion. Only the update of the access history for a state as outlined in section 5.8 is defined.

These functions are `read_account()`, `evaluate_account()`, `check_credit_history()`, `generate_contract()`, `alter_contract()`, `approve_contract()`. For example, the first of these functions is specified in the following function 10.1.

Alloy Function 10.1 *Action of reading a customer's account represented by the read_account() function.*

```
fun read_account (disj s,s' : State, p1 : Principal, acc1 : Account) {
 //only update the object history
   s.s_access_history.accessing_principal = p1 &&
   s.s_access_history.accessed_object = acc1 &&
 //Additional information must be provided according to context
   //s.s_access_history.used_role = ... &&
   //s.s_access_history.used_authorisation = ...
}
```

What must be noted is that we do not make any specific assumptions about the two fields `used_role` and `used_authorisation` which are also part of the `access_history` of a state as defined in section 5.10.2. Thus, whenever we use such a function, we may have to explicitly provide the context specific information, e.g. that the role `financial_clerk` and its authorisation `Auth_Account_Handling` were used by a `Principal_G` to access account `account_smith`.

These derived obligations and authorisations are later structured using roles. As in the first part of the case study, we distinguish between functional roles and positions. These are the roles of a `financal_clerk`, `service_clerk`, `financial_advisor` and a `share_trader`. The positions principals may occupy are those of the `Head_of_Branch`, `Senior` and `Junior` manager.

Two types of objects are involved in this process. These are customer accounts which we assume to contain data such as monthly salary and recent credit history required for the evaluation. The other type of object is that of the credit contract. We modelled these in Alloy by extending the `Object` signature with a `Contract` and an `Account`. These are then in turn extended with the corresponding objects for customers Smith and Miller, represented by the signatures `account_smith`, `account_miller`, `contract_smith` and `contract_miller` respectively. It is clear that a more refined approach may be to define the customer's name as an explicit attribute-like signature.

10.2.2 Identifying the relationships between objects

We now need to establish the relationships between the defined objects. For example, we declare the members of the involved roles through the following fact 10.1. Here we demand that in every state the set of members for the `financial_advisor` must consist of `Principal_A`, `Principal_B` and `Principal_C`. Similarly, `Principal_D`, `Principal_E`, `Principal_F` have the role of a `financial_clerk`, and `Principal_G`, `Principal_H`, `Principal_I` have that

of a service_clerk. Additionally, we assume Principal_B to be a member of the role share_trader. Using the state-based s_has_member relation and quantifying over all states we exclude any possibility of administrative changes or delegation of roles. If the effects of administrative activities as defined in appendix B.5 were to be explored, we would only require the above relationships to hold in the first state of a sequence as defined by State_Sequence.first.

Alloy Fact 10.1 *Assignment of roles and principals.*

```
fact {all s : State |
       financial_advisor.(s.s_has_member) = Principal_A +
                                            Principal_B +
                                            Principal_C &&
       financial_clerk.(s.s_has_member) = Principal_D +
                                          Principal_E +
                                          Principal_F &&
       service_clerk.(s.s_has_member) = Principal_G +
                                        Principal_H +
                                        Principal_I &&
       share_trader.(s.s_has_member) = Principal_B
}
```

This definition of sets and the exhaustive enumeration of their members has the benefit of reducing the search space when generating models of the specification, as the possible combinations of signatures are reduced to those constraints. As before, we do not provide all the detailed facts which, for example, relate authorisations to roles etc., but discuss these informally in the following.

Principal_A is a member of the position Head_of_Branch, Principal_B and Principal_C are Senior managers, while Principal_D, Principal_E, Principal_F, Principal_G, Principal_H and Principal_I are Junior managers.

The financial_advisor role is subject to the authorisations to approve credit contracts Auth_Approve_Credit_Contract and to alter contracts Auth_Alter_Credit_Contract, while the financial_clerk role is assigned with the authorisation to consult customers Auth_Initial_Consultation and to evaluate a credit Auth_Evaluate_Credit. The share_trader is subject to the authorisation Auth_Trade_Shares while the service_clerk is allowed to handle accounts as captured in the Auth_Account_Handling authorisation. We note that the authority to sign per procuration Auth_Sign_Per_Proc is assigned directly to Principal_A in the position of the Head_of_Branch.

As discussed in the previous section, this assignment of authority is based on the assignment of obligations, partially shown in figure 10.1. For example, the authorisation Auth_Evaluate_Credit corresponds to the obligation Obl_Evaluate_Credit and both have as their subject the financial_clerk role.

There are hierarchies between these roles and positions. The three positions `Head_of_Branch`, `Senior` and `Junior` form a supervision hierarchy, and there is also a role hierarchy between the roles `financial_advisor`, `financial_clerk` and `service_clerk`. This role hierarchy is interpreted using the semantics defined in section 5.6.3.1, where senior roles inherit the policy objects of junior roles. For example, any principal in the role `financial_advisor` has the `financial_clerk`'s policy objects at his discretion.

10.2.3 Identifying exclusive roles and critical authorisation sets

Exclusive roles are one of the concepts on the basis of which separation controls can be defined as discussed in section 5.7. One pair of exclusive roles within the context of the branch is the role of a `share_trader` which has been observed to be exclusive to the `financial_advisor`. This is because a share trader should not be involved in any financial activities, as he might otherwise misuse this authority for insider trading using borrowed money.

A different approach for obtaining control over the activities in the branch by means of separation is the identification of critical processes and the authorisations matching these processes, as discussed in section 6.2.3. In this case the credit application process has been defined as critical by making the involved authorisations part of a critical authorisation set. Also the three authorisations of the general handling of customer accounts as well as altering and evaluating a credit request have been defined as critical, as a principal may otherwise manipulate evaluation criteria to circumvent application controls or even create fictitious accounts he has control over. Both of these critical authorisation sets `Credit_Application_Critical_Authorisation_Set_1` and `Credit_Application_Critical_Authorisation_Set_2` are defined in the following fact 10.2.

Moreover, instead of declaring the two roles `share_trader` and `financial_advisor` as exclusive, we may alternatively specify a `Credit_Application_Critical_Authorisation_Set_3` which defines the authorisations of these roles as critical. This is also part of fact 10.2.

Alloy Fact 10.2 *Critical authorisation sets in the credit application context.*

```
fact {Credit_Application_Critical_Authorisation_Set_1.critical =
        Auth_Initial_Consultation +
        Auth_Evaluate_Credit +
        Auth_Alter_Credit_Contract +
        Auth_Approve_Credit_Contract &&
      Credit_Application_Critical_Authorisation_Set_2.critical =
        Auth_Evaluate_Credit +
        Auth_Alter_Credit_Contract +
        Auth_Account_Handling &&
      Credit_Application_Critical_Authorisation_Set_3.critical =
        Auth_Trade_Shares +
        Auth_Approve_Credit_Contract +
        Auth_Alter_Credit_Contract}
```

10.2.4 Alternative approaches to assigning authority

In the previous section we have assigned authorisations to the roles and positions principals occupy. These assignments are valid throughout the rest of this case study, but we want to outline alternative approaches to assigning authority.

As mentioned in section 5.2.1, we believe that organisational requirements also demand for the ability to assign authorisations directly to principals, a possibility which is excluded in other approaches [Sandhu et al., 1996]. Since in our framework principals are objects, and objects may be made subject to an authorisation, this direct assignment is possible. As an example of why a direct assignment of authority to a principal may be required, we refer to the authorisation to sign by power of procuration `Auth_Sign_Per_Proc`. Procuration is a legal construct reflecting the delegation of authority to a principal to represent his organisation in certain matters of business. By signing contracts with his name and adding *per proc.* this authority is explicitly indicated to the other contract participants, and they may confirm this authority by inspecting the commercial register. If captured in the form of an authorisation policy, then such power of procuration should be assigned directly to a principal. We refer to the example of a position description given in [Buehner, 1994] that shows a similar situation. We consider this to be an important issue with respect to the emerging field of electronic contracts, e.g. [Milosevic et al., 1995, Ao et al., 2002, Ungureanu, 2002].

This direct assignment may be further constrained by specifying that only principals in the position of a `Head_of_Branch` may be directly assigned with this authority. This can be expressed in the following fact 10.3.

Alloy Fact 10.3 *Only principals in the position of a* `Head_of_Branch` *may be directly assigned with the authority to sign per procuration.*

```
fact {all p: Principal |
      all s: State |
  p in Auth_Sign_Per_Proc.(s.s_subject) =>
    p in Head_of_Branch.(s.s_has_member)
}
```

Another example may be to constrain the assignment or use of authority considering some specific attributes. In this case the amount of the credit may be directly related to the authority of approving it. We may give a value to our credit by defining a new signature `Value` from which we statically extend the values `High` and `Low`. We then add the field `contract_value:Value` to the `Credit_Contract` signature. We do this because Alloy currently supports Integers only in its BNF specification, but they have not been fully integrated into its type system yet. We may then use this to constrain the actual use of the approval authority. For example, a principal may only approve a contract with a high value if he occupies the `Head_of_Branch` position, defined in the following fact 10.4.

Alloy Fact 10.4 *A principal may only approve a contract with a high value if he occupies the* `Head_of_Branch` *position.*

```
fact {all disj s,s': State |
      all p: Principal |
      all contract_1: Contract|
  approve_contract (s,s', p, contract_1) &&
  contract_1.contract_value = High =>
   p in Head_of_Branch.(s.s_has_member)
}
```

10.3 Separation controls in the credit application process

There are many possible separation controls that may be desirable to specify in this context. In section 10.3.1 we will show how, on the basis of exclusive roles and critical authorisations, we can introduce a set of incrementally relaxed separation controls within the context of the case study. This is done on the basis of the separation controls as presented in section 6.2. We then give some examples of alternative separation controls in section 10.3.2.

10.3.1 Incremental application of separation controls

We begin with the strict role-based separation control as defined in function 6.1. In the context of our case study this would not allow a principal to be member of the `financial_advisor` and `share_trader` roles at the same time, since these have been declared as exclusive. This may, however, be a too strict constraint as, perhaps depending on the size of the branch, a principal may well have to be able to perform both functions. A possible relaxation may be to use the static object-based separation constraint as defined in function 6.3 instead. This will allow a principal to be assigned to such exclusive roles. However, the set of authorisations determined by those roles may then not have the same target. This would not be true in this case, since both, the `financial_advisor` and the `share_trader`, must have access to customer accounts. Using the dynamic object-based separation constraint as defined in function 6.4 instead may mitigate this situation. This will now allow a principal to act in these two exclusive roles, as long as he does not use both roles and some of the assigned authorisations to access the same object.

As an alternative way to enforce control, we may consider the degree to which authorisations are shared between exclusive roles. We have discussed this in section 6.2.5 and identified four possible degrees of such a share. We note that in this context the strongest *Disjoint/Disjoint* property holds as defined in function 6.11. The two exclusive roles `financial_advisor` and `share_trader` do not share any authorisation policy, and none of the policies assigned to one of these exclusive roles is assigned to any other non-exclusive roles. If new exclusive roles and authorisations were to be introduced, these may then be incrementally checked against the described four degrees of shared authorisations to obtain the desired level of control.

A different scenario is that of considering separation controls based on the critical sets of authorisations as explained in section 6.2.3 and declared in fact 10.2. If we do not consider any role inheritance, then the static operational separation constraint as defined in function 6.5 will hold. If we do consider inheritance, then the constraint will, for example, not hold for principals in the role of a `financial_advisor`, since they would have all authorisations defined in the `Credit_Application_Critical_Authorisation_Set_1` and `Credit_Application_Critical_Authorisation_Set_2` at their discretion. However, this may again be too strict with respect to the day to day business of the branch. A possible relaxation is to use the dynamic operational separation function 6.6. This will allow a principal to have a set of critical authorisations available to him, but he must not use them all. Again this might be too restrictive as he may very well have to use all of them, just simply not on the same object. This can then be supported by using the history-based separation constraint specified in function 6.7, controlling that an object was either accessed more than once, but not using all critical authorisations, or that all critical policies have been used, but not on the same object.

10.3.2 Alternative separation controls

In the previous section we have described the incremental application of separation controls based on the notion of exclusive roles and critical authorisation sets. These are, however, not necessarily the only concepts that may be used for the enforcement of separation controls, and depending on the context we may even think about exclusive actions or principals as suggested in [Nyanchama and Osborn, 1999] and [Ahn, 2000]. With respect to the latter, we have observed that a major British mobile phone company does not allow related principals to work in the same branch, a constraint which may equally apply to the bank branch.

Chinese Walls as described in [Brewer and Nash, 1989] may also be considered as a kind of history-based separation control on the basis of conflicting company classes. In the context of the bank branch, a possible Chinese Wall may be that a financial advisor who advises a company A may not advise the competing company B. This should be easy to define in the context of our framework using the `Group` and `AccessHistory` signatures. For reasons of space we refrain from doing this here.

We believe that our notion of positions may also be a useful criterion to enforce separation controls considering the assignment of roles to principals. In the simplest case, positions may be declared as exclusive and the separation controls described in chapter 6 can be applied. Since positions are of the type `Role` no changes are required. It may also be desirable to declare certain roles and positions as exclusive. Similarly, we may want to control the degree to which authorisations are shared between positions.

A different approach is to consider the positions and roles that a principal is a member of, with respect to an existing supervision hierarchy. Certain roles may only be assigned to principals occupying a specific position. For example, we may specify that the role of a `financial_advisor` may only be assigned to principals in the position of a `Senior` or any position supervising a `Senior`. This can be expressed in the following fact 10.5.

Alloy Fact 10.5 *The role of a* `financial_advisor` *may only be assigned to principals in the position of a* `Senior` *or any position supervising a* `Senior`*.*

```
fact {all s : State | all p : Principal | some pos : Position |
  p in financial_advisor.(s._has_member) =>
   (p in Senior.(s.s_has_member)) ||
   (p in pos.(s.s_has_member) && Senior in pos.^(s.s_supervises))
}
```

In a similar way, we may want to define that the sets of roles assigned to principals occupying a `Senior` and `Junior` position are disjoint as shown in fact 10.6.

Alloy Fact 10.6 *Role sets for principals member of* `Senior` *and* `Junior` *positions are disjoint*

```
fact {all s : State |
  no ((s.s_has_member).(Senior.(s.s_has_member)) - Position) &
     ((s.s_has_member).(Junior.(s.s_has_member)) - Position)
}
```

Clearly the usefulness of this constraint has to be reviewed if a principal occupies both these positions, since there is no explicit constraint ruling this out, or if any inheritance hierarchies are involved. Nevertheless, it may be used to indicate a position-based separation of the functions principals carry out through their roles.

10.4 Delegation of policy objects

10.4.1 Delegation of authority and conflicting separation controls

The delegation of authority is supported by the delegation function 7.2 `delegate_auth()`. This allows a principal to delegate an authorisation policy he holds directly or through a role with the possibility of losing or retaining it. The various possible delegation scenarios have been discussed in detail in section 7.4.2. In the context of our example, we can observe that initially all but the `Auth_Sign_Per_Proc` authorisations are assigned to roles. If, for example, `Principal_B` delegates his authority `Auth_Trade_Shares` to `Principal_C` then the only assignments that change are that of `Principal_C` as a new subject to the `Auth_Trade_Shares` policy and the update to the delegation history.

As we further explored in section 7.7, the delegation of authority may conflict with separation controls. This may in fact be the case for the above delegation example. We consider the scenario supported by the following Alloy state sequence 10.1, where `Principal_B` delegates his authority to trade shares to `Principal_C`. Now, together with the authorisations obtained through his role of `financial_advisor`, `Principal_C` is in possession of all authorisations defined by the critical authorisation set

Credit_Application_Critical_Authorisation_Set_3. This set was defined in fact 10.2 as an alternative to declaring the two roles financial_advisor and share_trader as exclusive.

Thus, the static operational control defined by the function 6.5 static_op_separation() will not hold in this delegation scenario. The following state sequence 10.1 represents this delegation activity. No model will be generated when running this function credit_application_scenario_1() since the static_op_separation() will evaluate false.

State Sequence 10.1 *Delegation violates separation control*

```
fun credit_application_scenario_1 (){some disj s1, s2 : State |
 //Definition of sequence
   s1 -> s2 in State_Sequence.next &&
 //Step 1:
   delegate_auth(s1,s2, Principal_B, Principal_C, Auth_Trade_Shares) &&
   static_op_separation()
}
```

10.4.2 Controlling the delegation of policy objects

Before discussing review and supervision as post-hoc controls on delegation activities, we consider how our distinction between roles and positions may also be used to control the delegation of policy objects between principals. Positions may carry with them a set of attributes, characteristic of the position. They also participate in supervision relationships. We may use these attributes and supervision relationships to constrain the delegation of policy objects. Such constraints will be dependent on the given organisational context, and in the following we only give two possible examples for such a form of delegation constraint.

In the first example, principals occupying a position may only delegate to other principals in positions with identical attributes. In this context such an attribute may simply be that of the branch identifier. This would then only allow for the delegation of policy objects between principals occupying a position within the same branch as shown in fact 10.7. This is of course very simplistic, and other examples of such attributes may be specific qualifications such as that of a certified accountant.

Alloy Fact 10.7 *Only allow for the delegation of policy objects between principals occupying a position within the same branch, here* Branch_West.

```
fact {all disj s, s' : State | all disj p1, p2 : Principal |
 delegate_auth(s,s',p1, p2, Object) =>
   Branch_West in (((s.s_has_member).p1) - Role).(s.s_has_attribute) &&
   Branch_West in (((s.s_has_member).p2) - Role).(s.s_has_attribute)
}
```

Considering the supervision relationship between positions, another possibility may be to only allow for the delegation of authority from a principal to some other principal in the same or a lower position. This may be expressed in the following fact 10.8.

Alloy Fact 10.8 *Only allow for the delegation of authority from a principal to some other principal in the same or a lower position.*

```
fact {all disj p1,p2: Principal | all disj s, s': State |
      all pos1, pos2: Position |
  delegate_auth(s, s', p1, p2, Authorisation) =>
    p1 in pos1.(s.s_has_member) &&
    p2 in pos2.(s.s_has_member) &&
    (pos1 in pos2 || pos2 in pos1.(s.s_supervises))
}
```

If principals, positions, roles and authorisations have been assigned correctly at the time of system set-up, then this form of delegation constraint may ensure that principals in some specific position can never be delegated an authorisation. Also the reverse may be the case, principals should only be able to delegate to other principals in senior positions. Assuming that critical authorisations such as approving contracts are assigned with principals in higher position this may provide a further degree of control.

We have initially discussed such a control on the "flow" of delegated authorisation policies in [Schaad, 2001]. However, we did not further investigate possible relationships to (role-based) mandatory access control models, e.g. [Denning, 1976, Sandhu, 1996].

10.5 Review and supervision controls

The concepts of review and supervision were introduced in chapter 8 to achieve control over

- the delegation of specific obligation instances and;
- the delegation of general obligations.

We defined a review policy to be a specific type of obligation with a delegated obligation as its target. Supervision is the general obligation of a principal occupying a position to review the obligations of principals in supervised positions.

10.5.1 Review to support the *ad hoc* delegation of obligation instances

A review results out of an *ad hoc* delegation of an obligation instance between two principals, as we have formally described it in sections 7.4.3 and 8.2. The example we use here is that of a principal delegating his obligation to evaluate the credit application of a customer. To describe this situation we make use of the conceptual distinction between general and specific obligations discussed in section 5.3.2 and supported by signature 5.2 and facts 5.2, 5.3, 5.4 and 5.5.

Alloy Function 10.2 *In this sequence of states* `Principal_D` *delegates his obligation instance* `Obl_Inst_Evaluate_Credit_Smith` *to* `Principal_E`*.*

```
fun delegation_of_evaluation_obligation (){some disj s1, s2 : State |
   //The actual delegation of the obligation instance
   delegate_obligation(s1,s2, Principal_D, Principal_E,
                       Obl_Inst_Evaluate_Credit_Smith,
                       Review_Obl_Inst_Evaluate_Credit_Smith) =>
   //The delegated obligation must be reviewed...
     Obl_Inst_Evaluate_Credit_Smith in
     Review_Obl_Inst_Evaluate_Credit_Smith.(s2.s_target) &&
   //...by reviewing the evaluated contract as the evidence
     contract_Smith in (s2.s_reviewed_by).read_account_action &&
     some contract_Smith & Evidence &&
   //After the delegation, the review oblg. instance is held by Principal D
     Review_Obl_Inst_Evaluate_Credit_Smith.(s2.s_subject) = Principal_D &&
   //The delegated obligation instance is held by Principal E
     Obl_Inst_Evaluate_Credit_Smith.(s2.s_subject) = Principal_E
}
```

We consider `Principal_D` to have the specific obligation to evaluate the credit of the customer Smith, represented by the signature `Obl_Inst_Evaluate_Credit_Smith`. This signature is of the type `ObligationInstance` and may have `Principal_D` as its subject, because he is member of the role `financial_clerk`, which is subject to the corresponding general obligation `Obl_Evaluate_Credit`.

For some reason as, for example, one of those given in section 7.2, `Principal_D` is not able to evaluate this contract for customer Smith and delegates the specific obligation to `Principal_E` instead. This is possible because `Principal_E` is member of the same `financial_clerk` role and thus subject to the required general obligation `Obl_Evaluate_Account` as required by facts 5.3 and 5.4. As a result, `Principal_D` has the obligation to review this delegated obligation, here expressed in the signature `Review_Obl_Inst_Evaluate_Credit_Smith`. This signature is of the type `ObligationInstance` and may be held by `Principal_D` because he is assigned with the role of the `financial_clerk` which is subject to the corresponding general review obligation `Review_Obl_Evaluate_Credit`.

As we mentioned in section 8.2, it must have been defined beforehand how the initially delegating principal performs the review. This means that such a form of delegation cannot be totally *ad hoc*, but must happen within some given organisational constraints. The actual assignment of a general review obligation to a role and the membership of principals in that role is one of these constraints. If, in the above example, `Principal_E` was not in possession of the general obligation corresponding to the obligation instance delegated to him, then such a delegation could not happen. Additionally, as described in sections 5.4 and 8.2.3, the concept of evidence serves as an abstraction to the discharge of an obligation. In this context the

evidence that has to be provided might be simply the credit application contract of customer Smith itself. The above function 10.2 reflects this, defining the `contract_Smith` to be of the type `Evidence` and to be reviewed by the earlier defined action `read_account_action`.

We express this action as a signature since we are interested in structural aspects of review. We assume this review action to be available to `Principal_D` through the authorisation `Auth_Evaluate_Credit` he may use through his role as a `financial_advisor`.

10.5.2 Supervision to support the delegation of general obligations

In section 8.3, we identified supervision as the general obligation of a principal in a position to review the obligations of principals in supervised positions. This was assumed to be the result of some previous delegation of general obligations as described in the example given in section 8.3.2.

In this case this would, for example, mean that `Principal_A` in his position as the `Head_of_Branch` reviews credit contracts approved by principals in supervised positions. Here this would be any contract approved by `Principal_B` or `Principal_C`, as these have the authority to approve contracts and occupy the `Senior` position which is supervised by the `Head_of_Branch`. For economic reasons this review may be selective, for example, only every contract over Euro 50,000 or for a blacklisted customer is reviewed. A more detailed investigation into criteria for such a selective review is, however, outside the scope of our framework.

Before formally supporting this supervision relationship in the following fact 10.9, we need to clarify that the obligations and review obligations arising out of a supervision relation may not necessarily be assigned to positions only. In fact, in the above example the obligation to approve a contract `Obl_Approve_Credit_Contract` has as its subject the role of a `financial_advisor` and not the position of a `Head_of_Branch`, as shown in figure 10.1. One reason is that the authority to approve is with the `financial_advisor` role as well and in general, authority and obligations should always reside next to each other. Whether an obligation and its corresponding review obligation have a role or a position as their subject will, however, be dependent on the specific organisational context, and here obligations have only roles as their subject.

