SoDA: A Model for the Administration of Separation of Duty Requirements in Workflow Systems

Stephen Perelson

SoDA: A Model for the Administration of Separation of Duty Requirements in Workflow Systems

This dissertation comprises research work completed in the field of

INFORMATION TECHNOLOGY

and is submitted by

STEPHEN PERELSON

in accordance with the requirements for the degree of

MAGISTER TECHNOLOGIAE : INFORMATION TECHNOLOGY

at the

PORT ELIZABETH TECHNIKON

Promoter : R.A. Botha
Year : 2001

DECLARATION

I, **Stephen Perelson,** hereby declare that –

- The work in this dissertation is my own original work.
- All sources used or referred to have been documented and recognised.
- This dissertation has not been previously submitted in full or partial fulfilment of the requirements for an equivalent or higher qualification at any other recognised education institution.

Stephen Perelson

ABSTRACT

The increasing reliance on information technology to support business processes has emphasised the need for information security mechanisms. This, however, has resulted in an ever-increasing workload in terms of security administration. Security administration encompasses the activity of ensuring the correct enforcement of access control within an organisation. Access rights and their allocation are dictated by the security policies within an organisation. As such, security administration can be seen as a policy-based approach.

Policy-based approaches promise to lighten the workload of security administrators. Separation of duties is one of the principles cited as a criterion when setting up these policy-based mechanisms. Different types of separation of duty policies exist. They can be categorised into policies that can be enforced at administration time, viz. static separation of duty requirements and policies that can be enforced only at execution time, viz. dynamic separation of duty requirements.

This dissertation deals with the specification of both static separation of duty requirements and dynamic separation of duty requirements in role-based workflow environments. It proposes a model for the specification of separation of duty requirements, the expressions of which are based on set theory. The model focuses, furthermore, on the enforcement of static separation of duty. The enforcement of static separation of duty requirements is modelled in terms of invariant conditions. The invariant conditions specify restrictions upon the elements allowed in the sets representing access control requirements.

The sets are themselves expressed as database tables within a relational database management system. Algorithms that stipulate how to verify the additions or deletions of elements within these sets can then be performed within the database management system. A prototype was developed in order to demonstrate the concepts of this model. This prototype helps demonstrate how the proposed model could function and flaunts its effectiveness.

ACKNOWLEDGEMENTS

My sincerest gratitude goes to

Reinhardt Botha, for his guidance and perseverance.

The Port Elizabeth Technikon for their complete support.

The National Research Foundation, whose financial support made this research possible.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1.

Introduction

Businesses are beginning to realise that in order to remain competitive they need to manage their information more intelligently (Mohanty, 1998). Many methods exist which a business can make use of to accomplish this task. Such methods include knowledge management, total quality management and business process re-engineering (BPR).

BPR projects involve streamlining the current business processes in order to improve the efficiency of the business (Motwani, Kumar & Jiang, 1998). Often the implementation of information systems and technology is used to bring about the improvement of the business processes.

It is generally noted that the use of information systems and technology can enable better information management. One such information system is known as a workflow management system. A workflow management system is often seen as a suitable solution for BPR projects as it separates the business logic and the information technology support it requires (Hollingsworth, 1995).

Workflow is a growing area of business technology concerned with the automation of processes that involve various participants. Workflow automates the procedures where documents, data, or tasks are passed between participants according to defined rules to achieve, or contribute to, an overall business goal. Workflow can thus be seen as the computerised facilitation or automation of a business process, in whole or in part (Hollingsworth, 1995).

Workflow software coordinates workflow processes, which are comprised of a series of tasks that must be performed. These tasks form the basis of a business process. The workflow software will follow the defined process and will execute these tasks in the correct order as and when the conditions stipulate it. This will normally result in items of work that must be completed by the users that are part of the workflow environment.

These work items are created as the workflow progresses along the defined process. For example, a purchase order for a low value item may not require special approval, while orders for a high value item will require approval from the departmental manager. Due to the nature of the information being passed through the organisation, information security is an important issue within the workflow environment. The prevention of fraud and other detrimental activities within the workflow environment needs to be ensured.

As such, the workflow management coalition (WfMC) has recognised that information security is an important issue within workflow environments and has begun working on security specifications (WfMC, 1998). But rather than specify exactly how the security is to be implemented, the WfMC has left the decision and the implementation of security services to the software manufacturers.

Security services form a vital part of any information system within a business. Information in a business must be protected, and for this reason workflow management systems do implement security mechanisms (Bertino & Ferrari, 1999). Unfortunately, information security administration is a very complex and time-consuming activity. The complex nature of the administration of information security is eased through the introduction of role based access controls (Bertino & Ferrari, 1999).

The complex nature of information security administration could be alleviated more if the relationship between the workflow environment and the role-based environment was better understood. This understanding would result in a greater understanding of how and where security can be enforced.

In order to understand exactly what this research covers, an overview of the research domain is in order.

1.1 The Research Domain

The research reported on, in this dissertation, primarily addresses access control administration. In order to understand what access control is, it is necessary to review information security services.

Information security involves five generic services (ISO 7498–2, 1989). These services are: Authentication; Integrity; Non-repudiation; Access Control and Confidentiality.

An *authentication* service ensures that the person trying to gain access is properly identified and is in possession of the required access rights.

The *integrity* service ensures that the information being accessed is not corrupted in any way. Corruption may occur through the tampering of data.

A *non-repudiation* service ensures that the information received is from the correct source. This enables the receiving party to know without a doubt that the data came from the sending party and not a third party.

The *access control* service ensures that data is disseminated to persons who have valid access rights and not to those persons who do not have the correct access rights.

The *confidentiality* service ensures that the information remains secure.

Complex security features can be created by the careful combination of the basic security services. This project, however, will focus primarily on the access control service and its synergy with the integrity service.

Access control has to do with the control of access to resources. The demands required from access control mechanisms have quickly grown to not only include the specification of "*who has access"*, but also the "*type of access"* that is allowed. In the workflow environment, this can be extended to “who has what type of access to which information, *under which circumstances*” (Cholewka, Botha & Eloff, 2000).

In order to ease the administration of access rights to resources, the notion of user groupings has been defined. This type of access control is referred to as Role-Based Access Control (RBAC) (Sandhu, Coyne, Feinstein & Youman, 1996).

1.1.1 Role-Based Access Control

In essence RBAC associates permissions with roles, and users with roles. Users, therefore, receive permissions based on the roles with which they are

associated. Roles typically represent the various job functions in an organisation and users are therefore assigned roles based on their responsibilities and qualifications. Users can be re-assigned from one role to another with ease. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed (Sandhu, 1998). Rules, which are known as constraints, can be established to govern the relationships between roles.

Constraints are sometimes considered the principle motivation for RBAC. One common constraint is that of mutually disjointed roles, such as accounts payable manager and purchasing manager. If the same individual is allowed to be a member of both these roles, the possibility of committing fraud is enhanced. This forms the basis for a well-known and time-honoured principle known as Separation of Duty (SoD) (Ahn & Sandhu, 1999).

1.1.2 Separation of Duty

SoD is a fundamental security principle frequently exercised even in the paper-based world. The general idea is to ensure that no single person can be responsible for the completion of a business process (Simon & Zurko, 1997). Several types of SoD can be identified (Simon & Zurko, 1997) (Gligor, Gavrila & Ferraiolo, 1998).

These types of SoD can be grouped according to time of enforcement. SoD requirements that can be evaluated already at design time are known as static separation of duties. Requirements that can only be evaluated during the execution of workflow processes are known as dynamic separation of duties.

The specification of SoD constraints for either static or dynamic SoD can be accomplished within the access control administration environment. The enforcement of static SoD constraints can only be done within the access control administration environment. Dynamic SoD constraints are enforced during execution of the workflow processes. This is an important distinction for this research as it impacts on what can be accomplished within the access control administration environment.

1.1.3 Access Control Administration

As already mentioned, access control administration is a very complex problem. Although this administration problem is eased through the use of roles and permission groupings, the users of the administration systems can still make mistakes.

An access control administration environment that can prevent user errors would increase security and maintain ease of use. This is possible by controlling the complex relationships between the various entities in the workflow environment.

The entities involved would include those found within RBAC as well as those identified within the workflow environment. The linking of the workflow environment with RBAC only serves to increase the complexity of the access control administration environment. Information security concepts, in particular access control, will be discussed in more detail in chapter 3.

Now that the research domain has been established the research questions are examined.

1.2 Research Questions

The creation of a model for an advanced access control administration environment is envisaged. This model will require a number of problems to be researched. These problems are specified in the form of questions.

1.2.1 How can RBAC concepts be applied in the workflow environment?

RBAC environments formulate access control requirements in terms of Roles, Users and Permissions. Additional elements may need to be included into these formulations due to the fact that the propagation of work will have an effect on access control. The workflow environment will also dictate how the RBAC environment makes use of these elements.

1.2.2 How is the specification of SoD requirements influenced by the inclusion of the workflow entities?

With the inclusion of the additional workflow information into the RBAC environment, it will be necessary to examine the specification of SoD

principles. Specifically, what the impact upon the constraints between the different entities will be and what rules will govern how they inter-relate.

1.2.3 Can a single administration paradigm successfully formulate the range of separation of duty requirements?

The term *single administration paradigm* describes a shift towards an access control administration model that combines what used to be separate access control tasks into one logical entity. The single administration paradigm will involve the specification of access control requirements and will ensure the enforcement of these requirements. Other approaches could separate these tasks into two distinctly different administration entities. The single administration paradigm would allow for a consistent and easy to use approach to access control administration.

The answers to these questions all contribute towards the accomplishment of the primary research objectives.

1.3 Research Objectives

The objective of this research project is to propose a model that can be used to support the *specification* and *enforcement* of static SoD constraints in the workflow environment. The model should, furthermore, provide a single administration paradigm.

The proposed model will, furthermore, address the *specification* of dynamic SoD constraints employing the same administration paradigm. The proposed model does not, however, address the *enforcement* of dynamic SoD constraints, since this would be the responsibility of run-time components within the workflow environment.

1.4 Methodology

The research commenced with a literature study. The first part of the literature study developed an understanding of the workflow environment and technology currently applied. Thereafter, a study of access control, specifically focusing on the access control requirements of the workflow environment, followed.

This study of access control requirements resulted in a detailed study of the RBAC mechanisms that are frequently used in workflow environments. The focus of the literature study then shifted to examine SoD requirements.

A model that embodies a single administration paradigm was proposed for the specification of SoD requirements. The model has been extended to include the enforcement of static SoD requirements. The research will furthermore demonstrate the proposed model through the development of a security administration prototype.

The next section discusses how the results of this research have been incorporated into the dissertation.

1.5 Layout of Dissertation

The layout of the dissertation is depicted in Figure 1.1. The structure of the chapters is described below.

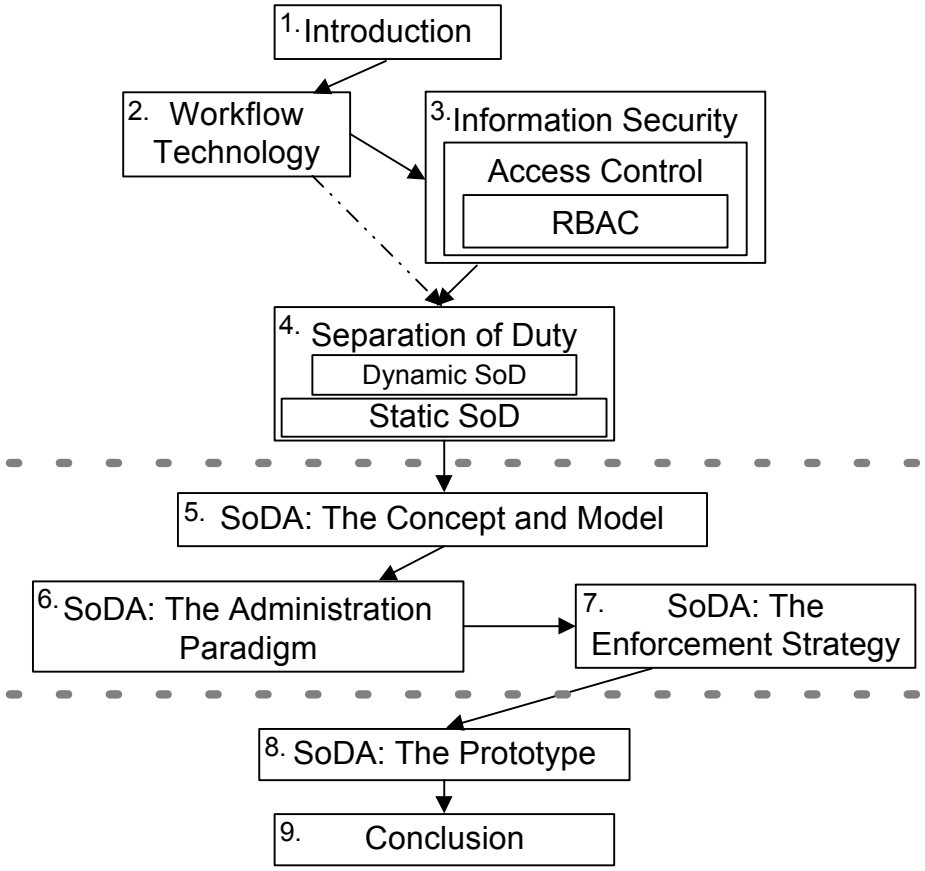

Figure 1.1 Layout of Dissertation

Chapter 1. Introduction

The problem statement, the objective of the dissertation and the research methodology are introduced here.

Chapter 2. Workflow Technology

A clarification of the terminology used in workflow environments is followed by a state of the art overview. A discussion of the concepts of workflow and a description of the WfMC's reference model is included.

Chapter 3. Information Security

An introduction to information security services is followed by a discussion of access control in broad terms. Focus then shifts to role-based access control (RBAC). It is shown how RBAC eases security administration through a policy-based approach.

Chapter 4. Separation of Duty

Separation of duty as a security policy is introduced. It is shown that the workflow environment provides an appropriate context for specifying SoD policies. This chapter introduces the difference between static and dynamic SoD.

Chapter 5. SoDA: The Concept and Model

The conceptual framework of the model is followed by a definition of the scope of the model.

Chapter 6. SoDA: The Administration Paradigm

An introduction to the entities that are involved is followed by descriptions and explanations of the relationships between these entities. It then describes the conflict paradigm that has been adopted. Thereafter, the conflict concept is incorporated as part of the model.

Chapter 7. SoDA: The Enforcement Strategy

This chapter introduces algorithms for maintaining integrity in the model.

Chapter 8. SoDA: The Prototype

A discussion of the design of the prototype is followed by an explanation of the implementation issues. The detailed workings of the prototype are shown. The lessons learnt and the shortcomings of the prototype conclude this chapter.

Chapter 9. Conclusion

A summary of the findings of this research is followed by a discussion of possible future research.

The workflow environment forms the basis of the access control administration model and, as such, the next chapter will discuss the workflow environment. In addition to describing the workflow environment, an example workflow process will be introduced in this chapter and will be referred to throughout this dissertation.

Chapter 2.

The Workflow Environment

Interest in workflow technology has been increasing as is evident with the plethora of workflow systems available. Companies such as Oracle, SAP, Microsoft and Adobe have all adopted some workflow functionality into their products. Workflow systems are also currently being used to facilitate other areas of business such as customer relationship management (Stadler, 2000). Many companies have adopted workflow systems as a means of implementing business process reengineering (BPR) projects (Teng, Jeong & Grover, 1998). Workflow technology is often used as the solution for BPR projects as it separates the business logic and the IT infrastructure (Hollingsworth, 1995).

It is necessary to examine the workflow environment in order to gain answers to the research questions that were formulated in the previous chapter. The knowledge gained about the workflow environment will enable role-based access control to be applied to the workflow environment.

In order to gather this knowledge of the workflow environment, a study of the Workflow Management Coalition's (WfMC) generic workflow product reference model will be carried out. This model identifies the characteristics, terminology, and components of the generic workflow system (Hollingsworth, 1995). Any developments in workflow technology can be based upon this generic model. As such, a look at the architecture of a workflow system is in order.

2.1 Architecture of a Workflow System

The workflow reference model proposed by the WfMC is an effort to standardise workflow management products. The reference model describes the concepts of workflow management, the reference architecture, and the interfaces between the various workflow components (Hollingsworth, 1995).

Many existing or future workflow systems may differ in implementation but they encompass some form of the architectural components described in the workflow reference model. These similarities enable the model to be used as a foundation for the study of existing and future workflow environments.

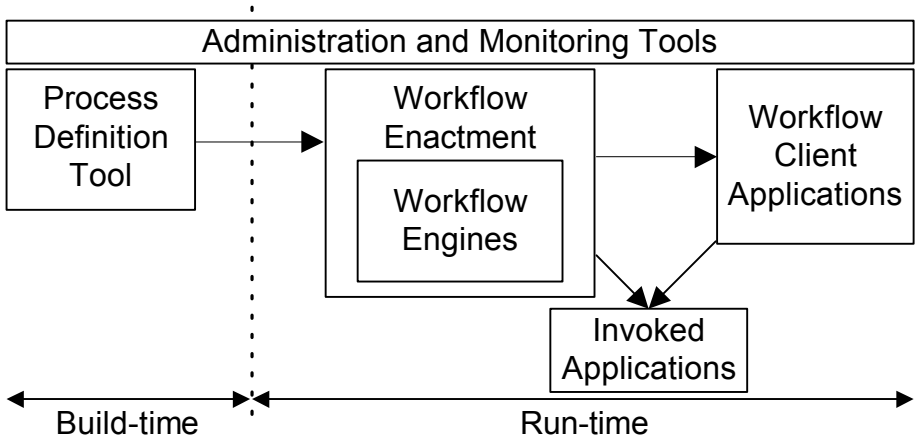

Figure 2.1 The workflow reference model and components

The *process definition tool* defines a process into a computer processable form. This defined process is known as a *process definition*. The process definition holds all necessary instructions to enable the workflow enactment software to correctly execute it. Information such as the starting and completion conditions and user tasks to be undertaken is included within the definition. In the workflow example, described in section 2.2, the process definition would be the tasks and the rules to transform between the tasks.

The *workflow enactment service* interprets the process description. Based on the process definition it will begin processes and sequence activities. The workflow enactment software will also add work items to the user work lists and invoke application tools as necessary.

A workflow engine is the part of the workflow enactment service that places items onto *worklists* when user interactions are needed. The worklist handler, which is part of the workflow client applications, will handle these items.

The *worklist handle*r manages the interaction between users and the workflow enactment service. The worklist handler progresses the work requiring user attention. It uses the worklist to interact with the workflow enactment software. The WfMC uses the term *workflow client application* in preference to "worklist handler" to reflect its larger potential usage (Hollingsworth, 1995).

The workflow environment may *invoke applications* to handle various tasks that do not require user interaction. The workflow environment would need sufficient logic built in to be able to communicate with the invoked application.

The final area of standardisation is a common interface for administration and monitoring functions. This will allow a single administration and monitoring tool from one vendor to be used, with multiple workflow environments from other vendors. These administration and monitoring tools must take into account the temporal nature of the workflow environment.

For the purposes of this research we distinguish events based upon the time that they occur. Events can be either build-time or run-time.

Build-time events are typically the process definition and the access control definition. This dissertation uses the term "build-time function" to refer to these events. These events are distinct from the run-time events that include the creation of process instances and the enforcement of access control.

2.1.1 Build-Time Functions

One aim of the build-time functions is to produce a computerised definition of a business process. This is accomplished by translating a business process from the real world into a computer processable definition. Various business analysis, modelling and system definition techniques may be used to produce the process definition. The process definition can be displayed in various means such as graphically or textually and is also called a process model, process template, or process metadata. A system administrator or the process owner may be responsible for the creation of the process definition.

A process definition comprises of various tasks[1] that have associated human or computer operations. Tasks are the building blocks of business processes and a user would perform each task. An example of a task that a user would perform is the 'Approve Order' task described in the workflow example in section 2.2. A computer-controlled task could possibly be the 'Check Stock'

[1] The term 'activity steps', which is used by the WfMC, is synonymous with the term 'tasks', which is more widely used within this text (Workflow Management Coalition. 1996).

task. The scenario depicts this task as being controlled by a user, however it could be controlled by a computer application if the stock system was automated. Tasks can be limited to certain sets of users ensuring a level of access control. Access control specification is another of the build-time functions and is a major area of focus for this research.

The constraints that ensure access control can also be used to govern the progression between tasks within a workflow process. These constraints govern when and if a task should be performed. In the workflow example in section 2.2, the 'Approve Order' task only gets performed if the order is above a certain value. This is an example of a constraint governing the progression of the process.

The process definition is subject to standardisation while the means of creating the definition can vary greatly between workflow products. This standardisation is necessary to allow for the interchange of data between other build-time and run-time components.

2.1.2 Run-Time Functions

The process definition is interpreted by software that will create and control instances of the process. This software will schedule the activities and will invoke the different human or IT applications as needed. The users who interact with the workflow environment are the organisation employees and other people who may form a part of the workflow processes. The run-time control functions link the modelled process to the real-world process.

In order to understand how a real world process functions within a workflow management system, an example workflow process will be discussed.

2.2 Example Workflow

The example workflow process definition, which will be used to explain the workflow concepts in more detail, is depicted in figure 2.2. It describes an internal order procedure within an organisation. It is made up of activities that can be performed by members of certain roles such as Employee or Manager roles. Roles are discussed in depth in chapter 3 but for now a role can be seen as a direct mapping to an organisational structure. For example, an

employee may place a purchase order, whilst only a departmental manager may approve the order.

The internal order process needs to be converted into a process definition for use by the workflow environment.

2.2.1 Defining the Process

The definition of the process will involve analysing all the tasks and conditions for every step of the process. The process definition will also define conditions that must be met during the performance of a task. These conditions are often expressed as entry and exit conditions for a task (Hollingsworth, 1995). For example, a purchase order for a low value item (i.e.. value <= 100) may not require special approval, whilst other orders with higher values would require approval from the departmental manager.

Note that a task definition may very well be a complete workflow. The "Order stock" task in Figure 2.2 may, for example, require its own process definition (WfMC, 1998).

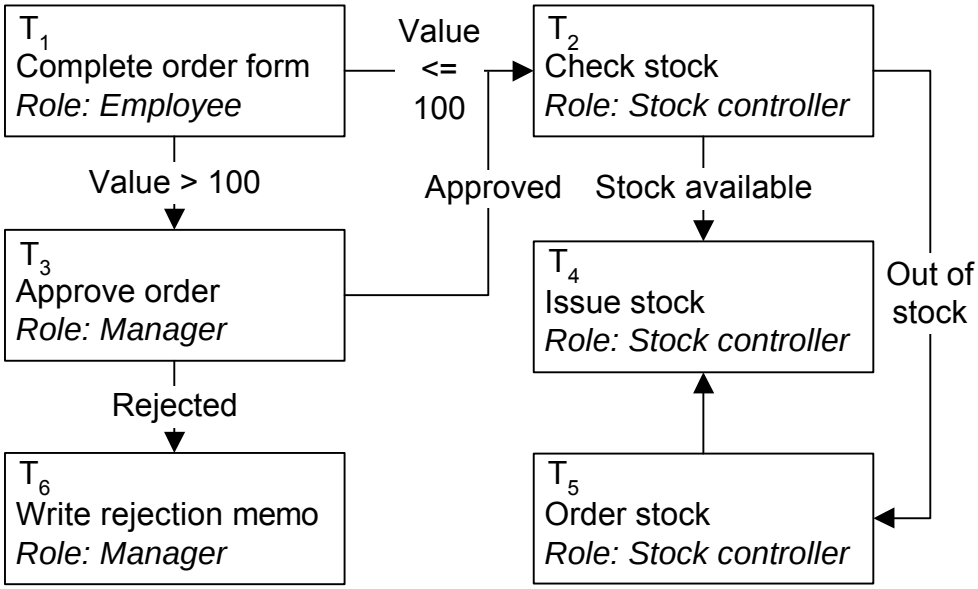

Figure 2.2 The “internal order” process

At any stage within the workflow environment, multiple instances of this workflow process may be in operation. Each instance of this workflow process will need to be interpreted by the workflow enactment service.

2.2.2 Interpreting the Definition

Each workflow process instance could be at a different point in execution as shown in figure 2.3. The route taken by the workflow process depends upon that particular instance's operational data. An instance of a task is created upon demand as the tasks are encountered during the processing of the workflow. In this case, whether the item requested is above a certain value or whether it is in stock, different tasks will be instantiated. In figure 2.3, process instance 1 depicts a task (t_{11}) that is waiting for a manager to approve it while in instance 2 the order has been approved (t_{23}) and is busy being checked by the stock controller (t_{22}).

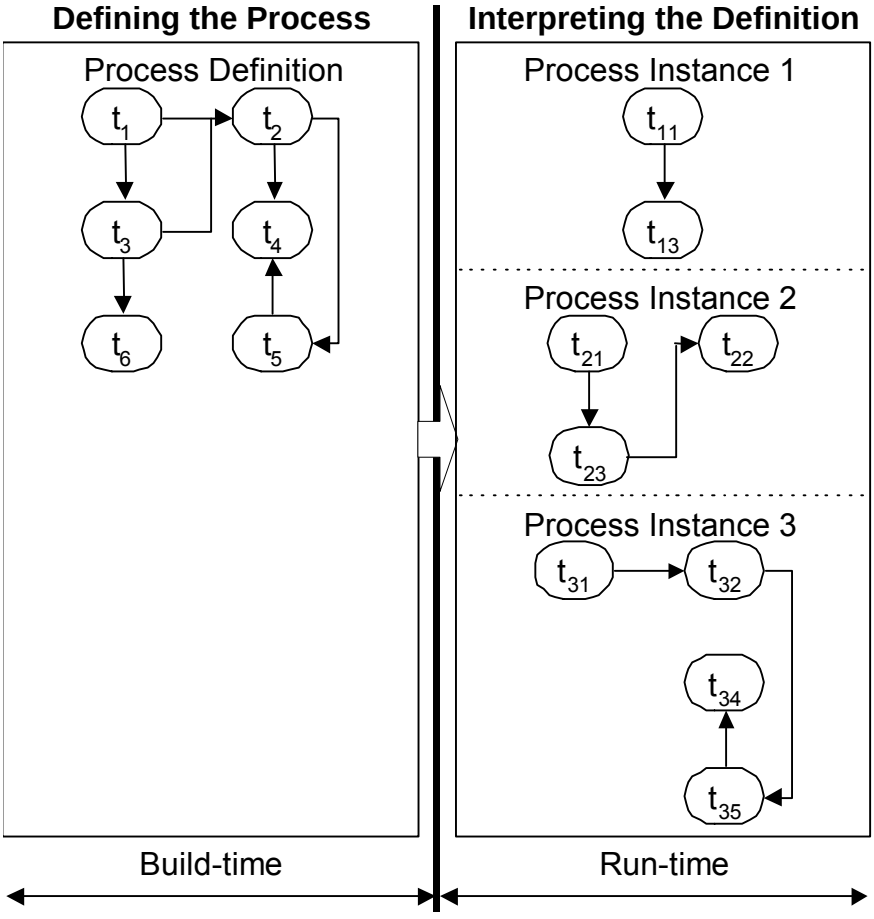

Figure 2.3 Example process instances

Process instance 3, however, was of a low value (<=100) and did not need approval from the manager. This means that task t_3 was not instantiated and thus the process bypassed the manager. Task t_2 was instantiated instead and so the stock controller is checking the order (t_{32}). But the item was not in

stock and so the stock controller had to order the item (t_{35}) before being able to issue the stock (t_{34}).

These tasks generally involve placing work items onto the user's worklists. Only a user that is associated to the correct role will receive the relevant work items.

2.2.3 Interacting with the User

The users in the example system, as shown in figure 2.4, are as follows:

- Users A and B are associated with the Manager role.
- User C is associated with the Stock Controller role.

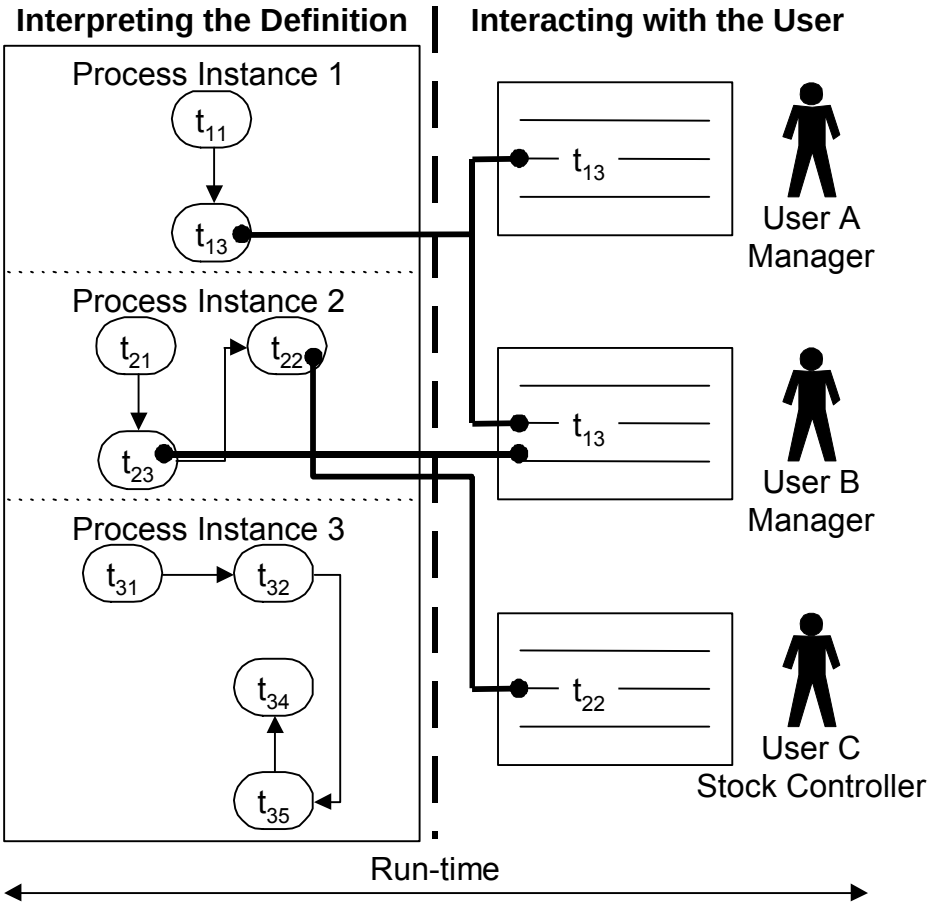

Figure 2.4 Task instances and user interaction

At various points during the execution of the workflow process instances, worklist items will appear on different users' worklists. These worklist items will only appear in certain users' worklists, which are determined by the relationships between the users and the roles, and the task and the roles.

The worklist items of a task may only be assigned to users who are associated with the same role that the task is associated with.

An example is depicted in figure 2.4. Since a worklist item based on task instance for task t_{13} appears on two worklists, user A's and user B's, both are managers. As soon as one of them completes the worklist item, it will disappear from both of their lists. Task instance t_{22} appears on user C's worklist as a worklist item, as user C is a stock controller within the organisation.

Task instance t_{23} of process instance 2 only appears on user B's worklist unlike task instance t_{13}, which appears on both user A and B's worklists. This is due to the fact that user A, as a member of the employee role, placed the order. That is, user A submitted the order form that instantiated process instance 2. A constraint is in place that prevents a user from approving self placed orders in order to reduce the possibility of fraud. This technique of restricting access rights is known as separation of duty and is discussed further in chapter 4.

This type of constraint closely follows the organisational security policies and would have to be programmed into the process definition through an administration tool. An organisation's access control policy needs to maintain the integrity between system objects and real world objects. This means that the organisation's workflow systems should only allow actions that would be allowed to occur in the real world. This can only happen when the business constraints that form the organisational policies are mirrored within the workflow access control decisions.

2.3 Conclusion

This chapter has defined the workflow environment and has introduced the core components of the workflow environment. These components include the build-time and run-time components as well as the concept of the task. An example workflow definition was used to illustrate how the workflow environment functions, as well as to demonstrate some access control principles. This workflow example will be referred to when describing information security.

The WfMC has identified that information security and the area of security administration are important for workflow systems (WfMC, 1998). An important part of information security is access control and the accompanying administration for the access control requirements.

While some information security services may be handled by the underlying operating system, there are still areas of information security within the workflow system that need specialised information security services. These information security services, and especially access control services, will be reviewed and discussed in the next chapter.

Chapter 3.

Information Security

Information security has been recognised as an important aspect of a workflow environment from an organisation's perspective. More importantly it should be noted that even though various types of security services exist, it was ascertained that access to the data and the integrity of the data are paramount. In order to correctly implement these two information security services, a proper understanding of all the information security services is needed.

Another reason for examining the information security services is the introduction of open systems such as the Internet. An organisation's critical data is being distributed through these open systems and as such a higher degree of security is required.

A lapse of security that causes some of this information to become freely accessible could be very detrimental to the day-to-day activities of the business. In order to properly protect an organisation's information, various security services need to be implemented and used by the organisation. Doing so will prevent many possible offences. Offences could include (Stallings, 1995):

- Modification: An unauthorised user gains access to valuable data and modifies that data. This constitutes an attack on integrity. An example of this could be the alteration of the contents of a purchase order through the worklist item in the workflow's worklist handler.
- Fabrication: An unauthorised party adds fake objects to the system. This constitutes an attack on authenticity. An example of this would be transmitting fabricated worklist items in order to gain from manipulating the workflow system.
- Interception: An unauthorised party gains access to an asset of the system. This constitutes an attack on confidentiality. An example would be

the monitoring of messages on the network. In so doing, values that would trigger certain checks could be altered in order to bypass those checks.

Five different security services are generally recognised. These security services are generally known as: authentication; confidentiality; non-repudiation; integrity; and access control (ISO 7498–2, 1989). These security services have evolved from within the real world and are also very relevant within the information technology realm.

3.1 Information Security Services

Through the correct use of various security services, a secure information environment can exist (Stallings, 1995). It is important to understand what each service tries to accomplish and how they can work together.

Authentication services ensure the users are who they claim to be. The correct identification of a user is important for maintaining accountability and access control. There are three discernable methods of authentication, which include: passwords; physical tokens; and biometric techniques (Sandhu & Samarati, 1996).

Passwords are normally utilised within the login process in software products. They are not a very secure method unless they are chosen carefully and changed very often. Passwords, besides being susceptible to guessing or automated dictionary attacks, are also susceptible to discovery through the use of network sniffers. The use of passwords alone does not ensure a secure environment.

Physical tokens, such as smart cards, are a better approach. Users now need to physically have a valid access token to gain control. This coupled with needing a password makes for better security. An example of a security token would be a bank's ATM card. However, tokens can be stolen or forged and as such are not as secure as they could be.

Biometric techniques offer another approach to the security token method. Instead of making a user keep a token such as a smart card, biometric techniques allow the user themselves to be the key. A variety of characteristics including fingerprints, retinas, voice patterns, and hand

geometry, are able to be measured and to uniquely identify a person. Biometric security methods are also not failsafe and as such should also be used in combination with other authentication methods.

Confidentiality services maintain the non-disclosure of information from unauthorised users. It also maintains the confidentiality of information that is transferred between parties. To keep information confidential it can either be kept hidden or it can be encrypted. Various encryption schemes exist including symmetrical key and public/private key systems, using a variety of enciphering mechanisms.

There are two levels of data confidentiality (Michener, 1999). Firstly, keep unauthorised users unaware of the existence of the data. This can be accomplished by proper access control to the data. As such, it is limited to the computer systems and applications that store the data. Secondly, keep unauthorised users unaware of the semantic content of the data. This can be accomplished through the encryption of the data by utilising various encryption techniques.

Non-repudiation services prevent denial of service to properly authenticated and authorised users (Zhou & Lam, 1999). Non-repudiation is normally achieved through the use of digital signatures (Zhou & Lam, 1999). These digital signatures are made up of a public/private key encrypted addition to the information being transmitted. If the sender's public key deciphers the encrypted signature, then it is proof that the message was from the sender. For example, if a recipient of a message returns a proof of delivery with a digital signature, then non-repudiation is provided. Due to the nature of public and private key encryption, only the recipient's private key could have created the digital signature. And this digital signature provides non-forgeable evidence of the delivery of the message. Symmetric encryption cannot guarantee non-repudiation since the sender and the receiver share the key. This enables either party to create a fake digital signature.

Integrity services are responsible for the sound state of the information. Integrity of information is dependant upon the timely, accurate, complete and consistent nature of the information. Integrity services provide for the basic

protection against corruption during storage or transfer through the effective use of various mechanisms such as a checksum. These mechanisms are typically already in place in data storage devices and communication protocols. Cryptographic methods provide for strong data integrity through the use of a calculated message hash of the data. If the data gets corrupted enroute then it will be identified as such when the message hash is decrypted. As the data is not encrypted it does not provide for confidentiality.

Integrity issues can be separated into three different aspects (Leymann & Roller, 1999): (1) operational integrity deals with concurrent access to data, (2) physical integrity implies protection from loss of data and (3) semantic integrity requires that the data complies with the appropriate business rules. Operational integrity can be maintained through concurrency controls as part of the integrity service. Physical integrity can be achieved by the use of checksums, provided by the integrity service, and cryptographic techniques provided by the confidentiality service. Semantic integrity relies heavily on the access control service, as it will require that data can only be changed according to certain business rules.

Access control services ensure that a user only has access to the information and resources for which the user has rights. It is important to ensure that users have access to the information that will allow them to do their work. It may also be important to restrict users from access to resources and information that they do not need. An organisation should be able to specify who can access information and how and when it can be accessed, including under which conditions (Sandhu et al., 1996). Access control is seen as an indispensable part of any information sharing system (Shen & Dewan, 1992).

This research focuses upon access control more than any of the other security services. An in-depth look at this particular information security service will be followed by an analysis of role based access control.

3.2 Access Control

Once authentication services are in place it becomes possible to implement access control services. There are several forms of access control, including

discretionary access control, lattice-based access control and role-based access control (Sandhu & Samarati, 1994).

Discretionary Access Controls (DAC) revolves around the concept of the data having an owner. The owner determines who will have access to this data. Access to a copy of the data can be obtained as DAC allows data to be copied without restriction from object to object.

Lattice-based access controls (Sandhu, 1993) are also known as mandatory access controls (MAC). This type of access control is based on rules for deciding whether a user may access requested data. The rules only allow the transfer of information in one direction in a lattice of security labels. Labels would include low, medium and high security and users would be assigned to a label. The confidentiality requirements of the military brought about MAC, but it has uses in many other applications.

In the workflow environment, access needs to be granted when a user needs it and should be revoked once the user has completed the work. In the workflow example shown in figure 2.2, the stock controller should only be able to issue stock when an order comes through. The rights needed to issue stock should only be granted when the internal order process has reached the task that requires it. Neither MAC nor DAC support this functionality.

If the workflow system made use of MAC for its access control requirements then a user would gain access to objects that they should not gain. This is due to the global nature of the access that is granted.

If the workflow system made use of DAC for its access control requirements the security administration of the system would become very complex. This is due to the fact that each object within the system could be owned by a unique user and as such would involve the complex administration of access rights (Osborn, Sandhu & Munawer, 2000). Administration of the access rights to the individual users is also a complex exercise.

Extensions to access control models such as role based access control have been introduced in order to add these types of functions (Nyanchama & Osborn, 1999). Access rights are assigned directly to the user with DAC, while with MAC there is less control over the particular access rights that are

assigned to a particular user. Role-based access control mechanisms deal with these issues by assigning the access rights to roles. A user will gain these access rights when he is assigned membership to the roles (Sandhu et al., 1996). This greatly simplifies information security administration.

RBAC can effectively enforce both MAC and DAC (Sandhu & Munawer, 1998) and is policy neutral. The policy that an RBAC based system enforces is a direct result of the RBAC components and their interactions (Sandhu et al., 1996). The RBAC model affords an administrator the opportunity to express an access control policy in terms of the way that the organisation is viewed.

The RBAC model makes use of roles, which are used to logically group permissions together so that they reflect the organisation's view or the application's view. Roles are basically a semantic construct around which a variety of different concepts and models can be formulated.

Since this dissertation will be utilising RBAC, a more in-depth look at RBAC is in order.

3.3 Role Based Access Control

RBAC greatly simplifies the permission to user information security administration. This simplification is achieved by only needing to administer the user to role assignments instead of needing to assign all required permissions to every user. Users get assigned to the roles with those permissions they need to do their work.

This can be seen in a DBMS such as Oracle where roles can be created with all the access rights and permissions necessary for a particular user group (Oracle, 1999). Any user that comes into the organisation as part of the group will be assigned to that role and will immediately have access to all required data.

RBAC also supports the following security principles: least privilege and separation of duties. It supports least privilege by only allowing the permissions required to perform a job to be assigned to a role. It ensures separation of duty by ensuring that mutually exclusive roles complete a sensitive task. The application of these principles cannot be enforced by

RBAC. These principles rely upon the correct use by the security administrator.

It is possible to predefine the role-permission relationships in order to simplify the assignment of users to predefined roles. The role-permission assignments tend to change slower than the user-role assignments (Sandhu et al., 1996). It is also seen to be desirable to allow administrators to be able to administer the user-role assignments but not the permission-role assignments. This is due to the fact that assigning users to roles is less technical than assigning permissions to roles.

It is important to differentiate between the concept of groups and roles. Groups are typically seen as groupings of users while roles are seen as groupings of permissions (Sandhu et al., 1996). It can also be seen that a role brings together groups of users and groups of permissions, and consequently acts as an intermediary.

In this dissertation the existing and well-accepted RBAC96 model (Sandhu et al., 1996) is used. Choosing an independently developed existing model for this exercise gives us an element of objectivity in assessing the power of the proposed administration paradigm. An overview of the RBAC96 model must therefore be given.

In order to better understand the RBAC model, the different concepts that form this model will now be described.

3.4 The RBAC96 Model

Sandhu et al (1996) describes four different conceptual RBAC models. The first is $RBAC_0$, which is the base model that specifies the minimum set of requirements for any system that implements RBAC fully. The second is $RBAC_1$, which adds the concept of role hierarchies that allow roles to inherit permissions from other roles. The third is $RBAC_2$, which adds the concept of the constraint. These constraints impose restrictions upon the configuration of various components within RBAC. The final model is $RBAC_3$, which includes $RBAC_1$, $RBAC_2$ and $RBAC_0$. This family of models is better known as RBAC96.

3.4.1 The RBAC96 Entities

The RBAC96 model consists of 4 main entities: users, roles, permissions and sessions. These entities and their relationships with one another can be seen in figure 3.1.

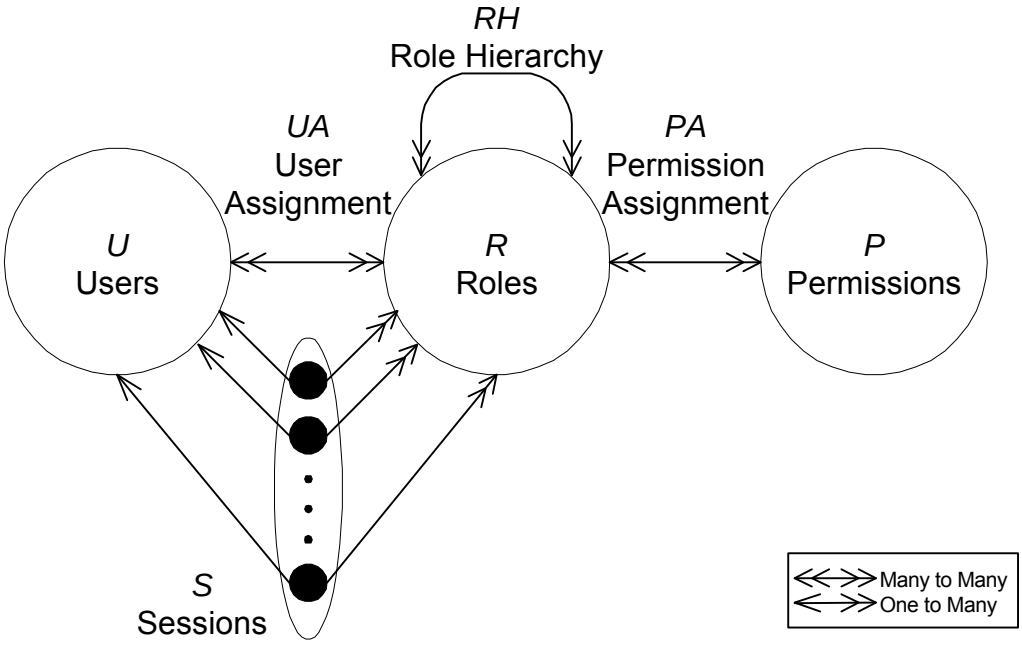

Figure 3.1 Role-based access control model (Sandhu, 1996)

The *user* denotes a human being but can also denote any automated process. To keep the model simple though, it will be assumed that it is a human being. Users are part of a set called *U* as shown in figure 3.1.

A *role* is generally a job function or title within an organisation. A user that is assigned to a particular role would generally be working within the job function that the role denotes. Roles are part of a set called *R* as shown in figure 3.1.

A *permission* is a privilege to access an object or objects in the organisation in a certain way. A permission is always positive and will allow the user to access the referenced object in the way defined by the permission. The objects that are referred to could be either data objects or resource objects. The concept of the permission can vary from being very coarse grained to being very fine grained. The particular nature of a permission depends upon the system being implemented. Permissions are part of a set called *P* as shown in figure 3.1.

A *session* is a mapping of a user to one or more roles. This means that a session is unique to a particular user. The user also receives the union of all the permissions of the role. This means that a user that belongs to a few sessions could be associated with many roles. Each role may, in turn, associate many permissions to the user and as such care must be taken to ensure that the user doesn't get the ability to commit fraud. Sessions are part of a set called *S* as shown in figure 3.1.

Users may have multiple sessions open simultaneously and users have control over which roles they activate. A user can belong to a powerful role but may leave it deactivated until required.

As can be seen in figure 3.1, user assignments and permission assignments are both many to many relations with roles. This is denoted by the double-headed arrows on both ends of the joining line. This means that roles can have many users and a user can be assigned to many roles, and that permissions can be assigned to many roles and many roles can have the same permission.

3.4.2 Role Hierarchies

Role hierarchies are a natural way of structuring roles so that they match the internal structure of an organisation. This dissertation refers to role hierarchies as role networks.

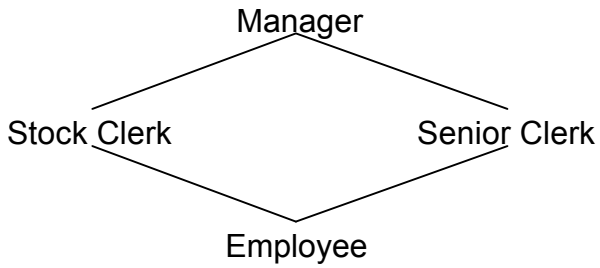

Figure 3.2 Role hierarchy example

Roles within the hierarchy inherit the permissions from lower roles. This would mean that the Stock Clerk role in figure 3.2 would have all of the Employee role's permissions as well as its own. The Manager role would have the Stock Clerk, Senior Clerk and the Employee role's permissions.

Sandhu et al (1996) have defined the entities and their relationships through a mathematical model. Users (U) are associated with roles (R) through the user-assignment relation (UA), which is shown in figure 3.1. Similarly, permissions (P) are associated with roles (R) through the permission-assignment relation (PA), which is also shown in figure 3.1. Roles are arranged in role hierarchies through the partial order RH. A role that is senior to another role inherits the permissions of the junior role. A user that is associated with a senior role in the role hierarchy, therefore, may also assume all the roles junior to the senior role in the RH partial order.

3.4.3 Constraints

Figure 3.1, however, lacks the definition of $RBAC_2$, which adds the concept of the constraints. Figure 3.3 depicts the addition of the concept of constraints into RBAC. Constraints affect every part of the relationships between the RBAC entities.

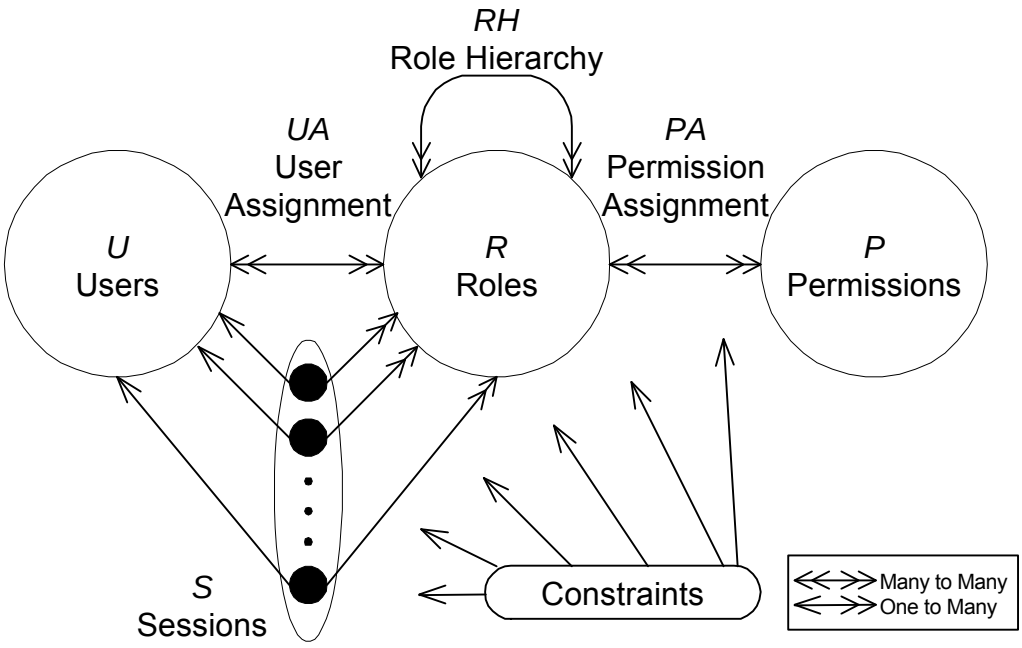

Figure 3.3 Role-based access control model with constraints (Sandhu, 1996)

Constraints are considered to be a very important aspect of RBAC. They are the mechanism through which the organisation's policies can be enforced. Constraints that are in place will ensure that associations between the RBAC entities will comply with what is allowable by the policies. As such, the

administrator's job will be eased, as the RBAC environment will ensure the integrity of the data.

An example of a constraint would be mutually exclusive roles such as the Manager and the Stock Controller roles. A user may not be a member of both of these roles, as this would create an opportunity for fraud. This principle is known as separation of duties.

Due to the nature of the concept of the constraint, numerous types can be identified. These types of constraints include:

Mutually exclusive roles, which ensure that two roles cannot be assigned to the same user. This helps to ensure separation of duty requirements.

Mutually exclusive permissions, which provides additional assurances for ensuring separation of duty requirements.

Dual constraint on permission assignment, which restricts a permission from being assigned to more than one role in a mutually exclusive set.

User assignment constraints, which, among many possible uses, could restrict the number of users that may be assigned to a role.

Not all of the possible constraints would be needed in a system. The constraints and the conflicts between entities that are used within this research are discussed in more detail in chapter 6.

Also, the concept of the session is not considered within this research, as access control requirements and separation of duty requirements do not need the session entity in order to be enforced.

3.4.4 Formal Summary of RBAC96

Consider a short formal summary of the relevant components in the RBAC96 model, based on the work of Sandhu et al (1996) and Ahn & Sandhu (1999):

Definition 3.1 formalises the entities that are evident.

Definition 3.1: RBAC entities

There are sets of users, roles and permissions.

U = set of users, $\{u_1,u_2,\ldots,u_l\}$

R = set of roles, $\{r_1,r_2,\ldots,r_m\}$

P = set of permissions, $\{p_1,p_2,\ldots,p_n\}$

Definition 3.2 formalises the associations that can exist between the entities.

Definition 3.2: RBAC associations

Users and permissions can be associated with roles and roles can form a hierarchy with other roles.

$UA \subseteq U \times R$, a many-to-many user-to-role assignment relation

$PA \subseteq P \times R$, a many-to-many permission-to-role assignment relation

$RH \subseteq R \times R$, a partial order on R called the role hierarchy, also written as £

Definition 3.3 formalises the functions that are used to ascertain the roles that a user or permission may be associated with. This is necessary in order to know which users are allowed to receive the different worklist items that are generated by the workflow enactment service.

Definition 3.3: Roles function

$roles$: $U \cup P \rightarrow 2^R$, a function mapping the sets U and P to a set of roles

$roles^*$: $U \cup P \rightarrow 2^R$ extends roles in the presence of a role hierarchy

$roles(u_i)=\{r \in R \mid (u_i,r) \in UA\}$

$roles(p_i)=\{r \in R \mid (p_i,r) \in PA\}$

$roles^*(u_i)=\{ r \in R \mid (\exists r' \pounds r')[(u_i,r') \in UA]\}$

$roles^*(p_i)=\{ r \in R \mid (\exists r' \pounds r)[(p_i,r') \in PA]\}$

Note that the definition of $roles^*$ is carefully formulated to reflect the role inheritance with respect to users going downwards and with respect to permissions going upwards.

Definition 3.4 formalises the functions that are used to ascertain the permissions that a user may be associated with through the user's role

association. This is necessary in order to know when a user may have access to a system object.

Definition 3.4: Permissions function

perm: $U \cup R \rightarrow 2^P$, a function mapping users and roles to a set of permissions.

*perm**: $U \cup R \rightarrow 2^P$, extends *perm* in the presence of a role hierarchy.

$perm(r_i) = \{p \in P \mid (p,r_i) \in PA\}$

$perm(u_i) = \{p \in P \mid (\exists r \in roles(u_i))[(p,r) \in PA]\}$

$perm^*(r_i) = \{p \in P \mid (\exists r' \leq r_i)[(p,r') \in PA]\}$

$perm^*(u_i) = \{p \in P \mid (\exists r \in roles^*(u_i))[(p,r) \in PA]\}$

Additions can now be made to these formalisations in order to extend RBAC's functionality. As such, RBAC can be adapted in order to enhance its effectiveness within a workflow environment.

3.5 Conclusion

This chapter contributed to this research by firstly, introducing information security and the information security services, and then discussing access control and role based access control. The formal definition of the RBAC environment will allow for the expansion of the RBAC environment to cater for the needs of the workflow environment.

RBAC is not considered to be perfect for every access control scenario. This is particularly evident with situations where sequences of operations will necessitate more complex control. An example of this is a purchase requisition where certain steps must be accomplished before the item can be issued. Other types of access control mechanisms could be built onto RBAC to extend its functionality for this purpose (Sandhu et al., 1996). Control of sequence is generally the domain of workflow systems.

Thomas and Sandhu have argued that it is time to move towards new paradigms in access control (1993). They have stated that authorisation (access control) in distributed applications should be distinguished in terms of tasks rather than individual objects. This observation is partly based upon the emergence of workflow type software applications within organisations. This

research takes the concept of tasks and integrates it within the RBAC environment to actively support workflow environments.

It is important for an organisation to prevent fraud; one way to accomplish this is to use separation of duty (Thomas & Sandhu, 1993). Separation of duty aims to prevent fraud by separating the responsibility for a process between different users. This enforcement of separation of duty requirements could be accomplished through the use of constraints within the RBAC environment. Separation of duty forms an essential component of this research and as such it will be discussed in the next chapter.

Chapter 4.

Separation of Duty

A dominant security issue within organisations is the prevention and restriction of fraud. The principle of separation of duty (SoD) is a time-honoured principle for achieving this goal. Separation of duty involves never allowing an individual to have enough privileges within the organisation to commit fraud on his own (Sandhu, 1990). This is achieved through the breaking up of business processes into smaller tasks and having these tasks performed by different users.

A more formal definition of SoD is that it is a security principle that is used to formulate multi-person control policies, requiring two or more distinct people to be responsible for the completion of a task or set of tasks (Simon & Zurko, 1997). In doing so, fraud is discouraged by the distribution of the responsibility between more than one user. The spreading of the responsibility raises the risks of fraud by necessitating the involvement of more than one individual.

Users only deal with smaller tasks that constitute a business process within a workflow environment. And as such, the workflow environment could support SoD principles.

This is achieved in the workflow example, which has been covered in chapter 2, by making use of three different roles to complete the various tasks when processing an internal purchase order. Firstly, there is the employee role that completes the order form, after which the order may be approved by the manager role, and finally there is the stock controller role that gets the ordered item to the employee. Fraud can be committed at various positions within this process.

An example of a fraudulent act is if the person who is the manager places an order for something expensive and then proceeds to approve the order. By preventing the manager from approving his own order, fraud is prevented. This is an example of one SoD requirement in the cited example.

To understand how SoD is used and how it fits into the workflow environment, it is imperative to consider its history.

4.1 Related SoD Research

The term "separation of privilege" was identified as one of eight design principles for the protection of information in computer systems by Saltzer and Schroeder (1975). They built on the observation that a security system with two keys is more robust and flexible than one that requires a single key. No single accident, deception or breach of trust is therefore sufficient to compromise the system.