Alloy Fact 10.9 *All principals* `p`*, that are member of the* `Senior` *position must have their obligation instances to approve a credit reviewed by the principal in the* `Head_of_Branch` *position.*

```
fact {all p : Principal | all s : State |
      all obl_inst : Obl_Approve_Credit_Contract & ObligationInstance |
 p in Senior.(s.s_has_member) &&
 p in obl_inst.(s.s_subject) =>
  some review_inst: Obl_Review_Approve_Credit_Contract & ObligationInstance|
  Head_of_Branch.(s.s_has_member) in review_inst.(s.s_subject) &&
  obl_inst in review_inst.(s.s_target)}
```

As a second example we may think of principals in `Senior` positions supervising principals in `Junior` positions, a relationship which, however, raises some questions. We observe that in the real world supervision is usually performed by one principal supervising several others, a requirement indirectly postulated in Urwick's 10 principles of organisation [Urwick, 1952], that we reviewed in section 2.3. Considering the cardinality of the position membership and the resulting effects on supervision, we do not place any restrictions on how many principals may occupy a position. In this context the `Senior` position supervises the `Junior` position and several principals are members of both these positions. This raises questions with respect to the actual principals performing the review resulting out of the supervision and who is supervised by whom. We have discussed a similar situation in section 8.3.3 and argued that other organisational factors must be considered to resolve any ambiguities.

For example, although `Principal_B` and `Principal_C` both occupy the same `Senior` position they may supervise different principals. In this context `Principal_B` may supervise those principals that occupy a `Junior` position and have the role of a `financial_clerk`, while `Principal_C` supervises those principals that occupy a `Junior` position and have the role of a `service_clerk`. Here supervision is performed on the basis of the position and function principals have.

Alloy Function 10.3 *If a principal is member of the* `Junior` *position and the* `financial_clerk` *role, then any of his obligation instances is reviewed by* `Principal_B`*. If a principal is member of the* `Junior` *position and the* `service_clerk` *role then, any of his obligation instances is reviewed by* `Principal_C`*.*

```
fact {all s : State | all p : Principal |
 some (p & Junior.(s.s_has_member) & financial_clerk.(s.s_has_member)) =>
 ((s.s_subject).p & ObligationInstance - Review) in //excl. review of review
 ((s.s_subject).Principal_B & Review & ObligationInstance).(s.s_target) &&

 some (p & Junior.(s.s_has_member) & service_clerk.(s.s_has_member)) =>
 ((s.s_subject).p & ObligationInstance - Review) in //excl. review of review
 ((s.s_subject).Principal_C & Review & ObligationInstance).(s.s_target)
}
```

The reason why `Principal_B` and `Principal_C` can supervise is that they have the three review obligations `Obl_Review_Evaluate_Credit`, `Obl_Review_Initial_Consultation` and `Obl_Review_Account_Handling` through their membership in the `financial_advisor` role and the role inheritance hierarchy. However, we believe that also other organisational criteria may be used to support supervision. For example, we could think of grouping principals in two distinct groups `front_office` and `back_office` with one `Senior` principal in each group supervising this group. Again, this depends on the given organisational context.

However, the supporting constraint 10.3 may be too strict as it requires every obligation of a supervised principal to be reviewed. This can also lead to further conflicts. We consider the previous example of delegating the obligation to evaluate the contract for customer Smith

from Principal_D to Principal_E, described in function 10.2. Principal_D would then have a review obligation as a result of this delegation. However, Principal_B would also have a review obligation as a result of his position membership and resulting participation in a supervision relationship. This may too restrictive and not economic, but any further discussion and resolution strategy is outside this scope.

10.5.3 Supervision in the semi-automated credit application process

As a third example of supervision we look at the scenario we described in section 9.9.2. A Credit_Application supports the process of applying for a credit by automatically evaluating the criteria for the credit and suggesting a contract. This is reflected in its direct assignment with the obligation and authority to evaluate a credit in figure 10.1.

A financial advisor may change the contract conditions suggested by the Credit_Application and approve the credit. He may even approve a credit rejected by the Credit_Application. If either of the two is the case, the Credit_Application will notify the supervisor of a principal of this change or approval. On a more formal level this means that the Credit_Application is subject to the obligation to notify the supervisor Obl_Notify_Supervisor as shown in figure 10.1. The triggering event for this notification would be the detection of a change to the contract by the principal.

This obligation to notify will then trigger an obligation on the side of the supervisor. Depending on the given circumstance, this second obligation may only require the supervisor to review the advisor's alterations to the contract or perhaps to provide a second approval. The main difference is that in the latter case the whole credit application process is put on hold, while in the first case the process may continue, but the Bank relies on corrective measures if the review is not satisfactory. We discuss these two possibilities in the following.

The first case is the review of a contract alteration based on a supervision relation, although it differs to some extent from the examples we have described in the previous section. In terms of our framework we observe that the Credit_Application triggers a new obligation instance for the principal's supervisor to review an alteration. This is possible because at some earlier stage the corresponding general review obligation has been assigned to a role or position of this supervising principal, allowing for the creation of this review obligation instance. In this context this is the assignment of Principal_A to the position of Head_of_Branch and the role of a financial_advisor. The general obligation to review any alterations to the contract by principals in supervised positions may then be expressed in a similar way as in the previous fact 10.9. The credit contract itself serves as the evidence that the financial advisor used his authority as defined in the organisational guidelines and thus obeyed his delegated general obligation.

The same observations hold for the second case, where the supervisor is requested to review any alterations. However, here he additionally has to approve these changes. This is a dual control encoded in the actions defined by the review obligation.

Of further interest are the observations on the `Credit_Application` and its obligation to evaluate credits. We believe that there are two underlying motivations for why it was delegated this general obligation. The first is simply the distribution of work and relief of principals from this task by means of automation. The second reason is that it is neither practical nor economic to review all the evaluation obligations of supervised principals. Control over the evaluation is achieved in the form of coding it into the `Credit_Application`, whose actions do not have to be reviewed. The supervisor will only have to review an evaluated credit if this application control does not suffice anymore.

This does not mean that applications do not have to be controlled in some form or other. However, we do not know what kind of more general application audit controls are enforced by the Bank.

10.6 Chapter summary and conclusion

This chapter and second part of our case study demonstrated the validity and possible practical application of our framework and the established concepts. We demonstrated how to partially support the modelling of the credit application process as initially described in section 9.9.2.

In particular we discussed the identification and modelling of the involved objects and their relationships in section 10.2, which also included the definition of exclusive roles, critical authorisation sets and role hierarchies as well as a discussion on alternative approaches to assigning authority. This was followed by a discussion on possible separation controls in section 10.3 and how these can be applied in an incremental fashion depending on the required degree of control.

We then looked at the delegation of policy objects, what possible conflicts this may cause and additional controls on delegation activities in section 10.4. This led to a discussion on review and supervision controls in section 10.5. We modelled the delegation of an obligation to evaluate the credit contract of a customer in section 10.5.1. This was followed by a discussion and specification of some supervision relationships in section 10.5.2, which in particular demonstrated how to resolve possible supervision ambiguities. In section 10.5.3 we looked at the role the `Credit_Application` plays in the context of review and supervision controls.

However, even this abstracted credit application process and the involved objects and their relationships proved to be complex to specify. Nevertheless, the semi-formal approach we took initially revealed some ambiguities in the supervision concept, which in turn led to amendments and changes in sections 8.3.3 and 10.5.2.

Chapter 11

Discussion and Conclusion

11.1 Introduction

Based on the theses defined in section 1.2, we have established a framework for organisational control principles.

1. This framework was developed from a review of relevant existing work in chapters 2-4.
2. It consisted of the definition, analysis and exploration of a control principle model and individual control principles in chapters 5-8.
3. A supporting case study validated the framework in chapters 9 and 10.

In this chapter we provide a final discussion and conclusion to the work presented in this thesis, and critically evaluate what has been achieved. In particular, this chapter includes:

- A general summary of our methodical approach and established concepts in section 11.2;
- A validation of the framework character of our work in section 11.3;
- An extended discussion and critical evaluation of selected aspects of this thesis which also gives an account of possible future work in section 11.4.

11.2 Thesis summary

Based on the theses stated in section 1.2, we established a framework for organisational control principles. The development of this framework was divided into an initial literature review; the definition, analysis and exploration of a control principle model and individual control principles; and a supporting and concept validating case study.

In **chapter 2** we provided a working definition of (an) organisation as a goal attainment system, characterised by the distribution of work which results in organisational structure. This was followed by a review of different types of organisational control and internal control systems, which in turn led to a definition of control principles and identification of their source. In **chapter 3** we argued that the delegation of work requires the delegation of the needed authority to be able to perform the arising obligations. In automated systems this authority is partially expressed in the access rights of a principal, and access control is a stringent requirement. This resulted in our review of role and policy-based approaches to access control and more general systems management and audit. In particular, we looked at the RBAC96, OASIS and Ponder frameworks since they all include a notion of roles; incorporate delegation mechanisms; allow for the specification of permissions/authorisation policies as well as obligation policies; and provide the means for the expression of other controls. These two chapters provided much of the conceptual groundwork for the definition of a control principle model, and **chapter 4** explored some specification languages and techniques we believe to be suitable to express some of these concepts. This resulted in our choice of a declarative approach, supported by the Alloy language and its automated analysis facilities as initially assessed in [Schaad and Moffett, 2002b]. We introduced Alloy by means of a short tutorial.

We defined a model for expressing control principles [Schaad and Moffett, 2002a] in **chapter 5**. The main components are policy objects, which can be authorisations or obligations. A review is a specific type of obligation. Principals and the roles that they are members of can be subject to these policies. In particular, we distinguished between the general and specific obligations of a principal. The model further includes the concepts of role and supervision hierarchies, exclusive roles, and defines approaches to maintaining a history of a principal's actions. As a first control principle we discussed separation controls in **chapter 6**. From the existing literature we extracted a set of separation controls and provided a unified representation. These controls were asserted and further explored by means of Alloy. In **chapter 7** we then discussed and formally defined the delegation and revocation of policy objects in the context of the control principle model. We explored possible relationships between delegation controls and separation controls based on our initial observations in [Schaad, 2001]. This included a critical review of delegation and revocation controls in the RBAC96, OASIS and Ponder frameworks. Particularly, the delegation of specific and general obligations requires to be controlled, and in **chapter 8** we introduced the novel concepts of review and supervision to support this. A review is an obligation with a previously delegated obligation as its target, as we initially defined it in [Schaad and Moffett, 2002c]. Evidence serves as an abstraction for the discharge of an obligation. Supervision was defined

as the general obligation of a principal to review the obligations of principals in supervised positions. We explored different scenarios for delegating obligations with respect to these newly established concepts.

In **chapter 9** we presented the first part of our case study of the large-scale role-based access control system of a major European bank [Schaad et al., 2001]. We described its basic architecture, its definition of a role as consisting of functions and positions, and possible role hierarchies, before we identified some general controls involved in the administration of the system. We then focused on the types of controls we observed in the context of a branch of that bank, in particular those within a credit application process. **Chapter 10** asserted the validity of our suggested framework and its suitability for analysing control principles. This included the identification and modelling of the objects and relations involved in the credit application process. This was followed by a discussion of possible separation and delegation controls, focusing on the identification and analysis of review and supervision controls.

11.3 A framework for organisational control principles

We claim to have developed a framework for organisational control principles. This adheres to our working definition of a framework which we presented in section 1.2 by clarifying, comparing, categorising, evaluating and integrating a range of existing and new concepts, models and techniques. Recalling the three underlying theses stated in section 1.2 and figure 1.1, we believe that we have successfully demonstrated the framework character of our work.

As part of our initial literature review in chapters 2-4, we have *clarified* the terms organisation and organisational structure. This included a discussion on the source and organisational background of control principles. We further provided an initial *comparison* and *evaluation* of selected role and policy-based models, as well as specification languages potentially suitable for control principle expression.

In chapters 5-8 we established our control principle model. This included the *integration* of existing separation controls and parts of the reviewed role and policy-based models into our control principle model. In particular, we *clarified* and gave a clear meaning to the concepts of delegation and revocation. We *categorised* different kinds of delegation and revocation activities according to the specific type of policy object to be delegated and the assignments of principals to it. We further *compared* and *evaluated* the reviewed role and policy-based models with respect to their support for these properties. We established the novel concepts of review and supervision and *evaluated* and *integrated* these.

Our case study presented in chapters 9 and 10 further *clarified* our work by identifying control principles in the context of a bank's access control system. We then used this as the basis for *evaluating* and demonstrating how our established control principle model and single control principles may be practically applied and analysed when *integrated*.

On a more technical level we have provided an exhaustive *evaluation* of the Alloy specification language and its support for automated analysis and exploration of control principles.

11.4 Discussion and critical evaluation

We will now discuss and provide a critical evaluation of selected aspects of this thesis. This discussion will follow the structure of the thesis and will also outline possible future work.

11.4.1 Organisations and organisational control

Our aim was to draw attention to the sociological aspects of organisations, their structure, control and control principles. However, we could only provide a very limited review of the existing work. Specifically section 2.7, discussing that organisational control principles support the control goals of an organisation, is only based on our observations and practical experiences. We believe that a more thorough review of organisational literature would reveal the existence of empirical data supporting our claims.

Organisational goals and their refinement has been treated in the computer science literature, specifically in the area of requirements engineering, e.g. [Moffett and Sloman, 1993, Anton, 1996, Lamsweerde, 2000b, Lamsweerde, 2001]. There is also initial work at the university Louvain-la-Neuve, investigating the integration of the KAOS requirements framework [Dardenne et al., 1993] and the Ponder policy framework [Damianou, 2002]. With this work as a starting point, it would be interesting to further investigate how far our suggested control principle framework could be used in the sparse area of security requirements engineering.

The organisational control principle model we suggested in chapter 5 is clearly limited. The tradeoff was to define a model that is suitable to support the expression of control principles at a structural and behavioural level, at the same time being small enough to be made subject to analysis. We believe that we have succeeded in that respect, but extensions to the model may be desirable. As initially discussed in chapter 2, an organisation may be understood as having a vertical structure of principals, roles, positions, departments, etc., participating in the workflows which are layered horizontally over this structure. We initially reviewed work in the workflow area, e.g. [Georgakopoulos and Hornick, 1995, Schael, 1996, Cichocki et al., 1997, Grefen et al., 1999] and investigated whether the established concepts could be integrated into our model. We decided not to do this as the definition of yet another workflow model would not have contributed directly to our aim of control principle expression and analysis. However, we noted in the second part of our case study that it would have been desirable to be able to capture the credit application process on a more formal level. The investigation of the theory and concepts underlying organisational workflows and their expression in the context of our framework thus remains as open issue.

11.4.2 Control principles

On the basis of the established control principle model we defined, analysed and explored a set of control principles. The first of these were *separation* controls. Here we mainly referred to existing work based on the concepts of exclusive roles and critical authorisation sets. It is these concepts that we believe should be investigated further.

More specifically, the existing literature remains silent about how exclusive roles and critical authorisation sets are methodically determined. There may be links to areas such as role, permission and more general policy engineering, e.g. [Coyne, 1995, Epstein and Sandhu, 1999, Beznosov, 2000, Roeckle et al., 2000], and we started initial investigations as documented in [Kern et al., 2002]. This, however, requires further improvements in determining, representing and evaluating aspects of organisational structures.

When analysing *delegation* and *revocation* controls for authorisation and obligation policies, we specifically focused on structural constraints as a result of integrating roles. We believe that our discussion is far more detailed than that provided in [Barka and Sandhu, 2000, Zhang et al., 2001], neither of which make a distinction between administrative delegation and *ad hoc* delegation; only insufficiently discuss the effects of delegating authorisations obtained through role-membership; do not effectively consider revocation; and do not provide a concept of delegating obligations. The latter point also applies to the Ponder framework, and we discussed the requirements for delegating obligations in Ponder as part of section 7.8.2.2.

We explicitly distinguished between delegating general and specific *obligations*, and within the context of this thesis this abstraction proved to be useful. Again, it would now be interesting to review in future work whether and how these general and specific obligations may be represented in existing workflow management systems and models. It is also not yet clear whether individual obligations may be delegated following a "push" or "pull" approach. The latter considers whether a principal may actually request an individual obligation being delegated to him. For example, in the context of a sales system we may assume that a clerk requests a specific customer order to be delegated to him if he has the available resources to process it.

Considering the implementation of mechanisms for delegating obligations, the next step would be to review existing workflow products. More specifically, this review should include an evaluation of the support existing workflow systems provide for the definition of obligations. Also the support for defining the rules constraining any delegation activities by means of scripting languages must be evaluated. With respect to delegating obligations it may be advisable to think about the support for the definition of possible protocols a delegation process has to follow. For example, there may be situations where the receiving principal should be able to refuse the acceptance of a delegated obligation.

Our interpretation of an obligation following the event-condition-action paradigm is clearly narrow. To begin with, we do not explicitly address the discharge of obligations but use our abstraction of evidence instead. We have done some initial research on the meaning of discharging an obligation but believe that this requires further investigation, possibly along the line of work presented in [Minsky and Ungureanu, 2000].

Additionally, as discussed in section 5.3.2.3, we exclude the splitting and sharing of obligations. The first is what we consider as policy-refinement, compare [Moffett and Sloman, 1993], which must be carefully assessed with respect to its organisational motivations and support by delegation controls. The sharing of obligations raises questions of who may ultimately

be held to account for a shared obligation, and how the involved principals coordinate their activities. Current research remains silent about this [Cole et al., 2001].

As indicated throughout the thesis, our concept of *review* has limitations. While we believe to have offered a real-world compliant conceptualisation of review as an obligation on a delegated obligation, a more extended discussion on how such a review is performed is missing. One reason for this is our declarative specification approach, ruling out any procedural descriptions. The second reason is that the knowledge about how a review is performed seems to be application specific. We have not yet investigated whether the standardisation of business processes by means of systems such as SAP's R/3 [Scheer, 1994] would leave room for standardised review procedures as well.

We described *supervision* as the generalised obligation to review. This description is problematic, as it may require extensive contextual information to resolve possible ambiguities in what position supervises which position, and which principal ultimately performs the review. It is not yet clear whether patterns in organisational structures as described in [Fowler, 1997] may be used to resolve possible ambiguities.

It should also be investigated how far both the concepts of review and supervision may be expressed through the policy composition features offered by Ponder, as described in section 3.4.1.3. In particular, this investigation would focus on the Ponder notion of relationships, defining how managers in organisational positions interact with each other.

A brief initial review on the necessity of *audit* controls to complement the above principles was given in section 3.5. Here we only focused on the logging character of audit, on the basis of which we defined the history-preserving mechanisms of our model. These were then used to store and retrieve the data required in the context of other controls such as separation or delegation controls. In some future work we would like to investigate the feasibility of general audit obligations. While review is a control that must have been defined before the delegation, it may be left unspecified how the audit is performed. Audit would be a true post-hoc control, and we would not give a detailed instruction on how to perform the audit, but merely state that some delegation activity has occurred which must be investigated.

11.4.3 Tools and integration into systems

11.4.3.1 Alloy

Although Alloy and its automated analysis facilities have been invaluable with respect to the definition of our control principle framework we believe it to have limits. We have indicated some of these throughout the thesis and will provide a more integrated discussion here.

To begin with, Alloy allows for the almost instantaneous checking of a specification for errors, ambiguities and inconsistencies. However, this feature may tempt the user to perceive Alloy as a procedural programming language. This will inevitably lead away from the actual problem, trying to cover all the (often not context-relevant) cases exposed through the declarative behaviour of the language. This 'overspecification' may in turn reduce the performance of

the analysis tool significantly. On the other hand, by trying to achieve a good performance, the specifier may oversimplify his specification, missing out important properties.

The control principle model we defined and documented in appendix B has been compiled, syntax and type checked. However, a more detailed analysis or exploration of specific control principles will require smaller specifications extracted from this general model. It would be desirable to assert such smaller parts of a specification and later compose these without having to re-assert them as, for example, in the PVS verification system [Owre et al., 1995].

If the specification is not too complex, then the Alloy tool will usually present us with a result within seconds or minutes. This means that the first-order logic specification has been translated to conjunctive normal form, which was then made subject to analysis with off-the-shelf SAT solvers. It is the first translation step we observed to be problematic. In particular, the translation of relations with a higher arity seems to be computationally demanding if combined with a more or less complex type system. A good example for this is our initial approach of defining a history signature 5.3, and these technical limitations have thus partially influenced our design.

Although Alloy has been identified as a powerful tool with respect to the definition and analysis of a control principle framework, it has its limitations, both in expressive power and computationally [Jackson, 2001]. Properties requiring higher-order logic cannot be expressed and it cannot be asserted that the precondition of an operation is not stronger than some predicate. Also, a field declared in a subsignature becomes implicitly a field of the supersignature. Thus, two subsignatures cannot declare fields of the same name. Since in our conceptual model defined in section 5.2.2 almost every signature was extended from a general `sig Object`, the definition of meaningful field names was sometimes difficult. With respect to the computational limitations we noticed a reduced performance of the analyser when using specification techniques such as objectification of state and signature extension.

A technique used throughout the thesis is the objectification of state introduced in sections 4.4.4 and 5.2.2. This also heavily influences the performance of the analyser due to the extension of signatures and their relations by a `State` field. Again, for each signature and each relation it needs to be individually considered whether it should be part of a general `State` with respect to the aims of automated analysis. For example, we often excluded administrative changes to relations such as `role_hierarchy` or `exclusive` by not making them part of the `State` signature. In general, it makes sense to constrain states where possible in order to reduce the possible search space. For example, we did this in our case study by clearly defining the assignment of principals to roles in the `has_member` relation.

Also, the frequent use of the signature extension mechanism seems to reduce performance. For a first analysis we thus commented the `extends` keyword out where this did not seem to have any side-effects. With respect to the actual running or checking of a specification, the careful choice of the search parameters will often make the difference between a useful result within seconds or several minutes. If, for example, we are interested in the properties of specific objects declared as static as in the case study, then it does not make sense to define a search space for more than the number of these static objects. Likewise, if we are only

interested in the before and after state of a specific function, then it does not make sense to define search spaces of more than two states.

There were restrictions we encountered that prohibited us from the expression of certain properties. We recall the discussion in section 5.8 of our model and in section 10.2.1 of our case study. There we defined a specific organisational authorisation of reading a customer's account as an Alloy function `fun read_account(s,s':State){...}` and used this function to model state changes. However, we would have also liked to be able to explicitly refer to this function, e.g. as part of a separation property stating that no principal should be directly subject to this authorisation. However, the Alloy language does not support the expression of constraints such as `fact {all p:Principal | p !in read_account.s_subject}`.

11.4.3.2 Prolog

We believe that we have exhausted the possibilities of Alloy and a realistic automated analysis. The further discussion and exploration of the defined control principles requires other approaches and supporting tools, specifically with respect to the issues of discharging an obligation; the performance of a review; the provision of selective reviews; the definition of application specific evidence and review actions on that evidence; and the resolution of ambiguities in the supervision relation. Having established the formal groundwork for our control principle framework in Alloy, we believe that the Prolog language offers us a more flexible and computationally powerful approach for the further declarative exploration of control principles, potentially addressing the issues discussed in sections 11.4.1 and 11.4.2.