Clark and Wilson (1987) identified separation of duty as one of the two major mechanisms that can be implemented to ensure data integrity. SoD serves as a mechanism to counteract fraud and error, whilst assuring correspondence between system objects and the real world objects that they represent. They asserted that, at the policy level, processes are divided into steps, with each step being performed by a different person. Separation of duty is thus tightly connected to application semantics.

The issue of separation of duty has been addressed from different perspectives by several authors. Examples can be found in various references: (Baldwin, 1990; Gligor, Gavrila & Ferraiolo, 1998; Nash & Poland, 1990; Sandhu, 1988; Sandhu, 1990; Simon & Zurko, 1997). Here, only work in which concepts are directly quoted in our interpretation, is examined.

Kuhn (1997) explored the mutual exclusion of roles as a means of expressing separation of duty requirements. He presented a taxonomy whereby separation of duty requirements are categorised according to the time at which mutual exclusion is applied (static vs. dynamic), as well as the degree to which privileges are shared by mutually exclusive roles (strong or partial exclusion).

Strong exclusion, on the one hand, implies no common permission or user assignments for exclusive roles. Partial exclusion, on the other hand, implies that mutually exclusive roles may share permissions (or users) but that each role should have permissions assigned that are unique to that role.

Nyanchama and Osborn (1999) discussed various types of conflicts that have to be considered when implementing separation of duty requirements. They evaluated the effect of role hierarchies in great depth in terms of their role-graph model.

Ahn and Sandhu (1999) defined the RSL99 language for specifying separation of duty constraints. They based their SoD requirements on the concepts of conflicting users, conflicting roles and conflicting permissions. The RSL99 language can be used to study SoD within RBAC environments as it helps with the identification of SoD properties that may not have been known or understood.

SoD requirements are enforced through constraints. It is necessary to understand the foundation for the constraints and the use of the constraints in order to apply separation of duties to its full effect.

4.2 A Taxonomy of SoD Constraints

Simon and Zurko (1997) have formally defined a variety of different forms of SoD. All of these variations fall under two main categories, which are: static SoD and dynamic SoD. This is shown in figure 4.1.

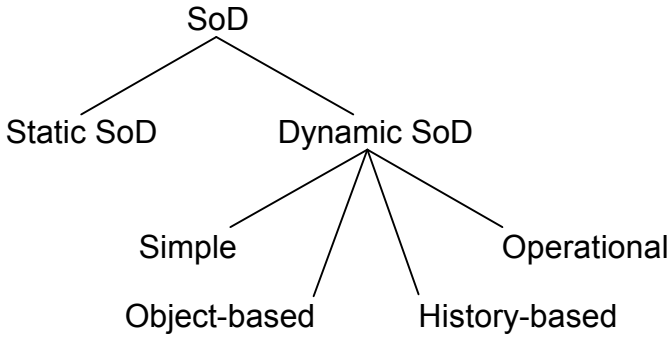

Figure 4.1 Categories of Separation of Duties.

Particular attention is paid to static SoD due to the scope of the research, which is outlined in chapter 5.

4.2.1 Static Separation of Duty

Static SoD constraints are considered the simplest variation of SoD (Ahn & Sandhu, 1999). Constraints can exist for each of the RBAC entities.

This includes constraints such as not allowing the same user to be a member of two conflicting roles. If a user were associated to both of the conflicting roles, then that user would be able to commit fraud. An example of this is the manager role and the stock controller role from the workflow example in chapter 2. These roles are conflicting and as such a user may not be associated to both of them.

Another constraint is that a user may not be associated to conflicting permissions. Since permissions are obtained through role associations, care must be taken when associating permissions to roles. An example of this is the permissions needed to submit an order and to approve an order. These permissions are conflicting, and as such may not be associated to a common user through the role associations. This means that conflicting permissions may not be associated to a common role.

These constraints are summarised in table 4.1.

Conflicting Entities	**Cannot be associated to:**
Conflicting Roles	Common User
Conflicting Permissions	Common Role

Table 4.1 Static SoD constraints.

Other constraints exist and are identifiable through the use of methods such as Ahn and Sandhu's RSL99 language (1999). Chapter 6 formalises the constraints that have been identified during this research.

These strongly exclusive relationships could be very useful in the administration environment as an aid to the security administrator. It can also form the basis for dynamic SoD.

4.2.2 Dynamic Separation of Duty

Dynamic SoD provides an improved set of possible policies. This is done by the controlled activation and use of the roles in a system. As long as the constraints are satisfied, users may be members of what would be considered strongly exclusive roles in a static SoD environment.

An example of this would be a user who places an order as a member of the Employee role but may not approve the order as a member of the manager role. This allows dynamic SoD to reflect the functioning of an organisation.

There are many variations of dynamic SoD.

Simple Dynamic SoD is the simplest variation that states that restricted roles may have the same members but a member may only be assigned to one role at a time.

Object-based SoD states that restricted roles may have common members and these members may assume both roles at the same time. Users may not act upon any system object that they have previously operated upon.

Operational SoD implies that restricted roles may contain common members as long as the union of the activities the roles perform does not contain all the activities in a complete business process. It prevents any one person from performing an entire business process.

History-based SoD handles various desirable actions that are prevented by the previously mentioned SoD variations. It does this by the combination of object-based SoD and operational SoD.

This research is only concerned with the specification and not the enforcement of dynamic SoD.

4.3 Conclusion

The concept of separation of duties was introduced within this chapter. This knowledge is essential in understanding how the RBAC environment needs to function within the workflow environment in order to ensure that SoD requirements are enforced.

The purpose of SoD policies is to prevent fraud. In doing so, an individual user should be prevented from receiving mutually exclusive tasks. Stated in another way, a user should not receive access rights to system objects that could enable the user to commit fraud. This dissertation refers to access rights as permissions, which is one of the entities of the RBAC model.

To prevent users from receiving mutually exclusive permissions, these permissions can be set as conflicting. This *conflicting* paradigm is applicable to the other RBAC entities and to workflow extensions to the RBAC model in this research. These conflicts and how they associate to the administration paradigm are discussed in-depth in chapter 6.

In order to clearly discuss the conflict paradigm, it is important to show the scope of the research, and its conceptual framework.

Chapter 5.

SoDA: The Concept and Model

With the increasing amount of information available electronically it is not only necessary to find a means to ease the job of the security administrator, but also to ensure that the information is protected and managed according to organisational policies. On the one hand, RBAC has been promoted as a possible solution to the administration nightmares that face security administrators (Ferraiolo, Barkley, & Kuhn, 1999). On the other hand, workflow technology has been boasted as a means of controlling the flow of information according to business process models. RBAC mechanisms employed in the workflow environment should thus be sensitised to the context of the work (Cholewka et al., 2000; Thomas & Sandhu, 1993).

The context of the work is determined by factors such as the sequence and history of events, as well as the organisational policies. One expression of organisational policy can be found in the age-old principle of separation of duty.

The primary objective of separation of duty is to prevent fraud, i.e. protect the integrity of the information (Clark & Wilson, 1987). SoD can be enforced through the correct use of access control mechanisms.

Access control is a two-phase process as depicted in figure 5.1. During phase one, users receive potential to perform certain activities – this is called access control administration. Phase two occurs when an application is used and the actual permissions are granted to the user – this is called run-time access control.

Chapter 4 showed that SoD requirements could be evaluated and enforced at two points in time. This can happen through the administration tool and through the run-time environment. When evaluation and enforcement occur in the administration tool it is referred to as *static separation of duty*. When evaluation and enforcement occur within the run-time environment it is referred to as *dynamic separation of duty*.

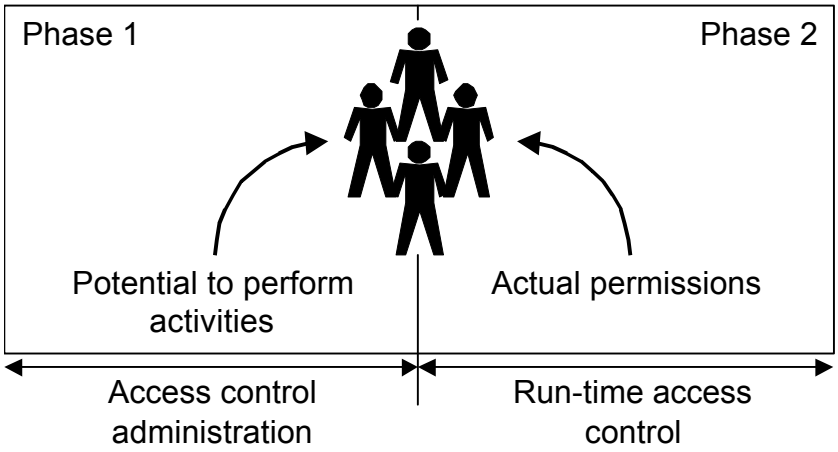

Figure 5.1 Access control phases

Static SoD requirements include constraints such as not allowing conflicting permissions to be assigned to a common user. This type of requirement would be evaluated when assigning a user to a role and when assigning permissions to roles. If the assignment is going to cause constraint errors, then the administration environment can prevent the assignment.

An example of a dynamic SoD requirement would be to prevent a common user from working on conflicting tasks in the same workflow process. This SoD requirement can be enforced within the run-time environment by not allowing the user access to the conflicting task.

These concepts are central to the scope of this dissertation. In order to understand the model it is important to appreciate the scope of the proposed solution.

5.1 Scope of Proposed Solution

The proposed model focuses on the technical enforcement of static separation of duty requirements in workflow systems. The scope of the proposed model is, however, best defined by considering its information technology scope and its information security scope.

5.1.1 Information Technology Scope

The information technology scope of the research, which is presented in this dissertation, is primarily described in terms of the information technology

domain it addresses, namely, workflow systems. Moreover, workflow systems have been identified in chapter 2 as consisting of administration and run-time components.

The information technology focus is thus on workflow systems, and in particular, the administration component of workflow systems. Figure 5.2 shows the information technology scope by means of concentric circles. The outer circle encapsulates information technology while the larger dashed inner circle encapsulates workflow systems. The workflow circle is divided into a run-time area and an administration area, which is shown as a grey disk in the centre of the circles.

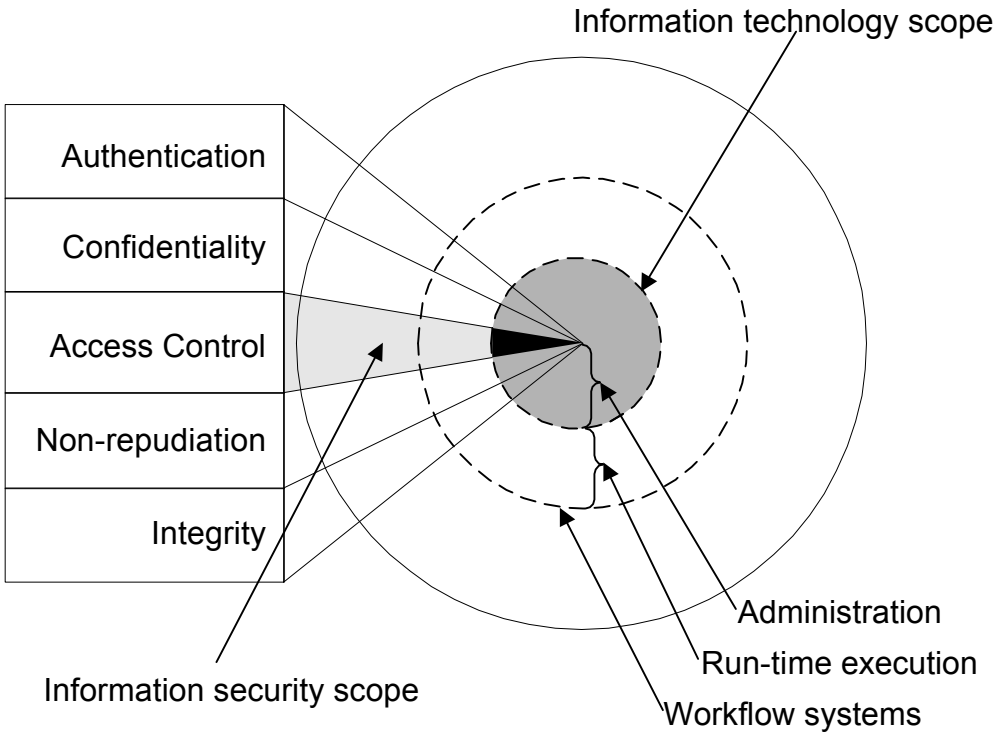

Figure 5.2 Scope of the proposed model

It is also important to know what areas of information security this research focuses on.

5.1.2 Information Security Scope

The triangle in figure 5.2 represents the field of information security. It is shown to overlap with information technology, in particular, also workflow systems.

Information security has been shown in chapter 3 to involve five security services: authentication, confidentiality, integrity, access control and non-repudiation. These services are depicted as slices of the information security triangle in figure 5.2. The access control service was chosen as the information security scope of this research, as indicated by the shaded portion of the triangle.

The darker shading indicates the overlap between the information technology scope and the information security scope. This shaded overlap represents the scope of this research and can be defined as the administration of access control for workflow systems.

A more detailed look at the research scope will shed light on the conceptual overview wherein this research was conducted.

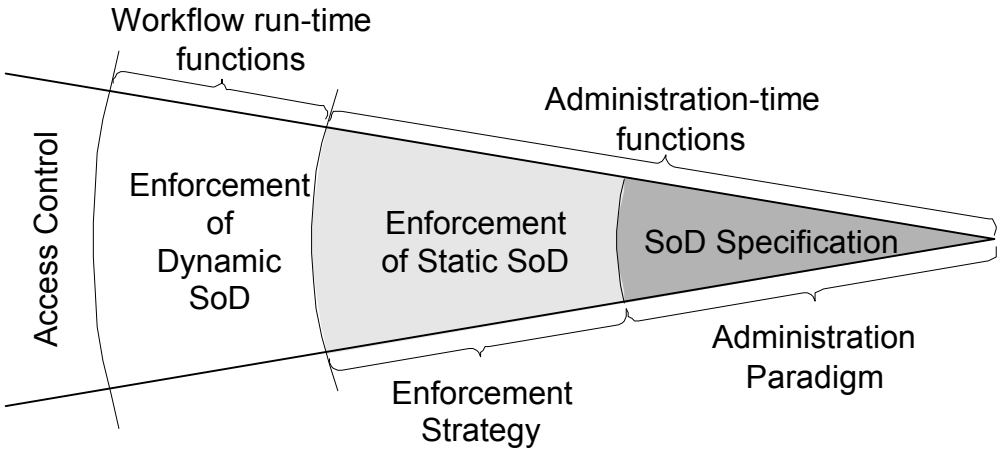

Figure 5.3 Detail scope for SoDA

5.2 Conceptual Overview

The proposed model constitutes two basic components: an administration paradigm and an enforcement strategy.

These components can be observed in terms of its relation to separation of duty in figure 5.3, which provides a magnified view of the darkly shaded area in figure 5.2. The administration paradigm involves the specification of access control, while the enforcement strategy is concerned with enforcing static access control requirements, specifically, requirements that can be checked at administration time. Note that the enforcement of access control

requirements that rely on the workflow execution environment falls explicitly outside the scope of this research.

Figure 5.4 depicts these two components graphically. Consider each in turn.

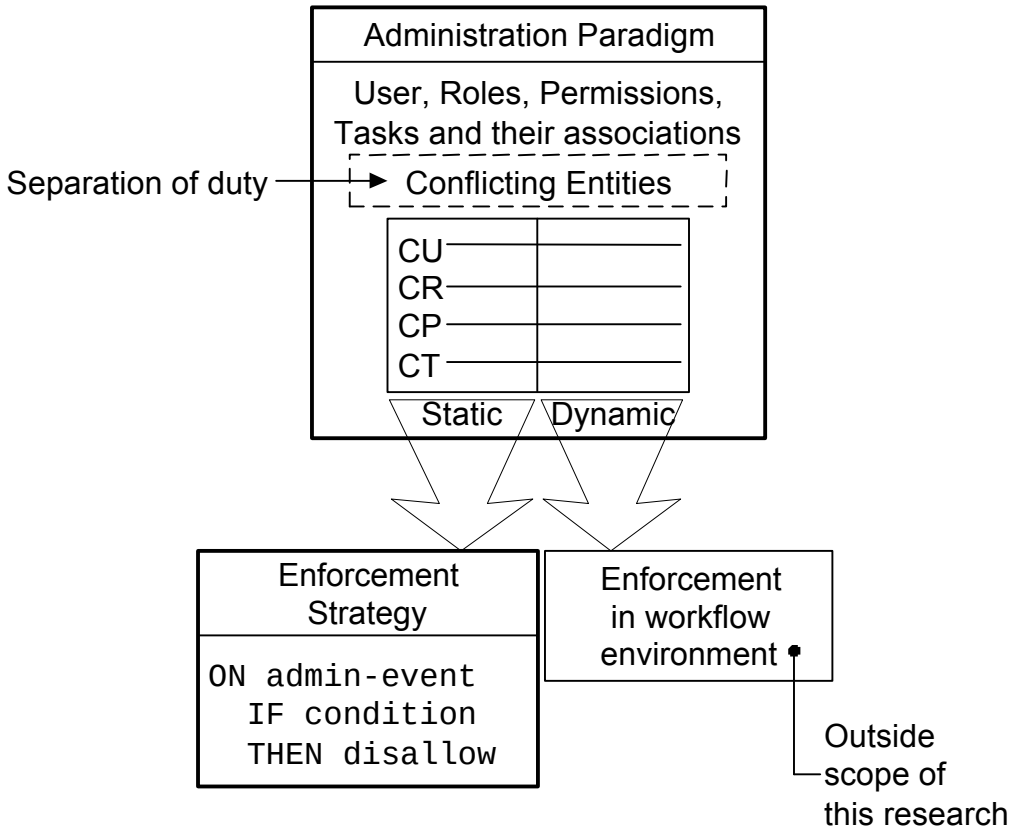

Figure 5.4 SoDA: Conceptual view

5.2.1 The administration paradigm

The proposed model strongly hinges on role-based access control. The identification of users, roles and permissions thus is integral to the model. In addition to these entities the concept of a task is introduced. Administration tools will allow for the associations between users, roles, permissions and tasks. These associations may be constrained according to static separation of duty requirements. The “Conflicting Entities” administration paradigm is introduced as a way of expressing these constraints. This paradigm is based on the concept of mutual exclusion in sets, i.e. that two tuples may not belong to the same set. A distinction is made between the time of exclusion, viz. at administration-time or run-time. Run-time exclusion must be managed by the workflow system and is thus not further addressed.

5.2.2 The enforcement strategy

The administration time exclusions are addressed in the second component by providing an enforcement strategy. The enforcement strategy is based on active database technology and is thus defined in terms of event-driven action rules.

The objective of the security administration paradigm is to disallow assignments or associations that will violate the static SoD requirements. It will, by default, disallow these prohibited actions. Algorithms to check the conditions have been developed as part of the enforcement strategy.

5.3 Conclusion

The current chapter set the scope of the SoDA model and provided a conceptual overview of its components. Chapters 6 to 8 will discuss these components in more detail.

Chapter 6 will develop an administration paradigm and state its properties for the static aspect mathematically.

Chapter 7 will develop algorithms for the use by an implementation of the SoDA model. These algorithms will ensure that the static separation of duty requirements remain valid.

Chapter 8 will describe the implementation details of the SoDA prototype implementation.

Chapter 6.

SoDA: The Administration Paradigm

As described in chapter 5, the scope of the model involves the specification of static and dynamic SoD and the enforcement of static SoD. These operations occur within the administration environment of a workflow system. SoD is built upon the access control specifications.

In order to specify the access control requirements, it is necessary to identify the entities involved.

This chapter will formalise the aspects of the model by:

- Introducing the entities involved.
- Introducing the conflicting entities administration paradigm.
- Developing a model of the integrity requirements for static SoD.

The information that the workflow environment adds to the RBAC environment is referred to as the workflow extensions.

6.1 Workflow extensions

The proposed extensions to the RBAC environment build upon the definitions already outlined in chapter 3. For notational convenience they are reproduced in table 6.1.

A typical process definition is a set of tasks linked together in a network, thus forming a business process (Hollingsworth, 1995). The workflow system is responsible for determining the route that work will follow through the organisation. From an access control perspective the basic building blocks are tasks that may be performed by a specific organisational role.

Figure 6.1 presents the RBAC entities with the task entity added. The constraints are not shown to maintain clarity.

RBAC entities U = set of users, $\{u_1,u_2,\ldots,u_l\}$ R = set of roles, $\{r_1,r_2,\ldots,r_m\}$ P = set of permissions, $\{p_1,p_2,\ldots,p_n\}$
RBAC associations $UA \subseteq U \times R$, a many-to-many user-to-role assignment relation $PA \subseteq P \times R$, a many-to-many permission-to-role assignment relation $RH \subseteq R \times R$, a partial order on R called the role hierarchy, also written as £
Roles function *roles*: $U \cup P \rightarrow 2^R$, a function mapping the sets U and P to a set of roles *roles** : $U \cup P \rightarrow 2^R$ extends roles in the presence of a role hierarchy $roles(u_i)=\{r \in R \mid (u_i,r) \in UA\}$ $roles(p_i)=\{r \in R \mid (p_i,r) \in PA\}$ $roles^*(u_i)=\{ r \in R \mid (\exists r \text{ £ } r')[(u_i,r') \in UA]\}$ $roles^*(p_i)=\{ r \in R \mid (\exists r' \text{ £ } r)[(p_i,r') \in PA]\}$
Permissions function *perm*: $U \cup R \rightarrow 2^P$, a function mapping users and roles to a set of permissions. *perm**: $U \cup R \rightarrow 2^P$, extends *perm* in the presence of a role hierarchy. $perm(r_i) = \{p \in P \mid (p,r_i) \in PA\}$ $perm(u_i) = \{p \in P \mid (\exists r \in roles(u_i))[(p,r) \in PA]\}$ $perm^*(r_i) = \{p \in P \mid (\exists r' \leq r_i)[(p,r') \in PA]\}$ $perm^*(u_i) = \{p \in P \mid (\exists r \in roles^*(u_i))[(p,r) \in PA]\}$

Table 6.1 RBAC definitions

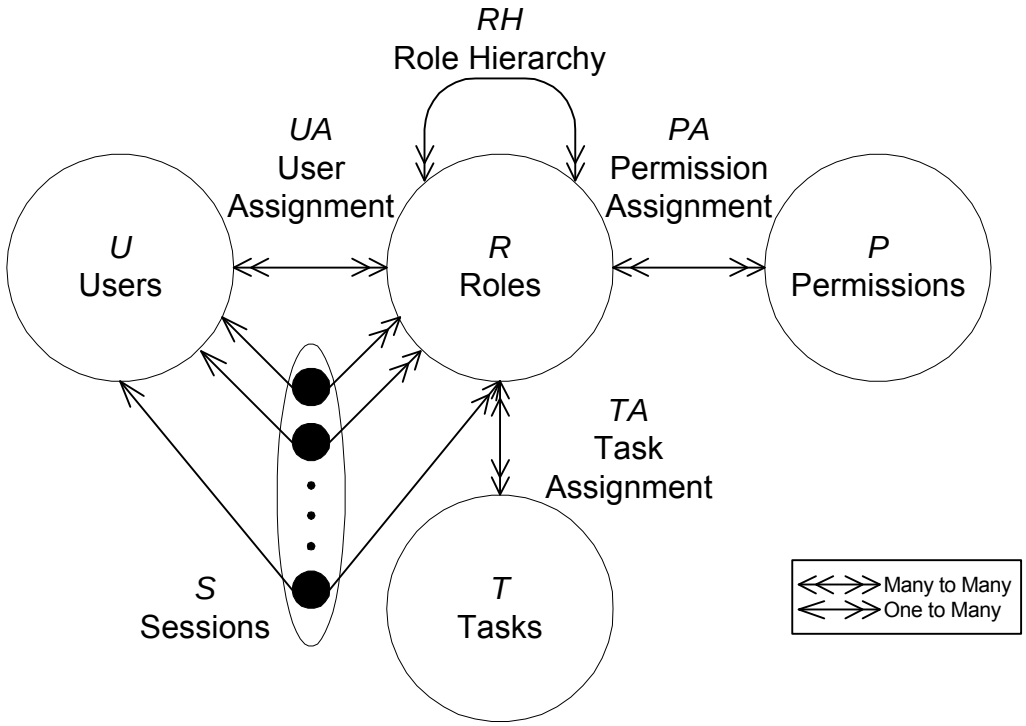

Figure 6.1 Role-based access control model extended with tasks

The task entities form a many-to-many relationship with the role entities just as the user and permission entities do. The following formalisations add the concept of the workflow's task to the formal RBAC definitions that were summarised in table 6.1.

Definition 6.1: Workflow entities

There are sets of tasks.

T = Set of tasks, $\{t_1,t_2,\ldots,t_n\}$, a set of task definitions.

The task entities will relate to the role entities as defined in the following definition.

Definition 6.2: Workflow Associations

Tasks can be associated with roles.

$TA \subseteq T \times R$, a many-to-many task-to-role assignment relation.

The set TA, as defined in definition 6.2, will contain the task to role associations that will be created. The RBAC96 functions must thus be extended to cater for the addition of this relationship.

Definition 6.3: Extended roles function

roles: $U \cup P \cup T \rightarrow 2^R$, a function mapping the sets U and P and T to a set of roles.

*roles** : $U \cup P \cup T \rightarrow 2^R$ extends *roles* in the presence of a role hierarchy.

roles(u_i),*roles*(p_i), *roles**(u_i) and *roles**(p_i) remain according to Definition 3.3.

roles$(t_i)=\{r \in R \mid (t_i,r) \in TA\}$

*roles**$(t_i)=\{ r \in R \mid (\exists r' \geq r')[(t_i,r') \in TA]\}$

The standard roles functions that are listed in table 6.1 have been adapted to cater for the addition of the task entity. The permission functions also need to be adapted.

Definition 6.4: Revised permissions function

perm: $U \cup R \cup T \rightarrow 2^P$, a function mapping users, roles and tasks to a set of permissions.

*perm**: $U \cup R \cup T \rightarrow 2^P$, extends *perm* in the presence of a role hierarchy.

perm(r_i), *perm*(u_i), *perm**(r_i) and *perm**(u_i) remain according to Definition 3.4.

$perm(t_i) = \{p \in P \mid (\exists r \in roles(t_i))[(p,r) \in PA]\}$

$perm^*(t_i) = \{p \in P \mid (\exists r \in roles^*(t_i))[(p,r) \in PA]\}$

The revised definitions give the elements which are essential to the administration of access control in the workflow environment. The following section will suggest a conflict paradigm to define further restrictions required to support static SoD requirements.

6.2 The conflicting entities administration paradigm

The concept of conflicting entities needs to be described in order to understand how SoD can be enforced. Any two of the identified entities may be conflicting. That is, any two permissions, users, roles, or tasks may be conflicting. Integrity is maintained by constraining the associations between entities based on the conflicts between entities. As such it is important to understand the conflicts that can occur.

The term *conflicting permissions* indicates two permission entities that are in conflict with one another. This means that if two conflicting permissions were associated to a common user, then that user would be able to commit fraud. An example of conflicting permissions is the permissions needed to submit an order and to approve an order.

If a user were associated to both of these permissions then that user would be able to approve a self-placed order. By assigning a conflict between the two permissions it becomes possible for the administration environment to constrain any associations between the permissions and the users.

The term *conflicting users* is misleading. The users are in fact not conflicting at all. Rather, it is the user's friendship that can result in an alliance to commit fraud. With this description it becomes relevant to consider that single users are in conflict with themselves.

It is important to realise that permissions are not directly associated to users. Users receive permissions through their associations with roles. Figure 6.1 displays how the permission, user and task entities all connect with the role entities with many-to-many relationships. In this way users will receive the permissions that have been associated with the roles that they are associated with. Permissions will also be associated with those tasks that share common roles. It is therefore important to understand the consequences of assigning conflicts between roles.

The term *conflicting roles* indicates two role entities that are in conflict with one another. It is through conflicting roles that the associations between all the entities can be controlled. An example of this is how conflicting permissions are prevented from being associated to conflicting users. This prevention is achieved by constraining the associations between permissions, roles, and users.

For example, the manager and employee roles have to be conflicting before being associated to the permissions needed to submit an order and to approve an order. But this constraint alone is not enough to prevent conflicting users from being associated with the conflicting permissions.

In order to prevent conflicting users from being associated with conflicting permissions, conflicting users must be constrained from being associated with conflicting roles. This constraint will prevent conflicting users from receiving conflicting permissions.

Another constraint for conflicting roles is that conflicting roles cannot form part of a role network. This is due to the fact that permissions get inherited from the roles in a role network. If the employee and manager roles were part of a role network, then the manager role will inherit the permissions that are associated to the employee role. In so doing, any user associated to the manager role would receive conflicting permissions.

From the workflow environment's point of view, the users and permissions required to perform a task are obtained from role associations. It is with *conflicting tasks* that SoD can be enforced within the workflow environment. The term *conflicting task* indicates two task entities that are in conflict with one another.

Tasks that are conflicting are workflow process tasks that may not be performed by conflicting users. If conflicting users were allowed to perform these tasks then there exists the possibility of fraud being committed. An example of two conflicting tasks can be seen in Figure 2.2. This figure shows the tasks that make up a workflow process and the "complete order form" task and the "approve order" task are conflicting tasks.

In order to prevent these tasks from receiving conflicting users it is necessary to restrict their associations with roles in the same way as conflicting permissions are constrained. This is accomplished by not allowing conflicting tasks to be associated with non-conflicting roles. Therefore, the "complete order form" task and the "approve order" task must be associated with conflicting roles such as the employee role and the manager role.

It has been shown that the conflicting entities paradigm can be used to enforce static and dynamic SoD. Although conflicts between entities can be used to enforce a variety of constraints within the run-time environment, this is beyond the scope of this research and as such is not covered.

The conflicting entities administration paradigm has been discussed in an informal manner. The following definitions formalise the aspects required for enforcement. Conflicting permissions are formalised first as power is vested in permissions.

Definition 6.5: Conflicting permissions are permissions that can result in unnecessary power if bestowed on the same person. Formally it is represented by

$CP \subseteq P \times P$, a many-to- many relation indicating conflict between permissions with

$(p_i,p_j) \in CP \Leftrightarrow (p_j,p_i) \in CP$ and $(p_i,p_i) \notin CP$.

We can now present the following axiom, which will represent our basic safety condition. The basic safety condition is the basic requirement for all proofs. If an association between entities violates this basic safety condition then that association is not to be allowed.

Basic Safety Condition: Conflicting permissions may not be assigned to a user.

Formally, $(perm^*(u) \times perm^*(u)) \cap CP = \varnothing)$

Since non-conflicting permissions cannot influence the basic safety condition the following axiom, to supplement the basic safety condition, is formulated.

Axiom 6.5: Non-conflicting permissions may be assigned to either conflicting or non-conflicting roles.

It is also important to look at the other conflicting entities that form part of the conflicting entity paradigm.

Definition 6.6: Conflicting users are users who are likely to conspire. Formally they are represented by

$CU \subseteq U \times U$, a many-to-many relation indicating conflict between users with

$(u_i,u_j) \in CU \Leftrightarrow (u_j,u_i) \in CU$ and $(u_i,u_i) \notin CU$.

It can be seen that a single user could commit fraud without conspiring with other users. As such, the following axiom is stated.

Axiom 6.6: Conflicting users are considered as a single user.

In practical terms conflicting users may be family members or people who are known to have conspired.

Definition 6.7: Conflicting roles are roles that together have the ability to conspire, i.e. they are assigned some (but not all) conflicting permissions. They are represented by

$CR \subseteq R \times R$, a many-to-many relation indicating conflict between roles with

$(r_i,r_j) \in CR \Leftrightarrow (r_j,r_i) \in CR$, $(r_i,r_i) \notin CR$ and

$(r_i,r_j) \in CR \Rightarrow perm^*(r_i) \times perm^*(r_j) \cap CP \neq \varnothing$

Note that roles are abstractions to ease administration. Although the conflicting permissions may not be identified as such in the administration tool, making roles conflict if they are not assigned some conflicting permissions is senseless. This principle thus is a logical principle, which in practice may not be checked literally in the administration tool.

Since conflicting roles must have some conflicting permissions we can state that non-conflicting roles do not have conflicting permissions. In the spirit of Axiom 6.5, the following axiom is formulated.

Axiom 6.7: Non-conflicting roles may be assigned either non-conflicting or conflicting users.

Definition 6.8: Conflicting tasks are tasks requiring conflicting permissions to complete. Formally they are represented by

$CT \subseteq T \times T$, a many-to-many relation indicating conflict between tasks with

$(t_i,t_j) \in CT \Leftrightarrow (t_j,t_i) \in CT$, $(t_i,t_i) \notin CT$ and

$(t_i,t_j) \in CT \Rightarrow perm^*(t_i) \times perm^*(t_j) \cap CP \neq \varnothing$

Note that conflicting tasks are assigned conflicting permissions. Since non-conflicting tasks can have only non-conflicting permissions assigned to them, we can see that they could not influence the basic safety condition, therefore, the following axiom is formulated.

Axiom 6.8: Non-conflicting tasks may be assigned to conflicting and non-conflicting roles.

These principles and definitions are essentially focused on the permissions exercised by the users. The integrity of the access control information is, however, determined by the associations in the access control model. In a RBAC environment users are never assigned directly to permissions. The role construct plays a pivotal role in linking tasks, users and permissions together. The next section will therefore show the integrity requirements pertaining to the associations allowed in the access control model.

6.3 Integrity Requirements

The previous section dealt with the definitions for conflicting entities. It also introduced the basic safety condition. This section presents a number of theorems reflecting integrity requirements that will have to be upheld in a security administration tool.

These theorems outline a model of enforcement that can be used to enforce SoD within the administration tool. These theorems can be considered as being rules or constraints. These theorems are enforced through the use of conflicting entities.

Theorem 6.9: Under the basic safety condition, conflicting roles may only have non-conflicting users assigned to them, i.e.

$$(u_i,r_k) \in UA \wedge (u_j,r_l) \in UA \wedge (r_k,r_l) \in CR \Rightarrow (u_i,u_j) \notin CU$$

Proof:

Assume that $(u_i,r_k) \in UA \wedge (u_j,r_l) \in UA) \wedge (r_k,r_l) \in CR. \wedge (u_i,u_j) \in CU$:

$perm^*(u_i) \supseteq perm^*(r_k)$ (Def 3.4)

$perm^*(u_j) \supseteq perm^*(r_l)$ (Def 3.4)

$(r_k,r_l) \in CR.$

$\Rightarrow$ $perm^*(r_k) \times perm^*(r_l) \cap CP \neq \varnothing$ (Def 6.7)

$\Rightarrow$ $perm^*(u_i) \times perm^*(u_j) \cap CP \neq \varnothing$

which contradicts the Basic Safety Condition. ***QED.***

Theorem 6.10: Under the basic safety condition, conflicting permissions may only be assigned to conflicting roles. Formally

$$(p_i,r_k) \in PA \wedge (p_j,r_l) \in PA \wedge (p_i, p_j) \in CP \Rightarrow (r_k, r_l) \in CR$$

Proof:

Assume that two conflicting permissions p_i and p_j are assigned to non-conflicting roles r_k and r_l, i.e.

$$(p_i,r_k) \in PA \wedge (p_j,r_l) \in PA \wedge (p_i,p_j) \in CP \wedge (r_k,r_l) \notin CR$$

Choose a user u_x and associate it with roles r_k and r_l. Since $(r_k,r_l) \notin CR$ this is allowed by Th. 6.9.

$\therefore \quad (u_x,r_k) \in UA \wedge (u_x,r_l) \in UA \wedge$

$(p_i,r_k) \in PA \wedge (p_j,r_l) \in PA$

$\Rightarrow \quad \{p_i, p_j\} \subseteq perm^*(u_x)$ (Def 6.6)

But $(p_i, p_j) \in CP$, which contradicts the Basic Safety Condition.

QED.

Theorem 6.11: Under the basic safety condition, conflicting tasks may only be assigned to conflicting roles. That is

$(t_i,r_k) \in TA \wedge (t_j,r_l) \in TA \wedge (t_i,t_j) \in CT \Rightarrow (r_k, r_l) \in CR$

Proof:

Assume that two conflicting tasks t_i and t_j are assigned to non-conflicting roles r_k and r_l.

$(t_i,r_k) \in TA \wedge (t_j,r_l) \in TA \wedge (t_i,t_j) \in CT \wedge (r_k,r_l) \notin CR$

Choose a user u_x and associate it with roles r_k and r_l. Since $(r_k,r_l) \notin CR$ this is allowed by Th. 6.9.

$perm^*(r_k) \subseteq perm^*(u_x)$ (Def 3.4)

$perm^*(r_l) \subseteq perm^*(u_x)$ (Def 3.4)

also $perm^*(t_i) \subseteq perm^*(r_k)$ (Def 6.4)

$perm^*(t_j) \subseteq perm^*(r_l)$ (Def 6.4)

$\therefore \quad perm^*(t_i) \subseteq perm^*(u_x)$

and $perm^*(t_j) \subseteq perm^*(u_x)$

$\Rightarrow \quad perm^*(t_i) \times perm^*(t_j) \cap CP \neq \varnothing$ (Def 6.8)

$\Rightarrow \quad perm^*(u_x) \times perm^*(u_x) \cap CP \neq \varnothing$

which contradicts the Basic Safety Condition. ***QED.***

Using truth table equivalence we state the following corollary.

Corollary 6.11: Under the basic safety condition, non-conflicting roles may only have non-conflicting tasks assigned to them. That is

$(t_i,r_k) \in TA \wedge (t_j,r_l) \in TA \wedge (r_k, r_l) \notin CR \Rightarrow (t_i,t_j) \notin CT$

These theorems limit the associations that can be allowed between users, roles, permissions and tasks. By using the conflicting entities they are able to ensure that integrity is maintained. These integrity requirements are essential for the enforcement of SoD.

The following table summarises the results of this chapter.

<table>
<tr><td colspan="2" rowspan="2">May be associated with</td><td colspan="4">Roles</td></tr>
<tr><td colspan="2">Conflicting</td><td colspan="2">Non-conflicting</td></tr>
<tr><td rowspan="2">Users</td><td>Conflicting</td><td rowspan="2">Th 6.9</td><td>×</td><td rowspan="2">Ax 6.7</td><td>✓</td></tr>
<tr><td>Non-Conflicting</td><td>✓</td><td>✓</td></tr>
<tr><td rowspan="2">Permissions</td><td>Conflicting</td><td colspan="2">✓</td><td colspan="2">×</td></tr>
<tr><td colspan="4">Theorem 6.10</td></tr>
<tr><td rowspan="2"></td><td>Non-Conflicting</td><td colspan="2">✓</td><td colspan="2">✓</td></tr>
<tr><td colspan="4">Axiom 6.5</td></tr>
<tr><td rowspan="2">Tasks</td><td>Conflicting</td><td colspan="2">✓</td><td colspan="2">×</td></tr>
<tr><td colspan="4">Theorem 6.11</td></tr>
<tr><td rowspan="2"></td><td>Non-Conflicting</td><td colspan="2">✓</td><td colspan="2">✓</td></tr>
<tr><td colspan="4">Axiom 6.8</td></tr>
</table>

Table 6.2 Conflicting entities matrix

A ✓ in the table indicates that an association is allowed, whilst a × shows that an association is prohibited. For example, theorem 6.9 proves that non-conflicting users may only be assigned to conflicting roles while axiom 6.8 states that non-conflicting tasks may be assigned to either conflicting or non-conflicting roles.

6.4 Conclusion

This chapter explored the model required for the conflicting entities administration paradigm. This paradigm is able to ensure static separation of duty requirements and enable the specification of dynamic separation of duty requirements in workflow environments.

This paradigm was enabled through the extension of the RBAC components with workflow specific components. In particular, it demonstrated how static separation of duty requirements specified through the use of conflicting users, conflicting roles, conflicting permissions and conflicting tasks could be enforced. Entities that are conflicting would imply certain constraints and as

such would ensure the integrity of the associations. Enforcement is based on maintaining the integrity of the associations allowed between entities.

Formulated algorithms, which would maintain the integrity of the associations, were developed next. Thereafter, a prototype that demonstrates that the SoDA model could be effective for the enforcement of static SoD in administration tools was developed.

Chapter 7.

SoDA: The Enforcement Strategy

The previous chapter identified the entities involved in the conflicting entities administration paradigm. It also introduced mathematical definitions and proofs that describe how different associations between entities must be evaluated and enforced. In so doing, it described a model that can be used to ensure the integrity of the data and to prevent the possibility of fraud.

The integrity constraints imposed by the conflict paradigm were summarised in the form of a matrix in chapter 6. This matrix shows the integrity constraints as allowable/disallowable associations.

The SoDA model outlines algorithms for each of the possible actions that may be performed upon the entities. These algorithms are designed according to the specifications that have been defined in chapter 6 and are described in detail in the current chapter.

Chapter 6 described the SoDA model in mathematical terms while this chapter describes the algorithms using relational database system terminology. As such, it is important to define a conceptual entity-relationship diagram (ERD) for the SoDA model.

7.1 The conceptual Entity Relationship Diagram

It is possible to express the mathematical relations between the sets of entities defined in chapter 6 as database tables. This is shown using an ERD in figure 7.1.

Expressing the mathematical relations in this way simplifies the development of the algorithms. There are three different categories of database tables depicted in figure 7.1. These are associations, conflicts, and entities.

The associations are depicted by the *UA*, *PA*, *TA* and *RH* database tables. Each association allows for a many-to-many relationship between a role and another entity. The *RH* association is slightly different to the others. It is used to create a role network.

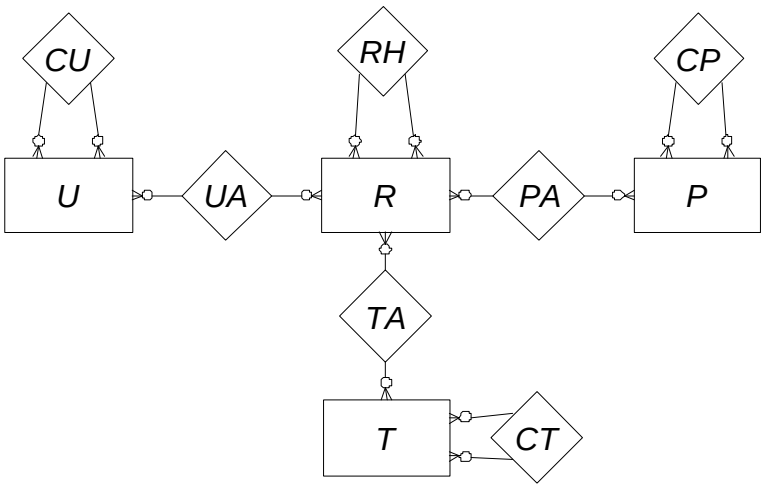

Figure 7.1 Conceptual ERD

The conflicts between entities are depicted by the *CU*, *CP*, *CR* and *CT* database tables. As can be seen, a conflict is a many-to-many relationship between the entities of a particular set.

Finally, there are the database tables that represent the sets of entities. These are denoted as *U*, *P*, *R* and *T*.

Within each of the categories, there exists the potential for a variety of conflicts. Each possibility needs an algorithm that will outline what an administration tool will do to test whether the action will violate an integrity requirement. Any action that will violate an integrity requirement will not be allowed to continue.

This includes actions such as adding or deleting entities from the system. With the integrity of the system being an important aim, it will not be possible to delete an entity without first checking the repercussions the deletion will cause.

The algorithms involved with the associations action will be evaluated first, followed by the algorithms for the conflicts action and finally, the algorithms for the entities action. The layout of this is shown in table 7.1.

This table has two columns that show the operations that can be performed upon the entities and their associations. These operations are *Add* and *Delete*. The SoDA model allows for updates or modifications through a twofold process of first deleting and then adding.

		Operations	
		Add	**Delete**
Associations (7.2)	UA	**7.2.1**	**7.2.2**
	PA		
	TA		
	RH	**7.2.3**	
Conflicts (7.3)	CU	**7.3.1**	**7.3.2**
	CP		
	CR		
	CT		
Entities (7.4)	Users	**7.4.1**	**7.4.2**
	Permissions		
	Tasks		
	Roles		

Table 7.1 The enforcement model chapter layout

7.2 Algorithms for Entity Associations

It is assumed that a set of entities and their conflicts already exist within the administration environment. The algorithms involved with the creation and deletion of entities and conflicts will be dealt with in the following sections.

7.2.1 Creating Associations

The first algorithm to be discussed will be the creation of a user to role association. As mentioned earlier, an example of this would be assigning user A to the manager role. This association is denoted as $(u_i,r) \in UA$. Before this association may be allowed to take place, the steps shown in figure 7.2 must be undertaken.

- Step 1: Does the user u_i have any conflicting users? This is done by iterating through all the conflicting user records where one of the users in the record is the user being added to the association. If the answer to this is yes, proceed to step 2, else create the association.
- Step 2: Find all the roles that have already been associated to the conflicting users found in step 1. This is done by iterating through the entire user to role association table where any of the associated users are in the group of users found in step 1. If no roles are found then the association can be safely made, else proceed to step three.

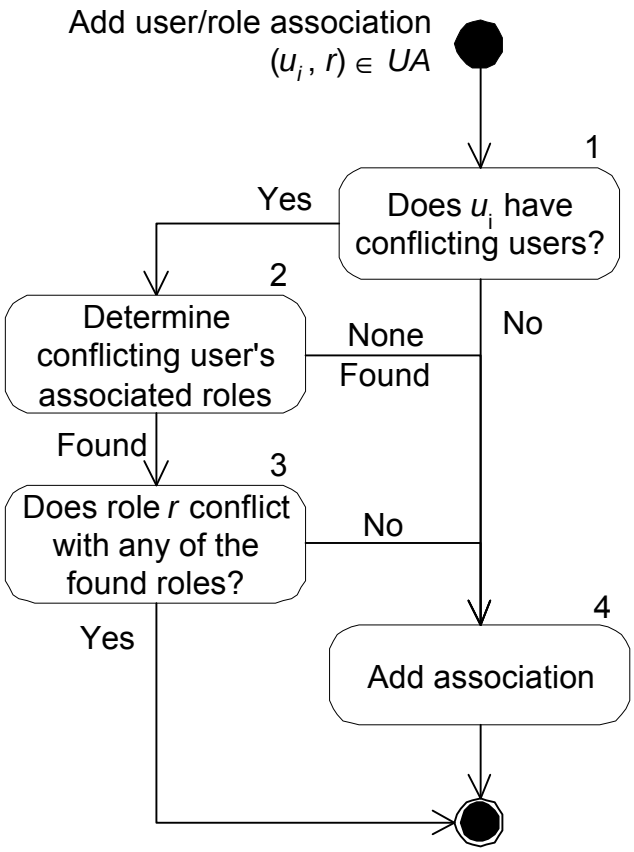

Figure 7.2 Adding user/role associations

- Step 3: Do any of the roles found in step 2 conflict with the added role *r*? This is accomplished by iterating through the conflicting roles records where the current role and any of the roles found in step 2 form a record. If none are found the association can be safely made, else the association may not proceed.
- Step 4: Add the association as a record to the appropriate database table.

The algorithm used when adding permission to role associations is discussed next. An example of this association would be assigning the permission that encompasses the access rights to approve an order to the manager role. This association is denoted as (p_i,r) ∈ PA. Before this association may be allowed to take place, the steps shown in figure 7.3 must be undertaken.

- Step 1: Does the permission p_i have any conflicting permissions? This is done by iterating through all the conflicting permission records where one of the permissions in the record is the permission being added to the association. If the answer to this is yes, proceed to step 2, else create the association.

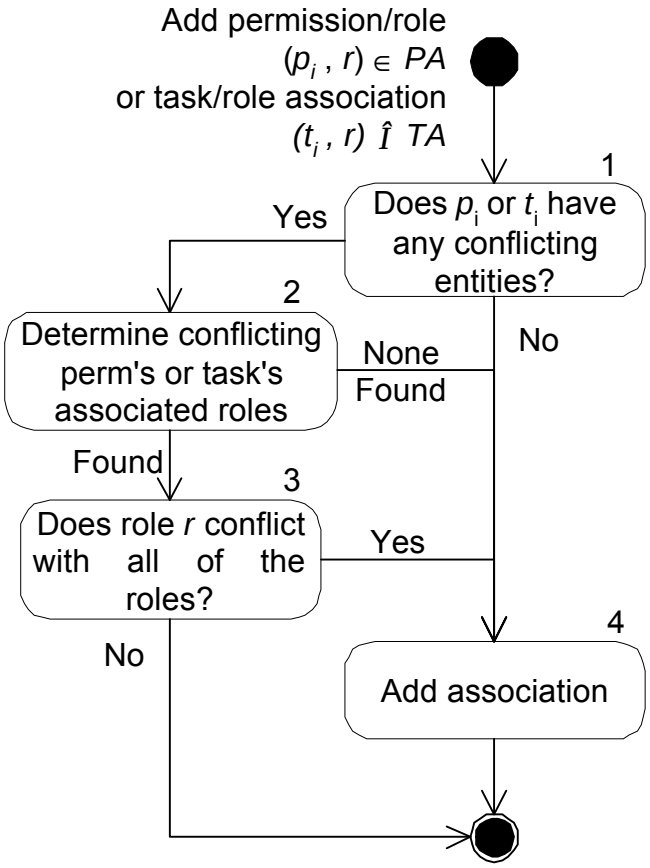

Figure 7.3 Adding permission/role or task/role associations

- Step 2: Find all the roles that have already been associated to the conflicting permissions found in step 1. This is done by iterating through the entire permission to role association table where any of the associated permissions is in the group of permissions found in step 1. If no roles are found then the association can be safely made else proceed to step three.
- Step 3: Do all of the roles found in step 2 conflict with the added role *r*? This is accomplished by iterating through the conflicting roles records where the current role and any of the roles found in step 2 form a record. If none is found the association can be safely made, else the association may not proceed. This step differs from the user to role association algorithm in that only if all the found roles are conflicting then may the association take place.
- Step 4: Add the association as a record to the appropriate database table.

Since the algorithm for adding a task to role association is so similar to the algorithm for adding a permission to role association, figure 7.3 will be used

as its reference. An example of this association would be assigning the task 'Approve Order' to the manager role. This association is denoted as $(t_i,r) \in TA$. Before this association may be allowed to take place, the steps shown in figure 7.3 must be undertaken.

- Step 1: Does the task t_i have any conflicting tasks? This is done by iterating through all the conflicting task records where one of the tasks in the record is the task being added to the association. If the answer to this is yes, proceed to step 2, else create the association.
- Step 2: Find all the roles that have already been associated to the conflicting tasks found in step 1. This is done by iterating through the entire task to role association table where any of the associated tasks are in the group of tasks found in step 1. If no roles are found then the association can be safely made, else proceed to step three.
- Step 3: Do all of the roles found in step 2 conflict with the added role *r*? This is accomplished by iterating through the conflicting roles records where the current role and any of the roles found in step 2 form a record. The association can be safely made if all of these roles conflict with the added role *r*, else the association may not proceed.
- Step 4: Add the association as a record to the appropriate database table.

7.2.2 Deleting Associations

There is no restriction upon the deletion of these associations, as it will not affect the secure nature of the environment. It may be necessary to ensure that certain associations remain intact for the work processes to execute correctly. The SoDA model does not cater for these situations.

7.2.3 Maintaining Role Networks

The last type of association that can be made is the role network, which is defined in chapter 3. Each role can have a parent role and a child role and it is important to not allow circular references to occur within a role network. Figure 7.4 displays the algorithm for adding roles to a role network, which is denoted as $(r_p,r_i) \in RH$. The role r_p is the parent role which is used to identify where the role r_i will be inserted.

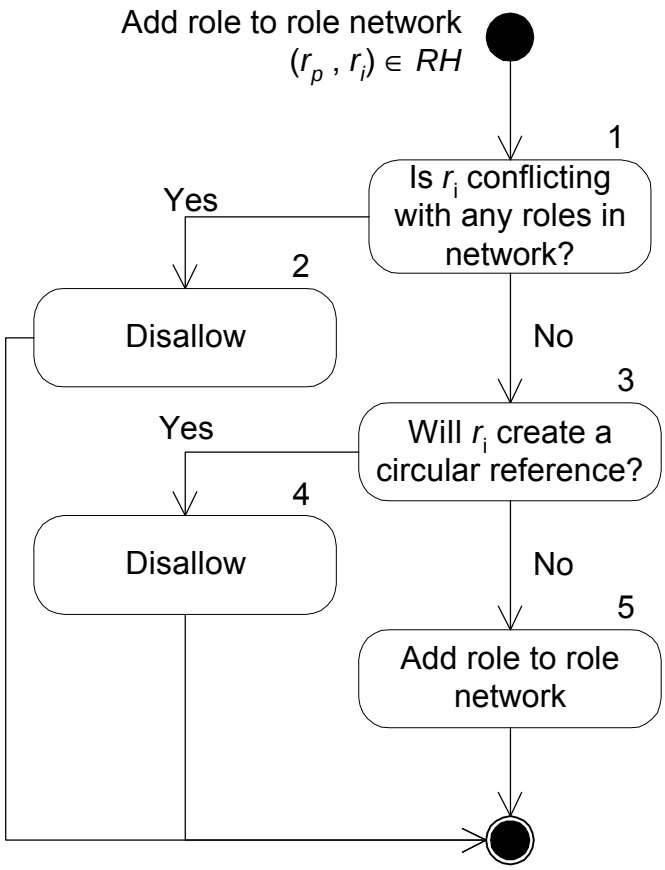

Figure 7.4 Adding roles to a role network

- Step 1: Does the role r_i conflict with any of the roles in the role network? This is done by iterating through the conflicting role database table where the role r_i and any of the roles in the role network form a record. The roles within the role network can be ascertained from the role network database table. If the answer to this is yes, proceed to step 2, else proceed to step 3.
- Step 2: Disallow the addition. This is accomplished in the form of an on screen dialog that warns the administrator.
- Step 3: Check whether the addition of the role r_i will create a circular reference. A circular reference is where the same role is referenced again along a branch of the role network. This check is accomplished by recursively checking the branch for any occurrences of the current role. If a circular reference is detected then the addition of this association will be disallowed through step 4, else it will be created in step 5.

- Step 4: Disallow the addition. This is accomplished by an on screen dialog that warns the administrator.
- Step 5: Create the role network association by writing the correct record into the role network database table.

When a role is deleted from the role network, it becomes important to make sure that a hole is not left behind. This is solved through making sure that the parent and child roles of the role that was removed get associated with each other. An algorithm that describes this logic is described with figure 7.5.

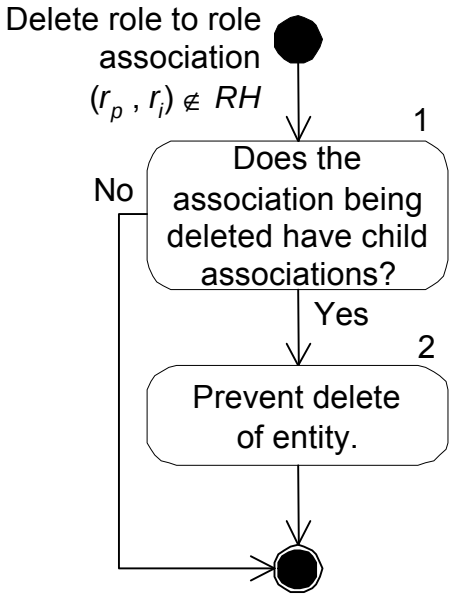

Figure 7.5 Deleting role associations from a role network

- Step 1: Does the association that is being deleted have any existing children associations? This is done by selecting all records where the parentid equals the deleted record's childid within the same role network. If child associations exist, proceed to step 2.
- Step 2: Raise an error and prevent the delete.

Other issues are evident and should be understood when working with roles in a role network. These issues include the inheritance of permissions and assigning ambiguous roles or roles with similar or identical permissions. These issues are not catered for in the SoDA model.

7.3 Algorithms for Entity Conflicts

It is important to be able to correctly maintain the conflicts between entities in order for an administration tool to control the integrity of the data effectively. We still assume that we have entities already in the administration environment.

7.3.1 Creating conflicting entities

Conflict assignments follow the same rules for all entities. The same general algorithm can be used to control all conflict assignments. These are depicted mathematically as $(t_i,t_j) \in CT$ for conflicting tasks, $(r_i,r_j) \in CR$ for conflicting roles, $(u_i,u_j) \in CU$ for conflicting users and $(p_i,p_j) \in CP$ for conflicting permissions. This algorithm is depicted in figure 7.6.

- Step 1: Check whether the entities are already conflicting with each other. This is done by checking the records in the relevant database table for one that matches the current entities. If a record is found, then this conflict assignment has already been done and should not be done again. If an existing conflict is not found, then it should proceed to step 2.
- Step 2: Check whether the entities form associations. This is done by iterating through all the applicable database tables, searching for any prior associations for either of the entities. If an association is found then step 3 must be performed, else the conflict may be created.
- Step 3: An association may become invalid if the conflict assignment goes ahead. It is therefore necessary to disallow any conflict assignment that will cause integrity problems. This step is expanded in figures 7.7, 7.8 and 7.9 to show exactly what processing needs to be done to accomplish this check for the different entities. If no problems will be caused by the conflict, then the conflict can be created, else the conflict must be disallowed.
- Step 4: Disallow the addition. This is accomplished in the form of an on screen dialog that warns the administrator.
- Step 5: Create the conflict assignment by writing a record into the appropriate database table.