Prolog is a powerful declarative language frequently used in the specification and analysis of systems, e.g. [Moffett and Sloman, 1988, Fox et al., 1998, Apt, 1999, Bandara et al., 2003], and we have already used Prolog for an initial exploration of delegation and separation controls in [Schaad, 2001]. Using recursion, the backtracking mechanisms provided by Prolog would allow us to further explore the cascading revocation of delegated policy objects as discussed in section 7.5.

We believe that the next step would be to assess to what extent the concepts defined in Alloy may be translated into Prolog. Having identified the formal properties underlying our control principles, Prolog would now allow us to express and explore these properties in a computationally more powerful environment. We initially experimented with an automated translation of Alloy to Prolog by using Prolog. This was based on the translation approach described in [Clocksin and Mellish, 1996]. There is, however, evidence from similar approaches based on the Z language, suggesting that such a translation is not trivial, e.g. [Moffett, 1990, West and Eaglestone, 1992, West, 1997, Ciancarini et al., 1997, Hewitt et al., 1997]. We did not pursue this any further but argue that a 'manual' translation as reported in [Moffett, 1990] would be sufficient.

Current Prolog programming environments, e.g. AMZI or LPA Prolog, provide a variety of interfaces to database systems and object-oriented/procedural programming languages. This would allow us to further investigate how control principles may be integrated into working

systems. Our initial approach [Schaad and Moffett, 2001], demonstrated how to define a set of separation controls in Prolog, operating on data about principals, roles and authorisations that were held in a relational database. Using a fully fledged database management system avoids potential integrity violations that may occur if the data were held in the form of Prolog rules. A commercial programming language, in this case Visual Basic, was used to access these data and the separation controls from an application.

However, we emphasise again that we still believe that Alloy was the right choice because it initially requires a more rigorous and formal approach than Prolog.

11.5 Conclusion

This thesis has successfully presented a framework which defines, clarifies, compares, categorises, evaluates and integrates a set of organisational control principles. Using the Alloy specification language supported their analysis and exploration. This thesis consists of three main parts:

- A review and discussion of organisational control principles, their origin and relationships, and existing role- and policy-based frameworks and specification languages that partially support their expression.
- A formal model for organisational control principles on the basis of which we defined, analysed, discussed and explored separation controls, delegation and revocation controls for authorisations and obligations, and review and supervision controls.
- A case study consisting of two parts, the first part describing the access control system and applied control principles of a major European bank, the second part using the domain example of a branch of that bank for the validation of our framework.

This framework can potentially be used to investigate the open issues that remain.

Appendix A

Simple Domain Model

```
module domain

//=============================================================================//
//-------------------------------Contents--------------------------------------//
//=============================================================================//

//A simple domain model used for explaining Alloy and the notion
//of objectifying state. Shows how to generate sequences of states
//constrained by defined functions. This extends the work presented in
//[Damianou, 2002] with an unrestricted notion of state.

  run add_object for 3 but 2 State    //Adding an object
  run rem_object for 3 but 2 State    //Removing an object
  run move_object for 4 but 2 State   //Moving an object
  check move_works for 5 but 3 State  //Assert that a move works

//=============================================================================//
//----------------------------Signatures and Facts-----------------------------//
//=============================================================================//

//Object Signature
  sig Object{}

//A domain which is an object and may contain other (domain) objects
  sig Domain extends Object {
      contains: set Object}

//State Signature with a ternary contains relation
  sig State{
      contains: Domain -> Object}

//No cycles in the containment relation over all states
  fact {all s : State | all o : Object | o !in o.^(s.contains)}
```

```
//=============================================================================//
//------------------------------A state sequence-------------------------------//
//=============================================================================//

  sig Domain_Operations {
    disj first, last : State,
    next: (State - last) !->! (State - first)
    }{
    all s : State | s in first.*next
     //pre(first) //Specify any conditions on first state here
     //post(last) //Specify any conditions on last state here
     (all states : State - last | some s = states | some s' = states.next |
        (some obj : Object | some d1,d2 : Domain |
         add_object(s, s', obj, d1)
         ||
         rem_object(s, s', obj, d1)
         ||
         move_object(s, s', obj, d1, d2))
      )
   }

//=============================================================================//
//----------------------------------Functions----------------------------------//
//=============================================================================//

//fun pre (s : State){Specify your preconditions here}
//fun post (s : State){Specify your postconditions here}

//Moving an object
  fun move_object (disj s, s' : State, disj o1,o2,o3 : Object){
    (o2->o1 in s.contains && o3->o1 !in s.contains) &&
    (s'.contains = (s.contains - o2->o1) + o3->o1)}

//Removing an object
  fun rem_object (disj s, s' : State, disj o1,o2 : Object){
    o2->o1 in s.contains && (s'.contains = s.contains - o2->o1)}

//Adding an object
  fun add_object (disj s, s' : State, disj o1,o2 : Object){
    (o2->o1 !in s.contains) && (s'.contains = s.contains + o2->o1)}

//Moving an object is like removing and adding
  assert move_works {all disj s,s',s'' : State |
         all disj o1,o2,o3: Object |
    rem_object(s, s', o1, o2) && add_object(s', s'', o1, o3) =>
     move_object (s, s'', o1, o2, o3)}
```

Appendix B

Control Principle Model

B.1 Structural model

```
module cp_structure

//=============================================================================//
//---------------------------------Contents------------------------------------//
//=============================================================================//

  //This module defines the basic structure of our control principle model as
  //discussed in chapter 5.
  //This includes the basic signatures and the corresponding transformation
  //into a state signature.

  //The module also defines a set of constraints. These include:
    --Constraints on groups
    --Constraints on roles
    --Constraints on principals
    --Constraints on positions
    --Constraints on policies in general
    --Constraints on history

  //We make the important conceptual distinction between general and
  //specific obligations as discussed in section 5.3.2.3. This requires the
  //definition of a set of additional constraints.

  //We also define a set of functions that deliver values we need throughout
  //the other modules as discussed in section 5.10.2
```

```
//================================================================================//
//--------------------------------Basic Signatures--------------------------------//
//================================================================================//

  //The following signatures are the initial, not state-based signatures.
  //These correspond to the control principle model defined in chapter 5.

  sig Object{
   member_of: set Group,
   has_attribute: set Attribute}

  disj sig PolicyObject extends Object{
   target: set Object,
   subject: set Object,
   defines: set Action}

  disj sig Obligation extends PolicyObject{
   specifies: set Evidence,
   has_instance: set ObligationInstance}

  disj sig ObligationInstance extends Obligation{}
  disj sig Authorisation extends PolicyObject{}
       sig Review extends Obligation{}

  disj sig Principal extends Object{
   holds_policy: set PolicyObject,
   has_active: set Role}

  disj sig Role extends Object {
   exclusive: set Role,
   role_hierarchy: set Role,
   has_member: set Principal}

  disj sig Position extends Role{
   supervises: set Role }

  disj sig Evidence extends Object{
   reviewed_by: ReviewAction}

  disj sig Operation{}

  sig Action{
   creates: set Evidence,
   consists_of: Object -> Operation}

  disj sig ReviewAction extends Action{}
  disj sig Group extends Object{}
  disj sig Attribute extends Object{}
```

```
//================================================================================//
//----------------------------------State Signature-------------------------------//
//================================================================================//

  //The above signatures are then transformed into a single state signature.
  sig State{
   s_member_of: Object -> Group,
   s_has_attribute: Object -> Attribute,
   s_target: PolicyObject -> Object,
   s_subject: PolicyObject -> Object,
   s_defines: PolicyObject -> Action,
   s_specifies: Obligation -> Evidence,
   s_holds_policy: Principal -> PolicyObject,
   s_exclusive: Role -> Role,
   s_role_hierarchy: Role -> Role,
   s_has_member: Role -> Principal,
   s_supervises: Position -> Role,
   s_reviewed_by: Evidence -> ReviewAction,
   s_creates: Action -> Evidence,
   s_has_active: Principal -> Role,
   s_assigned_to_role: Principal -> Role,
   s_has_instance: Obligation -> ObligationInstance,
   s_access_history: option AccessHistory,
   s_delegation_history: option DelegationHistory,
   s_revocation_history : option RevocationHistory
  }

  //A Sequence of states. Note that this is declared as static here. If we
  //want to consider several sequences of states, then the static keyword
  //must be removed.
  static sig State_Sequence {
   disj first, last : State,
   next: (State - last) !->! (State - first)}
   {all s : State | s in first.*next}

//================================================================================//
//----------------------------Additional Signatures-------------------------------//
//================================================================================//

  //The signature representing the history of delegations.
  sig DelegationHistory {
   delegating_principal : Principal,
   receiving_principal : Principal,
   based_on_role : option Role,
   delegated_policy_object : PolicyObject
  }
```

```
  //The signature representing the history of access attempts.
  sig AccessHistory {
   accessing_principal : Principal,
   accessed_object : Object,
   used_role : option Role,
   used_authorisation : option Authorisation
  }

 //The signature representing the history of revocations.
 sig RevocationHistory {
  revoking_principal : Principal,
  revoked_principal : Principal,
  revoked_policy_object : PolicyObject
 }

 //The signature representing critical authorisation sets
 sig Critical_Authorisation_Set {
    critical: set Authorisation
 }

//=============================================================================//
//-------------------------------------Facts-----------------------------------//
//=============================================================================//

//-------------------------------Constraints on groups-------------------------//

//Initial and state-based specification of a group membership constraint
  fact {all g: Group | g !in g.member_of}
  fact {all s: State | all g: Group | g !in g.(s.s_member_of)}

//-------------------------------Constraints on roles--------------------------//

//Accyclic role graph, we could also use Alloy's available template
//for partial/total orders.
  fact {all s : State | all r : Role | r !in r.^(s.s_role_hierarchy)}

//No transitive edges
  fact {all s : State | all disj r1,r2,r3 : Role |
   r3 in r1.^(s.s_role_hierarchy) && r2 in r1.(s.s_role_hierarchy) =>
     r2 !in r3.(s.s_role_hierarchy)}

//Exclusive roles must have no common senior
  fact {all s : State | all disj r1, r2, r3 : Role |
   r1->r2 in (s.s_exclusive) =>
    r3 !in (r1.~^(s.s_role_hierarchy)) & r2.~^(s.s_role_hierarchy)}

//A role is not exclusive to itself
  fact {all s : State | all r : Role |
   r !in r.(s.s_exclusive)}
```

```
//If two roles are exclusive in state s then they are exclusive in state s'
  fact {all disj s,s' : State | all disj r1, r2 : Role |
   r1 -> r2 in s.s_exclusive =>
    r1 -> r2 in s'.s_exclusive}

//Mutual exclusion for a role
  fact {all s : State | all disj r1, r2 : Role |
   r1->r2 in (s.s_exclusive) =>
    r2->r1 in (s.s_exclusive)}

//The above constraints are specified without our suggested notion of a
//role graph. If this is desired then they need to be changed, e.g.
  --fact {all s : State | all r : Role | r !in r.^(s.s_role_hierarchy)}
//would have to be changed to
  --fact {all s : State | all r : Role | all rg : RoleGraph|
  --r !in r.^(rg.(s.s_role_hierarchy))}

//---------------------------Constraints on principals---------------------------//

//A principal can only activate a role only if he is member of that role
  fact {all s : State | all p : Principal | all r : Role |
   r in p.(s.s_has_active) => p in r.(s.s_has_member)}

//Extended Role activation constraint. A principal may only be allowed to
//activate a role if he either is a direct member of the role or occupies
//a role which is senior to the role to be activated.
  fact {all s : State | all r2 : Role | all p : Principal |
        r2 in p.(s.s_has_active) =>
          p in r2.(s.s_has_member) ||
          (some r1: Role | (p in r1.(s.s_has_member) &&
                            r2 in r1.^(s.s_role_hierarchy)))
  }

//A principal may only access an object if he holds an authorisation which
//has the object as a target. Must be adjusted to include role hierarchies.
  fact {all s: State | all p: Principal | all s: State | all o: Object |
   some p & s.s_access_history.accessing_principal &&
   some o & s.s_access_history.accessed_object =>
   some s.s_access_history.used_authorisation &
       all_currently_available_policies_over_role(p, s)}

//----------------------------Constraints on positions-------------------------//

//No Position supervises itself, accyclic supervision graph
  fact {all s : State | all p : Position | p !in p.^(s.s_supervises)}

//No transitive edges
  fact {all s : State | all disj p1, p2, p3 : Position |
   p3 in p1.^(s.s_supervises) && p2 in p1.(s.s_supervises) =>
```

```
    p2 !in p3.(s.s_supervises)}
//-----------------------------Constraints on policies--------------------------//

//A policy is assigned to a role iff the policy has the role as a subject
  fact {all s : State | all pol : PolicyObject | all rol : Role |
   (pol-> rol) in s.s_assigned_to_role <=> (pol-> rol) in s.s_subject}

//A principal holds a policy iff the policy has the principal as a subject
  fact {all s : State | all pol : PolicyObject | all prin : Principal |
   (prin -> pol) in s.s_holds_policy <=> (pol -> prin) in s.s_subject}

//No policy has itself as a target -- This might however be desired...
  --fact {all s : State | all pol : PolicyObject |
  --pol !in pol.(s.s_target)}


//-----------------------------Constraints on history---------------------------//

//An authorisation is assigned either/or - no shared history
  fact {all auth : Authorisation | all s : State |
   all r : Role | all p : Principal |
    (some auth.(s.s_subject) & r &&
     p in r.(s.s_has_member)) =>
      no auth.(s.s_subject) & p}

//A state only has one history
  fact {all disj s : State | some s.s_delegation_history =>
   no s.s_revocation_history &&
   no s.s_access_history}

  fact {all disj s : State | some s.s_revocation_history =>
   no s.s_delegation_history &&
   no s.s_access_history}

  fact {all disj s : State | some s.s_access_history =>
   no s.s_delegation_history &&
   no s.s_revocation_history}

  fact {all disj s1, s2 : State |
       no s1.s_revocation_history & s2.s_revocation_history &&
       no s1.s_delegation_history & s2.s_delegation_history &&
       no s1.s_access_history & s2.s_access_history}

//A principal must have the authorisation to access some
//object and subsequently change the history of a state.
  fact {all s : State |  all p : Principal | all s : State | all o : Object |
   some p & s.s_access_history.accessing_principal &&
   some o & s.s_access_history.accessed_object =>
   some (s.s_access_history.used_authorisation &
```

```
        all_available_auth_policies(p, s))}
//---------------Constraints on General/Specific Obligations---------------------//

//Every specific obligation a principal holds must be an instance of a
//general obligation he is a subject of through one of his roles or directly.
  fact {all s: State | all disj p1: Principal | all o: ObligationInstance |
   some o & (s.s_subject).p1 =>
    (s.s_has_instance).o in (s.s_subject).p1  ||
    (s.s_has_instance).o in (s.s_subject).((s.s_has_member).p1)}

//A general obligation can only have a principal or one of his roles as a
//subject, but not both.
  fact {all s: State | all p1: Principal | all o: Obligation - ObligationInstance|
   some p1 & o.(s.s_subject) =>
    no o & (s.s_subject).((s.s_has_member).p1)}

//Only General obligations and authorisations may be assigned to a role
  fact {all s : State |
   s.s_assigned_to_role = (PolicyObject - ObligationInstance) -> Role}

//no policy object has itself as an instance or target
  fact {all pol_obj : PolicyObject | all s : State |
   pol_obj -> pol_obj !in s.s_has_instance &&
   pol_obj -> pol_obj !in s.s_target }

//Target not reflexive
  fact {all disj pol1, pol2 : PolicyObject | all s : State |
   pol1 -> pol2 in s.s_target => pol2 -> pol1 !in s.s_target}

//An obl o1 which has an obl o2 as its instance must not be o2's target
  fact {all pol_obj1 : Obligation | all pol_obj2 : ObligationInstance |
        all s : State |
    pol_obj1 -> pol_obj2  in (s.s_has_instance) =>
       pol_obj2 -> pol_obj1 !in s.s_target}

//Only one layer of instances
  fact {all pol_obj1 : Obligation | all pol_obj2 : ObligationInstance |
        all s : State |
    pol_obj1 -> pol_obj2 in s.s_has_instance =>
       no pol_obj2.(s.s_has_instance)}

//Principal must have role for instance
  fact {all s : State | all p1 : Principal | all o : ObligationInstance |
    o in p1.(s.s_holds_policy) =>
     (s.s_has_instance).o in (s.s_assigned_to_role).((s.s_has_member).p1)}

//No two principals hold the same ObligationInstance. Can't do that over
//holds_policy cardinality as this would also apply to authorisations
  fact {all s : State | all disj p1, p2 : Principal |
```

```
     no p1.(s.s_holds_policy) & p2.(s.s_holds_policy)}
//alternatively we may specify this as...
  --fact {all s : State | all obl : ObligationInstance |
  --one (obl.(s.s_subject) & Principal)}

//A Principal never holds a general obligation directly
//Can't do that in declaration as this would also apply to authorisations
  --fact {all s: State | all p1: Principal | all o: Obligation-ObligationInstance|
  -- o !in p1.(s.s_holds_policy)}

//An ObligationInstance has always one general obligation.
  fact {all s : State | all o : ObligationInstance |
    one ((s.s_has_instance).o & (Obligation - ObligationInstance))}

//An ObligationInstance has only principals as subject.
  fact {all s : State | all o : ObligationInstance |
   o.(s.s_subject) in Principal}

//An ObligationInstance has always one principal.
  fact {all s : State | all o : ObligationInstance | one o.(s.s_subject)}

//A policy a principal holds must not have this principal in its target.
  fact {all s : State | all p : PolicyObject | all prin : Principal |
   p -> prin in s.s_subject => prin -> p !in s.s_target}

//A policy object may only be assigned either directly or over a role
//At least with the existing delegation of authorisations this would cause
//a contradiction eg. p2 already holds o in a role but is assigned directly
//since we wrote s'.s_subject = s.s_subject + o ->p2 ... a contradiction

//=============================================================================//
//-----------------------------------Functions---------------------------------//
//=============================================================================//

//Determines all states before state s
  fun states_so_far(s: State) : set State {
   result = {states : State | states in (
             (State_Sequence.first).*(State_Sequence.next) -
             (s.*(State_Sequence.next)))}}

//Will be true if state s' is after s
  fun after (disj s,s' : State) {
   s' in s.*(State_Sequence.next)}

//---------------------------Functions for principals--------------------------//

//Return currently active roles for a principal
  fun all_currently_active_roles
    (p:Principal,s:State):set Role{
```

```
      result = {role : Role | role in p.(s.s_has_active)}}
//Return all authorisation based on exclusive roles
  fun all_excl_authorisations(p:Principal,s:State):set Authorisation{
     result = {auth : Authorisation |
     auth in (s.s_subject).(all_available_exclusive_roles(p,s))}}

//A variant which does not include any role hierarchies.
//Only authorisations
  fun all_currently_available_policies_over_role
     (p : Principal, s : State) : set Authorisation  {
      result = {auth : Authorisation |
        auth in (s.s_subject).((s.s_has_member).p)}}

//All available authorisation policies
  fun all_available_auth_policies (p:Principal, s:State): set Authorisation{
   result = {auth : Authorisation |
             (auth in (s.s_subject).((s.s_has_member).p)) ||
        (auth in (s.s_subject).p)}}

//All inherited policies
  fun all_inherited_auth_policies (p : Principal, s : State):set Authorisation{
    result = {auth : Authorisation |
      auth in (s.s_subject).(all_inherited_roles(p,s))}}

//All available policies
  fun all_available_policies (p:Principal, s:State): set PolicyObject{
   result = {pol : PolicyObject |
        (pol in (s.s_subject).((s.s_has_member).p)) ||
        (pol in (s.s_subject).p)}}

//------------------------------Functions for roles-----------------------------//

//Return all available roles for a principal
  fun all_available_roles(p:Principal,s:State):set Role{
   result = {role : Role | p in role.(s.s_has_member)}}

//Return all exclusive roles
  fun all_available_exclusive_roles(p:Principal,s:State):set Role{
   result = {role : Role | p in role.(s.s_has_member) &&
                   some role.(s.s_exclusive)}}

//Return all inherited roles
  fun all_inherited_roles(p:Principal,s:State):set Role{
   result = {r : Role | r in ((s.s_has_member).p).^(s.s_role_hierarchy)}}
```

```
//--------------------------------History Functions-----------------------------//

//Returns the set of authorisations used on an object.
  fun return_set_of_authorisations_used_on_object
    (p:Principal,o:Object):set Authorisation {
     result = {auth : Authorisation | some s : State |
                auth in s.s_access_history.used_authorisation &&
                p in s.s_access_history.accessing_principal &&
                o in s.s_access_history.accessed_object}}

//Returns the set of all authorisations used from the states
//where the access happened.
  fun return_set_of_all_authorisations_used
   (p : Principal, cstate : State) : set Authorisation {
     result = {auth : Authorisation | some s: State | some o : Object |
                s in states_so_far(cstate) &&
                auth in s.s_access_history.used_authorisation &&
                p in s.s_access_history.accessing_principal &&
                o in s.s_access_history.accessed_object}}

//Returns the set of all objects accessed by a principal through a role
  fun return_set_of_accessed_objects_through_a_role
   (p: Principal, r : Role) : set Object {
     result = {o : Object | some s : State |
                p in s.s_access_history.accessing_principal &&
                o in s.s_access_history.accessed_object &&
                r in s.s_access_history.used_role }}

//All objects accessed !sofar! through a role
  fun return_set_of_accessed_objects_through_a_role_so_far
   (p: Principal, r : Role, state : State) : set Object {
     result = {o : Object | some s : State |
                s in states_so_far(state) &&
                p in s.s_access_history.accessing_principal &&
                o in s.s_access_history.accessed_object &&
                r in s.s_access_history.used_role }}
```