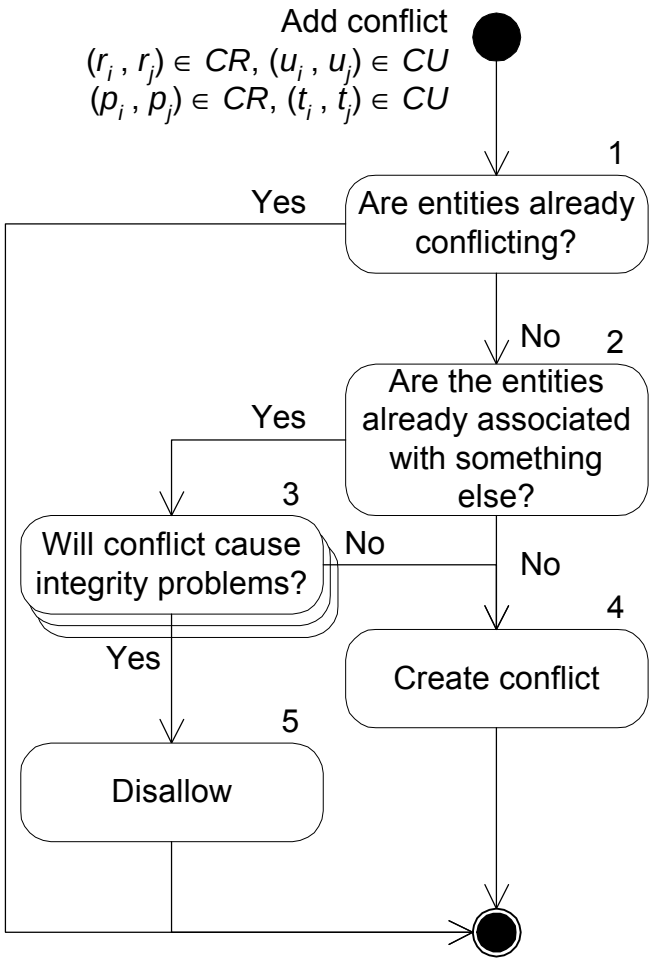

Figure 7.6 Adding conflicting entities

If the conflict assignment is created between two users ($(u_i,u_j) \in CU$) then the algorithm in figure 7.7 is performed.

- Step 3.1: We assume that the users u_i and u_j already conflict and we find all of the associated roles for both users. If associated roles are found for both users then go to step 3.2, else allow the conflict to be created.
- Step 3.2: Check whether any of the roles that are associated with the user u_i are conflicting with any of the roles that are associated with the user u_j. This is done by checking for the existence of the role pairing in the conflicting roles database table. If none of the roles are conflicting, then allow the conflict to be created.
- Step 3.3: Do not create the conflict, as it will cause an existing association to become illegal.

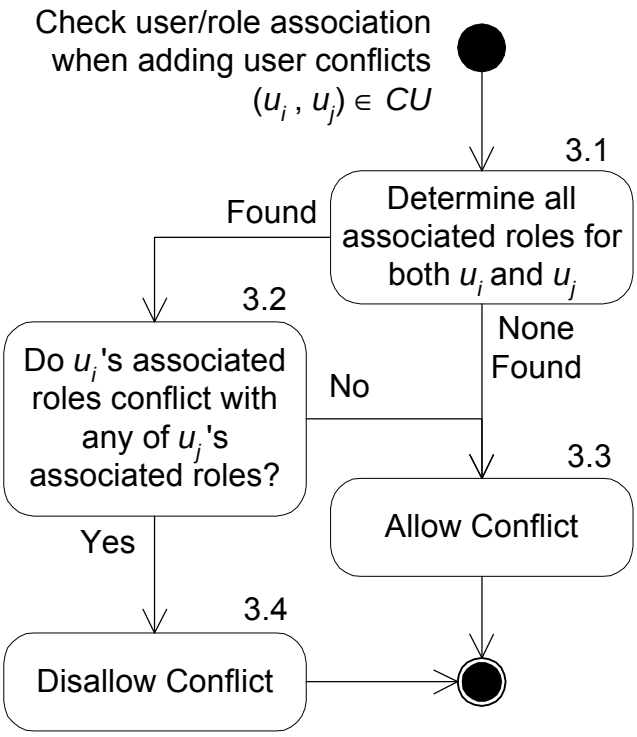

Figure 7.7 Checking user/role association

- Step 3.4: Create the conflict by writing a record into the appropriate database table.

If the conflict assignment is created between two permissions ((p_i,p_j) ∈ *CP*) or two tasks ((t_i,t_j) ∈ *CT*) then the algorithm in figure 7.8 is performed.

- Step 3.1: We assume that the entities p_i and p_j or t_i and t_j already conflict and we find all of the associated roles for both entities. If associated roles are found for both entities, then go to step 3.2, else allow the conflict to be created.
- Step 3.2: Check whether all of the roles that are associated with the first entity (p_i or t_i) are conflicting with all of the roles that are associated with the second entity (p_j or t_j). This is done by checking for the existence of the role pairing in the conflicting roles database table. If all of the roles are conflicting, then allow the conflict to be created.
- Step 3.3: Do not create the conflict, as it will cause an existing association to become illegal.
- Step 3.4: Create the conflict by writing a record into the appropriate database table.

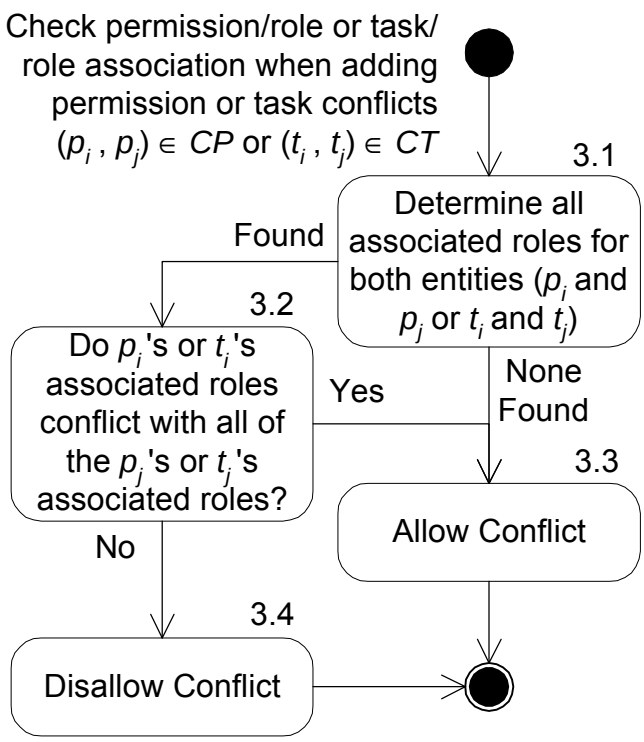

Figure 7.8 Checking permission/role and task/role associations

If the conflict is created between two roles, then the algorithm shown in figure 7.9 is used. It is only necessary to check whether conflicting users are assigned to the two roles, as this is the only association that will be affected by the roles becoming conflicting.

- Step 3.1: Check whether both roles (r_i and r_j) are already part of a role network. If they are, then it must go to step 3.5 to disallow the conflict, else it must go to step 3.2.
- Step 3.2: We assume that the roles r_i and r_j are already conflicting and we find all of the associated users for both roles. If associated users are found for both roles, then go to step 3.3, else allow the conflict to be created.
- Step 3.3: Check whether any of the users that are associated with the role r_i are conflicting with any of the users that are associated with the role r_j. This is done by checking for the existence of the user pairing in the conflicting users database table. If any of the users are conflicting, then do not allow the conflict to be created.

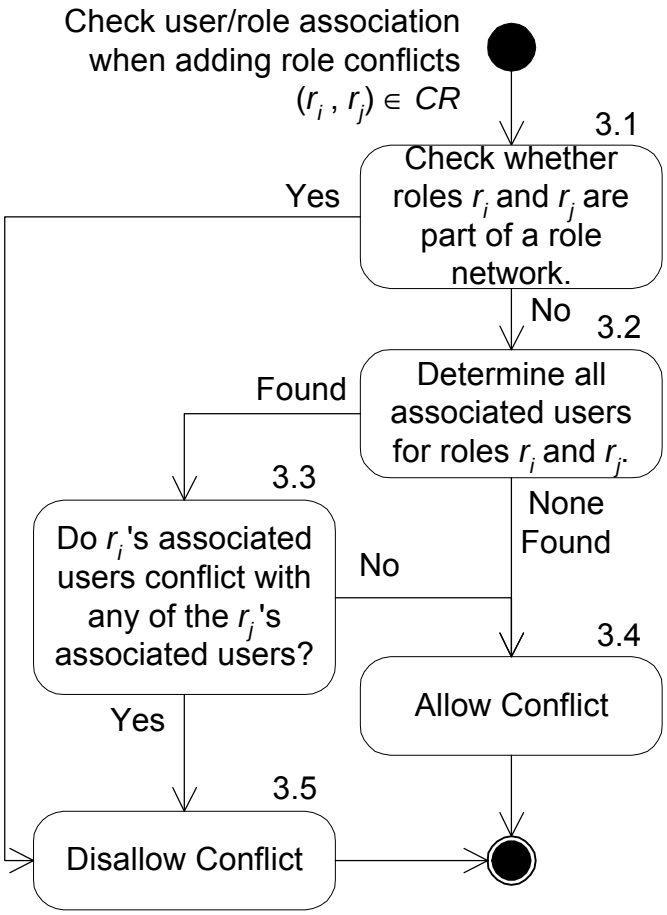

Figure 7.9 Checking user/role associations for role conflict assignments

- Step 3.4: Do not create the conflict, as it will cause an existing association to become illegal.
- Step 3.5: Create the conflict by writing a record into the appropriate database table.

Following these algorithms will ensure that the data's integrity remains intact.

7.3.2 Deleting conflicting entities

It is important to know what integrity requirements must be checked and maintained for each conflict type that is to be deleted.

User, permission and task conflicts may be safely removed without affecting the integrity of the system. This is because only the conflicting entities restrict how they are associated with the roles.

Deleting role conflicts require proper integrity checks to ensure that the existing associations remain valid. This algorithm is shown in figure 7.10.

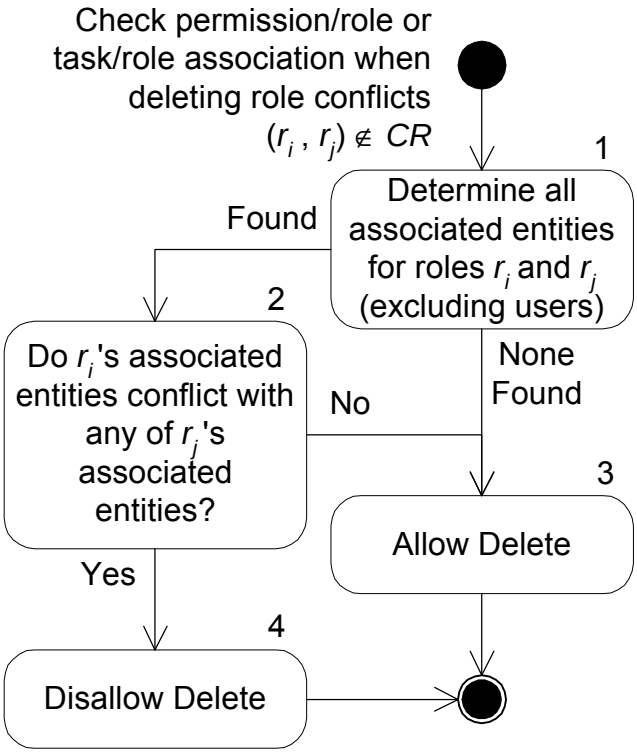

Figure 7.10 Checking entity associations for role conflict deletion

- Step 1: We find all of the associated permission and tasks for both roles r_i and r_j. If associated entities are found for both roles, then go to step 2, else allow the conflict to be deleted.
- Step 2: Check whether any of the permissions that are associated with the role r_i are conflicting with any of the permissions that are associated with the role r_j. Also, check whether any of the tasks that are associated with the role r_i are conflicting with any of the tasks that are associated with the role r_j. This is done by checking for the existence of the entity pairing in the correct conflicting entity database table. If any of the permissions or tasks are conflicting, then do not allow the conflict to be deleted.
- Step 3: Do not delete the conflict, as it will cause an existing association to become illegal.
- Step 4: Delete the conflict by removing the record from the appropriate database table.

The creation and deletion of entity associations as well as the creation and deletion of conflicting entities have now been covered. The final *set*

algorithms which needs to be discussed are those to do with the creation and deletion of entities.

7.4 Algorithms for Entity Maintenance

The algorithms that have been discussed so far have all assumed that there were entities already in an administration environment. This section discusses the issues surrounding the creation and deletion of entities within an administration environment.

7.4.1 Creating entities

Entities may be added an administration environment at any time. New entities do not affect the integrity of the data in the system. Care must be taken when deleting entities to ensure the integrity of the system remains intact.

7.4.2 Deleting entities

Any entity may be safely deleted if that entity does not form part of any associations. The SoDA model will not allow any type of entity from being removed if that entity has an association of any kind.

This logic is displayed in figure 7.11, which shows the algorithm used for the user ($u_i \notin U$), permission ($p_i \notin P$) and task ($t_i \notin T$) entity deletion. This algorithm also caters for deleting roles ($r_i \notin R$) as step 1 also handles role to role associations or role networks.

- Step 1: Check whether the entity has an association. This is done by checking the records in the relevant entity association database table for one where the entity matches the current entity. If the entity is a role, then it will be necessary to check whether the role forms part of a role network as well by checking the relevant database table. If a record is found, then this deletion may not continue, else if none are found then it can proceed to step 3.
- Step 2: Disallow the delete from continuing. The administration tool will display the reason for the error by prompting the administrator.

- Step 3: Check for the existence of any conflict assignments between the current entity and any others. This is done by iterating through the respective entity's conflict database table where any of the entities in the records matches the current entity. If an entry is found, then the record must be deleted. If none is found, then the entity can be safely deleted.

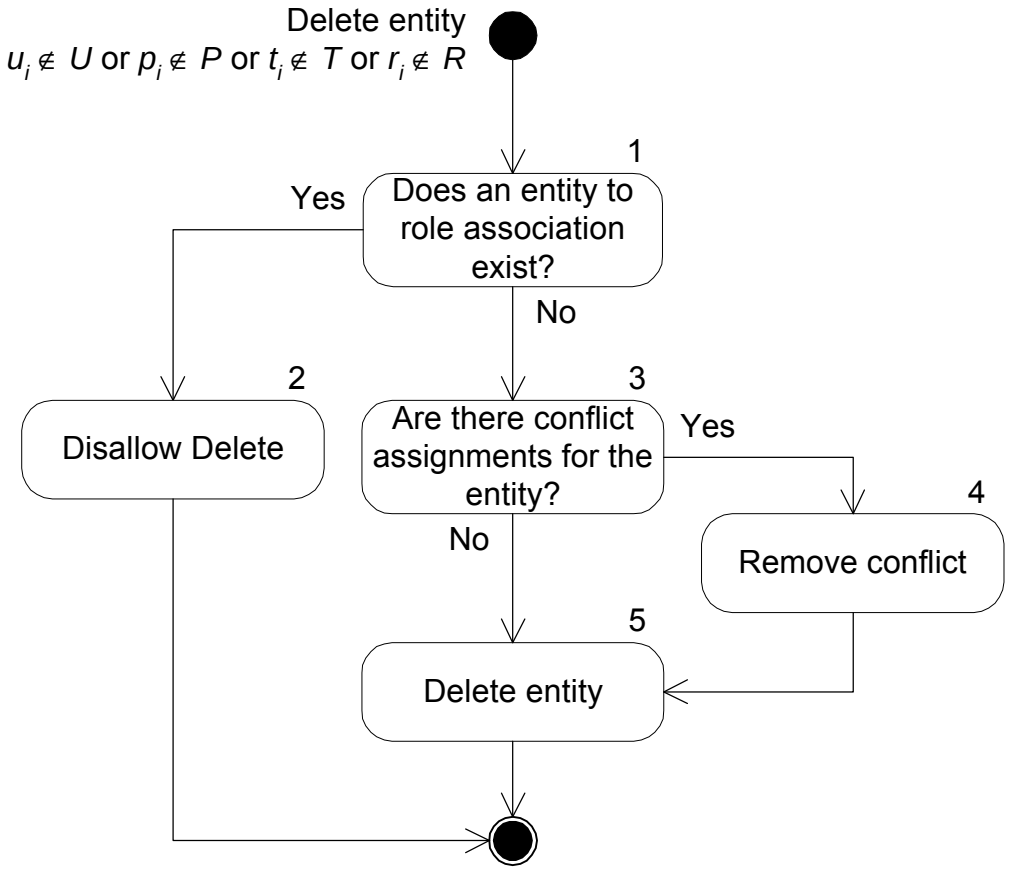

Figure 7.11 Deleting a user, permission or task entity

- Step 4: Remove any conflicts that are between the current entity and any other entities.
- Step 5: Delete the current entity. This is done by simply removing the entity's record from the respective entity's database table.

7.5 Conclusion

This chapter built algorithms according to the definitions, theorems and axioms in chapter 6 for use within the SoDA model. These algorithms describe exactly how the SoDA model maintains the integrity of the data.

Chapter 8 will describe the prototype that was used to demonstrate the SoDA model. This prototype was assembled following the concepts defined in the previous chapters. It demonstrates how the SoDA model can be implemented in an administration tool.

Chapter 8.

SoDA: The Prototype

Chapter 6 developed the concept of the integrity constraints while chapter 7 discussed various algorithms necessary to enforce these SoD requirements. This chapter introduces a prototype used to demonstrate the concepts of the SoDA model.

Two prototypes were developed for this purpose. The first prototype's design approach for enforcing the SoD requirements was markedly different to the approach taken by the second prototype.

It is necessary to discuss how the first prototype functioned and to show the lessons learnt through its design before discussing how the second prototype operates.

8.1 SoDA: The First Prototype

When development of the SoDA prototype first began it was envisaged that the prototype would be a workflow administration tool. It was decided that the algorithms, defined in chapter 7, would be built into a client application. This demanded extra attention to the design of the user interface for the client application. This approach affected the design of the first prototype.

8.1.1 Design approach

The first prototype was a Windows based client application with a database back-end. The client application was written using Visual Basic and the database management system used was Oracle. The design approach is depicted in figure 8.1.

The design approach for this prototype was an all-inclusive one. All of the data constraint and integrity logic was built into the prototype. The prototype itself enforced the various associations and would constrain the association from occurring if the particular association or assignment would cause integrity errors.

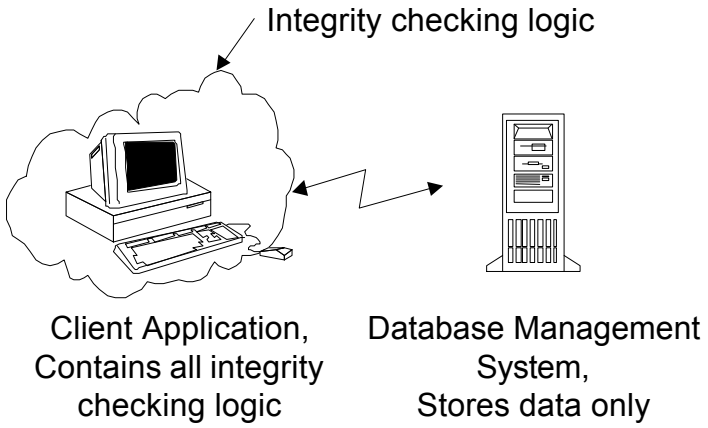

Figure 8.1 Design approach of the first SoDA prototype.

The client application would accomplish this by dynamically changing options depending upon what the user had selected. In so doing, the prototype gave the user only the options that were available for the actions the user was performing. With its design as a workflow administration tool, certain functions were included to support the environment it was administrating.

8.1.2 Functionality

One of the included functions was form and field creation. This function supported a certain type of workflow system that the prototype was created for.

Each field formed part of a hierarchy, which enabled permission groupings. This meant that a logical group of fields would all be accessed through the same permission. For example the “Edit Order Fields” permission would only have to be created for the field under which all the order fields fall. These fields would be automatically accessible by a user with rights to this particular permission.

This in turn would generally ease the administration of permissions. The form building front-end is shown in figure 8.2.

Certain additional workflow functions were taken into account due to the fact that the prototype was designed as a fully functional workflow administration tool. An example of these functions is the concept of a workflow process. Tasks had to belong to a process and only this process’ tasks could conflict with one another. The second prototype does not cater for this level of detail.

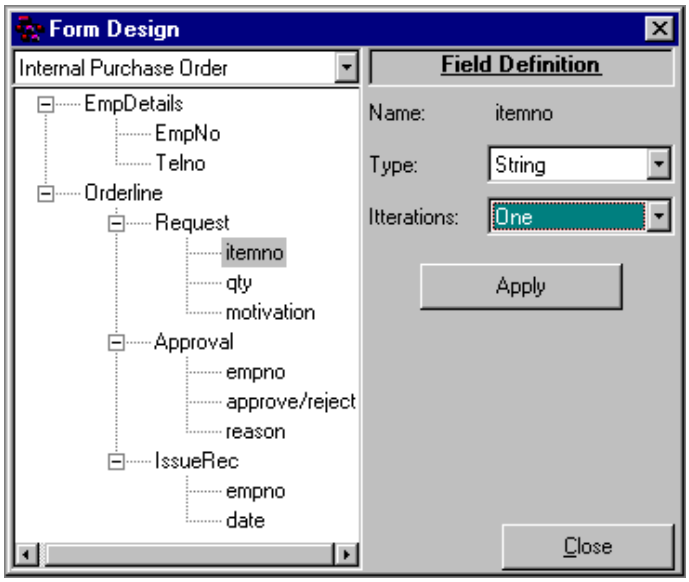

Figure 8.2 Form design environment used to create a "Purchase Order".

The first SoDA prototype contained all the integrity checking logic as depicted in figure 8.1. This resulted in complex user interface scenarios such as the task to role associations shown in figure 8.3.

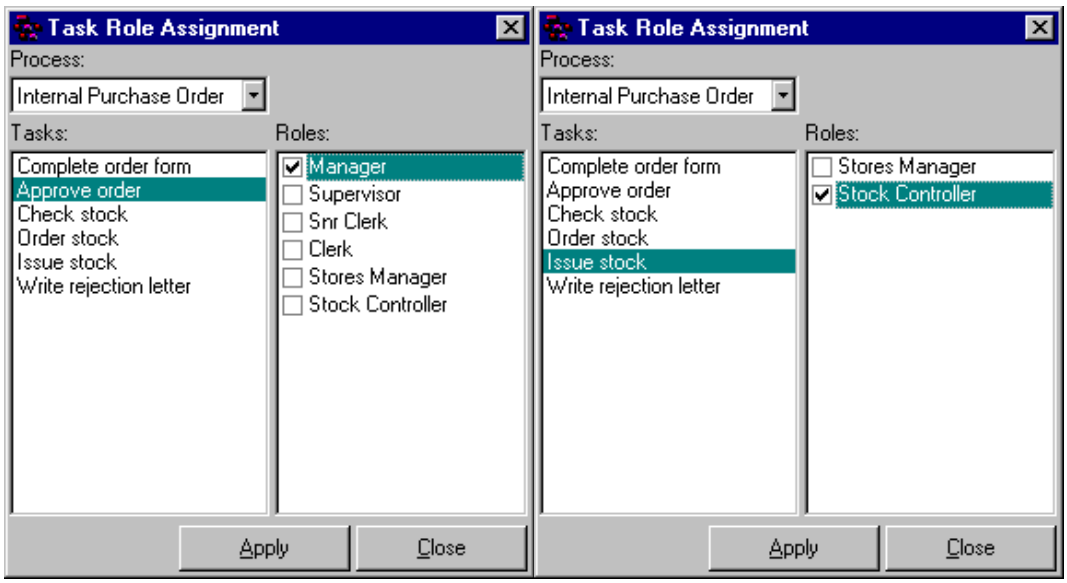

Figure 8.3 Ensuring conflicting task to conflicting role assignments.

As can be seen with the screenshot on the left of figure 8.3, the Manager role is being associated to the "Approve Order" task. Once this association has been made the user wants to then associate a role to the "Issue Stock" task. The screenshot on the right displays only those roles that are allowed to be associated with this task.

The application will only give the user the allowed choice of roles to choose from for a particular task. It does this by applying the necessary logic to select the roles that can be associated with the currently selected task.

This type of interactivity was considered a feature of the first prototype. It made the conflicting entity paradigm clear through how the user interface responded to the user's actions.

There were difficulties with the design of the first SoDA prototype due to the fact that all of the integrity checks were incorporated into the client application. This is most prevalent when considering the multitude of possibilities available due to the flexible functionality of the client.

8.1.3 Difficulties with this approach

The difficulties experienced with the first prototype included:

- The first prototype maintained the integrity of the data from the client side and not from the database side. This gave the whole system too much flexibility in what could be done and when it could be done. Every possible action needed to be catered for, thus causing the prototype to become unnecessarily complicated and cumbersome.
- Whenever an association was made it was necessary for the prototype to check every possible condition that could cause it to fail. This meant that a large section of program code was devoted to querying and checking the database tables. These database queries resulted in increased network usage and generally slowed the process of checking for integrity conflicts.
- There was also nothing preventing a user from accessing the database directly and making modifications to the data without the protective shell of the prototype to prevent them.

These issues forced the prototype to be redesigned resulting in version 2.

8.2 SoDA: The Second Prototype

The first SoDA prototype did all of the integrity checks from within the client application. Due to the difficulties experienced with this approach the second

SoDA prototype moved the integrity checks from the client side and into the database.

This was possible due to the fact that Oracle is able to extend its own functionality through the use of procedures and triggers. The difficulties experienced with the first prototype were thus solved through the use of a different design approach.

8.2.1 Design approach

One of the design decisions for the second prototype was to incorporate active database techniques. Active databases react to events in the database (Paton & Díaz, 1999). An event could be caused by the insertion of a record into a database table or by the timing out of a logged in user's session.

Another aspect of active database management systems is that instead of spreading organisational policies across various applications, these policies can be placed within the database (Kappel, Rausch-Schott, Retschitzegger & Sakkinen, 2001). Policies will be enforced for every business application without encoding the requirements that they impose into those business applications. The SoD requirements, as discussed in this dissertation, would thus be enforced by the active database management system.

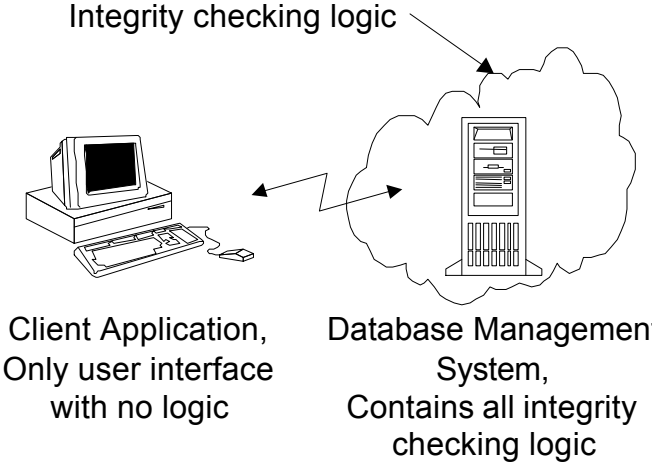

Figure 8.4 Design approach of the second SoDA prototype.

The second SoDA prototype uses the concept of an active database by implementing the integrity checks through database triggers. This approach is depicted in figure 8.4.

The algorithms in chapter 7, while capable of being implemented on the client side, were, in this second version of the SoDA prototype, implemented as triggers in an Oracle database. The redesigning of the original prototype to use triggers helped solve the problems of the first prototype in the following ways:

- Network traffic is reduced, as the client no longer has to perform extraneous data queries to perform SoD requirements checks.
- The client application is not required to ensure the validity of an assignment or association, as the database will control the inserts and deletes.
- Changes can be made to the triggers to extend the integrity constraint logic without necessitating a change in the client application.
- Users cannot bypass the client application to make changes to the data through the database as the database itself prevents it.

In order to illuminate the ensuing discussion, consider how database triggers can be used to enforce integrity constraints such as those imposed by static SoD requirements.