B.2 Separation controls

```
module separation

open cp_structure

//=============================================================================//
//-----------------------------------Contents----------------------------------//
//=============================================================================//

//This module defines a set of separation controls derived from the
//taxonomies of [Simon and Zurko, 1997] and [Kuhn 1997].

//These are based on the notion of exclusive roles and critical authorisation sets
//and include:
  --Strict/Relaxed separation controls
  --Static/Dynamic Object separation controls
  --Static/Dynamic Operational separation controls
  --History based separation controls

//We also define the degree of shared authorisations
  --Disjoint/Disjoint (complete)
  --Disjoint/Shared
  --Shared/Disjoint
  --Shared/Shared (partial)

//These are further analysed for their validity and composability in the
//analysis module.

//=============================================================================//
//-----------------------------Separation controls-----------------------------//
//=============================================================================//

//------------Strict/Relaxed Separation (as discussed in section 6.2.1)----------//

  fun strict_role_based_separation(){all s : State | all disj r1, r2: Role |
   r1->r2 in (s.s_exclusive) =>
    no r1.(s.s_has_member) & r2.(s.s_has_member)
  }

  fun relaxed_role_based_separation(){all s : State | all disj r1, r2 : Role |
   r1->r2 in (s.s_exclusive) =>
    no (s.s_has_active).r1 & (s.s_has_active).r2
  }
```

```
//--------Static/Dynamic Object Separation (as discussed in section 6.2.2)--------//

  fun static_obj_separation() {all s : State |
                               all disj r1, r2: Role |
                               all prin : Principal |
    r1->r2 in (s.s_exclusive) =>
     no ((Authorisation & (s.s_subject).r1)).(s.s_target) &
        ((Authorisation & (s.s_subject).r2)).(s.s_target)}

  fun static_obj_separation_variant() {all s : State |
                                       all disj r1, r2: Role |
                                       all prin : Principal |
                                       all o : Object |
                                       all disj auth1,auth2: Authorisation |
   some auth1 & (s.s_target).o &&
   some auth2 & (s.s_target).o &&
   some auth1 & (s.s_subject).r1 &&
   some auth2 & (s.s_subject).r2 &&
   some r1 + r2 & (s.s_has_member).prin =>
   no r1->r2 & (s.s_exclusive)
  }

  fun dynamic_obj_separation(cstate: State) {all disj prin : Principal |
                                             all disj r1,r2 : Role |
                                             all disj o : Object |
   some r1 -> r2 & (cstate.s_exclusive) &&
   some prin & r1.(cstate.s_has_member) &&
   some prin & r2.(cstate.s_has_member) =>
    no return_set_of_accessed_objects_through_a_role_so_far(prin,r1,cstate)&
    return_set_of_accessed_objects_through_a_role_so_far(prin,r2,cstate)}

  //The variant defined later in the analysis does not test for membership
    --fun dynamic_obj_separation() {all disj prin : Principal |
    --                              all disj r1,r2 : Role |
    --                              all disj o : Object |
    --                              all disj s : State |
    -- some r1 -> r2 & (s.s_exclusive) &&
    -- some o & return_set_of_accessed_objects_through_a_role(prin, r1) =>
    --   no o & return_set_of_accessed_objects_through_a_role(prin, r2)}

//----Static/Dynamic Operational Separation (as discussed in section 6.2.3)------//

  fun static_op_separation () {all s : State |
                               all p : Principal |
                               all c_set : Critical_Authorisation_Set |
    c_set.critical !in all_excl_authorisations(p, s)
   //c_set.critical !in all_available_auth_policies(p, s)
  }
```

```
  fun dynamic_op_separation (cstate: State) {
                              all p : Principal |
                              all c_set : Critical_Authorisation_Set |
   c_set.critical !in return_set_of_all_authorisations_used(p, cstate)
  }

//------------History Separation (as discussed in section 6.2.4.-----------------//

  fun hist_separation () {all p : Principal |
                           all c_set : Critical_Authorisation_Set |
                           //all s : State |
                           all o : Object |
   //c_set.critical in (all_available_auth_policies(p, s) & Authorisation) =>
   c_set.critical !in return_set_of_authorisations_used_on_object(p,o)
  }

//=============================================================================//
//-------Degree of shared authorisations (as discussed in section 6.3.3)----------//
//=============================================================================//

  fun dd (s:State, disj r1, r2: Role){
   r1->r2 in (s.s_exclusive) =>
    ds(s, r1,r2) &&
    sd(s, r1,r2)
  }

  fun ds (s:State, disj r1, r2: Role){
   r1->r2 in (s.s_exclusive) =>
    ss(s,r1,r2) &&
    no ((s.s_assigned_to_role).r1) & ((s.s_assigned_to_role).r2)
  } // ss() is not really needed but to avoid null role at assertion dsss

  fun sd (s:State, disj r1, r2: Role){
   r1->r2 in (s.s_exclusive) =>
    ss (s,r1,r2) &&
    no (((s.s_assigned_to_role).r1).(s.s_assigned_to_role) -r1 -r2)&&
    no (((s.s_assigned_to_role).r2).(s.s_assigned_to_role) -r1 -r2)
  }

  fun ss (s : State, disj r1, r2: Role){
   r1->r2 in(s.s_exclusive) =>
    some ((s.s_assigned_to_role).r1 - (s.s_assigned_to_role).r2) &&
    some ((s.s_assigned_to_role).r2 - (s.s_assigned_to_role).r1) &&
    (s.s_assigned_to_role).r1 in Authorisation &&
    (s.s_assigned_to_role).r1 in Authorisation
  }
```

```
//---Relationship between shared authorisations and strict/relaxed separation----//

  //Static complete - (Strict complete in our terminology)
    fun strict_complete() { all s : State | all disj r1, r2 : Role |
     r1->r2 in (s.s_exclusive) =>
      strict_role_based_separation() && dd(s,r1,r2)}

  //Dynamic complete - (Dynamic complete in our terminology)
    fun relaxed_complete() {all s : State | all disj r1, r2 : Role |
     r1->r2 in (s.s_exclusive) =>
      relaxed_role_based_separation() && dd(s,r1,r2)}

  //Static partial - (Strict partial in our terminology)
    fun strict_partial () {all s : State | all disj r1, r2 : Role |
     r1->r2 in (s.s_exclusive) =>
      strict_role_based_separation() && ss(s,r1,r2)}

  //Dynamic partial - (Dynamic partial in our terminology)
    fun relaxed_partial () {all s : State | all disj r1, r2 : Role |
     r1->r2 in (s.s_exclusive) =>
      relaxed_role_based_separation() && ss(s,r1,r2)}
```

B.3 Delegation and revocation controls

```
module delegation

open cp_structure

//=============================================================================//
//-----------------------------------Commands----------------------------------//
//=============================================================================//

run general_delegation for 12 but 2 State
run delegate_auth for 10 but 2 State
run delegate_obligation for 12 but 2 State
run delegate_role for 12 but 2 State
run weak_local_revoke for 10 but 2 State

run testing_basic_delegation1 for 12 but 2 State
run testing_basic_delegation2 for 12 but 2 State
run testing_basic_delegation3 for 12 but 2 State

run testing_auth_delegation1 for 12 but 2 State
run testing_auth_delegation2 for 12 but 2 State
run testing_auth_delegation3 for 12 but 2 State

run testing_revocation for 12 but 2 State

check delegation_revocation_sequence for 12 but 2 State

//=============================================================================//
//-----------------------------------Contents----------------------------------//
//=============================================================================//

//This Module defines the delegation and revocation controls of the control
//principle framework.

//We begin with the definition of a general delegation function as discussed in
//section 7.4.1. This general delegation function is then used for
  --the definition of a function for delegating authorisations, discussed in 7.4.2
  --the definition of a function for delegating obligations, discussed in 7.4.3

//We further provide a function for delegating roles as outlined 7.4.4

//Considering revocation controls we only define a weak local revocation function
//since we argued that the remaining types of revocation can be modelled in terms
//of series of stepwise weak local revocations.
//Appendix B.7 will give an example of a simple global (recursive) revocation.
```

```
//=============================================================================//
//-----------Basic Delegation Function (as discussed in 7.4.1)-----------------//
//=============================================================================//
  fun general_delegation(disj s,s' : State, disj p1,p2: Principal,
                         pol : PolicyObject){
   //Case 1: If principal p1 holds pol directly in state s then p2 will
   //have pol in s' while p1 may (not) lose pol.
     (p1 in pol.(s.s_subject) =>
      (s'.s_subject = s.s_subject + pol -> p2 ||
       s'.s_subject = s.s_subject + pol -> p2
                                  - pol -> p1))          &&
   //Case2: If pol is in one of p1's roles then p2 must have pol in s'
   //but p1 retains pol.
     (pol in (s.s_subject).((s.s_has_member).p1) =>
       s'.s_subject = s.s_subject + pol -> p2)           &&
   //Case 3: If p1 does not hold pol in state s then nothing changes in s'
   //with respect to the s_subject relation.
     ((p1 !in pol.(s.s_subject) &&
      pol !in (s.s_subject).((s.s_has_member).p1) =>
       s'.s_subject = s.s_subject))                      &&
   //Frame Conditions
    general_delegation_frame(s,s')}

  fun general_delegation_frame(disj s,s': State){
   //Here it should be specified what frame conditions are supposed to hold.
   //This, however, depends on the given situation. An example for a set of
   //frame conditions are the following three frames:
    s'.s_target = s.s_target &&
    s'.s_defines = s.s_defines &&
    s'.s_exclusive = s.s_exclusive //etc.
   }

  fun no_double_delegation (cstate: State, disj p1, p2: Principal,
                            pol: PolicyObject){
   //A principal may delegate the same object more than once to the same
   //principal. If there is a revocation of the object in between those
   //delegations than this is valid, but if not, then such a second delegation
   //should be ruled out. The following function must be adapted depending on
   //how it is used in combination with a delegation function.
   //If there has been a delegation in the past
    some disj s : states_so_far(cstate) |
     (s.s_delegation_history).delegating_principal = p1 &&
     (s.s_delegation_history).receiving_principal = p2 &&
     (s.s_delegation_history).delegated_policy_object = pol =>
  //Then there must be a revocation
    some s' : states_so_far(cstate) |
     after(s,s') &&
     (s'.s_revocation_history).revoking_principal = p1 &&
     (s'.s_revocation_history).revoked_principal = p2 &&
```

```
    (s'.s_revocation_history).revoked_policy_object = pol} --checked
//=============================================================================//
//------------Authorisation Delegation Function (as discussed in 7.4.2)----------//
//=============================================================================//
  fun delegate_auth(disj s,s': State, disj p1,p2: Principal, aut: Authorisation){
   //Precondition, that authorisation aut is held by p1
     (aut in (s.s_subject).p1 ||
     (aut in (s.s_subject).((s.s_has_member).p1))) &&
   //Delegate using the basic delegation function
     general_delegation(s,s',p1,p2,aut) &&
   //History Updates
     (s.s_delegation_history).delegating_principal = p1 &&
     (s.s_delegation_history).receiving_principal = p2 &&
     (s.s_delegation_history).delegated_policy_object = aut &&
     (some r: Role | (aut in (s.s_subject).p1 =>  //If it was direct
                      no (s.s_delegation_history).based_on_role) &&
                      (aut in (s.s_subject).r &&
                     p1 in r.(s.s_has_member) =>  //If a role was used
                      (s.s_delegation_history).based_on_role = r)) &&
    //Frames
      individual_auth_delegation_frame(s,s')}

  fun individual_auth_delegation_frame(disj s,s':State){
   //Again, this depends on the situation see example of frames in the
   //general_delegation_frame() function
 }

//=============================================================================//
//-------------Obligation Delegation Function (as discussed in 7.4.3)-----------//
//=============================================================================//
  fun delegate_obligation (disj s,s':State, disj p1,p2:Principal, obl:Obligation){
   //Preconditions
     (obl in (s.s_subject).p1 ||
     (obl in (s.s_subject).((s.s_has_member).p1))) &&
   //Delegation
     general_delegation(s,s',p1,p2,obl) &&
   //History
     (s.s_delegation_history).delegating_principal = p1 &&
     (s.s_delegation_history).receiving_principal = p2 &&
     (s.s_delegation_history).delegated_policy_object = obl &&
     (some r : Role | (obl in (s.s_subject).p1 =>
                       no (s.s_delegation_history).based_on_role) &&
                      (obl in (s.s_subject).r && p1 in r.(s.s_has_member) =>
                       (s.s_delegation_history).based_on_role = r)) &&
   //Frames
     individual_obl_delegation_frame(s,s')}

  fun individual_obl_delegation_frame(disj s,s':State){
   //Again, this depends on the situation see example of frames in the
```

```
   //general_delegation_frame() function
  }

//=============================================================================//
//-----Delegation Function for General Obligation (as discussed in 7.4.3)--------//
//=============================================================================//

  fun delegate_general_obligation (disj s,s': State,
                                   disj p1,p2: Principal,
                                   obl: Obligation - ObligationInstance ){
   //Delegate
       (all obl_inst : ObligationInstance |
            ((obl -> obl_inst) in s.s_has_instance &&        --Case 1
             (obl_inst -> p1) in s.s_subject =>
                (s'.s_subject = s.s_subject + (obl -> p2)       -- a)
                                            - (obl_inst -> p1)
                                            + (obl_inst -> p2))||
                (s'.s_subject = s.s_subject + (obl -> p2)       -- b)
                                            - (obl -> p1)
                                            - (obl_inst -> p1)
                                            + (obl_inst -> p2))||
                (s'.s_subject = s.s_subject + (obl -> p2))) &&  -- c)
            ((obl -> obl_inst) !in s.s_has_instance =>       --Case 2
                (s'.s_subject = s.s_subject + (obl -> p2))||    -- a)

                (s'.s_subject = s.s_subject + (obl -> p2)       -- b)
                                            - (obl -> p1)))) &&
   //History Updates
       (s.s_delegation_history).delegating_principal = p1 &&
       (s.s_delegation_history).receiving_principal = p2 &&
       (s.s_delegation_history).delegated_policy_object = obl &&
        some r: Role | (obl in (s.s_subject).p1 =>
                            no (s.s_delegation_history).based_on_role) &&
                       (obl in (s.s_subject).r && p1 in r.(s.s_has_member) =>
                          (s.s_delegation_history).based_on_role = r) //&&
   //Frame Condition
   //delegate_general_obligation_frame (s, s')
  }

//=============================================================================//
//------------------Delegate Roles (as discussed in 7.4.4)---------------------//
//=============================================================================//

  fun delegate_role (disj s,s' : State,
                     disj p1, p2 : Principal,
                     disj r : Role){
   //Precondition
       some ((r -> p1) & (s.s_has_member))  &&
   //Delegation of the role
```

```
     (s'.s_has_member = s.s_has_member  + (r -> p2) ||
      s'.s_has_member = s.s_has_member  + (r -> p2)
                                       - (r -> p1)) &&
   //Change the subject relationships for all obligation instances
   //with respect to delegating/receiving principals. In this case
   //all obligation instances for p1 through the general obligation
   //are delegated to p2.
      all iob : ObligationInstance |
      all gob : Obligation - ObligationInstance |
      gob -> r in s.s_assigned_to_role &&
      iob -> p1 in s.s_subject &&
      gob -> iob in s.s_has_instance =>
       s'.s_subject = s.s_subject - (iob -> p1)
                                  + (iob -> p2) &&
   //Frames and Updates
       role_delegation_frame(s,s') &&
       (s.s_delegation_history).delegating_principal = p1 &&
       (s.s_delegation_history).receiving_principal = p2 &&
       //Note the double use of .based_on_role
       (s.s_delegation_history).based_on_role = r &&
       no (s.s_delegation_history).delegated_policy_object
  }

  fun role_delegation_frame(disj s,s' : State){
   //Again, this depends on the situation see example of frames in the
   //general_delegation_frame() function
  }


//=============================================================================//
//----------------------------The revocation part------------------------------//
//=============================================================================//

  //This part demonstrates the definition of a weak delegation function
  //based on the subset of the signatures, relations and functions and
  //facts of our control principle model.
  //The function considers the various delegations (and revocations) that
  //might have preceded the revocation of a policy object pol from a
  //principal p2 by p1 in a current state cstate

  //The challenge is that unlike in a procedural approach we have no
  //direct information about 'how many times' a principal holds
  //an object as a result of multiple delegations since there are no sets
  //with double entries. This needs to be inferred from the information given
  //in the histories and context such as current state!

  //The weak local revocation function we are going to defined consists of
  //four functions as we discussed in section 7.5.3.1. These consider:
    -- the precondition of a revocation;
```

```
  -- whether a roles was used for a delegation;
  -- whether there were other delegations not related to the revoker;
  -- whether the principal a policy is to be revoked from initially held it.
//The precondition of revocation. This function will evaluate true if
//p1 delegated pol to p2 in a state before the current state cstate and
//did not revoke pol from p2 in between.
  fun revocation_precondition (cstate: State,
                               disj p1, p2: Principal,
                               pol: PolicyObject){
   (some disj s : states_so_far(cstate) |
    (s.s_delegation_history).delegating_principal = p1 &&
    (s.s_delegation_history).receiving_principal = p2 &&
    (s.s_delegation_history).delegated_policy_object = pol &&
      no s' : states_so_far(cstate) |
        after(s,s') &&
        (s'.s_revocation_history).revoking_principal = p1 &&
        (s'.s_revocation_history).revoked_principal = p2 &&
        (s'.s_revocation_history).revoked_policy_object = pol)
  } --checked

//A role was used for the delegation. This function will
//evaluate true if p1 delegated pol to p2 in a state before the
//current state cstate on basis of a role-based assignment to pol.
  fun role_was_used_for_del_of_pol (cstate: State,
                                    disj p1, p2: Principal,
                                    pol: PolicyObject){
   some s : states_so_far(cstate) |
   some r : Role |

   ((s.s_delegation_history).delegating_principal = p1 &&
    (s.s_delegation_history).receiving_principal = p2 &&
    (s.s_delegation_history).based_on_role = r &&
    (s.s_delegation_history).delegated_policy_object = pol) &&

    (no s' : states_so_far(cstate) |  after(s,s') &&
    (s'.s_revocation_history).revoking_principal = p1 &&
    (s'.s_revocation_history).revoked_principal = p2 &&
    (s'.s_revocation_history).revoked_policy_object = pol)
  } --checked

//The object to be revoked was revoked by some other principal.
//This function will evaluate true if some other principal p
//delegated pol to p2 and did not revoke it before the current state
//cstate.
  fun pol_was_delegated_by_other_p (cstate: State,
                                    disj p1, p2: Principal,
                                    pol: PolicyObject){
   some s : states_so_far(cstate) |
   some p : Principal - p1 |
```

```
     ((s.s_delegation_history).delegating_principal = p &&
      (s.s_delegation_history).receiving_principal = p2 &&
      (s.s_delegation_history).delegated_policy_object = pol) &&

      (no s' : states_so_far(cstate) |  after(s,s') &&
      (s'.s_revocation_history).revoking_principal = p &&
      (s'.s_revocation_history).revoked_principal = p2 &&
      (s'.s_revocation_history).revoked_policy_object = pol)
    } --checked

  //The revoked principal held object initially. This function will evaluate
  //true if principal p2 the policy object pol is revoked from already held
  //pol in the first state.
    fun rev_p_held_pol_initially(cstate: State,
                                 disj p1, p2: Principal,
                                 pol: PolicyObject){
     pol -> p2 in ((State_Sequence.first).s_subject)
    }

  //Will update the history  for cstate with the given variables
    fun update_rev_history(cstate: State,
                           disj p1, p2: Principal,
                           pol: PolicyObject){
     (cstate.s_revocation_history).revoking_principal = p1 &&
     (cstate.s_revocation_history).revoked_principal = p2 &&
     (cstate.s_revocation_history).revoked_policy_object = pol
    }

//=============================================================================//
//-------------------Weak Revocation (as discussed in 7.5.3.1)-----------------//
//=============================================================================//

  fun weak_local_revoke (disj s1, s2: State,
                         disj p1, p2: Principal,
                         pol: PolicyObject){
  //Precondition: p1 must have delegated pol to p2 before s1
  revocation_precondition(s1, p1, p2, pol) &&

  //Case I: No role was used for this initial delegation
   (!role_was_used_for_del_of_pol(s1, p1, p2, pol) =>
   //Case I.1: No other delegations occurred
     (!pol_was_delegated_by_other_p(s1, p1, p2, pol) =>
       //Case I.1.a: p2 did hold pol initially
         (rev_p_held_pol_initially(s1, p1, p2, pol)    =>
            update_rev_history(s1, p1, p2, pol) &&
            s2.s_subject = s1.s_subject + pol -> p1) &&
       //Case I.1.b: p2 did not hold pol initially
         (!rev_p_held_pol_initially(s1, p1, p2, pol)   =>
```

```
            update_rev_history(s1, p1, p2, pol) &&
            s2.s_subject = s1.s_subject + pol -> p1
                                        - pol -> p2)
     ) &&
    //Case I.2: Some other delegation occurred
     (pol_was_delegated_by_other_p(s1, p1, p2, pol)  =>
         update_rev_history(s1, p1, p2, pol) &&
         s2.s_subject = s1.s_subject + pol -> p1)
    ) &&
   //Case II: A role was used for this initial delegation
    (role_was_used_for_del_of_pol(s1, p1, p2, pol) =>
    //Case II.1: No other delegations occurred
     (!pol_was_delegated_by_other_p(s1, p1, p2, pol)  =>
       //Case I.1.a: p2 did hold pol initially
         (rev_p_held_pol_initially(s1, p1, p2, pol)    =>
            update_rev_history(s1, p1, p2, pol) &&
            s2.s_subject = s1.s_subject + pol -> p1) &&
       //Case I.1.b: p2 did not hold pol initially
         (!rev_p_held_pol_initially(s1, p1, p2, pol)   =>
            update_rev_history(s1, p1, p2, pol) &&
            s2.s_subject = s1.s_subject + pol -> p1
                                        - pol -> p2)
     ) &&
     //Case II.2: Some other delegations occurred.
     //The two subcases (as in I.1.a/b) do not need to be considered since
     //a role was used for the delegation by p1
       (pol_was_delegated_by_other_p(s1, p1, p2, pol) =>
            update_rev_history(s1, p1, p2, pol) &&
            s2.s_subject = s1.s_subject
       )
    )
   }

//=============================================================================//
//------Delegation and Revocation Sequences (as discussed in section 7.6)--------//
//=============================================================================//

   //This sequence was used to test the behaviour of the function
     fun sequence1 (){some disj s1,s2,s3,s4,s5 : State |
                      some disj p1, p2, p3, p4 : Principal |
                      some disj pol : PolicyObject |
                      some r : Role |
   //Some Initialisations
     one State_Sequence &&
     State_Sequence.first = s1 &&
     s1 -> s2 + s2 -> s3 + s3 -> s4 + s4 -> s5  in State_Sequence.next &&

   //The actual sequence of delegations
   //Could also use delegate() instead but this form is easier to manipulate
```

```
  //In the current example p2 will lose pol while p1 will be assigned with
  //it in state s5. This is because there is no other delegation of pol to
  //p2 and p1 used a direct assignment for the delegation in state s1

    s1.s_subject = pol -> p1 && //&&+ pol -> p2 &&
    s2.s_subject = pol -> p2 &&
    s3.s_subject = pol -> p2 &&
    s4.s_subject = pol -> p2 &&
    //s5.s_subject = pol -> p1  &&

    some s1.s_delegation_history.delegating_principal & p1 &&
    some s1.s_delegation_history.receiving_principal & p2 &&
  //Always state explicitly that there is no role !!!!
    no s1.s_delegation_history.based_on_role &&
    some s1.s_delegation_history.delegated_policy_object & pol &&
  //Always state explicitly if there is no history !!!!
    no s1.s_revocation_history &&

  //some s3.s_delegation_history.delegating_principal & p3 &&
  //some s3.s_delegation_history.receiving_principal & p2 &&
  //some s3.s_delegation_history.based_on_role & r &&
  //some s3.s_delegation_history.delegated_policy_object & pol &&
    no s3.s_revocation_history &&
    no s3.s_delegation_history &&

    some (s2.s_revocation_history).revoking_principal & p3 &&
    some (s2.s_revocation_history).revoked_principal & p2 &&
    some (s2.s_revocation_history).revoked_policy_object & pol &&
    //no s2.s_revocation_history &&
    no s2.s_delegation_history &&
    //no s4.s_revocation_history &&
    no s4.s_delegation_history &&
    no s5.s_revocation_history &&
    no s5.s_delegation_history &&

  //Call the weak local revocation
    weak_local_revoke(s4, s5, p1, p2, pol)
  }

//=============================================================================//
//---------------Some additional tests for delegation and revocation-----------//
//=============================================================================//

  fun testing_basic_delegation1(){some disj p1,p2: Principal |
                                  some disj s,s' : State |
                                  some pol : PolicyObject |
  //Use this to show a direct delegation
   some (pol -> p1) & s.s_subject &&
   no (pol -> p1) & s'.s_subject &&
```

```
 general_delegation(s,s',p1,p2,pol)
} -- Checked 31/10/02

fun testing_basic_delegation2(){some disj p1,p2: Principal |
                                some disj s,s' : State |
                                some pol : PolicyObject |
//Use this to show a role-based delegation
 no (pol -> p1) & s.s_subject &&
 no (pol -> p1) & s'.s_subject &&
 no (pol -> p2) & s.s_subject &&
 some (pol -> p2) & s'.s_subject &&
 general_delegation(s,s',p1,p2,pol)
} -- Checked 31/10/02

fun testing_basic_delegation3(){some disj p1,p2: Principal |
                                some disj s,s' : State |
                                some pol : PolicyObject |
//Use this to show no delegation s'.s_subject = s.s_subject
 pol !in all_available_policies(p1,s) &&
 general_delegation(s,s',p1,p2,pol)
} -- Checked 31/10/02

fun testing_auth_delegation1(){some disj p1,p2: Principal |
                               some disj s,s' : State |
                               some auth: Authorisation |
//Use this to show a direct delegation
 some (auth -> p1) & s.s_subject &&
 no (auth -> p1) & s'.s_subject &&
 delegate_auth(s,s',p1,p2,auth)
}-- Checked 31/10/02

fun testing_auth_delegation2(){some disj p1,p2: Principal |
                               some disj s,s' : State |
                               some auth: Authorisation |
//Use this to show a role-based delegation
 no (auth -> p1) & s.s_subject &&
 no (auth -> p1) & s'.s_subject &&
 no (auth -> p2) & s.s_subject &&
 some (auth -> p2) & s'.s_subject &&
 delegate_auth(s,s',p1,p2,auth)
} -- Checked 31/10/02

fun testing_auth_delegation3(){some disj p1,p2: Principal |
                               some disj s,s' : State |
                               some auth: Authorisation |
                               some disj r1, r2 : Role |
//Use this to show a direct delegation
//We found a bug in our delegate_auth() function and changed
//the universal quantifier all to some when maintaining a history
```

```
//of using roles as basis for delegation
 no (auth -> p1) & s.s_subject &&
 no (auth -> p1) & s'.s_subject &&
 no (auth -> p2) & s.s_subject &&
 some (auth -> p2) & s'.s_subject &&
 some (auth -> r1) & s.s_subject &&
 some (auth -> r2) & s.s_subject &&
 some (r1 -> p1) & s.s_has_member &&
 some (r2 -> p1) & s.s_has_member &&
 delegate_auth(s,s',p1,p2,auth)
} -- Checked 31/10/02

fun testing_revocation(){some disj p1,p2: Principal |
                         some disj s,s',s'' : State |
                         some auth: Authorisation |
//Use this to show a direct delegation
 some s.s_delegation_history.delegating_principal & p1 &&
 some s.s_delegation_history.receiving_principal & p2 &&
 no s.s_delegation_history.based_on_role &&
 some s.s_delegation_history.delegated_policy_object & auth &&

 some (auth -> p1) & s.s_subject &&
 no (auth -> p1) & s'.s_subject &&

 delegate_auth(s,s',p1,p2,auth) &&
 weak_local_revoke(s',s'',p1,p2,auth)
} -- Checked 31/10/02

assert delegation_revocation_sequence{all disj s1,s2,s3 : State |
                                      all disj p1, p2, p3, p4 : Principal |
                                      all auth : Authorisation |

//Some assumptions about our states
  State_Sequence.first = s1 &&
  s1 -> s2 + s2 -> s3 in State_Sequence.next &&

//Some initialisations - Must be adjusted depending on analysis
  (auth -> p1) !in s1.s_subject &&
  auth in all_available_policies  (p1, s1) &&
  auth !in all_available_policies (p2, s1) &&
  auth !in all_available_policies (p3, s1) &&
  auth !in all_available_policies (p4, s1) &&

//The actual assertion of the sequence of delegations
  delegate_auth(s1,s2, p1,p2, auth) &&
  weak_local_revoke(s2,s3, p1,p2, auth) =>
   s1.s_subject = s3.s_subject
} -- Checked 31/10/02
```