8.2.1.1 Using triggers to achieve integrity

Using database triggers to enforce the conflict paradigm constraints was the primary focus of the second prototype. The client application was a secondary goal and its only purpose was to test that the SoD requirements, which were implemented within the database, functioned correctly.

SoD requirements are, where possible, enforced through foreign key constraints and through primary or unique key constraints. This type of integrity constraint is known as declarative integrity constraints. This integrity constraint prevents entities from being deleted if they are already used in associations. If an entity is deleted but only has conflicts with another entity and no other associations, then the conflicts are removed automatically through the use of cascading deletes.

An example of this would be a database table that relies on values in another database table. The database can be instructed to ensure that the relied

upon records may not be deleted or to delete all records in the first table that make use of the relied upon record's values. But there are integrity constraint situations that the database cannot manage. In these situations a database trigger has to be used to enforce the integrity constraint.

This type of integrity constraint is known as a procedural integrity constraint. An example of this type of integrity constraint would be checking to see if an action is allowed to take place at a certain time of day. In this case a business process, such as a sale, could be restricted to business hours.

Triggers have restrictions on what they can do and how they can be used. Therefore, when designing database triggers, special considerations must be taken into account.

8.2.1.2 Special considerations when using triggers

There are two types of triggers: row-based triggers and statement-based triggers (ORACLE, 1999). Row-based triggers are executed for every row or record in a table that a SQL statement affects. While statement-based triggers are only executed once for a SQL statement, no matter how many records are affected, row-based triggers are more difficult to work with, as the possibility for errors is more prevalent.

One such error is the mutating table error. A mutating table error is caused when a row-based trigger attempts to read data from a database table that is in the process of undergoing change (Oracle, 1999). The database management system raises an error and prevents the trigger from continuing since it might read invalid data.

For example, the following SQL statement, which sets the hourly rate for wage earning employees, could have repercussions:

```
UPDATE staff SET rate = rate + 10;
```

There may be procedural integrity constraints that ensure that the rate increase is allowed for under organisational policies. This could be a statement-based trigger, as it would most likely only need checking once for the entire table. Another procedural integrity constraint could check the increase for every employee that is affected. This would be a row-based

trigger, as it will need to execute once for every record that is affected. The creation of this trigger would be hampered if it needed to examine values within the staff database table due to the mutating table error.

In designing the triggers for the prototype it was realised that they would have to examine the contents of their own changing database tables. This forced them to be designed in such a way to prevent mutating table errors. This problem can be demonstrated practically with the following example.

A trigger will fire when two conflicting roles are inserted into the conflicting role (CR) database table. This trigger will have to execute every time a conflicting pair of roles gets inserted into this table. This means that this trigger is a row-based trigger.

This trigger has to find all of the roles that are already conflicting with the conflicting roles that are being added to the CR database table in order for it to check the SoD requirements. This requires it to query the CR table that is presently in the process of changing. Oracle will not allow this query to proceed and will generate a mutating table error.

In order to work around this problem it was necessary to create an Oracle package to hold PL/SQL table variables for each trigger. A package is a globally accessible collection of variables and routines (ORACLE, 1999). A PL/SQL table is an array-like structure that can be used to hold the values of database records.

In order to make use of the PL/SQL tables in the package it was necessary to design the triggers correctly. This design involves the creation of three triggers, one for each type of action for each database table. This solution is represented graphically in figure 8.5.

The package defines two PL/SQL tables for each database table and for each action that will be handled. So there are two PL/SQL tables for inserting a record into the role to role database table when creating role networks. There are also two PL/SQL tables for the triggers that handle deleting records from the role to role database table.

Of the two PL/SQL tables for each trigger, the first table holds the actual data while the second is simply an empty structure used to reinitialise the first.

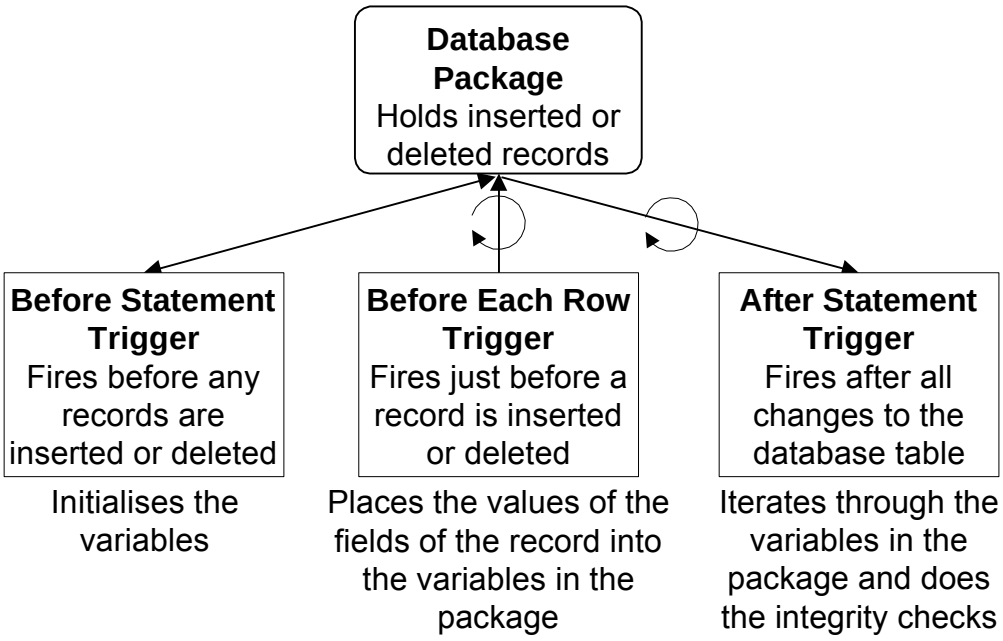

Figure 8.5 A solution to the problem of mutating tables.

The first of the three triggers does the reinitialising of the first PL/SQL table. This trigger is a statement trigger that will run at the beginning of the insert or delete action on a database table.

The second trigger is a row-based trigger that gets activated for every row that is affected. This trigger adds the values of the fields of the affected records to the first PL/SQL table's fields and increments the first PL/SQL table's counter by one each time.

The third trigger is a statement-based trigger that fires after the changes on the database table. This trigger loops through all the records in the first PL/SQL table and checks the SoD requirements for the record that was inserted or deleted. It must be remembered that at this point the record is either already in the database table if it was inserted or it is already deleted from the database table if deleted. If the SoD requirement check fails, then it is necessary for the third trigger to raise an exception to cause the database to cause a rollback to reverse the changes.

The client application for this approach has no SoD requirement checking logic. It relies upon the database in order to operate effectively and securely. This does have the added disadvantage that the client tends to show every possibility as being available. Only after the user attempts an assignment or association does the client show the error raised by the database.

It is possible to address this issue by using stored procedures for the SQL queries that will return a sensitised record set. However, this is a user interface issue and not an integrity issue, and as such falls outside the scope of this dissertation.

It is also possible to combine the approach taken with the first prototype and the approach taken with the second prototype. Certain SoD requirements, such as not allowing the same role to be conflicting with itself, could very easily be guarded against from the client side as well as from the database.

To demonstrate the effectiveness of the second prototype's approach, it was necessary to make the client application as simple as possible. The client application for the second SoDA prototype is just a user interface that facilitates the creation of entities, conflict assignments and associations. The client application contains no SoD requirements checking logic. As such, only the triggers, which were coded in Oracle's PL/SQL programming language, are listed in appendix C along with the database tables and the declarative constraints.

An in-depth discussion of the functionality of the triggers is required in order to understand how this solution creates an environment that handles separation of duty constraints.

8.2.2 Functionality

The second SoDA prototype implements the algorithms in chapter 7 as a set of triggers. A scenario based on the workflow example in chapter 2 is introduced in order to describe the functioning of these triggers.

The security administrator will create all of the conflict assignments. Once the conflicts between entities have been identified, it will be up to the workflow administrator, whose job it is to manage the workflow process definitions, to create the associations between the entities. These employee assignments are depicted in figure 8.6.

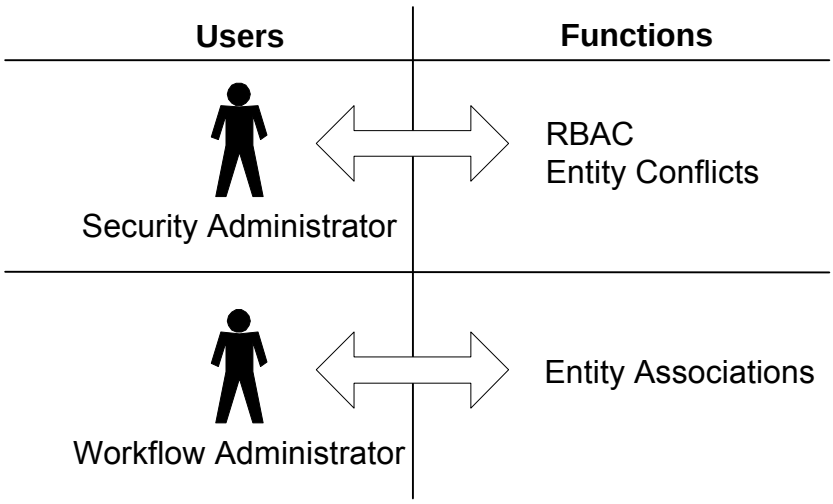

Figure 8.6 Anticipated employee assignments for the SoDA environment.

To facilitate the scenario, the following pre-existing entities shall be used:

Users	UserID
Thomas	1
Peter	2
Frank	3

Tasks	TaskID
Complete Order Form	1
Approve Order	2
Check Stock	3
Issue Stock	4
Order Stock	5
Write Rejection Memo	6

Roles	RoleID
Employee	1
Stock Controller	2
Manager	3

Permissions	PermID
Edit Order Fields	1
Edit Approve Order Fields	2
Edit Rejection Fields	3
Read Order Form	4
Edit Order Completed Fields	5

The permissions declared here are similar to the permissions defined for the first prototype. That is, they are named groupings of access rights to objects. In this example, the permissions are derived from the concept of form-based workflow. The permission “Edit Approve Order Fields” is thus a group of access rights for a user to achieve a specific activity.

Within the first prototype, the security administrator would assign access rights to the fields within a form to a named permission group. The second prototype does not cater for this step, but rather treats permissions at a higher level of abstraction.

The outline of this scenario is shown in table 8.1.

	Operation	Section
Conflicts	Creating role conflicts	8.2.2.1
	Creating user conflicts	8.2.2.2
	Creating permission conflicts	8.2.2.3
	Creating task conflicts	8.2.2.4
	Deleting conflicting entities	8.2.2.5
Role Networks	Creating the role network	8.2.2.6
	Deleting a role from a role network	8.2.2.7
Associations	Creating user to role associations	8.2.2.8
	Creating permission to role associations	8.2.2.9
	Creating task to role associations	8.2.2.10
	Deleting associations	8.2.2.11

Table 8.1 The SoDA model prototype scenario layout

It is possible to begin with any of the associations or conflict assignments, but in order to keep this explanation simple, the scenario will begin with the creation of conflicts.

8.2.2.1 Creating role conflicts

In the example, only two roles are conflicting: the manager role and the employee role. If these roles were not conflicting then they would be able to exist in the same role network. Since permissions are inherited from the role network, it would mean that one of the roles would have the permissions that would permit fraud.

In order to set the conflict, which is denoted as $(r_1,r_3) \in CR$, a SQL statement will insert a record into the CR database table. For these roles this can be coded as:

```
INSERT INTO cr VALUES(1,3);
```

The “crstatafter_trig” trigger will check that this is possible by first checking that they are not already conflicting. The algorithm for this trigger can be found in chapter 7, section 7.3.1, figures 7.6 and 7.9. The primary key constraint already present for the database table will capture this error sometimes, but the roles may be reversed for the current record and so this check is necessary.

It will then check whether the two roles form part of the same role network. If they are, then the following error is raised:

```
ORA-20223: Cannot insert role conflict - both roles are
already part of a role network.
ORA-06512: at "STEPHEN.CRSTATAFTER_TRIG", line 25
ORA-04088: error during execution of trigger
'STEPHEN.CRSTATAFTER_TRIG'
```

The trigger will proceed to check whether any of the users that are associated with the first role are conflicting with any of the users that are associated with the second role. If there are any then it will raise the following error:

```
ORA-20205: Cannot insert role conflict - both roles are
already associated to conflicting users.
ORA-06512: at "STEPHEN.CRSTATAFTER_TRIG", line 43
ORA-04088: error during execution of trigger
'STEPHEN.CRSTATAFTER_TRIG'
```

Only once these integrity constraint checks have passed will the trigger allow the conflict to be inserted.

Conflicting roles restrict which associations may be made with other entities and the roles that may form part of a role network.

8.2.2.2 Creating user conflicts

For this example it is not necessary to create conflicts between any of the users. For thoroughness, a description of how the logic operates for an example is described.

If the users Thomas and Frank were brothers it may be necessary to create a conflict between them. In order to set the conflict, which is denoted as $(u_1,u_3) \in CU$, a SQL statement will insert a record into the CU database table. For these users this can be coded as:

```
INSERT INTO cu VALUES(1,3);
```

The “custatafter_trig” trigger will check that this is possible by first checking that they are not already conflicting. The algorithm for this trigger can be found in chapter 7, section 7.3.1, figures 7.6 and 7.7. The primary key constraint already present for the database table will capture this error sometimes, but the users may be reversed for the current record and so this check is necessary.

The trigger will then check if the first user has any associated roles that are conflicting with any of the second user’s associated roles. If so, it generates the following error:

```
ORA-20207: Cannot insert user conflict - both users are
already associated to conflicting roles.
ORA-06512: at "STEPHEN.CUSTATAFTER_TRIG", line 43
ORA-04088:    error    during    execution    of    trigger
'STEPHEN.CUSTATAFTER_TRIG'
```

If conflicting users have been defined, these conflicts will restrict which roles may be associated with these users as well as the roles that may be part of a role network.

8.2.2.3 Creating permission conflicts

The primary means for preventing fraud is by the manipulation of access rights to the objects within a system. These access rights are known as permissions and creating conflicts between certain permissions is the means of manipulation necessary to enforce static separation of duty.

The security administrator needs to decide which permissions are conflicting. Once the conflicting permissions have been identified it is necessary to set them as conflicting.

In this example the permissions that are conflicting are:

Permission	**Conflicts with**
Edit Order Fields	Edit Approve Order Fields
	Edit Rejection Fields

In order to set these conflicts, which are denoted as $(p_1,p_2) \in CP$ and $(p_1,p_3) \in CP$, a SQL statement will insert two records into the CP database table. For these permissions this can be coded as:

```
INSERT INTO cp VALUES(1,2);

INSERT INTO cp VALUES(1,3);
```

The “cpstatafter_trig” trigger will check that this is possible by first checking that they are not already conflicting. The algorithm for this trigger can be found in chapter 7, section 7.3.1, figures 7.6 and 7.8. The primary key constraint already present for the database table will capture this error sometimes, but the permissions may be reversed for the current record and so this check is necessary.

The trigger will then check that all of the first permission's associated roles are conflicting with all of the second permission's associated roles. If this is not so, it will generate the following error:

```
ORA-20209: Cannot insert permission conflict - one or more
of the associated roles of the two permissions are not
conflicting.
ORA-06512: at "STEPHEN.CPSTATAFTER_TRIG", line 47
ORA-04088: error during execution of trigger
'STEPHEN.CPSTATAFTER_TRIG'
```

It is important to note that if one of the permissions does not have any associated roles, then the conflict may be created.

Conflicting permissions will restrict which roles may be associated with them, which causes a cascading effect that restricts the permissions assigned to users.

Up to now this scenario has only dealt with the entities that are normally a part of role based access control. Now it is necessary to include the concept of a task.

8.2.2.4 Creating task conflicts

The workflow task helps to incorporate the separation of duty constraints into the workflow environment. The enforcement of separation of duty requirements becomes easier for workflow processes once the tasks have been incorporated into conflict assignments and into the associations.

The security administrator will need to create conflicts between the following tasks:

Task	**Conflicts with**
Complete Order Form	Approve Order
	Write Rejection Memo

It could be argued that even more conflicts could be created between various other tasks. These conflicts could be made but they are dependent upon the level of security needed.

In order to set these conflicts, which are denoted as $(t_1,t_2) \in CT$ and $(t_1,t_6) \in CT$, a SQL statement will insert two records into the CT database table. For these tasks this can be coded as:

```
INSERT INTO ct VALUES(1,2);

INSERT INTO ct VALUES(1,6);
```

The “ctstatafter_trig” trigger will check that this is possible by first checking that they are not already conflicting. The algorithm for this trigger can be found in chapter 7, section 7.3.1, figures 7.6 and 7.8. This logic is identical as that for the permission conflicts.

The trigger will then check that all of the first task’s associated roles are conflicting with all of the second task’s associated roles. If this is not so, it will generate the following error:

```
ORA-20211: Cannot insert task conflict - one or more of the
associated roles of the two tasks are not conflicting.
ORA-06512: at "STEPHEN.CTSTATAFTER_TRIG", line 48
ORA-04088:    error    during    execution    of    trigger
'STEPHEN.CTSTATAFTER_TRIG'
```

As with the permission conflict assignment, it is important to note that if one of the tasks does not have any associated roles then the conflict may be created.

Conflicting tasks will prevent the possibility of conflicting permissions from being assigned to conflicting users within the workflow environment.

At this point it is necessary to explain what will happen if it is necessary to remove conflicts.

8.2.2.5 Deleting conflicting entities

Most conflict assignments can be removed without any repercussions. This is due to the fact that the removal of most conflicts will not impact the data integrity of the SoDA data.

The exception to this rule is the deletion of a role conflict assignment. A role conflict cannot be removed if the first role’s associated tasks or permissions conflict with the second role’s associated tasks or permissions. This means that if two conflicting tasks or permissions are associated with two conflicting roles, then the role conflict may not be deleted.

If the security administrator attempts to delete this role conflict then the “dcrstatafter_trig” trigger generates the following error:

```
ORA-20212: Cannot delete role conflict - the roles form part
of associations to tasks or permissions that will become
invalid if the role conflict is deleted.
ORA-06512: at "STEPHEN.DCRSTATAFTER_TRIG", line 66
ORA-04088:    error    during    execution    of    trigger
'STEPHEN.DCRSTATAFTER_TRIG'
```

The algorithm for this trigger can be found in chapter 7, section 7.3.2, figure 7.10. This ensures that the data integrity of the SoDA associations remains intact.

The security administrator has created all of the assignments up to this point in the scenario. Once the conflicts have been created it is no longer the security administrator's job to continue with the associations. The creation of the association is the job of the workflow administrator.

The workflow administrator must create the associations between all the entities in order to support the workflow processes. These associations can be made in any order. The creation of the role network will be demonstrated first to continue the scenario.

8.2.2.6 Creating the role network

This scenario's role network will be very small but the logic works on very small or very large role networks.

The first step to creating a role network is to give the role network a name. Inserting a record into the "RoleNet" database table does this. For his scenario the description of the role network will be "Order Fulfilment". This role network's ID will be used for every role inserted into this specific role network.

The SQL statement that will insert this role network is:

```
INSERT INTO rolenet VALUES(1,'Order Fulfilment');
```

The database table that holds the actual roles assigned to a role network is named "RH". This database table currently holds no roles for this role network.

The first role to be inserted must be the root role. This record's parentid field is null with the childid field equal to the roleid being inserted. In this case it will be the manager role. The SQL statement to insert this record is:

```
INSERT INTO rh VALUES(1,null,3);
```

This record will not be inserted if the role network doesn't already exist. This integrity constraint is enforced by foreign key constraints. The "rhstatafter_trig" trigger will check for a variety of other SoD requirements. Firstly it will check whether an existing root role already exists for the particular role network. If one does exist, then the trigger generates the following error:

```
ORA-20217: Cannot insert role into role network - root role
already exists for the particular role network.
ORA-06512: at "STEPHEN.RHSTATAFTER_TRIG", line 42
ORA-04088: error during execution of trigger
'STEPHEN.RHSTATAFTER_TRIG'
```

The next role to be inserted into the RH database table for this role network must be a child role of the root role. In this case it will be the stock controller role. The SQL statement to insert this record is:

```
INSERT INTO rh VALUES(1,3,2);
```

The "rhstatafter_trig" trigger will check first for the existence of a root role for the role network. The algorithm for this trigger can be found in chapter 7, section 7.2.3, figure 7.4. Once found it then checks whether the parent role of the inserted record is already in the role network as a child. In this case the parent role is the manager role and it is in the role network as the root role (parentid is null and childid is 3). If the parent role is not already there, then Oracle will raise an error. The following SQL statement tries to add a role to the role network where the parent role does not exist:

```
INSERT INTO rh VALUES(1,1,2);
```

And it generates the following error:

```
ORA-20213: Cannot insert role into role network - the parent
role is not an existing child role.
ORA-06512: at "STEPHEN.RHSTATAFTER_TRIG", line 87
ORA-04088: error during execution of trigger
'STEPHEN.RHSTATAFTER_TRIG'
```

The trigger then checks for the possibility of a circular reference. It does this by executing a hierarchical query that will fail if a circular reference has been created. If it will cause a circular reference, the trigger captures the exception and raises the following error:

```
ORA-20214: Cannot insert role into role network - it will
cause a circular reference to occur.
ORA-06512: at "STEPHEN.RHSTATAFTER_TRIG", line 94
ORA-04088: error during execution of trigger
'STEPHEN.RHSTATAFTER_TRIG'
```

The trigger's final check is to see whether any of the roles already in the role network are already conflicting with the role just inserted. If there are any then the following error is raised:

```
ORA-20215: Cannot insert role into role network -
conflicting role(s) are already present in the role network.
ORA-06512: at "STEPHEN.RHSTATAFTER_TRIG", line 81
ORA-04088: error during execution of trigger
'STEPHEN.RHSTATAFTER_TRIG'
```

The role network for this scenario is very small. It only contains two roles, namely the manager role as the root role, and the stock controller role as the child role.

As with the creation of a role network, it is important to understand the SoD requirement checks required when deleting roles from a role network.

8.2.2.7 Deleting a role from a role network

If a role needs to be deleted from a role network then the "drhstatafter_trig" trigger checks that the role to be deleted has no children roles in the role network. The delete statement below tries to delete the root role (manager) from the role network.

```
DELETE FROM rh WHERE parentid IS NULL AND rolenetid = 1 AND
childid = 3;
```

And it generates the following error:

```
ORA-20222: Cannot delete role association - children
associations already exist for the role been deleted.
ORA-06512: at "STEPHEN.DRHSTATAFTER_TRIG", line 17
ORA-04088: error during execution of trigger
'STEPHEN.DRHSTATAFTER_TRIG'
```

A role network may only be removed when it no longer contains any roles.

Roles can be associated with all other entities and the scenario will now continue with the steps involved with these associations.

8.2.2.8 Creating user to role associations

The workflow administrator will now create associations between the users and the roles. The user to role associations that need to be created for this workflow example are displayed in the following table.

User	**Role**
Thomas	Employee
Peter	Stock Controller
Frank	Manager

A user to role association is denoted mathematically as $(u_i,r_i) \in UA$. The SQL statements to insert these associations are:

The Employee role to the user Thomas:

```
INSERT INTO ua VALUES(1,1);
```

The Stock Controller role to the user Peter:

```
INSERT INTO ua VALUES(2,2);
```

The Manager role to the user Frank:

```
INSERT INTO ua VALUES(3,3);
```

Certain SoD requirements need to be enforced when adding these associations and the “uastatafter_trig” trigger checks these SoD requirements. The algorithm for this trigger can be found in chapter 7, section 7.2.1, figure 7.2. This trigger will generate an error if the user in the user to role association has conflicting users that are associated to roles that are conflicting with the role being added.

What this means is that if Frank were being associated to the Manager role then it would be necessary to do the following checks:

- Ascertain if Frank is conflicting with any other users. For example, Frank may be conflicting with Thomas.
- Check whether any of Thomas’s associated roles are conflicting with the Manager role. The association may continue if Thomas has no associated roles. If a problem was detected then the following error will be raised by the trigger:

```
ORA-20201: Cannot insert user to role association - a
conflicting user is already associated to a conflicting
role.
ORA-06512: at "STEPHEN.UASTATAFTER_TRIG", line 48
ORA-04088: error during execution of trigger
'STEPHEN.UASTATAFTER_TRIG'
```

These associations will restrict the permissions and the tasks that can be associated with the roles and in effect, the users.

8.2.2.9 Creating permission to role associations

The workflow administrator will now create associations between the permissions and the roles. The permission to role associations that need to be created for this workflow example are displayed in the following table.

Permission	**Role**
Edit Order Fields	Employee
Edit Order Completed Fields	Stock Controller
Edit Approve Order Fields	Manager
Edit Rejection Fields	

A permission to role association is denoted mathematically as $(p_i,r_i) \in PA$. The SQL statements to insert these associations are:

The Employee role to the “Edit Order Fields” permission:

```
INSERT INTO pa VALUES(1,1);
```

The Stock Controller role to the “Edit Order Completed Fields” permission:

```
INSERT INTO pa VALUES(2,5);
```

The Manager role to the “Edit Approve Order Fields” and “Edit Rejection Fields” permissions:

```
INSERT INTO pa VALUES(3,2);

INSERT INTO pa VALUES(3,3);
```

The “pastatafter_trig” trigger will check the role to permission associations that are inserted into the database table. The algorithm for this trigger can be found in chapter 7, section 7.2.1, figure 7.3. This trigger will generate an error if the permission in the permission to role association has conflicting permissions and that some of these permission’s associated roles are not conflicting with the role been added.

What this means is that if the “Edit Approve Order Fields” permission is being associated with the Manager role, then the following steps need to be taken:

- Identify all of the “Edit Approve Order Fields” permission’s conflicting permissions. This would identify the “Edit Order Fields” permission.
- Now identify all of the associated roles for the previously identified permissions. This would identify the Employee role.
- All of the previously identified roles must conflict with the Manager role. In this case they do as the Employee role does conflict with the Manager role. If this were not the case, the trigger would generate the following error:

```
ORA-20202: Cannot insert permission to role association - a
conflicting permission must be assigned to a conflicting
role.
ORA-06512: at "STEPHEN.PASTATAFTER_TRIG", line 48
ORA-04088: error during execution of trigger
'STEPHEN.PASTATAFTER_TRIG'
```

The task to role association is conceptually equivalent to the permission to role association.

8.2.2.10 Creating task to role associations

The workflow administrator will now create associations between the tasks and the roles. The task to role associations that need to be created for this workflow example are displayed in the following table.

Task	**Role**
Complete Order Form	Employee
Check Stock	Stock Controller
Issue Stock	
Order Stock	
Approve Order	Manager
Write Rejection Memo	

A task to role association is denoted mathematically as $(t_i,r_i) \in TA$. The SQL statements to insert these associations are:

The Employee role to the “Complete Order Form” task:

```
INSERT INTO ta VALUES(1,1);
```

The Stock Controller role to the “Check Stock”, “Issue Stock” and “Order Stock” tasks:

```
INSERT INTO ta VALUES(2,3);

INSERT INTO ta VALUES(2,4);

INSERT INTO ta VALUES(2,5);
```

The Manager role to the “Approve Order” and “Write Rejection Memo” tasks:

```
INSERT INTO ta VALUES(3,2);

INSERT INTO ta VALUES(3,6);
```

The “tastatafter_trig” trigger will check the role to task associations that are inserted into the database table. The algorithm for this trigger can be found in chapter 7, section 7.2.1, figure 7.3. This trigger will generate an error if the task in the task to role association has conflicting tasks and if some of these task’s associated roles are not conflicting with the role being added.

What this means is that if the “Approve Order” task is being associated with the Manager role, then the following steps need to be taken:

- Identify all of the “Approve Order” task’s conflicting tasks. This would identify the “Complete Order Form” task.
- Now identify all of the associated roles for the previously identified tasks. This would identify the Employee role.
- All of the previously identified roles must conflict with the Manager role. In this case they do as the Employee role does conflict with the Manager role. If this were not the case, the trigger would generate the following error:

```
ORA-20203: Cannot insert task to role association - a
conflicting task must be assigned to a conflicting role.
ORA-06512: at "STEPHEN.TASTATAFTER_TRIG", line 49
ORA-04088: error during execution of trigger
'STEPHEN.TASTATAFTER_TRIG'
```

As previously mentioned, both the permission to role association and the task to role association are very similar. The correct usage of these associations will help ensure a secure workflow environment.

The SoDA prototype also allows for associations to be deleted.

8.2.2.11 Deleting associations

Associations may be deleted without compromising the data integrity of the SoDA system.

This scenario demonstrated the effectiveness of the second version of the SoDA prototype. By ensuring that separation of duty requirements are met through the use of database triggers that guarantee the enforcement of the integrity constraints, it has exhibited an ability to help and ease the administrative burden.

8.2.3 Further functionality

While the triggers that form the basis of the second version of the SoDA prototype ensure the integrity of the data, they do not cater for many different security and management issues.

The first of these issues is the fact that a user can still connect directly to the database and can update any of the records in the database. This is generally easy to solve through a few different approaches.

For the first approach, triggers could be added to the database tables to prevent updates from occurring. This is not an ideal situation as it complicates the whole system unnecessarily.

Another approach is to maintain security through the proper use of users and roles within Oracle. A specific user with enough access rights and privileges would install the SoDA prototype. Then the users of the SoDA prototype would only be given select, insert and delete rights to the database tables. This would ensure that even if a user does access the database directly, they can do no more damage than if they were using the client application.

8.3 Implementation Issues

The SoDA prototype was developed according to the proposed model. As such, the design of the database tables followed the design of the sets of entities used by the model. The triggers could be simplified if the database tables were denormalised. This would involve adding redundant information

into the database tables so as to improve the performance and to simplify the algorithms utilised in the triggers.

The prototype also had to conform to the restrictions of the database management system. These restrictions dictated how the database triggers would function. The logic of the triggers could be simplified if the database management system handled the restrictions differently.

8.4 Conclusion

The concept of the SoDA prototype has changed markedly throughout this research. The current version has improved upon the previous version and has tried to correct problems inherent in the older design.

More importantly though, both versions of the SoDA prototypes help demonstrate the effectiveness of the conflicting entities paradigm. This paradigm is discussed in chapter 6. The prototypes show that it is possible to help prevent separation of duty conflicts within the workflow environment, and in so doing, it could help enforce company policy. Company policies can also be enforced through the use of active database technology.

It is evident that database management systems are incorporating active database techniques and technologies (G. Kappel et al., 2001). The capabilities of database management systems will expand to encompass the ability to enforce policy driven rules for the protection of an organisation's data.

With these developments in database technology it also becomes possible for the database management system to encompass elements of a workflow system. It is foreseen that the SoDA prototype would become abstracted into the database management system as part of a workflow component. Without tasks it could be implemented as part of an active relational DBMS (ARDBMS).

Chapter 9.

Conclusion

Due to the increasing volume and importance of information available electronically it has become necessary to facilitate an organisation's information security administration. Along with easing the administration of information security is the need to enforce organisational policies. This ensures that the information is managed and protected according to the needs of the organisation. For this research, these needs took the form of the enforcement of separation of duties within an organisation's workflow environment.

Separation of duties and the administration thereof can be eased through the utilisation of RBAC. In order to tie this ease of use to the managing of an organisation's information, it was necessary to ascertain if and how RBAC could be applied to a workflow environment.

A workflow environment controls the flow of information according to rules defined within business process models. RBAC employed within the workflow environment should thus be sensitised to the context of the work (Cholewka et al., 2000; Thomas & Sandhu, 1993). In order to employ RBAC into the workflow environment a workflow specific entity, the task, was added to the normal RBAC entities.

Through the use of the task entity it became possible to enforce separation of duty requirements in the workflow environment. In order to enforce the separation of duty requirements it was necessary to ascertain the different associations between the entities that may be allowed to take place.

These associations became the methodology required to implement the conflicting entities administration paradigm. The emphasis of this paradigm is to help the security administrator ensure that separation of duty constraints get enforced.

An implementation of the conflicting entities administration paradigm would help security administration by enforcing the integrity constraints for the administrators. The environment becomes sensitive to the requirements for enforcing the integrity constraints and is able to prevent potential errors from occurring.

This research relied upon answers for the research questions that were asked in the first chapter. These questions were asked in order to understand the requirements for accomplishing the primary research objective. This objective was the creation of a model for an advanced access control administration environment. These questions are now reviewed.

9.1 Research Questions Reviewed

The model of an advanced access control administration environment required a number of problems to be researched. These problems were specified in the form of questions.

9.1.1 How can RBAC concepts be applied in the workflow environment?

RBAC environments formulate access control requirements in terms of Roles, Users and Permissions. An additional element, from the workflow environment, was identified and included into these formulations. This identified workflow element, the task, was incorporated into the RBAC environment and the RBAC environment was adapted to make use of it.

9.1.2 How is the specification of SoD requirements influenced by the inclusion of the workflow entities?

It was identified that SoD requirements could be imposed by integrity constraints between the RBAC entities. These integrity constraints and the additional integrity constraints required by the addition of the workflow task were identified and formalised. Algorithms based upon these formalisations were developed for use in a prototype in order to demonstrate the model's effectiveness.

9.1.3 Can a single administration paradigm successfully formulate the range of separation of duty requirements?

The required range of SoD requirements can be formulated in the single administration paradigm. The single administration paradigm has been shown to handle the specification of SoD requirements and to ensure the enforcement of these requirements. The single administration paradigm does allow for a consistent and easy to use approach to access control administration.

These questions and their subsequent answers have contributed towards the accomplishment of the primary research objectives.

9.2 Contribution of this dissertation

The outcome of this research was a model for an advanced access control administration environment that handled the administration of SoD constraints in a workflow environment. This model has contributed various concepts. These concepts include the conflicting entities administration paradigm and enforcement algorithms for static SoD.

9.2.1 Development of conflicting entities administration paradigm

Once the concept of the task entity had been adapted into the RBAC environment, it became necessary to identify a model for the use of the extended RBAC environment. The solution that is presented is one where an administration environment will enforce associations and assignments between entities based upon the concept of conflicting entities.

Conflicting entities may only be related to other entities according to certain rules. The development of these rules was an integral part of this research. Once the rules were defined, it became possible to create generic algorithms to provide integrity constraint checking.

9.2.2 Development of enforcement algorithms for static SoD

These algorithms include the logic required to check for every possible entity association or conflict assignment that could occur. Each of these algorithms takes into account various likely scenarios due to the fact that an association or assignment needs to do a variety of checks before final approval.

These algorithms are generic enough to be realised in a variety of implementations. In order to demonstrate their functionality two prototypes were developed. The first prototype attempted to enforce the integrity constraint rules in a client-side application. In order to improve upon the design of the first prototype, it was decided to demonstrate the SoDA model using a different approach. This approach involved the use of active database techniques.

Active database techniques can be used to allow an organisation's data to abide by the organisation's policies (Kappel et al., 2001). In so doing, application systems do not need to be developed to support these policies as the database management system will support them for all application systems.

The second prototype was developed with these motives in mind. This prototype implemented the algorithms as database triggers in order to enforce the integrity constraints. This prototype demonstrated that the implementation of the conflicting entities administration paradigm is feasible at the database level.

9.3 Future Research

There are many areas of this research that require further analysis. This is partly a result of the scope taken. With this in mind it is clear that more integration with the workflow environment is required.

Future research would undertake to study how to evolve the conflicting entities administration paradigm to include the dynamic workflow environment. Topics such as the temporality of access rights and dynamic constraints would need to be researched.

Another area of future research is the issue of management. Management issues will become increasingly important once the SoDA model has been extended to include dynamic SoD.

Understanding how an organisation can manage their resources efficiently and maintain a high level of security would be advantageous. Research

would help in the design of a model that is flexible enough to adapt to any organisation's requirements while still remaining as secure as possible.

Another future research proposal for consideration is the concept of abstraction. This would involve researching the possibilities and benefits involved with abstracting workflow and its information security requirements into lower level applications. These applications include the operating system or the database management system.

Database management systems are evolving into active environments, able to handle the enforcement of organisational policies and as such could theoretically be extended to support the required workflow and security services. Understanding what should be abstracted and what should remain separate would also have to be assessed.

Bibliography

Ahn, G.J. & Sandhu, R.S. (1999). The RSL99 language for role-based separation of duty constraints. In proceedings of the 4th ACM Workshop on Role-based Access Control, Fairfax, Virginia, 28-29 October 1999.

Baldwin, R.W. (1990). Naming and grouping privileges to simplify security management in large database. In proceedings 1990 IEEE Symposium on Security and Privacy, 116 - 132, May 1990.

Bertino, E & Ferrari, E. (1999). The specification and enforcement of authorization constraints in workflow management systems. ACM Transactions on Information and System Security, 2 (1), February 1999, 65–104.

Cholewka, D.G., Botha R.A. & Eloff, J.H.P. (2000). A context-sensitive access control model and prototype implementation. In proceedings of the 16th IFIP Information Security Conference, Beijing, China, August 2000.

Clark, D.D. & Wilson, D.R. (1987). A comparison of commercial and military computer security policies. In proceedings of IEEE Symposium on Security and Privacy, April 1987, 184 - 194.

Ferraiolo, D., Barkley, J.F., & Kuhn, D.R. (1999). A role-based access control model and reference implementation within a corporate intranet. ACM Transactions on Information and System Security, 2 (1), February 1999, 34–64.

Gligor, V.D., Gavrila, S.I. & Ferraiolo, D. (1998). On the formal definition of separation-of-duty policies and their composition. In proceedings of IEEE Symposium on Security and Privacy, Oakland, California, 3 – 6 May 1998.

Hollingsworth, D. (1995). Workflow Management Coalition: The workflow reference model [online]. Word Doc. Issue 1.1. Workflow Management Coalition. [Retrieved 12 March 1998]. Available from internet: URL http://www.wfmc.org.

ISO 7498–2. (1989). Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 2: Security Architecture. International Standards Organisation, Geneva, Switzerland.

Kappel, G., Rausch-Schott, S., Retschitzegger, W., & Sakkinen, M. (2001). Bottom-up design of active object-oriented databases, Communications of the ACM, 44 (4), April 2001, pg 99.

Kuhn, D.R. (1997). Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In proceedings of the 2nd ACM Workshop on Role-based Access Control, Fairfax, VA, October 1997.

Leymann, F., & Roller, D. (1999). Production Workflow: Concepts and Techniques. Prentice Hall PTR (ECS Professional) Copyright 2000 ISBN 0-13-021753-0

Michener, J. (1999). System insecurity in the internet age. IEEE Software, July/August, 62-69.

Mohanty, R. P. (1998). BPR – Beyond industrial engineering? Work Study, 47 (3), 90–96.

Motwani, J., Kumar, A., & Jiang, J. (1998). Business process reengineering - A theoretical framework and an integrated model. International Journal of operations & Production Management, 18 (9/10), 1998, 964-977.

Nash, M.J. & Poland, K.R. (1990). Some conundrums concerning separation of duty. In proceedings of the IEEE Symposium on Security and Privacy, 201 - 207, May 1990.

Nyanchama, M., & Osborn, S. (1999). The role graph model and conflict of interest. ACM Transactions on Information and System Security, 2 (1), February 1999, 3–33.

Oracle. (1999). Application developer's guide – Fundamentals. Oracle Corporation, [Cited 01 April 2001]. Available online at http://www.oracle.com.

Osborn, S., Sandhu, R., & Munawer, Q. (2000). Configuring role-based access control to enforce mandatory and discretionary access control policies. ACM Transactions on Information and System Security, 3 (2), May 2000.

Paton, N.W., & Díaz, O. (1999). Active database systems. ACM Computing Surveys, 31 (1), March 1999, 63-103.

Saltzer, J.H. & Schroeder, M.D. (1975). The protection of information in computer systems. In proceedings of IEEE, 63, 1278-1308.

Sandhu, R. (1988). Transaction control expressions for separation of duties. In proceedings of 4th Aerospace Computer Security Conference, 282 - 286, Dec 1988.

Sandhu, R. (1990), Separation of duties in computerized information systems. In proceedings of IFIP WG11.3 Workshop on Database Security, September 1990.

Sandhu, R. (1993). Lattice-based access control models. IEEE Computer, 26 (11), November 1993, 9 – 19.

Sandhu, R. (1998). Role-based access control. Advances in Computers, 46, Academic Press, 1998.

Sandhu, R., Coyne, E., Feinstein, H. L., & Youman, C. E. (1996). Role-based access control models. IEEE Computer, 29 (2), 38-47, February 1996.

Sandhu, R., & Munawer, Q. (1998). How to do discretionary access control using roles. In Proceedings of the 3rd ACM Workshop on Role-based Access Control, Fairfax, Virginia, 22 – 23 September 1998, 47 – 54.

Sandhu, R., & Samarati, P. (1996). Authentication, access control, and audit. ACM Computing Surveys. 28 (1).

Shen, H., & Dewan, P. (1992) Access control for collaborative environments, CSCW 92 Proceedings, pg 51.

Simon, R & Zurko, M.E. (1997). Separation of duty in role-based environments. In proceedings of 10th Computer Security Foundation Workshop, Rockport, Massachusetts, 10 – 12 June 1997.

Stadler, C. (2000, Sept/Oct). Workflow: The missing link between customer intelligence and customer interaction. Call Centre IQ, pp. 28 – 35.

Stallings, W. (1995). Network and Internetwork Security Principles and Practice. Englewood Cliffs: Prentice Hall.

Teng, J., Jeong, S., & Grover, V. (1998). Profiling successful reengineering projects. Communications of the ACM, 41 (6), June 1998 pg. 99

Thomas, R. & Sandhu, R. (1993). Towards a task-based paradigm for flexible and adaptable access control in distributed applications. In Proceedings of the 1992–1993 ACM SIGSAC New Security Paradigms Workshop, Little Compton, RI, 138 – 142.

von Solms, R. (1999). Information security management: Why standards are important. Information management & computer security, 7 (1), 50-57.

WfMC (1996). Workflow Management Coalition: Terminology and glossary [online]. Document Number WFMC-TC-1011. Issue 2.0. Workflow Management Coalition. [Cited 25 May 2000]. Available from internet: URL http://www.wfmc.org

WfMC (1998). Workflow Management Coalition: Workflow security considerations - white paper [online]. Document Number WFMC-TC-1019. Issue 1.0. Workflow Management Coalition. [Cited 25 May 2000]. Available from internet: http://www.wfmc.org

Zhou, J., & Lam, Y. (1999). Securing digital signatures for non-repudiation, Computer Communications, 22 (8), 25 May 1999, 710-716.

Appendix A. Paper at Conference

The paper titled “Conflict Analysis as a Means of Enforcing Static Separation of Duty Requirements in Workflow Environments” was presented at the SAICSIT 2000 conference in Cape Town, South Africa, 1-3 November 2000. It was published as a short paper in a special conference issue of the South African Computer Journal, number 26, November 2000.

Conflict Analysis as a Means of Enforcing Static Separation of Duty Requirements in Workflow Environments

Stephen Perelson and Reinhardt A. Botha

Faculty of Computer Studies, Port Elizabeth Technikon, Port Elizabeth, South Africa
rbotha@computer.org

Abstract

The increasing reliance on information technology to support business processes has emphasised the need for information security mechanisms. This, however, has resulted in an ever-increasing workload in terms of security administration. Policy-based approaches have been proposed, promising to lighten the workload of security administrators. Separation of duty is one of the principles cited as a requirement when setting up these policy-based mechanisms. Different types of separation of duty policies exist. They can be categorised into policies that can be enforced at administration time, viz. static separation of duty requirements and policies that can be enforced only at execution time, viz. dynamic separation of duty requirements. This paper deals with specifying static separation of duty requirements in role-based workflow environments. It proposes a mathematical model based on the concept of "conflicting entities" to express static separation of duty requirements. It provides a detailed explanation of the integrity checking that must take place at administration time to ensure that specified separation of duty requirements are honoured.

Keywords: *Access Control, Separation of duty, Authorisation constraints, Workflow*
Computing Review Category: *D4.6, G2.3, H4.1, K6.5*

1 Introduction

With the increasing amount of information available electronically it is not only necessary to find a means to ease the job of the security administrator, but also to ensure that the information is protected and managed according to organisational policies.

On the one hand, Role-based Access Control (RBAC) has been promoted as a possible solution to the administration nightmares that face security administrators [5]. On the other hand, however, workflow technology has been boasted as a means of controlling the flow of information according to business process models. RBAC mechanisms employed in the workflow environment should thus be sensitised to the context of the work [3,16].

The context of the work is determined by factors such as the sequence and history of events, as well as the organisational policies. One expression of organisational policy can be found in the age-old principle of separation of duty (SoD). Separation of duty's primary objective is to prevent fraud, i.e. protect the integrity of the information [4]. It can, however, be largely enforced by means of appropriate access control mechanisms.

Access control is a two-phase process. During phase one, users receive potential to perform certain activities – this is called access control administration. Phase two occurs when an application is used and the actual permission is granted to the user – this is called run-time access control.

SoD requirements can, similarly, be evaluated in two ways. In the first instance, the access control administration tool can check that specified requirements are met. This is referred to as *static separation of duty*. With static SoD the user would thus not even receive the potential to ever perform an activity. In the second instance the SoD requirements can be enforced at run-time. This is referred to as *dynamic separation of duty*. With dynamic SoD the roles that the user may activate are thus controlled.

This paper focuses on static SoD requirements, i.e. it addresses enforcement from an access control administration perspective.

2 Related SoD Research

The term "separation of privilege" was identified as one of eight design principles for the protection of information in computer systems by Saltzer and Schroeder [14]. They built on the observation that a security system with two keys is more robust and flexible than one that requires a single key. No single accident, deception or breach of trust is therefore sufficient to compromise the system.

Clark and Wilson [4] identified separation of duty as one of the two major mechanisms that can be implemented to ensure data integrity. SoD serves as a mechanism to counteract fraud and error, whilst assuring correspondence between system objects and the real world objects that they represent. They asserted that, at the policy level, processes are divided into steps, with each step being performed by a different person. Separation of duty is thus tightly connected to application semantics.

The issue of separation of duty has been addressed from different perspectives by several authors. Interested readers are referred to [2],[6],[10],[11], [12] and [15]. This paper only examines related work in which concepts directly called upon in our interpretation are discussed.

Kuhn [8] explored the mutual exclusion of roles as a separation of duty mechanism. He presented a taxonomy whereby a separation of duty requirement is categorised according to the time at which mutual exclusion is applied (static vs. dynamic), as well as the degree to which privileges are shared by mutually exclusive roles (strong or partial exclusion). Strong exclusion implies no common permission or user assignments for exclusive roles. Partial exclusion, on the other hand, implies that mutually exclusive roles may share permissions (or users) but that each role should have permissions assigned that are unique to that role.

Nyanchama and Osborn [9] discussed various types of conflicts that have to be considered when implementing separation of duty policies. They evaluated the effect of role hierarchies in great depth in terms of their role-graph model.

Ahn and Sandhu [1] defined the RSL99 language for specifying separation of duty constraints. They based their SoD requirements on the concepts of conflicting users, conflicting roles and conflicting permissions. New static separation of duty properties are discovered through the application of their RSL99 language.

The observation that existing separation of duty models do not take into consideration the work processes has been made by [1] and [9]. The relevance of work process models on the dynamic enforcement of separation of duty requirements are easily recognisable, however the impact of work process models is not identified. This paper will extend the typical RBAC model to include the notion of a task which represents the basic building block of work models. It will show how the task concept influences static SoD. In particular a single administration paradigm that includes the task will be presented.

The administration paradigm presented hinges on the understanding of role-based access control. A detailed look at RBAC is thus in order.

3 Role-Based Access Control

The basic premise of RBAC is that access permissions are assigned to roles rather than to individuals. Individuals are assigned to roles in order to obtain the access permissions that the individual requires in order to work. This greatly reduces the administration burden.

In this paper the existing and well-accepted RBAC96 model [13] is used. Choosing an independently developed existing model for this exercise gives us an element of objectivity in assessing the power of the proposed administration paradigm. An overview of the RBAC96 model must therefore be given.

3.1 RBAC96

The RBAC96 model consists of 4 main components: users, roles, permissions and sessions. Since a session is a run-time concept it is irrelevant to the administration environment and thus to this paper.

Users (U) are associated with roles (R) through the user-assignment relation (UA). Similarly, permissions (P) are associated with roles (R) through the permission-assignment relation (PA). Roles are arranged in role hierarchies through the partial order RH. A role that is senior to another role inherits the permissions of the junior role. A user that is associated with a senior role in the role hierarchy therefore may also assume all the roles junior to the senior role in the RH partial order.

Consider a short formal summary of the relevant components in the RBAC96 model [1,13]:

Definition 3.1: RBAC entities

U = set of users, $\{u_1,u_2,\ldots,u_l\}$

R = set of roles, $\{r_1,r_2,\ldots,r_m\}$

P = set of permissions, $\{p_1,p_2,\ldots,p_n\}$

Definition 3.2: RBAC associations

$UA \subseteq U \times R$, a many-to-many user-to-role assignment relation

$PA \subseteq P \times R$, a many-to-many permission-to-role assignment relation

$RH \subseteq R \times R$, a partial order on R called the role hierarchy, also written as $\leq$

Definition 3.3: Roles function

$roles$: $U \cup P \rightarrow 2^R$, a function mapping the sets U and P to a set of roles

$roles^*$: $U \cup P \rightarrow 2^R$ extends roles in the presence of a role hierarchy

$roles(u_i)=\{r \in R \mid (u_i,r) \in UA\}$

$roles(p_i)=\{r \in R \mid (p_i,r) \in PA\}$

$roles^*(u_i)=\{ r \in R \mid (\exists r \leq r')[(u_i,r') \in UA]\}$

$roles^*(p_i)=\{ r \in R \mid (\exists r' \leq r)[(p_i,r') \in PA]\}$

Note that the definition of $roles^*$ is carefully formulated to reflect the role inheritance with respect to users going downwards and with respect to permissions going upwards.

Definition 3.4: Permissions function

$perm$: $U \cup R \rightarrow 2^P$, a function mapping users and roles to a set of permissions.

$perm^*$: $U \cup R \rightarrow 2^P$, extends $perm$ in the presence of a role hierarchy.

$perm(r_i) = \{p \in P \mid (p,r_i) \in PA\}$

$perm(u_i) = \{p \in P \mid (\exists r \in roles(u_i))[(p,r) \in PA]\}$

$perm^*(r_i) = \{p \in P \mid (\exists r' \leq r)[(p,r') \in PA]\}$

$perm^*(u_i) = \{p \in P \mid (\exists r \in roles^*(u_i))[(p,r) \in PA]\}$

Subsequently we introduce the workflow extensions.

3.2 Workflow extensions

A typical workflow is a set of tasks linked together in a network, thus forming a business process [7]. The workflow system is responsible for determining the route

that work will follow through the organisation. From an access control perspective the basic building blocks are tasks that may be performed by a specific organisational role.

Definition 3.5: Workflow entities

T = Set of tasks, $\{t_1,t_2,\ldots,t_n\}$

Definition 3.6: Workflow Associations

$TA \subseteq T \times R$, a many-to-many task-to-role assignment relation

The RBAC96 functions must thus be extended.

Definition 3.7: Extended roles function

$roles$: $U \cup P \cup T \rightarrow 2^R$, a function mapping the sets U and R and T to a set of roles.

$roles^*$: $U \cup P \cup T \rightarrow 2^R$ extends $roles$ in the presence of a role hierarchy

$roles(u_i)$,$roles(p_i)$, $roles^*(u_i)$ and $roles^*(p_i)$ remain according to Definition 3.3

$roles(t_i)=\{r \in R \mid (t_i,r) \in TA\}$

$roles^*(t_i)=\{ r \in R \mid (\exists r \leq r')[(t_i,r') \in TA]\}$

Definition 3.8: Revised permissions function

$perm$: $U \cup R \cup T \rightarrow 2^P$, a function mapping users, roles and tasks to a set of permissions.

$perm^*$: $U \cup R \cup T \rightarrow 2^P$, extends $perm$ in the presence of a role hierarchy.

$perm(r_i)$, $perm(u_i)$, $perm^*(r_i)$ and $perm^*(u_i)$ remain according to Definition 3.4

$perm(t_i) = \{p \in P \mid (\exists r \in roles(t_i))[(p,r) \in PA]\}$

$perm^*(t_i) = \{p \in P \mid (\exists r \in roles^*(t_i))[(p,r) \in PA]\}$

The above definitions give the elements essential to the administration of access control in the workflow environment. The following paragraph will suggest a conflict paradigm to define further restrictions required to support static SoD requirements.

4 The conflict paradigm

Separation of duty is concerned with the prevention of fraud by ensuring that a single user does not have too much power. Power is vested in permissions, therefore the essence of our paradigm lies with conflicting permissions.

Definition 4.1: Conflicting permissions are permissions that can result in unnecessary power if bestowed on the same person. Formally it is represented by

$CP \subseteq P \times P$, a many-to- many relation indicating conflict between permissions with

$(p_i,p_j) \in CP \Leftrightarrow (p_j,p_i) \in CP$ and $(p_i,p_i) \notin CP$.

We can now present the following axiom which will represent our basic safety condition

Basic Safety Condition: Conflicting permissions may not be assigned to a user.

Formally, $(perm^*(u) \times perm^*(u)) \cap CP = \varnothing)$

Since non-conflicting permissions cannot influence the basic safety condition the following axiom to supplement the basic safety condition is formulated.

Axiom 4.1: Non-conflicting permissions may be assigned to both conflicting or non-conflicting roles.

Now consider the other conflicting entities that form part of the conflicting entity paradigm.

Definition 4.2: Conflicting users are users who are likely to conspire. Formally they are represented by

$CU \subseteq U \times U$, a many-to-many relation indicating conflict between users with

$(u_i,u_j) \in CU \Leftrightarrow (u_j,u_i) \in CU$ and $(u_i,u_i) \notin CU$.

Axiom 4.2 Conflicting users are considered as a single user.

In practical terms conflicting users may be family members or people who are known to have conspired.

Definition 4.3: Conflicting roles are roles that together have the ability to conspire, i.e. they are assigned some (but not all) conflicting permissions. They are represented by

$CR \subseteq R \times R$, a many-to-many relation indicating conflict between roles with

$(r_i,r_j) \in CR \Leftrightarrow (r_j,r_i) \in CR$, $(r_i,r_i) \notin CR$ and

$(r_i,r_j) \in CR \Rightarrow perm^*(r_i) \times perm^*(r_j) \cap CP \neq \varnothing$

Note that roles are abstractions to ease administration. Although the conflicting permissions may not be identified as such in the administration tool, making roles conflict if they are not assigned some conflicting permissions is senseless. This principle thus is a logical principle, which in practice may not be checked literally in the administration tool.

Since conflicting roles must have some conflicting permissions we can state that non-conflicting roles do not have conflicting permissions. In the spirit of Axiom 4.1 we thus formulate the following axiom.

Axiom 4.3: Non-conflicting roles may be assigned either non-conflicting or conflicting users.

Definition 4.4: Conflicting tasks are tasks requiring conflicting permissions to complete. Formally they are represented by

$CT \subseteq T \times T$, a many-to-many relation indicating conflict between tasks with

$(t_i,t_j) \in CT \Leftrightarrow (t_j,t_i) \in CT$, $(t_i,t_i) \notin CT$ and

$(t_i,t_j) \in CT \Rightarrow perm^*(t_i) \times perm^*(t_j) \cap CP \neq \varnothing$

Note that conflicting tasks are assigned conflicting permissions. Since non-conflicting tasks can have only non-conflicting permissions assigned to them we can see that they could not influence the basic safety condition, therefore the following axiom is formulated.

Axiom 4.4: Non-conflicting tasks may be assigned to conflicting and non-conflicting roles.

These principles and definitions are essentially focused on the permissions exercised by the users. The integrity of the access control information is, however, determined by the associations in the access control model. In a

RBAC environment users are never assigned directly to permissions. The role construct plays a pivotal role in linking tasks, users and permissions together. The next paragraph will therefore show the integrity requirements pertaining to the associations allowed in the access control model.

5 Integrity Requirements

This paragraph presents a number of theorems reflecting integrity requirements that will have to be upheld in a security administration tool.

Theorem 5.1: Under the basic safety condition, conflicting roles may only have non-conflicting users assigned to them, i.e.

$(u_i,r_k) \in UA \wedge (u_j,r_l) \in UA \wedge (r_k,r_l) \in CR \Rightarrow (u_i,u_j) \notin CU$

Proof:

Assume that $(u_i,r_k) \in UA \wedge (u_j,r_l) \in UA) \wedge (r_k,r_l) \in CR. \wedge (u_i,u_j) \in CU$:

$perm^*(u_i) \supseteq perm^*(r_k)$ (Def 3.4)

$perm^*(u_j) \supseteq perm^*(r_l)$ (Def 3.4)

$(r_k,r_l) \in CR.$

$\Rightarrow \quad perm^*(r_k) \times perm^*(r_l) \cap CP \neq \varnothing$ (Def 4.3)

$\Rightarrow \quad perm^*(u_i) \times perm^*(u_j) \cap CP \neq \varnothing$

which contradicts the Basic Safety Condition. ***QED.***

Theorem 5.2: Under the basic safety condition, conflicting permissions may only be assigned to conflicting roles. Formally

$(p_i,r_k) \in PA \wedge (p_j,r_l) \in PA \wedge (p_i, p_j) \in CP \Rightarrow (r_k, r_l) \in CR$

Proof:

Assume that two conflicting permissions p_i and p_j are assigned to non-conflicting roles r_k and r_l, i.e.