B.4 Review and supervision controls

```
module review_and_supervision

open cp_structure
open delegation

//=============================================================================//
//-----------------------------------Commands----------------------------------//
//=============================================================================//
run general_delegation_with_review for 12 but 2 State

//=============================================================================//
//---------------------------------Contents------------------------------------//
//=============================================================================//
  //This module defines the review and supervision controls discussed in chapter 8
  //Since the introduction of a review obligtaion has effects on the subject and
  //target relations, we cannot reuse the delegation functions defined in chapter 7

  //A new general delegation function general_delegation_with_review() is defined
  //here which caters for the delegation of general and obligation instances.

  //A set of constraints in the strcuture of review, evidence and review actions
  //is also part of this module

//=============================================================================//
//--------------Review Constraints (as discussed in section 8.2.3)-------------//
//=============================================================================//
  //A review object may only have an obligation as its target.
    fact {all s : State | all r : Review | r.(s.s_target) in Obligation}

  //A review object only defines review actions, and an obligation which is
  //not a review only defines actions which are not review actions.
    fact {all s : State | all r : Review |
     r.(s.s_defines) in ReviewAction}
    fact {all s : State | all o : Obligation - Review |
     no o.(s.s_defines) & ReviewAction}

  //If some evidence is created by some action then this action must
  //be defined by the obligation specifying the evidence
    fact {all e : Evidence | all s : State |
     (s.s_creates).e = ((s.s_specifies).e).(s.s_defines)}

  //Evidence is reviewed by the actions that are defined by the review
  //that has the obligation specifying the evidence as its target.
    fact {all s: State | all o: Obligation | all e: Evidence | all a: Action |
     a in e.(s.s_reviewed_by) =>
      a in ((s.s_target).((s.s_specifies).e)).(s.s_defines)}
```

```
  //If some review has a target then its actions must review it.
    fact {all s : State | all disj o1, o2 : Obligation |
     o1 in Review &&
     o2 in o1.(s.s_target) =>
      o2.(s.s_specifies) in (s.s_reviewed_by).(o1.(s.s_defines))}

  //An obligation instance is a review if and only if the general obligation
  //is a review.
    fact {all s : State | all o : ObligationInstance |
     o in Review <=> (s.s_has_instance).o in Review}

  //The general obligation of a review instance should have the general
  //obligation of the delegated obligation as its target.
    fact {all s : State |
          all rev : Review & ObligationInstance |
          all o : ObligationInstance |
    o in rev.(s.s_target) =>
      (s.s_has_instance).o in ((s.s_has_instance).rev).(s.s_target)}

//=============================================================================//
//---Obligation Delegation Function with Review (as defined in section 8.2.4)----//
//=============================================================================//

  fun general_delegation_with_review (disj s,s' : State,
                                      disj p1, p2 : Principal,
                                      disj obl : Obligation,
                                      disj rev' : Review) {
   //p1 holds the obligation to be delegated
     (obl in (s.s_subject).p1 ||
     (obl in (s.s_subject).((s.s_has_member).p1))) &&
   //Delegation and assignment of review
   //Case 1: If principal p1 holds obl directly in state s then p2 will
   //have obl in s' while p1 may (not) lose obl.
     (p1 in obl.(s.s_subject) =>
      (s'.s_subject = s.s_subject + obl -> p2
                                  + rev' -> p1 ||
       s'.s_subject = s.s_subject + obl -> p2
                                  - obl -> p1
                                  + rev' -> p1))          &&
   //Case2: If obl is in one of p1's roles then p2 must have obl in s'
   //but p1 retains obl.
     (obl in (s.s_subject).((s.s_has_member).p1) =>
       s'.s_subject = s.s_subject + obl -> p2
                                  + rev' -> p1)           &&
   //The target relationship needs to be updated for the new review obligation
     s'.s_target = s.s_target + rev' -> obl &&
   //There is some evidence after delegation
     some e : Evidence | e in obl.(s'.s_specifies) &&
                         no obl.(s.s_specifies) &&
```

```
   //The review object did not exist in previous states
     new_review(s, rev') &&
   //Other frames and updates
     delegate_obl_with_review_frame(s,s',p1, p2, obl, rev') &&
   //Update of the delegation history
     (s.s_delegation_history).delegating_principal = p1 &&
     (s.s_delegation_history).receiving_principal = p2 &&
     (s.s_delegation_history).delegated_policy_object = obl &&
     (some r : Role | (obl in (s.s_subject).p1 =>  //No role was used
                        no (s.s_delegation_history).based_on_role) &&
                      (obl in (s.s_subject).r &&
                        p1 in r.(s.s_has_member) => //A role was used
                        (s.s_delegation_history).based_on_role = r))
  }

  fun new_review(s : State, rev' : Review){
   // this depends on the situation
  }

  fun delegate_obl_with_review_frame(disj s,s':State, disj p1, p2: Principal,
                                     obl:Obligation, rev':Review){
   //Again, this depends on the situation see example of frames in the
   //general_delegation_frame() function
  }

//=============================================================================//
//-------------------------------Supervision-----------------------------------//
//=============================================================================//

  //If a position pos2 is supervised by a position pos1 then all obligations
  //obl which have pos2 as a subject, must be the target of a review obligation
  //rev which has pos1 as its subject.
    fact {all disj pos1, pos2: Position | all o: Obligation | all s: State |
     pos2 in pos1.(s.s_supervises) &&
     pos2 in o.(s.s_subject) =>
      some rev : Review | rev -> o in (s.s_target) &&
                          pos1 in rev.(s.s_subject)}


  //If a position pos2 is supervised by a position pos1 then all obligations
  //obl which have pos2, or a role occupied by a principal in pos2, as their
  //subject, must be the target of a review obligation rev which has pos1
  //as its subject.
    fact {all disj pos1, pos2: Position | all o: Obligation | all s: State |
     pos2 in pos1.(s.s_supervises) &&
     (pos2 in o.(s.s_subject) ||
     ((s.s_has_member).(pos2.(s.s_has_member))) in o.(s.s_subject)) =>
       some rev : Review | rev -> o in (s.s_target) &&
                           pos1 in rev.(s.s_subject)}
```

B.5 Administrative behaviour

```
module adm_behaviour

open cp_structure

//=============================================================================//
//-----------------------------------Contents----------------------------------//
//=============================================================================//

  //This module defines some examples of the basic administrative activities a
  //principal can perform. This includes:
    --The assignment of principals to roles;
    --The removal of principals from roles;
    --The assignment of supervision relationships between positions;
    --The assignment of exclusive relationships between roles;
    --The assignment of roles in role hierarchies;
    --The assignment of policy objects to roles;

  //These functions are supported by frame conditions, the content of which must
  //be specified dependent on the context.

//=============================================================================//
//-----------------------------------Functions---------------------------------//
//=============================================================================//

//------------------------Principal\Role Assignment----------------------------//

  fun assign_prin_role (disj s, s' : State, prin : Principal, r : Role) {
   (r -> prin) !in s.s_has_member &&
   s'.s_has_member = s.s_has_member + (r -> prin) &&
   prin_role_frame (s',s)
  }

  fun rem_prin_role (disj s, s' : State, prin : Principal, r : Role) {
   s'.s_has_member = s.s_has_member - (r -> prin) &&
   prin_role_frame(s',s)
  }

  fun prin_role_frame(disj s',s : State){
   s'.s_member_of = s.s_member_of &&
   s'.s_target = s.s_target &&
   s'.s_subject = s.s_subject &&
   s'.s_defines = s.s_defines &&
   s'.s_specifies = s.s_specifies &&
   s'.s_holds_policy = s.s_holds_policy &&
   s'.s_exclusive = s.s_exclusive &&
   s'.s_role_hierarchy = s.s_role_hierarchy &&
   //s'.s_has_member = s.s_has_member &&
```

```
   s'.s_supervises = s.s_supervises &&
   s'.s_reviewed_by = s.s_reviewed_by &&
   s'.s_creates = s.s_creates &&
   s'.s_has_active =s.s_has_active &&
   s'.s_assigned_to_role = s.s_assigned_to_role &&
   no s.s_delegation_history &&
   no s.s_revocation_history &&
   no s.s_access_history
  }

//-------------------------Position\Position Assignment-------------------------//

  fun assign_pos_pos (disj s, s' : State, disj pos1, pos2: Position) {
   (pos1->pos2) !in s.s_supervises //Not essential, but gives better results
   s'.s_supervises = s.s_supervises + (pos1->pos2)
   pos_pos_frame(s',s)
  }

  fun rem_pos_pos (disj s, s' : State, disj pos1, pos2: Position) {
   s'.s_supervises = s.s_supervises - (pos1->pos2)
   pos_pos_frame(s',s)
  }

  fun pos_pos_frame(disj s',s : State){
   //As the previous frame, but will depend on context
  }

//-------------------------Role\Role exclusive Assignment-----------------------//

  fun assign_role_role_excl (disj s, s' : State,
                             disj role1, role2: Role ) {
   (role1->role2) !in (s.s_exclusive) &&
   (s'.s_exclusive) = (s.s_exclusive) + (role1->role2) &&
    role_role_excl_frame(s',s)
  }

  fun rem_role_role_excl (disj s, s' : State,
                          disj role1, role2: Role) {
   (s'.s_exclusive) = (s.s_exclusive) - (role1->role2)
    role_role_excl_frame(s',s)
  }

  fun role_role_excl_frame(disj s',s : State){
   //As the previous frame, but will depend on context
   //s'.s_exclusive = s.s_exclusive
  }
```

```
//------------------------Role\Role Hierarchy Assignment-------------------------//

  fun assign_role_role_hierachy (disj s, s' : State,
                                 disj role1, role2: Role) {
   (role1->role2) !in (s.s_role_hierarchy)
   (s'.s_role_hierarchy) = (s.s_role_hierarchy) + (role1->role2)
   role_role_excl_frame(s',s)
  }

  fun rem_role_role_hierachy (disj s, s' : State,
                              disj role1, role2: Role) {
   (s'.s_role_hierarchy) = (s.s_role_hierarchy)  - (role1->role2)
   role_role_hierachy_frame(s',s)
  }

  fun role_role_hierachy_frame(disj s',s : State){
   //As the previous frame, but will depend on context
   //s'.s_role_hierarchy = s.s_role_hierarchy &&
  }

//------------------------PolicyObject\Role Assignment---------------------------//

  fun assign_polobj_role (disj s, s' : State,
                          polobj: PolicyObject, r : Role) {
   (polobj->r) !in s.s_assigned_to_role &&
    s'.s_subject = s.s_subject  + (polobj->r) &&
    polobj_role_frame(s',s)
  }

  fun rem_polobj_role (disj s, s' : State,
                       polobj: PolicyObject, r : Role) {
   s'.s_subject = s.s_subject - (polobj->r) &&
   polobj_role_frame(s',s)
  }

  fun polobj_role_frame(disj s',s : State){
   //As the previous frame, but will depend on context
   //s'.s_subject = s.s_subject &&
  }
```

B.6 User behaviour

```
module user_behaviour

open cp_structure

//=============================================================================//
//----------------------------------Contents-----------------------------------//
//=============================================================================//

  //This module defines the basic activities a principal may perform.
  //This includes:
  --The general access of an object and corresponding update of
  --the history mechanism;
   --The(de)activiation of roles.

//=============================================================================//
//---------------------------------Functions-----------------------------------//
//=============================================================================//

  //A principal p1 accesses an object o uaing auth
  //Note that we only say that he possibly uses one his roles, but if the
  //authorisation has several roles he is a member of then we have to provide
  //additional information.
   fun access_object (disj s, s' : State, disj p1 : Principal,
                      disj auth : Authorisation, disj o : Object){

  //Is the auhtorisation available to him? Yes, then check wether it was
  //directly availbale to him or via a role.
   (auth in all_available_auth_policies (p1, s) =>
    s.s_access_history.accessing_principal = p1 &&
    s.s_access_history.accessed_object = o &&
    s.s_access_history.used_authorisation = auth &&
      (some r : Role | (auth in (s.s_subject).p1 =>
             no (s.s_access_history).used_role)  &&
                      (auth in (s.s_subject).r &&
               p1 in r.(s.s_has_member) =>
             (s.s_access_history).used_role = r)) &&
              access_frame(s,s')
   )&&
  //Is the authorisation available to him? No, then no change to history
   (auth !in all_available_auth_policies (p1, s) =>
    no s.s_access_history.accessing_principal &&
    no s.s_access_history.accessed_object &&
    no s.s_access_history.used_authorisation &&
    no s.s_access_history.used_role
   )
  }
```

```
  //Principal p1 activates a role
   fun principal_role_activation (disj s, s' : State, p1 : Principal,
                                  r : Role, pos : Position) {
    no r & p1.(s.s_has_active) &&
    s'.s_has_active = s.s_has_active + (p1 -> r) &&
    principal_role_activation_frame (s,s')}

  //Principal p1 deactivates a role
   fun principal_role_de_activation (disj s, s' : State, p1 : Principal,
                                     r : Role, pos : Position) {
   some r & p1.(s.s_has_active) &&
   s'.s_has_active = s.s_has_active - (p1 -> r) &&
   principal_role_activation_frame (s,s')}

//=============================================================================//
//-------------------------------------Frames----------------------------------//
//=============================================================================//

  fun access_frame(disj s,s' : State){
   s'.s_target = s.s_target &&
   s'.s_subject = s.s_subject &&
   s'.s_defines = s.s_defines &&
   s'.s_specifies = s.s_specifies &&
   s'.s_exclusive = s.s_exclusive &&
   s'.s_role_hierarchy = s.s_role_hierarchy &&
   s'.s_has_member = s.s_has_member &&
   //....
   no s.s_delegation_history &&
   no s.s_revocation_history
   //Again definition will depend on context
  }

  fun principal_role_activation_frame (disj s',s : State){
   s'.s_target = s.s_target &&
   s'.s_subject = s.s_subject &&
   s'.s_defines = s.s_defines &&
   s'.s_specifies = s.s_specifies &&
   s'.s_holds_policy = s.s_holds_policy &&
   s'.s_exclusive = s.s_exclusive &&
   s'.s_role_hierarchy = s.s_role_hierarchy &&
   s'.s_has_member = s.s_has_member &&
   //....
   no s.s_delegation_history &&
   no s.s_access_history &&
   no s.s_revocation_history
   //Again definition will depend on context
  }
```

B.7 Static and dynamic analysis

```
module dynamic_analysis

open delegation
open cp_structure
open separation
open user_behaviour
open review_and_supervision

//=============================================================================//
//---------------------------------Contents------------------------------------//
//=============================================================================//

  //This module describes the static and dynamic analysis of control principles.

  //As discussed in chapter 7 "Delegation and Revocation" we analysed and
  //explored control principles by means of asserting and checking state
  //sequences. In this module we first specify a general approach to exploration
  //of delegation and revocation controls, followed by more explicit state
  //sequences as defined in section 7.7 showing how:
  --a static operational separation control prevents delegation
  --delegation and revocation of a policy circumvents a static operational control

  //We further analyse and explore
  --separation controls as discussed in section 6.3.1 and 6.3.2
  --and the degree of shared authorisations as discussed in 6.3.3
  --role hierarchies as discussed in section 6.3.4
  --the delegation of a delegated obligation as discussed in 8.2.6.1
  --and the delegation of a review obligation as discussed in section 8.2.6.2

//=============================================================================//
//-----------------------------------Commands----------------------------------//
//=============================================================================//

  //Commands to run check of delegation and revocation assertions
    run some_state_sequences for 8 but 4 State
    run static_operational_separation_prevents_delegation for 8 but 2 State
    run circumvent_static_operational_separation for 8 but 6 State
    run obligation_delegation_chain for 8 but 3 State

  //Commands to run check of separation assertions
    //single assertions
    check strict_role_based_separation_assert for 8 but 2 State
    check static_obj_separation_assert for 8 but 2 State
    check dynamic_obj_separation_assert for 8 but 3 State
    check static_op_separation_assert for 8 but 3 State
    check history_separation_assert for 8 but 3 State
```

```
    //composed assertions
    check strict_separation_implies_relaxed_separation for 8 but 3 State
    check static_obj_separation_implies_dynamic_obj_separation for 8 but 2 State
    check static_op_separation_implies_dynamic_op_separation for 8 but 2 State
    check dynamic_op_separation_implies_history_separation for 8 but 3 State

  //Commands to run check of degree of shared authorisations assertions
    check implication_ddss for 8
    check implication_dsss for 8
    check implication_ddsd for 8
    check implication_ddds for 8
    check implication_sdss for 8

  //Commands to run checks on relationships between shared authorisations
  //and strict/relaxed separation.
    check strict_complete_implies_strict_partial for 8
    check strict_partial_implies_relaxed_partial for 8
    check strict_complete_implies_relaxed_complete for 8
    check relaxed_complete_implies_relaxed_partial for 8

//=============================================================================//
//----------------------A general analysis approach--------------------------//
//=============================================================================//

fun some_state_sequences() {
 //Quantification
   some disj s1,s2,s3, s4, s5, s6  : State |
   some disj s,s': State - State_Sequence.last |
   some obj : Object |
   some disj p1, p2 : Principal |
   some auth : Authorisation |
 //The state sequence
   s1->s2 + s2->s3 + s3->s4 + s4->s5 + s5->s6 in State_Sequence.next &&
 //General constraints
   some s -> s' & State_Sequence.next &&
   static_op_separation() &&
 //Preconditions
 //e.g. no p2 & auth.(s1.s_subject) &&
 //Postconditions
 //e.g. some & auth.(s1.s_subject) &&
 //Functions
   (access_object(s,s',p1,auth,obj)
    || //access an object or delegate it
    delegate_auth(s,s',p1,p2,auth)
    || //or weakly local revoke it
    weak_local_revoke(s,s',p1,p2,auth))
}
```

```
//=============================================================================//
//----Scenario1: A static operational separation control prevents delegation-----//
//=============================================================================//

  fun static_operational_separation_prevents_delegation(){
  //Quantifications
    some disj s1,s2 : State |
    some disj p1, p2 : Principal |
    some disj auth1, auth2, auth3, auth4, auth5 : Authorisation |
    some c_set: Critical_Authorisation_Set|

  //The state sequence
    s1 -> s2 in State_Sequence.next &&

  //Explicit Definition of a critical authorisation set
    c_set.critical = auth1 + auth2 + auth3 + auth4 &&

  //Definition of an explicit dual control for principals p1, p2 on the
  //defined  critical authorisation set
  //  all s:State| c_set.critical = (all_available_auth_policies(p1,s) +
  //                                 all_available_auth_policies(p2,s)) &&

  //Some definitions on the initial state
    no (s1.s_has_member).p1 &&         --No roles for p1
    no (s1.s_has_member).p2 &&         --No roles for p1
    some p1 & auth1.(s1.s_subject) &&  --auth1 assigned to p1
    some p1 & auth2.(s1.s_subject) &&  --auth2 assigned to p1
    some p1 & auth3.(s1.s_subject) &&  --auth3 assigned to p1
    some p2 & auth4.(s1.s_subject) &&  --auth4 assigned to p2
    some p2 & auth5.(s1.s_subject) &&  --auth5 assigned to p2
    no p1 & auth5.(s1.s_subject) &&    --auth5 not assigned to p1
    no p1 & auth4.(s1.s_subject) &&    --auth4 not assigned to p1
    no p2 & auth3.(s1.s_subject) &&    --auth3 not assigned to p2
    no p2 & auth2.(s1.s_subject) &&    --auth2 not assigned to p2
    no p2 & auth1.(s1.s_subject) &&    --auth1 not assigned to p2

  //Static operational separation must hold
    static_op_separation() &&

  //p2 delegates auth4 to p1
    delegate_auth(s1,s2,p2,p1,auth4) //&&

 } --checked 07/11/02
```

```
//=============================================================================//
//-------------Scenario2: Circumvention of static operational controls----------//
//=============================================================================//

  fun circumvent_static_operational_separation(){
  //Quantifications
    some disj s1,s2,s3,s4,s5,s6 : State |
    some disj p1, p2 : Principal |
    some disj auth1, auth2, auth3 : Authorisation |
    some c_set: Critical_Authorisation_Set|
    some o : Object |

  //The state sequence
    s1->s2 + s2->s3 + s3->s4 + s4->s5 + s5->s6 in State_Sequence.next &&

  //Explicit Definition of a critical authorisation set
     c_set.critical = auth1 + auth2 + auth3 &&

  //Some definitions on the initial state
    some p1 & auth1.(s1.s_subject) &&  --auth1 assigned to p1
    some p1 & auth2.(s1.s_subject) &&  --auth2 assigned to p1
    some p1 & auth3.(s1.s_subject) &&  --auth3 assigned to p1
  //  some p1 & auth4.(s1.s_subject) &&  --auth4 assigned to p2

  //Static operational separation must hold
    static_op_separation_variant() &&

  //p1 delegates auth2 to p1
    delegate_auth(s1,s2,p1,p2, auth2) &&
  //p1 accesses o using auth1
    access_object (s2,s3, p1, auth1, o) &&
  //p1 revokes auth2 from p1
    weak_local_revoke(s3,s4,p1,p2, auth2) &&
  //p1 delegates auth2 to p1
    delegate_auth(s4,s5,p1,p2, auth1) &&
  //p1 accesses o using auth2
    access_object (s5,s6, p1, auth2, o)
  } --checked 08/11/02

  //The variant of the static operational separation we initially specified
  fun static_op_separation_variant(){all disj s1, s2 : State |
                  all c_set: Critical_Authorisation_Set |
                      all p1 : Principal |
                  all o : Object |
                  all auth : Authorisation |
      s1 -> s2 in State_Sequence.next &&
      access_object (s1,s2, p1, auth, o) =>
        c_set.critical !in all_available_auth_policies(p1,s1)
  }
```

```
//============================================================================//
//--------------------------Delegation of Obligations-------------------------//
//============================================================================//

  //Run this sequence of delegation operations if you want to observe how:
  //- an obligation is delegated two times (or more)
  //- the review of a delegated obligation is delegated
  //- a combination of the two

  fun obligation_delegation_chain (){some disj s1,s2,s3 : State |
                                     some disj p1, p2, p3 : Principal |
                                     some obl : Obligation |
                                     some disj rev1, rev2 : Review |

  //Some assumptions about our states
   State_Sequence.first = s1 &&
   s1 -> s2 + s2 -> s3 in State_Sequence.next &&

  //Some initialisations - Must be adjusted depending on analysis
   no State_Sequence.first.s_target &&                //No delegations
   no State_Sequence.first.s_specifies &&             //so far and thus no
   no State_Sequence.first.s_reviewed_by &&           //specific evidence
   no State_Sequence.first.s_creates &&               //or any reviews
   no State_Sequence.first.s_defines & Review->Action &&
   no Review.((State_Sequence.first).s_subject) &&
   (obl -> p1) in (State_Sequence.first).s_subject && //p1 has obl

  //The actual sequence of delegations (1)
   -- Principal p1 delegates obligation obl to p2
   general_delegation_with_review (s1, s2, p3, p2, obl, rev1) &&
   -- Principal p1 then delegates obligation obl to p3
   general_delegation_with_review (s2, s3, p2, p1, obl, rev2)

  //The actual sequence of delegations (2)
   -- Principal p1 delegates obligation obl to p2
  //general_delegation_with_review (s1, s2, p1, p2, obl, rev1) &&
   -- Principal p1 then delegates review rev1
  //general_delegation_with_review (s2, s3, p1, p3, rev1, rev2)
  }
```

```
//=============================================================================//
//---------------------------Separation Assertions-----------------------------//
//=============================================================================//

//----------------------Strict/Relaxed Separation Assertion----------------------//

  assert strict_role_based_separation_assert {all p1 : Principal |
                                              all disj r1, r2 : Role |
                                              all s : State |
   some r1 -> r2 & (s.s_exclusive) &&
   p1 in r1.(s.s_has_member) &&
   strict_role_based_separation() =>
    no p1 & r2.(s.s_has_member)} --checked

  --This will only apply if done in combination with the activation rule
  --requiring a principal to be a member of roles he activates.
   assert strict_separation_implies_relaxed_separation {
   strict_role_based_separation() =>
    relaxed_role_based_separation()} --checked

//----------------------Object-based Separation Assertion-----------------------//

  assert static_obj_separation_assert{all disj p1 : Principal |
                                      all disj r1, r2 : Role |
                                      all disj s : State |
                                      all disj o : Object |
   some r1 -> r2 & (s.s_exclusive) &&
   static_obj_separation() &&
   some o & (Authorisation & (s.s_subject).r1).(s.s_target) =>
     no o & (Authorisation & (s.s_subject).r2).(s.s_target)
  } --checked (no counterexample)

  --the dynamic object separation as we defined it requires a current state
  --as its input value. By quantifying over all states and making these the
  --input values we effectively consider every state.
   assert dynamic_obj_separation_assert{all disj p1 : Principal |
                                        all disj r1, r2 : Role |
                                        all disj s : State |
                                        all disj o : Object |
   some r1 -> r2 & (s.s_exclusive) &&
   dynamic_obj_separation(s) =>
    no o & (return_set_of_accessed_objects_through_a_role(p1, r1) &
            return_set_of_accessed_objects_through_a_role(p1, r2))
  } --checked (no counterexample)

  assert static_obj_separation_implies_dynamic_obj_separation{all s : State |
   static_obj_separation() => dynamic_obj_separation(s)
  } --checked
```

```
//----------------------Operational Separation Assertion-------------------------//

  assert static_op_separation_assert{all s : State |
                                     all p1 : Principal |
                                     all c_set: Critical_Authorisation_Set |
                                     all disj a1, a2, a3 : Authorisation |
   some p1 & Principal &&
   a1 + a2 + a3 = c_set.critical &&
   some a1 & all_available_auth_policies(p1, s) &&
   some a2 & all_available_auth_policies(p1, s) &&
   static_op_separation() =>
     no a3 & all_available_auth_policies(p1, s)}

  assert static_op_separation_implies_dynamic_op_separation{all s : State |
   some Critical_Authorisation_Set &&
   some Principal &&
   some State &&
   static_op_separation() => dynamic_op_separation(s)
  } --checked

//-------------------------History Separation Assertion--------------------------//

  assert history_separation_assert {all s : State |
                                    all p1 : Principal |
                                    all o : Object |
                                    all c_set: Critical_Authorisation_Set |
                                    all disj a1, a2, a3 : Authorisation |
   some p1 & Principal &&
   a1 + a2 + a3 = c_set.critical &&
   (a1 + a2 + a3) in all_available_auth_policies(p1, s) &&
   hist_separation() =>
   (a1 + a2 + a3) !in return_set_of_authorisations_used_on_object(p1,o)}

   assert dynamic_op_separation_implies_history_separation {all s : State |
    //This will return a counterexample since a principal may have used
    //all critical authorisations but not on the same object.
    dynamic_op_separation(s) => hist_separation()}

//----------------------Degree of shared authorisations-------------------------//
  assert implication_ddds {all s : State | all disj r1, r2 : Role |
   dd(s, r1, r2) => ds(s, r1, r2)}
  assert implication_ddsd {all s : State |  all disj r1, r2 : Role |
   dd(s, r1, r2) => sd(s, r1, r2)}
  assert implication_sdss {all s : State |  all disj r1, r2 : Role |
   sd(s, r1, r2) => ss(s, r1, r2)}
  assert implication_dsss {all s : State |  all disj r1, r2 : Role |
   ds(s, r1, r2) => ss(s, r1, r2)}
  assert implication_ddss {all s : State |  all disj r1, r2 : Role |
   dd(s, r1, r2) => ss(s, r1, r2)}
```

```
//-----Relat.ship between shared authorisations and strict/relaxed separation----//
  assert strict_complete_implies_strict_partial {
   strict_complete() => strict_partial()}
  assert strict_partial_implies_relaxed_partial {
   strict_partial() => relaxed_partial()}
  assert strict_complete_implies_relaxed_complete {
   strict_complete() => relaxed_complete()}
  assert relaxed_complete_implies_relaxed_partial {
   relaxed_complete() => relaxed_partial()}

//=============================================================================//
//----------------------------Role Hierarchy Analysis--------------------------//
//=============================================================================//

//----------------------Role-based separation assertion-------------------------//

  //This is under the assumption that role hierarchies are defined as
  //senior roles may activate junior roles
   assert role_based_separation_assert_variation {all s : State |
                                                  all disj r1, r2 : Role |
                                                  all p1 : Principal |
    r1 -> r2 in (s.s_exclusive) &&
     strict_role_based_separation() =>
      no r2 & p1.(s.s_has_active) &&
      no r1 & p1.(s.s_has_active)
  }

  //This then however requires an extended activation rule
  //fact {all s : State |
  //      all disj r1, r2 : Role |
  //      all p : Principal |
  // r2 in p.(s.s_has_active) => p in r2.(s.s_has_member) ||
  //                   (p in r1.(s.s_has_member) &&
  //                r2 in r1.^(s.s_role_hierarchy))
  //}

//--------Hierarchy aware definition of static operational separation------------//

  //This is under the assumption that role hierarchies are defined as
  //senior roles inherit authorisations from junior roles
  fun static_op_separation_hierarchy_variant () {all s : State |
                                                 all p : Principal |
                                            all c_set : Critical_Authorisation_Set |
   c_set.critical !in (all_available_auth_policies(p, s) +
                       all_inherited_auth_policies (p,s))
  }
```

B.8 A simple recursive revocation control

```
module recursive_revocation

//=============================================================================//
//---------------------------------Contents------------------------------------//
//=============================================================================//
  //This module describes as simple recursive revocation by means of
  //an explicit sequence of prior delegations. It is outside the defined
  //structure of our framework but uses the same naming conventions.

  //The most important issue that are clarified here are:
  // a) How to keep track of any possible recursive revocations through a static
  //    state CState
  // b) To only allow our revocation function to perform updates on CState
  // c) To provide a frame condition to obtain the state after the delegation

  run sequence for 6 but 1 Object, 4 Principal
//=============================================================================//
//-----------------------------------Signatures--------------------------------//
//=============================================================================//
  sig Principal{}
  static sig RevPrincipal extends Principal{}

  sig Object{}

  static sig State_Sequence {
   disj first, last : State,
   next: (State - last) !->! (State - first)} {all s : State | s in first.*next}

  sig State{
   holds: Principal -> Object,
   s_delegation_history: option DelegationHistory}

  static sig CState extends State{}

  sig DelegationHistory {
   delegating_principal : Principal,
   receiving_principal : Principal,
   delegated_object : Object}

//=============================================================================//
//--------------------------Supporting functions-------------------------------//
//=============================================================================//
  //Make sure that a principal p and an object o are only in CState if at some
  //stage o was delegated (in)directly to p by the revoking principal RevPrincipal
  fact {all p : Principal | all o : Object |
   some p -> o & CState.holds => (true2(RevPrincipal,p,o))}
```

```
  //Use CState to now update from state s to s'
  fun frame (s,s' : State) { all p : Principal | all o : Object |
   p -> o in s.holds &&
   p -> o !in CState.holds =>
     p -> o in s'.holds
   else
     p -> o !in s'.holds}

  //Is it true that at some stage a principal rec_p was delegated an object o from
  //a principal del_p either directly or via delegation path stemming from del_p?
  fun true2 (disj del_p,rec_p: Principal, o: Object){
    some prin : Principal | some s : State - CState |
       ((s.s_delegation_history).delegating_principal = del_p &&
        (s.s_delegation_history).receiving_principal = rec_p &&
        (s.s_delegation_history).delegated_object = o )
       ||
       ((s.s_delegation_history).delegating_principal = prin &&
        (s.s_delegation_history).receiving_principal = rec_p &&
        (s.s_delegation_history).delegated_object = o  &&
        true2 (del_p, prin , o))
  }

  //Basic Delegation function
  fun delegate(disj s,s' : State, disj p1, p2 : Principal, o : Object){
   s'.holds = s.holds + p2 -> o}

//=============================================================================//
//-----------------------------Recursive Revocation----------------------------//
//=============================================================================//
  //The recursive revocation function
  fun revoke(disj state: State, disj p1,p2: Principal, o: Object) {
   all disj s,s' : State - CState |
   all p : Principal |
   //Case1: No further delegation by the revoked principal
     ((s.s_delegation_history).delegating_principal = p1 &&
      (s.s_delegation_history).receiving_principal = p2 &&
      (s.s_delegation_history).delegated_object = o  =>
        p2 -> o in CState.holds ) &&
   //Case2: Further delegation and thus subsequent revocations
     ((s.s_delegation_history).delegating_principal = p1 &&
      (s.s_delegation_history).receiving_principal = p2 &&
      (s.s_delegation_history).delegated_object = o &&
      (s'.s_delegation_history).delegating_principal = p2 &&
      (s'.s_delegation_history).receiving_principal = p &&
      (s'.s_delegation_history).delegated_object = o &&
       s -> s' in (State_Sequence.next) =>
        revoke (s,p2,p,o) )
  }
```

```
//=============================================================================//
//------------------------------A Delegation Sequence--------------------------//
//=============================================================================//

  //A sequence of delegations to check the revocation function
   fun sequence(){some disj p1,p2,p3,p4 : Principal |
                  some disj s1,s2,s3,s4,s5 : State |
                  some o : Object |
   //CState is not in sequence
      CState !in (s1+s2+s3+s4+s5) &&
   //Setting up the sequence
      one State_Sequence &&
      State_Sequence.first = s1 &&
      s1 -> s2 + s2 -> s3 + s3 -> s4  in State_Sequence.next &&
   //Initialisations
      s1.holds = p1 -> o &&
      s2.holds = p1 -> o + p2 -> o &&
      s3.holds = p1 -> o + p2 -> o + p3 -> o &&
      s4.holds = p1 -> o + p2 -> o + p3 -> o + p4 ->o &&
   //The delegations we assume to have happened
      some s1.s_delegation_history.delegating_principal & p1 &&
      some s1.s_delegation_history.receiving_principal & p2 &&
      some s1.s_delegation_history.delegated_object & o &&

      some s2.s_delegation_history.delegating_principal & p2 &&
      some s2.s_delegation_history.receiving_principal & p3 &&
      some s2.s_delegation_history.delegated_object & o &&

      some s3.s_delegation_history.delegating_principal & p3 &&
      some s3.s_delegation_history.receiving_principal & p4 &&
      some s3.s_delegation_history.delegated_object & o &&

      no s4.s_delegation_history &&
      no s5.s_delegation_history &&
      no CState.s_delegation_history &&
   //Just to check again that out delegation function works
      delegate(s1,s2,p1,p2,o) &&
      delegate(s2,s3,p2,p3,o) &&
      delegate(s3,s4,p3,p4,o) &&
   //The revoking principal is p2 and he revokes in state s4
      RevPrincipal = p1 &&
      revoke(s4, p1, p2, o) &&
      frame(s4,s5)
  }
```

Appendix C

Case Study

```
module case_study

open cp_structure
open delegation
open review_and_supervision
open separation

//=============================================================================//
//---------------------------------Contents------------------------------------//
//=============================================================================//

  //This is the formal specification supporting our case study in chapter 10. It
  //uses the example of the credit application process as described in 9.9.2

  //We begin with the identification and modelling of the objects/signatures that
  //are involved and their relationships. This will also include the definition of
  //exclusive roles, critical authorisation sets and role hierarchies.

  //This rest supports our discussion on
    --possible separation controls in section 10.3.
    --delegation of policy objects in section 10.4
    --review and supervision controls in section 10.5

//=============================================================================//
//---------------------Context Specific Signatures-----------------------------//
//=============================================================================//
  disj sig Account extends Object {}

  disj sig Contract extends Object {
   contract_value : Value}

  disj sig Value{}
  static disj sig High, Low extends Value {}
```

```
//================================================================================//
//---------------------Extended Disjoint Static Signatures----------------------//
//================================================================================//
  static disj sig Branch_West extends Attribute{}

  static disj sig Principal_A, Principal_B, Principal_C,
                  Principal_D, Principal_E, Principal_F,
                  Principal_G, Principal_H, Principal_I,
                  Credit_Application extends Principal{}

  static disj sig share_trader,
                  financial_advisor,
                  financial_clerk,
                  service_clerk extends Role{}

  static disj sig Head_of_Branch,
                  Senior,
                  Junior extends Position{}

  static disj sig account_Smith,
                  account_Miller extends Account {}

  static disj sig contract_Smith,
                  contract_Miller extends Contract {}

  static disj sig Auth_Sign_Per_Proc,
                  Auth_Trade_Shares,
                  Auth_Approve_Credit_Contract,
                  Auth_Initial_Consultation,
                  Auth_Evaluate_Credit,
                  Auth_Alter_Credit_Contract,
                  Auth_Account_Handling extends Authorisation{}

  static disj sig Obl_Initial_Consultation,
                  Obl_Evaluate_Credit,
                  Obl_Alter_Credit_Contract,
                  Obl_Approve_Credit_Contract,
                  Obl_Account_Handling extends Obligation {}

  static disj sig Obl_Inst_Evaluate_Credit_Smith,
                 Review_Obl_Inst_Evaluate_Credit_Smith extends ObligationInstance{}

  static disj sig Obl_Review_Initial_Consultation,
                  Obl_Review_Evaluate_Credit,
                  Obl_Review_Alter_Credit_Contract,
                  Obl_Review_Approve_Credit_Contract,
                  Obl_Review_Account_Handling extends Review{}
```

```
  fact {Review_Obl_Inst_Evaluate_Credit_Smith in Review}

  static disj sig read_account_action extends ReviewAction {}

  static disj sig Credit_Application_Critical_Authorisation_Set_1,
                  Credit_Application_Critical_Authorisation_Set_2,
                  Credit_Application_Critical_Authorisation_Set_3
                  extends Critical_Authorisation_Set{}

//=============================================================================//
//------------------Establishing the relationships between objects-------------//
//=============================================================================//

//----------------------------Principal/Role assignment-------------------------//
  fact {all s : State |
        financial_advisor.(s.s_has_member) = Principal_A +
                                             Principal_B +
                                             Principal_C &&
        financial_clerk.(s.s_has_member) = Principal_D +
                                           Principal_E +
                                           Principal_F &&
        service_clerk.(s.s_has_member) = Principal_G +
                                         Principal_H +
                                         Principal_I &&
        share_trader.(s.s_has_member) = Principal_B}

//-----------------------Principal/Position assignment--------------------------//
  fact {all s : State |
        Head_of_Branch.(s.s_has_member) = Principal_A &&
     Senior.(s.s_has_member) = Principal_B +
                               Principal_C &&
     Junior.(s.s_has_member) = Principal_D +
                               Principal_E +
                               Principal_F +
                               Principal_G +
                               Principal_H +
                               Principal_I }

//--------------------Authorisation/Role/Position assignment-------------------//
  //This would not allow for any delegation/change a sequence of models
  //If this is required we should defined that these assignments are only
  //valid in 'State_Sequence.first'
  fact {all s : State |
    Auth_Sign_Per_Proc.(s.s_subject) = Principal_A &&
    Auth_Trade_Shares.(s.s_subject) = share_trader &&
    Auth_Alter_Credit_Contract.(s.s_subject) = financial_advisor &&
        Auth_Approve_Credit_Contract.(s.s_subject) = financial_advisor &&
        Auth_Evaluate_Credit.(s.s_subject) = financial_clerk &&
    Auth_Initial_Consultation.(s.s_subject) = financial_clerk &&
```

```
        Auth_Account_Handling.(s.s_subject) = service_clerk }
//-------------------------Obligation/Role/Position assignment-----------------//
  //This would not allow for any delegation/change a sequence of models
  //If this is required we should defined that these assignments are only
  //valid in 'State_Sequence.first'
  fact {all s : State |
     Obl_Initial_Consultation.(s.s_subject) = financial_clerk &&
     Obl_Evaluate_Credit.(s.s_subject) = financial_clerk &&
     Obl_Alter_Credit_Contract.(s.s_subject) = financial_advisor &&
     Obl_Approve_Credit_Contract.(s.s_subject) = financial_advisor &&
     Obl_Account_Handling.(s.s_subject) = service_clerk &&

     Obl_Review_Initial_Consultation.(s.s_subject) = financial_clerk &&
     Obl_Review_Evaluate_Credit.(s.s_subject) = financial_clerk &&
     Obl_Review_Alter_Credit_Contract.(s.s_subject) = financial_advisor &&
     Obl_Review_Approve_Credit_Contract.(s.s_subject) = financial_advisor &&
     Obl_Review_Account_Handling.(s.s_subject) = service_clerk
  }

//---------------------Definition of authorisation targets----------------------//
  fact {all s : State |
    Auth_Sign_Per_Proc.(s.s_target) = Contract &&
    Auth_Trade_Shares.(s.s_target) = Account &&
    Auth_Approve_Credit_Contract.(s.s_target) = Account + Contract &&
    Auth_Alter_Credit_Contract.(s.s_target) = Account + Contract &&
    Auth_Evaluate_Credit.(s.s_target) = Account + Contract &&
    Auth_Initial_Consultation.(s.s_target) = Account + Contract &&
        Auth_Account_Handling.(s.s_target) = Account
  }

//----------------------Definition of obligation targets------------------------//
  //Is not required for this case study but would be as for authorisations

//--------------------------Definition of exclusive roles----------------------//
  fact {share_trader in financial_advisor.exclusive}

//-------------------------Definition of role hierarchy-------------------------//
  fact {financial_clerk in financial_advisor.role_hierarchy}

//----------------------Definition of supervision hierarchy--------------------//
  fact {Junior in Senior.role_hierarchy &&
        Senior in Head_of_Branch.role_hierarchy}

//--------------------Definition of critical authorisation sets----------------//
  fact {Credit_Application_Critical_Authorisation_Set_1.critical =
        Auth_Initial_Consultation +
        Auth_Evaluate_Credit +
        Auth_Alter_Credit_Contract +
        Auth_Approve_Credit_Contract}
```

```
  fact {Credit_Application_Critical_Authorisation_Set_2.critical =
        Auth_Evaluate_Credit +
        Auth_Alter_Credit_Contract +
        Auth_Account_Handling}

  fact {Credit_Application_Critical_Authorisation_Set_3.critical =
        Auth_Trade_Shares +
        Auth_Evaluate_Credit +
        Auth_Alter_Credit_Contract +
        Auth_Approve_Credit_Contract}

//=============================================================================//
//--------------Functions that reflect the actions of our framework--------------//
//=============================================================================//

  fun read_account (disj s,s' : State, p : Principal, ac : Account) {
   //only update the object history
   s.s_access_history.accessing_principal = p &&
   s.s_access_history.accessed_object = ac &&
   s.s_access_history.used_authorisation = Auth_Initial_Consultation
  }

  fun evaluate_account (disj s,s' : State, p : Principal, account : Account){
   //only update the object history
   s.s_access_history.accessing_principal = p &&
   s.s_access_history.accessed_object = account &&
   s.s_access_history.used_authorisation = Auth_Evaluate_Credit
  }

  fun check_credit_history (disj s,s': State,p: Principal,account: Account){
   //only update the object history
   s.s_access_history.accessing_principal = p &&
   s.s_access_history.accessed_object = account &&
   s.s_access_history.used_authorisation = Auth_Evaluate_Credit
  }

  fun generate_contract (disj s,s': State,p:Principal,contract: Contract) {
   //only update the object history
   s.s_access_history.accessing_principal = p &&
   s.s_access_history.accessed_object = contract &&
   s.s_access_history.used_authorisation = Auth_Evaluate_Credit
  }

  fun alter_contract (disj s,s' : State,p:Principal,contract : Contract) {
   //only update the object history
   s.s_access_history.accessing_principal = p &&
   s.s_access_history.accessed_object = contract &&
   s.s_access_history.used_authorisation = Auth_Alter_Credit_Contract
```

```
  }

  fun approve_contract (disj s,s': State,p: Principal,contract : Contract) {
   //only update the object history
   s.s_access_history.accessing_principal = p &&
   s.s_access_history.accessed_object = contract &&
   s.s_access_history.used_authorisation = Auth_Approve_Credit_Contract
  }

//=============================================================================//
//-------------------Case Study specific functions and facts-------------------//
//=============================================================================//

  //Only principals in the position of a Head_of_Branch may be directly
  //assigned with the authority to sign per procuration.
   fact {some p : Principal | some s : State |
    p in Auth_Sign_Per_Proc.(s.s_subject) =>
     p in Head_of_Branch.(s.s_has_member)}

  //A principal may only approve a contract with a high value if he occupies
  //the Head_of_Branch position.
   fact {some p: Principal | some disj s,s':State | some contract_1:Contract|
    approve_contract (s,s', p, contract_1) &&
    contract_1.contract_value = High =>
     p in Head_of_Branch.(s.s_has_member)}

  //The role of a financial_advisor may only be assigned to principals in the
  //position of a \texttt{Senior} or any position supervising a Senior.
  //stateless...
   fact {all p : Principal | some pos : Position |
    p in financial_advisor.has_member =>
     (p in Senior.has_member) ||
     (p in pos.has_member && Senior in pos.^supervises)}
  //or with states...
   fact {all s : State | all p : Principal | some pos : Position |
    p in financial_advisor.has_member =>
     (p in Senior.(s.s_has_member)) ||
     (p in pos.(s.s_has_member) && Senior in pos.^(s.s_supervises))}

  //Role sets for positions are disjoint.
  //stateless...
   fact { no ((Senior.has_member).~has_member - Position) &
             ((Junior.has_member).~has_member - Position)}
  //or with states...
   fact {all s : State |
    no ((s.s_has_member).(Senior.(s.s_has_member)) - Position) &
       ((s.s_has_member).(Junior.(s.s_has_member)) - Position)}
```

```
 //A simple scenario in which Principal B delegates the Auth_Trade_Shares
 //to Principal C, this will then contradict our static_op_separation
 //because of the critical authorisation set 3.
  fun credit_application_scenario_1 (){some disj s1, s2 : State |
   s1 -> s2 in State_Sequence.next &&
 //Delegation Step 1:
   delegate_auth(s1,s2, Principal_B, Principal_C, Auth_Trade_Shares) &&
   static_op_separation()
 }

 //Only allow for the delegation of policy objects
 //between principals occupying a position within the same branch.
 //stateless...
 fact {all disj s, s' : State | all disj p1, p2 : Principal |
  delegate_auth(s,s',p1, p2, Object) =>
     Branch_West in ((p1.~has_member) - Role).has_attribute &&
     Branch_West in ((p2.~has_member) - Role).has_attribute}
 //or with states...
 fact {all disj s, s' : State | all disj p1, p2 : Principal |
  delegate_auth(s,s',p1, p2, Object) =>
    Branch_West in (((s.s_has_member).p1) - Role).(s.s_has_attribute) &&
    Branch_West in (((s.s_has_member).p2) - Role).(s.s_has_attribute)}

//Only allow for the delegation of authority from a principal to some other
//principal in the same or a lower position.
//stateless...
 fact {some pos1, pos2: Position |
       all disj p1, p2: Principal | all disj s1, s2: State |
  delegate_auth(s1, s2, p1, p2, Authorisation) =>
   p1 in pos1.has_member &&
   p2 in pos2.has_member &&
  (pos1 in pos2 || pos2 in pos1.supervises)}
//or with states...
 fact {some pos1, pos2: Position |
       all disj p1, p2: Principal | all disj s, s': State |
  delegate_auth(s, s', p1, p2, Authorisation) =>
   p1 in pos1.(s.s_has_member) &&
   p2 in pos2.(s.s_has_member) &&
   (pos1 in pos2 || pos2 in pos1.(s.s_supervises))}

 //Delegation of the obligation instance
 //Review_Obl_Inst_Evaluate_Credit_Smith from Principal_B to Principal_C
 //must result in the following:
 fun delegation_of_evaluation_obligation (){some disj s1, s2 : State |
   //The actual delegation of the obligation instance
   general_delegation_with_review(s1,s2,
                            Principal_B, Principal_C,
                           Obl_Inst_Evaluate_Credit_Smith,
```

```
        Review_Obl_Inst_Evaluate_Credit_Smith) =>
    //The delegated obligation must be reviewed
      Obl_Inst_Evaluate_Credit_Smith in
      Review_Obl_Inst_Evaluate_Credit_Smith.(s2.s_target) &&
    //By reading the evaluated contract as the evidence
      contract_Smith in (s2.s_reviewed_by).read_account_action &&
      some contract_Smith & Evidence &&
    //The review obligation instance is held by Principal B
      Review_Obl_Inst_Evaluate_Credit_Smith.(s2.s_subject) = Principal_B &&
    //The delegated obligation instance is held by Principal C
      Obl_Inst_Evaluate_Credit_Smith.(s2.s_subject) = Principal_C
  }

//=============================================================================//
//--------------An example of a general supervision relationship-----------------//
//=============================================================================//

  //All principals that occupy a Senior position and have a obligation
  //instance to evaluate a credit must have that instance reviewed by their
  //next senior, here the Head of Branch.
  fact {all p : Principal | all s : State |
        all obl_inst : Obl_Approve_Credit_Contract & ObligationInstance |
   p in Senior.(s.s_has_member) &&
   p in obl_inst.(s.s_subject) =>
    some review_inst: Obl_Review_Approve_Credit_Contract & ObligationInstance|
      Head_of_Branch.(s.s_has_member) in review_inst.(s.s_subject) &&
      obl_inst in review_inst.(s.s_target)}

//=============================================================================//
//---------------Resolving ambiguities in the supervision relation-------------//
//=============================================================================//

  //If a principal is member of the Junior position and the
  //financial_clerk role then any of his obligation instances is
  //reviewed by Principal_B. If a principal is member of the Junior
  //position and the service_clerk role then any of his obligation instances
  //is reviewed by Principal_C.
  fact {all s : State | all p : Principal |
   some (p & Junior.(s.s_has_member) & financial_clerk.(s.s_has_member)) =>
    ((s.s_subject).p & ObligationInstance - Review) in
    ((s.s_subject).Principal_B & Review & ObligationInstance).(s.s_target) &&

   some (p & Junior.(s.s_has_member) & service_clerk.(s.s_has_member)) =>
    ((s.s_subject).p & ObligationInstance - Review) in
    ((s.s_subject).Principal_C & Review & ObligationInstance).(s.s_target)}
```

List of References

[Agostini and De Michelis, 2000] Agostini, A. and De Michelis, G. (2000). A Light Workflow Management System Using Simple Process Models. *In Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9(3-4):335–363.

[Agostini et al., 1994] Agostini, A., De Michelis, G., Grasso, M. A., and Patriarca, S. (1994). Reengineering a Business Process with an Innovative Workflow Management System: a Case Study. (Extended Version). *Collaborative Computing*, 1:163–190.

[Ahn, 2000] Ahn, G. (2000). *RCL 2000.* PhD thesis, George Mason University.

[Ahn and Sandhu, 1999] Ahn, G. and Sandhu, R. (1999). The RSL99 language for role-based separation of duty constraints. In *Proceedings of the fourth ACM workshop on role-based access control*, pages 43–54.

[Amoroso, 1994] Amoroso, E. (1994). *Fundamentals of Computer Security Technology.* Prentice Hall, New Jersey.

[Anderson, 1972] Anderson, J. (1972). Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Vols. I and II, Air Force Electronic Systems Division.

[Anderson, 2001] Anderson, R. (2001). *Security Engineering.* John Wiley & Sons.

[Anton, 1996] Anton, A. (1996). Goal-Based Requirements Analysis. In *2nd IEEE Conferenec on Requirements Engineering*, pages 136–144, Colorado Springs, Colorado.

[Ao et al., 2002] Ao, X., Minsky, N., and Nguyen, T. (2002). A Hierarchical Policy Specification Language, and Enforcement Mechanism, for Governing Digital Enterprises. In *3rd IEEE Workshop on Policies for Distributed Systems and Networks*, Monterey, CA, USA.

[Apt, 1999] Apt, K. (1999). *The logic programming paradigm: a 25 year research perspective.* Springer, Berlin.

[Atluri and Huang, 1996] Atluri, V. and Huang, W. (1996). An Authorization Model for Workflows. *Lecture Notes in Computer Science*, 1146:44–64.

[Awischus, 1997] Awischus, R. (1997). Role based access control with the security administration manager (SAM). In *Proceedings of the second ACM workshop on Role-based access control*, pages 61–68, Fairfax, Virginia, USA.

[Bacon and Moody, 2002] Bacon, J. and Moody, K. (2002). Toward Open, Secure, Widely Distributed Services. *Communications of the ACM*, 45(6):59–64.

[Bacon et al., 2000] Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O., and Spiteri, M. (2000). Generic support for distributed applications. *IEEE Computer*, 33(3):68–76.

[Bacon et al., 2002] Bacon, J., Moody, K., and Yao, W. (2002). A Model of OASIS Role-Based Access Control and its Support for Active Security. *ACM Transactions on Information and System Security (TISSEC)*, 5(4):492–540.

[Bailey et al., 1983] Bailey, A., Gerlach, J., Preston McAfee, R., and Whinston, A. (1983). An OIS model for internal accounting control evaluation. *ACM Transactions on Information Systems*, 1(1).

[Baldwin, 1990] Baldwin, R. (1990). Naming and Grouping Privileges to Simplify Security Management in Large Databases. In *IEEE Symposium on Security and Privacy*, pages 116–132, Oakland.

[Balzert, 1996] Balzert, H. (1996). *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, Heidelberg.

[Bandara et al., 2003] Bandara, A., Lupu, E., and Russo, A. (2003). Using Event Calculus to Formalise Policy Specification and Analysis. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks.*, Como, Italy.

[Barka, 2002] Barka, E. (2002). *Framework for Role-based Delegation Models*. PhD thesis, George Mason University.

[Barka and Sandhu, 2000] Barka, E. and Sandhu, R. (2000). Framework for Role-Based Delegation Models. In *IEEE Annual Computer Security Applications Conference*, New Orleans, LA, USA.

[Beckert et al., 2002] Beckert, B., Keller, U., and Schmitt, P. (2002). Translating the Object Constraint Language into First-order Predicate Logic. In *VERIFY, Workshop at Federated Logic Conference (FLoC)*, Copenhagen, Denmark.

[Bell and La Padula, 1973] Bell, D. and La Padula, L. (1973). Secure Computer Systems: Mathematical Foundations. Technical Report MTR 2547 Vol. 1, MITRE Corporation.

[Belokosztolszki and Moody, 2002] Belokosztolszki, A. and Moody, K. (2002). Meta-Policies for Distributed Role-Based Access Control Systems. In *3rd IEEE Workshop on Policies for Distributed Systems and Networks*, Monterey, CA, USA.

[Bertino et al., 2001] Bertino, E., Bonatti, P., and Ferrari, E. (2001). TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191 – 233.

[Bertino et al., 1997a] Bertino, E., Ferrari, E., and Atluri, V. (1997a). A flexible model supporting the specification and enforcement of role-based authorization in workflow management systems. In *Proceedings of the second ACM workshop on Role-based access control*, pages 1–12, Fairfax, Virginia, USA.

[Bertino et al., 1999a] Bertino, E., Ferrari, E., and Atluri, V. (1999a). The specification and enforcement of authorization constraints in workflow management systems. *Transactions on Informations Systems Security*, 2(1):65–104.

[Bertino et al., 1999b] Bertino, E., Jajodia, S., and Samarati, P. (1999b). A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems*, 17(2):101–140.

[Bertino et al., 1997b] Bertino, E., Samarati, P., and Jajodia, S. (1997b). An Extended Authorization Model for Relational Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(1):85–101.

[Bezevin and Muller, 1998] Bezevin, J. and Muller, P., editors (1998). *The Unified Modeling Language - UML98*, volume 1618 of *Lecture Notes in Computer Science*. Springer, Mulhouse, France.

[Beznosov, 2000] Beznosov, K. (2000). *Engineering Access Control for Distributed Enterprise Applications*. PhD thesis, Florida International University.

[Biba, 1975] Biba, K. (1975). Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, Mitre Corporation.

[Biddle, 1979] Biddle, B. (1979). *Role Theory: Expectations, Identities and Behaviours.* Academic Press, New York.

[BIS, 1998] BIS (1998). Framework for Internal Control Systems in Banking Organisations. Technical Report Basel Committee Publications No. 40, Bank for International Settlements, Basle Committee on Banking Supervision.

[Blau and Scott, 1962] Blau, P. and Scott, W. (1962). *Formal Organizations*. Chandler, San Francisco.

[Bowen, 1996] Bowen, J. (1996). *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, International Thomson Publishing.

[Bratko, 2000] Bratko, I. (2000). *Prolog Programming for Artificial Intelligence*. Longman, 3rd edition.

[Brewer and Nash, 1989] Brewer, D. and Nash, M. (1989). The Chinese Wall Security Policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA.

[Buehner, 1994] Buehner, R. (1994). *Betriebswirtschaftliche Organisationslehre*. Oldenbourg, Muenchen.

[Castano et al., 1994] Castano, S., Fugini, M., Martella, G., and Samarati, P. (1994). *Database Security*. Addison Wesley.

[Cengarle and Knapp, 2001] Cengarle, M. and Knapp, A. (2001). A Formal Semantics for OCL 1.4. *UML2001*, 2185.

[Chen and Sandhu, 1995] Chen, F. and Sandhu, R. (1995). Constraints for RBAC. In *1st ACM Workshop on Role-Based Access Control*, pages 39–46, Gaithersburg, MD.

[Child, 1988] Child, J. (1988). *Organization: A Guide to Problems and Practice*. Paul Chapman, 2nd edition.

[Chomicki and Lobo, 2001] Chomicki, J. and Lobo, J. (2001). Monitors for History-Based Policies. *Policies for Distributed Systems and Networks*, 1995.

[Chomicki et al., 2000] Chomicki, J., Lobo, J., and Naqvi, S. (2000). A Logic Programming Approach to Conflict Resolution in Policy Management. In *KR2000: Principles of Knowledge Representation and Reasoning*.

[Ciancarini et al., 1997] Ciancarini, P., Cimato, S., and Mascolo, C. (1997). Engineering Formal Requirements: an Analysis and Testing Method for Z Documents. *Annals of Software Engineering*, 3:189–219.

[CICA, 1995] CICA (1995). Criteria of Control (CoCo). Technical report, The Canadian Institute of Chartered Accountants.

[Cichocki et al., 1997] Cichocki, A., Helal, A., Rusinkiewicz, M., and Woelk, D. (1997). *Workflow and Process Automation*. Kluwer Academic, Boston.

[Clark and Wilson, 1987] Clark, D. and Wilson, D. (1987). A Comparison of Commercial and Military Security Policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, California.

[Clark and Wilson, 1988] Clark, D. and Wilson, D. (1988). Evolution of a Model for Computer Integrity. *Working Draft Paper distributed at the 11th National Computer Security Conference, Baltimore, October 1988*.

[Clark and Warmer, 2002] Clark, T. and Warmer, J. (2002). *Object Modeling with the OCL: The Rationale Behind the Object Constraint Language*. Lecture Notes in COmputer Science 2263. Springer.

[Clarke et al., 2000] Clarke, E., Grumberg, O., and Peled, D. (2000). *Model Checking*. The MIT Press.

[Clarke and Wing, 1996] Clarke, E. and Wing, J. (1996). Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643.

[Clocksin and Mellish, 1996] Clocksin, W. and Mellish, C. (1996). *Programming in Prolog*. Springer, 4th edition.

[Colbert and Bowen, 1996] Colbert, J. and Bowen, P. (1996). A Comparison of Internal Controls: COBIT, SAC, COSO and SAS 55/78. *IS Audit and Control Journal*, IV:26–35.

[Cole et al., 2001] Cole, J., Derrick, J., Milosevic, Z., and Raymond, K. (2001). Author Obliged to Submit Paper before 4 July: Policies in an Enterprise Specification. In *Policies for Distributed Systems and Networks*, volume 1995, Bristol, UK. Springer Lecture Notes.

[CommonCriteria, 1999] CommonCriteria (1999). Common Criteria for Information Technology Security Evaluation V 2.1. Technical report, In the UK: Communications-Electronics Security Group Compusec Evaluation Methodology.

[COSO, 1992] COSO (1992). Internal Control - Integrated Framework. Technical report, Committee of the Sponsoring Organisations (COSO) of the Treadway Commission.

[Covington et al., 2001] Covington, M., Long, W., Srinivasan, S., Key, A., Ahamad, M., and Abowd, G. (2001). Securing Context-Aware Applications Using Environment Roles. In *6th ACM Symposium on Access Control Models and Technologies*, Chantilly, Virginia, USA.

[Coyne, 1995] Coyne, E. (1995). Role Engineering. In *1st ACM Workshop on Role-based Access Control*, pages 115–116, Gaithersburg.

[Crampton and Loizou, 2002] Crampton, J. and Loizou, G. (2002). Administrative Scope and Role Hierarchy Operations. In *7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Naval Postgraduate School, Monterey, CA, USA.

[Cresson Wood, 1990] Cresson Wood, C. (1990). Principles of Secure Information Systems Design. *Computers & Security*, 9:13–24.

[Dalton and Lawrence, 1971] Dalton, G. and Lawrence, P. (1971). *Motivation and Control in Organizations*. Irwin, Homewood.

[Damianou, 2002] Damianou, N. (2002). *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College, UK.

[Damianou et al., 2001] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. (2001). The Ponder Policy Specification Language. In *Policies for Distributed Systems and Networks*, volume 1995, pages 18–38, Bristol. Springer Lecture Notes in Computer Science.

[Dardenne et al., 1993] Dardenne, A., Lamsweerde, A., and Fickas, S. (1993). Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20(1-2):3–50.

[Date, 1994] Date, C. (1994). *An Introduction to Database Systems.* Addison Wesley, 6th edition.

[Davies and Woodcock, 1996] Davies, J. and Woodcock, J. (1996). *Using Z: Specification, Refinement and Proof.* Prentice Hall International Series in Computer Science.

[Denning, 1976] Denning, D. (1976). A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243.

[Dobson, 1993] Dobson, J. (1993). New Security Paradigms: What Other Concepts Do We Need as Well? In *1st New Security Paradigms Workshop*, Little Compton, Rhode Island. IEEE Press.

[Douglas, 1983] Douglas, I. (1983). *Audit and Control of Systems Software*. N.C.C., Manchester.

[Douglas, 1995] Douglas, I. (1995). *Computer Audit and Control Handbook*. Butterworth-Heinemann.

[Dunlop et al., 2002] Dunlop, N., Indulska, J., and Raymond, K. (2002). Dynamic Conflict Detection in Policy-Based Management Systems. In *6th International Enterprise Distributed Object Computing Conference EDOC 2002*, Lausanne, Switzerland.

[Dupuy et al., 1998] Dupuy, S., Ledru, Y., and Chabre-Peccoud, M. (1998). Translating the OMT dynamic model into Object-Z. *11th International Conference of Z Users (ZUM'98)*, 1493.

[Dupuy et al., 2000] Dupuy, S., Ledru, Y., and Chabre-Peccoud, M. (2000). An overview of RoZ - a tool for integrating UML and Z specifications. In *12th Conference on Advanced information Systems Engineering (CAiSE'2000)*.

[Dwyer et al., 1998] Dwyer, M., Avrunin, G., and Corbett, J. (1998). Property Specification Patterns for Finite-State Verification. In *2nd Workshop on Formal Methods in Software Practice*, pages 7–15.

[Dwyer et al., 1999] Dwyer, M., Avrunin, G., and Corbett, J. (1999). Patterns in Property Specifications for Finite-State Verification. In *21st International Conference on Software Engineering*, Los Angeles, California.

[Elliot, 1980] Elliot, D. (1980). The organization as a system. In Salaman, G. and Thompson, K., editors, *Control and Ideology in Organizations*, pages 85–102. The Open University Press.

[Epstein and Sandhu, 1999] Epstein, P. and Sandhu, R. (1999). Towards a UML based approach to role engineering. In *4th ACM Workshop on Role-based Access Control*, pages 135–143, Fairfax, Virginia, USA.

[Etzioni, 1964] Etzioni, A. (1964). *Modern Organizations*. Prentice Hall, N. J.

[Evans et al., 2000] Evans, A., Kent, S., and Selic, B., editors (2000). *3rd International Conference on the Unified Modeling Language*, volume 1939. Springer Lecture Notes, York, UK.

[Fagin, 1978] Fagin, R. (1978). On an Authorization Mechansism. *ACM Transactions on Database Systems*, 3(3):310–319.

[Fayol, 1949] Fayol, H. (Transl. Stours, C. (1949). *General and Industrial Management*. Pitman, London.

[Ferraiolo et al., 1995] Ferraiolo, D., Cugini, J., and Kuhn, D. (1995). Role-Based Access Control (RBAC): Features and Motivations. In *Computer Security Applications*, pages 241–248.

[Ferraiolo and Kuhn, 1992] Ferraiolo, D. and Kuhn, R. (1992). Role-Based Access Control. In *15th MNCSC National Computer Security Conference*, pages 554–563, Baltimore.

[Ferraiolo et al., 2001] Ferraiolo, D., Sandhu, R., Gavrila, S., Kuhn, D., and Chandramouli, R. (2001). Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and Systems Security*, 4(3).

[Fowler, 1997] Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Addison Wesley Object-Oriented Software Engineering Series. Addison Wesley.

[Fox et al., 1998] Fox, M., Barbuceanu, M., Gruninger, M., and Lin, J. (1998). An Organizational Ontology for Enterprise Modeling. In Prietula, M., Carley, K., and Gasser, L., editors, *Simulating Organizations - Computational Models of Institutions and Groups*. MIT Press.

[France and Rumpe, 1999] France, R. and Rumpe, B., editors (1999). *UML99 - The Unified Modeling Language*, volume 1723. Springer Lecture Notes, Fort Collins, CO, USA.

[Galbraith, 1977] Galbraith, J. (1977). *Organization Design*. Addison-Wesley.

[Gallaire et al., 1984] Gallaire, H., Minker, J., and Nicolas, J. (1984). Logic and Databases: A Deductive Approach. *Computing Surveys*, 16(2):153–185.

[Gallegos et al., 1999] Gallegos, F., Manson, D., and Allen-Senft, S. (1999). *Information Technology Control and Audit*. CRC Press.

[Georgakopoulos and Hornick, 1995] Georgakopoulos, D. and Hornick, M. (1995). Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153.

[George, 1995] George, E. (1995). *Report of the Banking Supervision Inquiry into the Circumstances of the Collapse of Barings*. Ordered by the House of Commons, The Stationery Office, London.

[Giuri, 1995] Giuri, L. (1995). A new model for role-based access control. In *11th Annual Computer Security Application Conference*, pages 249–255, New Orleans, LA.

[Gligor et al., 1998] Gligor, V., Gavrila, S., and Ferraiolo, D. (1998). On the Formal Definition of Separation-of-Duty Policies and their Composition. In *IEEE Symposium on Security and Privacy*, pages 172–185, Oakland, CA.

[Gray and Schach, 2000] Gray, J. G. and Schach, S. R. (2000). Constraint Animation Using an Object-Oriented Declarative Language. In *38th Annual ACM SE Conference*, pages 1–10, Clemson, SC.

[Grefen et al., 1999] Grefen, P., Pernici, B., and Sanchez, G. (1999). *Database Support for Workflow Management*. Kluwer Academic, Boston.

[Griffiths and Wade, 1976] Griffiths, P. and Wade, B. (1976). An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, 1(3):243–255.

[Hagstrom et al., 2001] Hagstrom, A., Jajodia, S., Parisi-Presicce, F., and Wijesekera, D. (2001). Revocations - A Categorization. In *Computer Security Foundations Workshop*. IEEE Press.

[Harrison et al., 1976] Harrison, M., Ruzzo, W., and Ullman, J. (1976). Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471.

[Hayton et al., 1998] Hayton, R., Bacon, J., and Moody, K. (1998). Access Control in an Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA.

[Hewitt et al., 1997] Hewitt, M., O'Halloran, C., and Sennett, C. (1997). Experiences with PiZA, an Animator for Z. *Lecture Notes in Computer Science*, 1212:37–51.

[Hickson and McCullough, 1980] Hickson, D. and McCullough, A. (1980). Power in Organizations. In Salaman, G. and Thompson, K., editors, *Control and Ideology in Organizations*. The Open University Press.

[Hopwood, 1974] Hopwood, A. (1974). *Accounting and Human Behaviour*. Prentice Hall, London.

[Huth and Ryan, 2000] Huth, M. and Ryan, M. (2000). *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press.

[ISACA, 2000] ISACA (2000). COBIT Framework - Control Objectives for Information and related Technology. Technical report, Information Systems Audit and Control Association and Foundation (ISACA).

[ITU, 1994] ITU (1994). Basic reference model of open distributed procesing, part 2: Descriptive model. Technical Report ISO/IEC JTC1/SC21, ITU-T X.902 ISO/IEC 10746-2.

[Jackson, 2000] Jackson, D. (2000). Automating First-Order Relational Logic. In *ACM SIGSOFT Conference Foundations of Software Engineering*, San Diego.

[Jackson, 2001] Jackson, D. (2001). A Micromodularity Mechanism. In *8th Joint Software Engineering Conference*, Vienna, Austria.

[Jackson, 2002] Jackson, D. (2001/2002). Micromodels of Software: Lightweight Modelling and Analysis with Alloy. Technical report, Software Design Group, MIT Lab for Computer Science.

[Jackson and Fekete, 2001] Jackson, D. and Fekete, A. (2001). Lightweight Analysis of Object Interactions. In *Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan.

[Jackson and Rinard, 2000] Jackson, D. and Rinard, M. (2000). Software Analysis: A Roadmap. In Finkelstein, A., editor, *The Future of Software Engineering*. ACM Press.

[Jackson et al., 2000] Jackson, D., Schechter, I., and Shlyakhter, I. (2000). Alcoa: the Alloy Constraint Analyzer. In *International Conference on Software Engineering*, Limerick, Ireland.

[Jacobson, 1995] Jacobson, I. (1995). *The Object Advantage: Business Process Reengineering with Object Technology*. Addison Wesley.

[Jajodia, 1997] Jajodia, S. (1997). *Integrity and Internal Control in Information Systems*. Kluwer Academic Publishers.

[Jajodia et al., 1997a] Jajodia, S., Samarati, P., and Subrahmanian, V. (1997a). A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–43, Oakland, CA.

[Jajodia et al., 1997b] Jajodia, S., Samarati, P., Subrahmanian, V., and Bertino, E. (1997b). A Unified Framework for Enforcing Multiple Access Control Policies. In *ACM SIGMOD Conference on Management of Data*, Tucson, Arizona.

[Janssen et al., 1999] Janssen, W., Mateescu, R., Mauw, S., Fennema, P., and Stappen, P. (1999). Model Checking for Managers. *Lecture Notes in Computer Science*, 1680.

[Jayaratna, 1994] Jayaratna, N. (1994). *Understanding and Evaluating Methodologies: NIMSAD, A Systematic Framework*. Mc Graw-Hill, London.

[Johnson and Gill, 1993] Johnson, P. and Gill, J. (1993). *Management Control and Organizational Behaviour*. Paul Chapman Publishing.

[Jones et al., 1976] Jones, A., Lipton, R., and Snyder, L. (1976). A linear time algorithm for deciding security. In *17th Annual Symposium on Foundations of Computer Science*.

[Jonscher, 1998] Jonscher, D. (1998). *Access Control in Object-Oriented Federated Database Systems*. PhD thesis, University of Zurich.

[Katz and Kahn, 1966] Katz, D. and Kahn, R. (1966). *Social Psychology of Organizations*. Wiley.

[Kern et al., 2002] Kern, A., Kuhlmann, M., Schaad, A., and Moffett, J. (2002). Observations on the Role life-cycle in the context of Enterprise Security Management. In *7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Monterey, CA.

[Kern et al., 2003] Kern, A., Schaad, A., and Moffett, J. (2003). An Administration Concept for the Enterprise Role-Based Access Control Model. In *8th ACM Symposium on Access Control Models and Technologies (SACMAT)*.

[Knight, 1977] Knight, K., editor (1977). *Matrix Management: A Cross-functional Approach to Organisation*. Gower Press, Westemead, UK.

[Knorr and Weidner, 2001] Knorr, K. and Weidner, H. (2001). Analyzing Separation of Duties in Petri Net Workflows. *Lecture Notes in Computer Science*, 2052.

[KPMG, 2002] KPMG (2002). Fraud Survey Reports 1996-2002. Technical Report http://www.kpmg.ca/english/about/press/, KPMG International Canada.

[Krishnakumar and Sloman, 2001] Krishnakumar, K. and Sloman, M. (2001). Constraint-Based Configuration of Proxylets for Programmable Networks. *Lecture Notes in Computer Science*, 2158.

[Kuhn, 1997] Kuhn, R. (1997). Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of the second ACM workshop on Role-based access control*, pages 23–30.

[Lampson, 1971] Lampson, B. (1971). Protection. In *5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University (Reprinted in ACM Operating Systems Review 8(1): 18-24, 1974).

[Lamsweerde, 2000a] Lamsweerde, A. (2000a). Formal Specification: a Roadmap. In Finkelstein, A., editor, *The Future of Software Engineering*. ACM Press.

[Lamsweerde, 2000b] Lamsweerde, A. (2000b). Requirements Engineering in the Year00: A Research Perspective. In *22nd International Confernce on Software Engineering*, Limerick.

[Lamsweerde, 2001] Lamsweerde, A. (2001). Goal-Oriented Requirements Engineering: A Guided Tour. In *International Joint Conference on Requirements Engineering*, pages 249–263, Toronto.

[Liu, 1999] Liu, M. (1999). Deductive database languages: problems and solutions. *ACM Computing Surveys*, 31(1).

[Lobo and Naqvi, 1999] Lobo, J. and Naqvi, R. (1999). A Policy Description Language. In *American Association for Artificial Intelligence / IAAI*, pages 291–298.

[Lorsch, 1970] Lorsch, J. (1970). Introduction to the structural design of organizations. In Dalton, G., editor, *Organisation Structure and Design*. Irwin, Homewood.

[LSE, 1999] LSE (1999). Internal Control - Guidance for Directors on the Combined Code. Technical Report ISBN 1841520101, Institute of Chartered Accountants.

[Lupu, 1998] Lupu, E. (1998). *A Role-Based Framework for Distributed Systems Management*. PhD thesis, Imperial College, UK.

[Lupu and Sloman, 1997] Lupu, E. and Sloman, M. (1997). A Policy Based Role Object Model. In *EDOC97*.

[Lupu and Sloman, 1999] Lupu, E. and Sloman, M. (1999). Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management*, 25(6):852–869.

[Marriott et al., 1994] Marriott, D., Mansouri-Samani, M., and Sloman, M. (1994). Specification of Management Policies. In *5th IFIP/IEEE Int. Workshop on Distributed Systems Operations and Management (DSOM '94)*, Toulouse, France.

[Marshall, 2002] Marshall, A. (2002). A Financial Institution's Legacy Mainframe Access Control System in Light of the Proposed NIST RBAC Standard. In *18th Annual Computer Security Applications Conference*, pages 382–390, Las Vegas, Nevada, USA.

[Merton, 1952] Merton, R. (1952). "Bureacratic structure and personality". In Merton, R., Gray, A., Hockey, B., and Selvin, H., editors, *Reader in Bureaucracy*, pages 361–71. Free Press, New York.

[Mikhailov and Butler, 2002] Mikhailov, L. and Butler, M. (2002). Combining B and Alloy. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Springer Lecture Notes*, Grenoble, France.

[Milosevic et al., 1995] Milosevic, Z., Berry, A., Bond, A., and Raymond, K. (1995). Supporting Business Contracts in Open Distributed Systems. In *Services in Distributed and Networked Environments*, pages 60–67, Whistler, British Columbia-CA.

[Minsky and Ungureanu, 2000] Minsky, N. and Ungureanu, V. (2000). Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(3).

[Mintzberg, 1979] Mintzberg, H. (1979). *The structuring of organizations.* Prentice-Hall, NJ.

[Mintzberg et al., 1998] Mintzberg, H., Quinn, J., and Ghoshal, S. (1998). *The strategy process.* Prentice Hall, London, revised european edition.

[Moeller, 1997] Moeller, R. (1997). Changing definitions of Internal Control and Information Systems Integrity. In *Integrity and Internal Control in Information Systems*, volume 1, pages 255–272. Chapman & Hall.

[Moffett, 1998] Moffett, J. (1998). Control Principles and Role Hierarchies. In *3rd ACM Workshop on Role Based Access Control (RBAC)*, pages 63–72, George Mason University, Fairfax, VA.

[Moffett and Lupu, 1999] Moffett, J. and Lupu, E. (1999). The Uses of Role Hierarchies in Access Control. In *4th ACM Workshop on Role-Based Access Control*, pages 153–160, Fairfax, Virginia.

[Moffett and Sloman, 1988] Moffett, J. and Sloman, M. (1988). The Source of Authority for Commercial Access Control. *IEEE Computer*, 21(2):59–69.

[Moffett and Sloman, 1993] Moffett, J. and Sloman, M. (1993). Policy Hierarchies for Distributed Systems Management. *IEEE Journal on Selected Areas in Communication*, 11(9 (Special Issue on Network Management)):1404–1414.

[Moffett and Sloman, 1994] Moffett, J. and Sloman, M. (1994). Policy Conflict Analysis in Distributed System Management. *Ablex Publishing Journal of Organisational Computing*, 4(1):1–22.

[Moffett, 1990] Moffett, J. D. . (1990). *Delegation of Authority Using Domain Based Access Rules*. PhD thesis, Imperial College, University of London.

[Muller, 1981] Muller, J. (1981). Delegation and Management. *British Journal of Administrative Management*, 31(7):218–224.

[Mullins, 1999] Mullins, L. (1999). *Management and Organisational Behaviour*. Prentice Hall, London, 5th edition.

[Nash and Poland, 1990] Nash, M. and Poland, K. (1990). Some Conundrums Concerning Separation of Duty. In *IEEE Symposium on Security and Privacy*, pages 201–209, Oakland, CA.

[Neumann and Strembeck, 2002] Neumann, G. and Strembeck, M. (2002). A Scenario-driven Role Engineering Process for Functional RBAC Roles. In *7th ACM Symposium on Access Control Models and Technologies*, pages 33–43, Monterey, CA, USA.

[NIST, 1999] NIST (1999). An Introduction to Computer Security: The NIST Handbook. Technical Report (NIST Special Publication 800-12), National Institute of Standards and Technology.

[Nyanchama and Osborn, 1999] Nyanchama, M. and Osborn, S. (1999). The role graph model and conflict of interest. *Transactions on Information Systems Security*, 2(1):Pages 3 – 33.

[Oh and Sandhu, 2002] Oh, S. and Sandhu, R. (2002). A Model for Role Administration Using Organization Structure. In *7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Naval Postgraduate School, Monterey, CA, USA.

[Ortalo, 1998] Ortalo, R. (1998). A Flexible Method for Information System Security Policy Specification. In *5th European Symposium on Research in Computer Security*, volume 1485, pages 67–84. Lecture Notes in Computer Science.

[Owre et al., 1995] Owre, S., Rushby, J., Shankar, N., and Henke, F. v. (1995). Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2).

[Oxford, 1990] Oxford (1990). *The Oxford Paperback Dictionary*. Oxford University Press, 3rd edition.

[Parsons, 1970] Parsons, T. (1970). Social Systems. In Grusky, O. and Miller, G., editors, *Sociology of Organizations*. The Free Press, New York.

[Perwaiz and Sommerville, 2001] Perwaiz, N. and Sommerville, I. (2001). Structured Management of Role-Permission Relationships. In *6th ACM Symposium on Access Control Models and Technologies*, pages 163–171, Chantilly, VA, USA.

[Pfeffer and Salancik, 1997] Pfeffer, J. and Salancik, G. (1997). The Design and Management of Externally Controlled Organisations. In Pugh, D., editor, *Organization Theory: Selected Readings*. Penguin Books, 4th edition.

[Pfleeger, 1997] Pfleeger, C. (1997). *Security in Computing*. Prentice Hall, New Jersey.

[Pugh, 1966] Pugh, D. (1966). Role Activation Conflict: A Study of Industrial Inspection. *American Sociological Review*, 31:835–42.

[Pugh, 1990] Pugh, D. (1990). *Organization Theory: Selected Readings*. Penguin Business. Penguin Books, 3rd edition.

[Pugh, 1997] Pugh, D. (1997). *Organization Theory: Selected Readings*. Penguin Business. Penguin Books, 4th edition.

[Pugh and Hickson, 1973] Pugh, D. and Hickson, D. (1973). The comparative study of organizations. In Salaman, G. and Thompson, K., editors, *People and Organizations*, pages 50–66. Longmans, London.

[Ramakrishnan and Ullman, 1993] Ramakrishnan, R. and Ullman, J. (1993). A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*.

[Ribeiro et al., 2001a] Ribeiro, C., Zuquete, A., and Ferreira, P. (2001a). Enforcing obligation with security monitors. In *The Third International Conference on Information and Communication Security (ICICS2001)*, Xian, China.

[Ribeiro et al., 2001b] Ribeiro, C., Zuquete, A., Ferreira, P., and Guedes, P. (2001b). SPL: An access control language for security policies with complex constraints. In *Network and Distributed System Security Symposium (NDSS01)*, San Diego, California.

[Richters and Gogolla, 2000] Richters, M. and Gogolla, M. (2000). Validating UML models and OCL constraints. In *3rd International Conference on the Unified Modeling Language*, volume 1939, York. Springer LNCS.

[Robbins et al., 1998] Robbins, J., Hilbert, D., and Redmiles, D. (1998). Extending design environments to software architecture design. *Automated Software Engineering*, 5(3):261–390.

[Roeckle et al., 2000] Roeckle, H., Schimpf, G., and Weidinger, R. (2000). Process-Oriented Approach for Role-Finding to Implement Role-Based Security Administration in a Large Industrial Organisation. In *5th ACM Workshop on Role-Based Access Control*, pages 103–110, Berlin, Germany.

[Rueschemeyer, 1986] Rueschemeyer, D. (1986). *Power and the Division of Labour*. Polity Press.

[Salaman, 1980] Salaman, G. (1980). Roles and rules. In Salaman, G. and Thompson, K., editors, *Control and Ideology in Organizations*, pages 128–151. The Open University Press.

[Salaman and Thompson, 1980] Salaman, G. and Thompson, K., editors (1980). *Control and Ideology in Organizations*. The Open University Press.

[Saltzer and Schroeder, 1975] Saltzer, J. and Schroeder, M. (1975). The protection of Information in Computer Systems. In *IEEE*, volume 63(9), pages 1278–1308.

[Samarati and Vimercati, 2001] Samarati, P. and Vimercati, S. (2001). Access Control: Polcies, Models and Mechanisms. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, pages 137–196. Springer Lecture Notes 2171.

[Sandhu, 1988] Sandhu, R. (1988). Transaction Control Expressions for Separation of Duties. In *4th Aerospace Computer Security Conference*, pages 282–286, Arizona.

[Sandhu, 1990] Sandhu, R. (1990). Separation of Duties in Computerized Information Systems. In *IFIP WG11.3 Workshop on Database Security*, Halifax, UK.

[Sandhu, 1992] Sandhu, R. (1992). The Typed Access Matrix Model. In *IEEE Symposium on Security and Privacy*, pages 122–136, Oakland.

[Sandhu, 1996] Sandhu, R. (1996). Role Hierarchies and Constraints for Lattice-Based Access Controls. *Lecture Notes in Computer Science*, 1146:65–79.

[Sandhu, 1998] Sandhu, R. (1998). Role activation hierarchies. In *3rd ACM Workshop on Role-Based Access Control*, pages 33–40, Fairfax, VA.

[Sandhu et al., 1999] Sandhu, R., Bhamidipati, V., and Munawer, Q. (1999). The ARBAC97 model for role-based administration of roles. *Transactions on Information Systems Security*, 2(1):105 – 135.

[Sandhu et al., 1996] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. (1996). Role-based access control models. *IEEE Computer*, 29(2):38–47.

[Sandhu and Munawer, 1999] Sandhu, R. and Munawer, Q. (1999). The ARBAC99 Model for Administration of Roles. In *15th Annual Computer Security Applications Conference*, Phoenix, Arizona.

[Schaad, 2001] Schaad, A. (2001). Conflict Detection in a Role-based Delegation Model. In *17th Annual Computer Security Applications Conference*, New Orleans.

[Schaad and Moffett, 2001] Schaad, A. and Moffett, J. (2001). The Incorporation of Control Principles into Access Control Policies (Extended Abstract). In *Hewlett Packard Policy Workshop*, Bristol.

[Schaad and Moffett, 2002a] Schaad, A. and Moffett, J. (2002a). A Framework for Organisational Control Principles. In *18th Annual Computer Security Applications Conference*, Las Vegas, Nevada, USA.

[Schaad and Moffett, 2002b] Schaad, A. and Moffett, J. (2002b). A Lightweight Approach to Specification and Analysis of Role-based Access Control Extensions. In *7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Monterey, CA.

[Schaad and Moffett, 2002c] Schaad, A. and Moffett, J. (2002c). Delegation of Obligations. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, Monterey, CA.

[Schaad et al., 2001] Schaad, A., Moffett, J., and Jacob, J. (2001). The access control system of a European bank - a case study. In *6th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Chantilly, VA, USA.

[Schael, 1996] Schael, T. (1996). *Workflow Management Systems for Process Organisations*, volume 1096 of *Lecture Notes in Computer Science*. Springer.

[Schael and Zeller, 1993] Schael, T. and Zeller, B. (1993). Workflow management systems for financial services. In *Conference on Organizational computing systems*, pages 142–154, Milpitas, CA, USA.

[Scheer, 1994] Scheer, A. (1994). *Business Process Engineering: Reference Models for Industrial Enterprises*. Springer-Verlag Berlin and Heidelberg GmbH & Co. KG.

[Sibley et al., 1991] Sibley, E., Michael, J., and Wexelblat, R. (1991). Use of an Experimental Policy Workbench: Description and Preliminary Results. *DBSec*, pages 47–76.

[Simon and Zurko, 1997] Simon, R. and Zurko, M. (1997). Separation of Duty in Role-Based Environments. In *Computer Security Foundations Workshop X*, Rockport, Massachusetts.

[Sloman, 1994a] Sloman, M. (1994a). *Network and Distributed Systems Management*. Addison Wesley.

[Sloman, 1994b] Sloman, M. (1994b). Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2(4):333–360.

[Sloman and Moffett, 1991] Sloman, M. and Moffett, J. (1991). Delegation of Authority. In Krishnan, I. and Zimmer, W., editors, *Integrated Network Management II*, pages 595–606. North Holland.

[Sloman and Twidle, 1994] Sloman, M. S. and Twidle, K. P. (1994). Domains: A Framework for Structuring Management Policy. In Sloman, M. S., editor, *Network and Distributed Systems Management*, pages 433–453. Addison Wesley.

[Smith, 1999] Smith, E. (1999). *Network Auditing: A Control Assessment Approach*. John Wiley.

[Smith, 2000] Smith, G. (2000). *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers.

[Sommerville, 2001] Sommerville, I. (2001). *Software Engineering*. Addison-Wesley.

[Spivey, 1992] Spivey, M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 2nd edition.

[Steen and Derrick, 1999] Steen, M. and Derrick, J. (1999). Formalising ODP Enterprise Policies. In *3rd International Enterprise Distributed Object Computing Conference (EDOC)*, Mannheim, Germany.

[Strati, 2000] Strati, A. (2000). *Theory and Method in Organization Studies*. Sage Publications.

[Strens and Dobson, 1993] Strens, R. and Dobson, J. (1993). How Responsibility Modelling Leads to Security Requirements. In *2nd New Security Paradigms Workshop*, Little Compon, Rhode Island.

[Strous, 1997] Strous, L. (1997). Integrity: Definition, Subdivision, Challenge. In Jajodia, S., List, W., McGregor, G., and Strous, L., editors, *Integrity and Internal Control in Information Systems*, volume One, pages 187–194. Kluwer Academic Press.

[Swanson and Guttman, 1996] Swanson, M. and Guttman, B. (1996). Generally Accepted Principles and Practices for Securing Information Technology Systems. Technical Report NIST Special Publication 800-14.

[The Times, 2002] The Times, U. (2002). 'Rogue' trader denies fraud allegations.

[Tidswell and Jaeger, 2000] Tidswell, J. and Jaeger, T. (2000). An access control model for simplifying constraint expression. In *7th ACM conference on Computer and communications security*, pages 154–163, Athens, Greece.

[Ungureanu, 2002] Ungureanu, V. (2002). Regulating E-Commerce through Certified Contracts. In *18th Annual Computer Security Applications Conference*, Las Vegas, Nevada.

[Urwick, 1952] Urwick, L. (1952). *Notes on the Theory of Organization*. American Management Association.

[Vaziri and Jackson, 1999] Vaziri, M. and Jackson, D. (1999). Some Shortcomings of OCL, the Object Constraint Language of UML (Response to Object Management Group's Request for Information on UML 2.0). Technical report, MIT, Software Design Group.

[Warmer and Kleppe, 1998] Warmer, J. and Kleppe, A. (1998). *The Object Constraint Language: Precise modeling with UML*. Addison Wesley.

[Weber, 1947] Weber, M. (1947). *The Theory of Social and Economic Organization*. Free Press, New York.

[Weber, 1998] Weber, R. (1998). *Information Systems Control and Audit*. Prentice Hall.

[West, 1997] West, M. (1997). Animation of Z with Prolog. In *Invited talk at AFPL et AFCET-GT Programmation en Logique Seminaire: Specifications formelles et programmation en logique (available under http://scom.hud.ac.uk/scommmw/)*, AFCET, Paris.

[West and Eaglestone, 1992] West, M. and Eaglestone, B. (1992). Software development: two approaches to animation of Z specifications using Prolog. *Software Engineering Journal*, pages 264–276.

[Wies, 1994] Wies, R. (1994). Policies in Network and Systems Management. *Network and Systems Management*, 2(1):63–83.

[Woehe and Doering, 1996] Woehe, G. and Doering, U. (1996). *Einfuehrung in die Allgemeine Betriebswirtschaftslehre*. Verlag Franz Vahlen, Muenchen.

[Yao et al., 2001] Yao, W., Moody, K., and Bacon, J. (2001). A Model of OASIS Role-Based Access Control and its Support for Active Security. In *6th ACM Symposium on Access Control Models and Technologies*, Chantilly, Virginia, USA.

[Zaniolo et al., 1997] Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R., Subrahmanian, V., and Zicari, R. (1997). *Advanced Database Systems*. Morgan Kaufmann.

[Zhang et al., 2001] Zhang, L., Ahn, G., and B., C. (2001). A Rule-based Framework for Role-Based Delegation. In *6th ACM Symposium on Access Control Models and Technologies*, Chantilly, VA, USA.