$(p_i,r_k) \in PA \wedge (p_j,r_l) \in PA \wedge (p_i,p_j) \in CP \wedge (r_k,r_l) \notin CR$

Choose a user u_x and associate it with roles r_k and r_l. Since $(r_k,r_l) \notin CR$ this allowed by Th. 5.1.

$\therefore \quad (u_x,r_k) \in UA \wedge (u_x,r_l) \in UA \wedge (p_i,r_k) \in PA \wedge (p_j,r_l) \in PA$

$\Rightarrow \quad \{p_i, p_j\} \subseteq perm^*(u_x)$ (Def 4.2)

But $(p_i, p_j) \in CP,$ which contradicts the Basic Safety Condition.

QED.

Theorem 5.3: Under the basic safety condition, conflicting tasks may only be assigned to conflicting roles. That is

$(t_i,r_k) \in TA \wedge (t_j,r_l) \in TA \wedge (t_i,t_j) \in CT \Rightarrow (r_k, r_l) \in CR$

Proof:

Assume that two conflicting tasks t_i and t_j are assigned to non-conflicting roles r_k and r_l.

$(t_i,r_k) \in TA \wedge (t_j,r_l) \in TA \wedge (t_i,t_j) \in CT \wedge (r_k,r_l) \notin CR$

Choose a user u_x and associate it with roles r_k and r_l. Since $(r_k,r_l) \notin CR$ this is allowed by Th. 5.1.

$perm^*(r_k) \subseteq perm^*(u_x)$ (Def 3.4)

$perm^*(r_l) \subseteq perm^*(u_x)$ (Def 3.4)

also $perm^*(t_i) \subseteq perm^*(r_k)$ (Def 3.8)

$perm^*(t_j) \subseteq perm^*(r_l)$ (Def 3.8)

$\therefore \quad perm^*(t_i) \subseteq perm^*(u_x)$

and $perm^*(t_j) \subseteq perm^*(u_x)$

$\Rightarrow \quad perm^*(t_i) \times perm^*(t_j) \cap CP \neq \varnothing$ (Def 4.4)

$\Rightarrow \quad perm^*(u_x) \times perm^*(u_x) \cap CP \neq \varnothing$

which contradicts the Basic Safety Condition. ***QED.***

Using truth table equivalence we state the following corollary.

Corollary 5.3: Under the basic safety condition, non-conflicting roles may only have non-conflicting tasks assigned to them. That is

$(t_i,r_k) \in TA \wedge (t_j,r_l) \in TA \wedge (r_k, r_l) \notin CR \Rightarrow (t_i,t_j) \notin CT$

Above theorems limit the associations allowed between users, roles, permissions and tasks.

6 Conclusion

This paper explored the static separation of duty requirements in workflow environments. This was done through extending the RBAC components with workflow specific components. In particular it demonstrated how static separation of duty requirements specified through the use of conflicting users, conflicting roles, conflicting permissions and conflicting tasks could be enforced. Enforcement is based on maintaining the integrity of the associations allowed between components.

The following table summarises the work explored in this paper.

<table>
<tr><td colspan="2" rowspan="2">May be associated with</td><td colspan="4">Roles</td></tr>
<tr><td colspan="2">Conflicting</td><td colspan="2">Non-conflicting</td></tr>
<tr><td rowspan="2">Users</td><td>Conflicting</td><td rowspan="2">Th 5.1</td><td>×</td><td rowspan="2">Ax 4.3</td><td>✓</td></tr>
<tr><td>Non-Conflicting</td><td>✓</td><td>✓</td></tr>
<tr><td rowspan="2">Permis-sions</td><td>Conflicting</td><td colspan="2">✓</td><td colspan="2">×</td></tr>
<tr><td>Non-Conflicting</td><td colspan="2">✓</td><td colspan="2">✓</td></tr>
<tr><td rowspan="2">Tasks</td><td>Conflicting</td><td colspan="2">✓</td><td colspan="2">×</td></tr>
<tr><td>Non-Conflicting</td><td colspan="2">✓</td><td colspan="2">✓</td></tr>
</table>

A ✓ in the table indicates that an association is allowed, whilst a × shows that an association is prohibited. For example, theorem 5.1 proves that non-conflicting users may only be assigned to conflicting roles while axiom 4.4 states that non-conflicting tasks may be assigned to either conflicting or non-conflicting roles.

A security administration tool may allow conflicting assignments to be made if it can ensure the integrity of the association by the use of remedial actions. For example, two conflicting permissions may be assigned to two non-conflicting roles. The tool would provide the administrator with an option to not continue with the assignment or to set the two roles to be conflicting and to continue with the assignment

Usability factors in security administration tools will be considered in future work. The extension of the paradigm to allow for the specification of dynamic separation of duty requirements also needs to be considered.

References

[1] Ahn, G.-J. and Sandhu, R.S.. *The RSL99 Language for Role-based Separation of duty constraints.* In Proceedings of the 4th ACM Workshop on Role-based Access Control, Fairfax, Virginia, 28 –29 October 1999, pp. 43 – 53.

[2] Baldwin, R.W. *Naming and Grouping Privileges to Simplify Security Management in Large Database.* Proc 1990 IEEE Symposium on Security and Privacy, , May 1990, pp. 116 – 132

[3] Cholewka, D.G., Botha R.A. and Eloff, J.H.P. *A context-sensitive access control model and prototype implementation.* Proc of the 15th IFIP TC11 Information Security Conference, Beijing, China, August 2000, pp. 141 – 150.

[4] Clark, D.D. and Wilson, D.R.. *A comparison of commercial and military computer security policies.* Proc. of IEEE Symposium on Security and Privacy, April 1987, pp. 184 – 194.

[5] Ferraiolo, D., Barkley, J.F. and Kuhn, D.R. *A Role-Based Access Control Model and Reference Implementation Within a Corporate Intranet.* ACM Transactions on Information and System Security, Vol. 2, No. 1, February 1999, pp. 34–64.

[6] Gligor, V.D, Gavrila, S.I. and Ferraiolo, D. *On the Formal Definition of Separation of Duty Policies and their composition.* In Proceedings IEEE Symposium on Security and Privacy, Oakland, California, 3 – 6 May 1998, pp. 172 – 183.

[7] Hollingsworth, D. *The Workflow Reference Model.* Document Number TC-00-1003. Issue 1.1. 19 Jan 1995. Available from www.wfmc.org

[8] Kuhn, D.R. *Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems.* In Proceedings of the 2nd ACM Workshop on Role-based Access Control, Fairfax, VA, October 1997, pp. 23 – 30.

[9] Nyanchama, M. and Osborn, S. *The role-graph model and conflict of interest.* ACM Transactions on Information and Systems Security, 2(1), February 1999, pp. 3-33.

[10] Nash, M.J. and Poland, K.R. *Some Conundrums Concerning Separation of Duty.* In Proceedings 1990 IEEE Symposium on Security and Privacy, May 1990, pp. 201 – 207.

[11] Sandhu, R. *Transaction Control Expressions for Separation of Duties.* Proc. of 4th Aerospace Computer Security Conference, 282 – 286, Dec 1988.

[12] Sandhu, R.S. *Separation of Duties in Computerized Information Systems.* S. Jajodia, C.E. Landwehr (Eds.): Database Security, IV: Status and Prospects. Results of the IFIP WG 11.3 Workshop on Database Security, Halifax, U.K. 18-21 September 1990. pp. 179 – 190.

[13] Sandhu, R.S., Coyne, E.J., Fenstein, H.L. and Youman, C.E.. *Role-based Access Control Models.* IEEE Computer, Volume 29, Number 2, February 1996, pp. 38 – 47.

[14] Saltzer, J.H. and Schroeder, M.D. *The Protection of Information in Computer Systems.* In Proceedings of IEEE, 63(9), 1975, pp. 1278–1308,

[15] Simon, R. and Zurko, M.E. *Separation of duty in Role-based Environments.* In Proceedings of 10th Computer Security Foundation Workshop, Rockport, Massachusetts, 10 – 12 June 1997.

[16] Thomas, R.K. and Sandhu, R.S. *Towards a task-based paradigm for flexible and adaptable access control in distributed applications.* In Proceedings. of 1992–1993 ACM SIGSAC New Security Paradigms Workshop, Little Compton, RI, 1993, pp. 138 – 142.

[17] Workflow Management Coalition. *Workflow Security Considerations - White Paper.* Document Number WFMC-TC-1019. Issue 1.0. Feb 1998. Available from www.wfmc.org

Appendix B. Paper published

The paper titled “Separation of Duty Administration” was accepted for publication in the South African Computer Journal, number 27.

Separation of Duty Administration

Stephen Perelson[a] Reinhardt Botha[a] Jan Eloff[b]

[a] *Faculty of Computer Studies, Port Elizabeth Technikon, Port Elizabeth*
{stephen,reinhard}@petech.ac.za
[b] *Department of Computer Science, Rand Afrikaans University, Johannesburg*
eloff@rkw.rau.ac.za

Abstract

Access control administration is a huge task. Administration tools should assist the administrator in ensuring that the access control requirements are met. One example of an access control requirement is Separation of Duty (SoD). SoD requirements specify that no single person may have sufficient authority to complete a business process unilaterally.

The SoDA prototype administration tool has been developed to assist administrators with the administration of SoD requirements. It demonstrates how the specification of both Static and Dynamic SoD requirements can be done based on the "conflicting entities" paradigm. Static SoD requirements must be enforced in the administration environment. The SoDA prototype, therefore, enforces the specified static SoD requirements.

Keywords: *Information Security, Access Control Administration, Separation of Duty*
Computing Review Categories: *D4.6, H2.7, H4.1, K6.5*

1 Introduction

Security administrators must manage an ever-increasing number of systems under their control. In recent years, Role-based Access Control (RBAC) has been promoted as a possible solution to the resultant administration nightmares [5]. With the increasing amount of information available electronically, it is necessary not only to find a means to ease the job of the security administrator, but also to ensure that the information is protected and managed according to organizational policies.

One expression of organizational policy can be found in the age-old principle of Separation of Duty (SoD). Saltzer and Schroeder [10] identified SoD, or "separation of privilege" as they called it, as one of eight design principles for the protection of information in computer systems. They built on the observation that a security system with two keys is more robust and flexible than one that requires a single key. No single accident, deception or breach of trust is therefore sufficient to compromise the system. Clark and Wilson [4] identified SoD as one of the two major mechanisms that can be implemented to ensure data integrity. SoD serves as a mechanism to counteract fraud and error, while assuring correspondence between system objects and the real world objects that they represent.

Furthermore, they [4] asserted that, at the policy level, processes are divided into tasks, with each task being performed by a different person. [1] and [8] observed that existing SoD models do not take work processes into consideration. Work processes are often facilitated through the use of workflow systems. Workflow systems are constructed around tasks that are linked according to business rules to represent a business process. This paper introduces the task as an additional building block for expressing SoD requirements in workflow systems.

Even with the introduction of the task abstraction, the administration of SoD requirements remains a mammoth task. In a large organization, there may be thousands of objects that require protection. The organization may have thousands of users, filling hundreds of different positions in the organization. The identification of all the access requirements requires a huge effort. It is virtually impossible to maintain consistency when performing such a huge task, unless the administration tools provide appropriate assistance.

The SoDA prototype is introduced to assist security administrators with the specification of access control requirements according to Role-based Access Control principles. More specifically, the SoDA prototype is intended to assist with the administration of SoD requirements. In order to demonstrate the "conflicting entities" administration paradigm as used within the SoDA prototype, the remainder of the paper is structured as follows. First, a brief review of role-based access control principles is provided. Thereafter, the additional concept of a task is introduced. This is followed by a discussion on the use of the "conflicting entities" paradigm to specify SoD requirements. Finally, we illustrate how the SoDA prototype is used to administer SoD requirements.

2 Basic Concepts

This section will provide the necessary background to explain the principle of separation of duty within role-

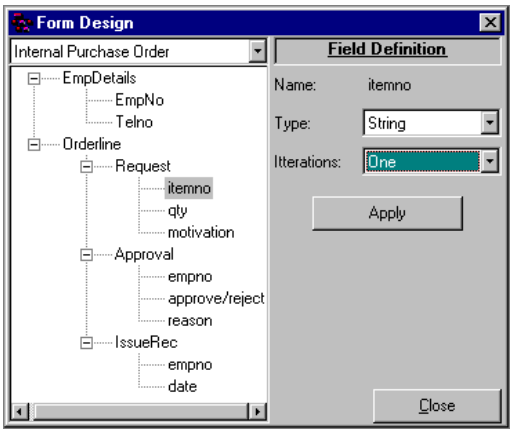

Figure 1: Form design environment used to create a "Purchase Order"

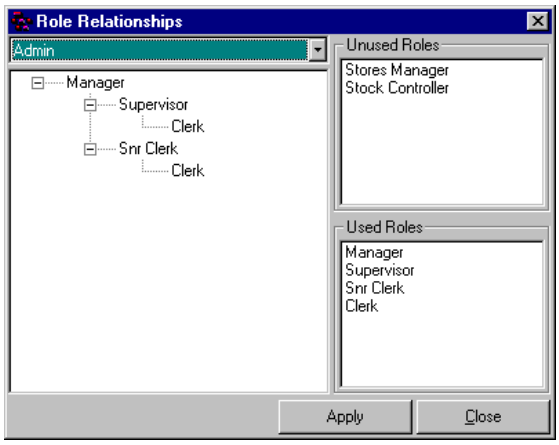

Figure 2: SoDA associates roles according to named role networks

based workflow systems.

2.1 Role-based Access Control

The concept of a role is pivotal in role-based access control. Users receive access permissions based on the roles that they may assume. Users are anyone/anything that accesses resources in the system. A user may, therefore, be an individual or another program. Roles often correspond to positions in the organizational structure. It is thus a semantic construct, created to ease the management of access rights. Permissions can be interpreted as the right to execute a certain method of an object.

The SoDA prototype considers an object to be a document containing various field objects. Users may perform different actions on the field objects, e.g. add another instance of the field object, delete a field object, edit the contents of a field object or view the contents of a field object. Individual field objects may be grouped, resulting in composite objects. Figure 1 shows how a hierarchical view, representing object containment, can be used to create the 'Internal Purchase Order' object. Permissions could relate to any of the field objects, or composite field objects, in the 'Internal Purchase Order' object. Permissions assigned to an object are inherited for objects contained by that object. For example, the permission to edit Employee Details will imply the permission to edit all fields that form part of Employee Details on the form.

Roles may be related through a partial order. A role inherits permissions assigned to the roles that are junior to it in the partial order. For example, the 'Manager' role may be considered senior to the 'Supervisor' role. The 'Manager' role will, therefore, inherit the permissions assigned to the 'Clerk' role. Figure 2 shows how the SoDA prototype manages the associations between roles. In SoDA, roles are related to other roles within disjoint, named role networks. The combination of all named role networks is similar to the role-graph presented by [8], if an artificial maximum and an artificial minimum role were introduced.

The concepts employed in RBAC are indeed very powerful. However, Sandhu et al. [11] observed that:

> "RBAC is not a panacea for all access control issues. More sophisticated methods are required to deal with situations that control operation sequences. [...] Other forms of access control can be layered on top of RBAC for this purpose."

Workflow Systems provides an environment where the sequences of operations are controlled according to business rules. The next section introduces workflow concepts, paving the way for the expression of access control policies in terms of sequence of operations.

2.2 Workflow Concepts

Workflow Systems are concerned with the automation and facilitation of business processes [6]. Business processes are defined through process definitions. A process definition consists of sets of tasks, connected according to business rules.

The process definition is enacted by the workflow engine. For each enactment of the business process, e.g. for each 'Internal Purchase Order' that is issued, a process instance is generated. Task instances are generated on demand, based on the business rules encapsulated as part of the process definition.

SoDA is a tool that focuses on supporting access control administration. Access control requirements are, typically, described within the general context of a business process and not for a specific enactment of the workflow. The SoDA prototype is, therefore, only concerned with the process and task definitions.

The "conflicting entities" paradigm relies on restricting the associations between all the entities that are involved, namely user, roles, permissions and tasks.

3 SoDA – The "conflicting entities" paradigm

Separation of duty requirements are implemented by restricting the associations allowed between entities. This is to ensure that a single user may not receive too many permissions. An example of such a constraint may specify that "the permission to approve an order and the permission to issue an order may not be assigned to the same role".

Kuhn [7] explained how mutual exclusive roles, i.e. roles that may not be assigned to the same user, can be used to enforce SoD. Ahn and Sandhu [1] showed through their RSL99 specification language that there are several ways of expressing similar SoD requirements. SoDA builds on these observations, and extends it with the concept of conflicting tasks.

The term "conflicting entities" does not indicate that there are any disharmony between the entities. The "conflict" refer, rather to the disharmony that the entities could cause between the actual and the desired state of the system. Conflict thus indicates a potential undesirable state of integrity. The "conflicting entities" paradigm, as employed in the SoDA prototype, identifies four types of conflict [3]:

Conflicting permissions are permissions that can result in unnecessary power if bestowed on the same person. For example, a person with the permissions required for financial audits should not receive permissions to approve financial transactions. If this were allowed, auditors could lose their independence.

Conflicting users are users who will together have sufficient power to collude, and are likely to do so. In practice, this may be family members or previously known accomplices.

Conflicting roles are roles that together possess the ability to conspire. This means that they are assigned conflicting permissions. Consider, for example, the 'Auditor' and 'Financial Manager' roles. It is common practice that auditors and financial managers should be independent. The roles may have certain permissions, e.g. 'view order', in common. However, the 'approve order' and 'approve audit' permissions may be assigned only to one of these roles.

Conflicting tasks are tasks requiring conflicting permissions to complete. This would, for example, imply that the 'Audit Purchase Order' task and the 'Approve Purchase Order' task would be conflicting since they require the 'approve order' and 'approve audit' permissions. These permissions are, in turn, conflicting.

The "conflicting entities" paradigm is based on the observation that power is vested in permissions. The essence of the "conflicting entities" paradigm lies, therefore, in conflicting permissions. It is argued, however, that tasks provide a more natural abstraction for the specification of SoD requirements. The "conflicting entities" paradigm allows for the specification of both Static and Dynamic SoD requirements.

Static SoD requirements, on the one hand, control the associations between entities during administration time. They would, for example, disallow a user to be assigned to a role if an SoD requirement would be violated. Dynamic SoD, on the other hand, does not restrict associations between entities at administration time. Instead, it controls the execution of permissions at run-time. It would, for example, allow a user to belong to the 'Manager' and 'Clerk' roles. However, during run-time, the user that initiated the purchase order (using the 'Clerk' role) will not be able to approve that purchase order (using the 'Manager' role).

The specification of both Static and Dynamic SoD requirements within the SoDA prototype is similar. This will be discussed in Section 4. Static SoD requirements must, however, also be enforced in the administration environment. The enforcement of Static SoD requirements in the SoDA prototype is thus discussed in Section 5.

4 Separation of duty specification in SoDA

The SoDA prototype allows for the specification of conflicting users, conflicting roles, conflicting permissions and conflicting tasks. A distinction is made between static and dynamic SoD. Conflicts are based on the sets U, R, P and T, representing the user, role, permission and task entities respectively. P is defined as $P \subseteq 2^{O \times M}$, where O represents the objects and M the methods that may be performed. Note that not all the methods may necessarily be defined on all objects. Thus, the set of permissions is a subset of the power set.

The specification of the conflicts is done through the sets:

$$CU_D, CU_S, CR_D, CR_S, CP_D, CP_S, CT_D, CT_S.$$

The same naming convention is followed. CX denotes conflicting entities of type X, and the subscript indicates whether the conflict must be checked statically (CX_S) or dynamically (CX_D). The "conflicting entities" relations are defined in a symmetric and non-reflexive fashion:

$$CX_Y \subseteq X \times X \text{ such that } \forall x_i \neq x_j$$
$$(x_i, x_j) \in CX_Y \iff (x_j, x_i) \in CX_Y$$

The specification for all 8 sets can be derived by replacing X with the appropriate entity (U,R,P or T)

and Y with S or D, for Static and Dynamic respectively.

Figure 3 shows how conflicting tasks are identified within the SoDA prototype. The other conflicts are specified in a similar manner. The interpretation of the various conflicts is summarized in Table 1.

The enforcement of Dynamic SoD requires interpretation of the process instance. Thus it is the responsibility of the workflow system. Consequently, it falls outside the scope of the administrative tool. For a more detailed discussing regarding dynamic SoD the interested reader are refered to [3]. Static SoD must, however, be enforced in the administration environment. The next section discusses how this is implemented in the SoDA prototype.

5 Static Separation of Duty enforcement in SoDA

In order to enforce Static SoD, the SoDA prototype ensures that the integrity of the associations between entities is maintained. If an action cannot be performed, remedial actions are suggested. For example, if conflicting tasks are assigned to non-conflicting roles, the user is given the option of making the roles conflicting. The associations that are allowed are summarized in Table 2 [9].

To illustrate how the SoDA prototype maintains the associations, this section will review different static SoD implementations of the requirement: "A person who issues stock may never approve an order". Three approaches to enforcing this SoD requirement in a static fashion are proposed. This is done by rephrasing the SoD requirement in the following ways:

(SoD1) A manager and a stock controller may not perform the same tasks.

(SoD2) The 'Issue Stock' permission and the 'Approve Order' permission may not be assigned to the same user.

(SoD3) The 'Issue Stock' task may not be performed by someone who performs the 'Approve Order' task.

These SoD constraints will be implemented as conflicting roles, conflicting permissions and conflicting tasks. Conflicting users can be used in combination with these.

Conflicting users are interpreted in the same way as in [AS99]. If two users are conflicting, it means that the chances of them colluding are very high. In essence, they should, therefore, be treated as if they were one user. For example, if two tasks may not be performed by the same user, two conflicting users may not perform them either as the chances of a conspiracy are high. We shall now consider how each of the approaches can, in turn, be handled in the prototype.

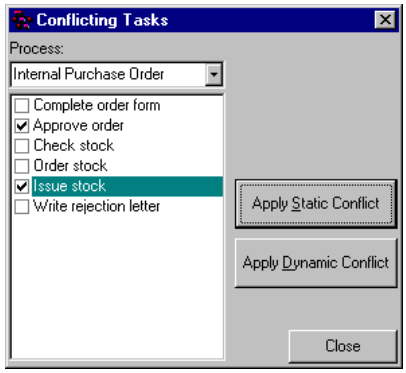

Figure 3: Specifying conflicting tasks

5.1 Conflicting Roles

First consider **(SoD1)** - A manager and a stock controller may not perform the same tasks.

Since managers approve orders, and stock controllers issue stock, the 'Manager' role in the 'Admin' role network and the 'Stock Controller' role in the 'Stores' role network may be set to conflict. Due to the inheritance property of role networks, conflicting roles cannot exist in the same role network. If conflicting roles were allowed in one role network, the topmost role in that role network would inherit the permissions of both conflicting roles. This clearly defeats the purpose. A role may conflict with more than one role in another network. Conflicts are, however, inherited up the partial order and setting more than one conflict, as such, may not be necessary. The SoDA prototype will remove any unnecessary conflict.

In Figure 4, the 'Stores Manager' inherits the conflict set upon 'Stock Controller'. 'Stores Manager' will, therefore, also conflict with the 'Manager' role in the 'Admin' role network. In Figure 3, the 'Approve order' and 'Issue stock' tasks were made conflicting tasks. Conflicting roles and conflicting tasks impact on the allowable associations as follows. Only non-conflicting users may be assigned to conflicting roles. Conflicting tasks must be performed by conflicting roles. Recall that the 'Stock Controller' role and the 'Stores Manager' role were identified as conflicting with the 'Manager' role. Figure 5 depicts the 'Manager' role as being assigned to the 'Approve Order' task. Figure 5 shows, furthermore, that subsequently only the two roles conflicting with the 'Manager' role, namely the 'Stock Controller' and 'Stores Manager' roles, may be assigned to the 'issue stock' task. If two tasks are initially not indicated to be conflicting, but they are assigned to conflicting roles, the tasks are made conflicting tasks.

5.2 Conflicting Permissions

Now consider **(SoD2)** – The 'Issue Stock' permission and the 'Approve Order' permission may not be assigned to the same user.

Conflict	Static	Dynamic
Conflicting Roles	May not have the same user (or conflicting users) as members	May not be assumed by the same user (or conflicting users) in one process instance
Conflicting Permissions	Must be assigned to conflicting roles	May not be exercised by the same user (or conflicting users) for a specific process instance
Conflicting Users	May not belong to the same role or conflicting roles	May not perform conflicting tasks in the same process instance
Conflicting Tasks	Must be assigned to conflicting roles	May not be executed by the same user (or conflicting users) in the same process instance

Table 1: Interpretation of conflicts according to the "conflicting entities" paradigm

May be associated with		Roles	
		Conflicting	Non-conflicting
Users	Conflicting	N	Y
	Non-conflicting	Y	Y
Permissions	Conflicting	Y	N
	Non-conflicting	Y	Y
Tasks	Conflicting	Y	N
	Non-conflicting	Y	Y

Table 2: Static SoD – Allowable associations

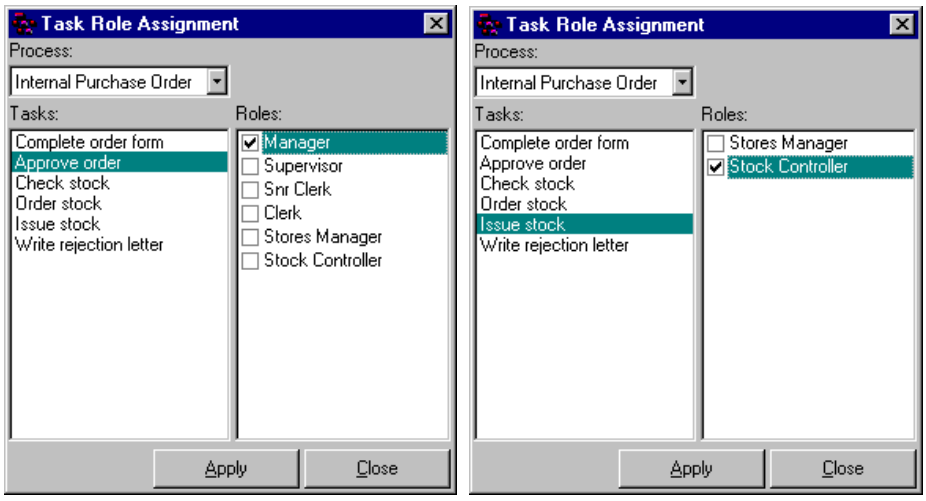

Figure 5: Conflicting tasks must be assigned to conflicting roles

Figure 4: Conflicting roles

The permissions involved are editing the 'Approval' and 'IssueRec' field groups on the 'Internal Order Form' object. Conflicting permissions may only be assigned to conflicting roles. If this is not enforced, conflicting permissions could be assigned to conflicting users. These conflicting users belong to non-conflicting roles, which have conflicting permissions that were incorrectly assigned to the non-conflicting roles. This clearly opens the door for a conspiracy. The SoDA prototype, therefore, only allows conflicting roles to receive conflicting permissions.

If the roles are not conflicting, they are made conflicting, subject to additional integrity checking. Roles cannot be made conflicting if conflicting users are assigned to the said roles. It can, therefore, be seen that even if the 'Manager' and 'Stock Controller' roles were not initially identified to be conflicting, they will be made conflicting when the two conflicting permissions are assigned to these two roles. Similar to section 4, the tasks assigned to these two roles will also be made conflicting.

5.3 Conflicting Tasks

Consider **(SoD2c)** – The 'Issue stock' task may not be performed by someone who may perform the 'Approve order' task. In section 5.1, it was shown how conflicting roles could only be assigned to conflicting tasks. If conflicting roles were assigned to tasks, these tasks were automatically made conflicting. This approach can be considered to be the reverse of that. Two tasks are defined to be conflicting. Subsequently, the roles that must be assigned to the user must be conflicting. If two non-conflicting roles are assigned, the roles are made conflicting, subject to a series of integrity checks being performed. It is evident that the same result is achieved, irrespective of the approach used, since automatic maintenance of conflict relationships is performed.

The results of the conflicting role and conflicting task approaches are thus identical. The conflicting permission approach can, however, be considered stricter. Conflicting permissions must be performed by conflicting roles. However, conflicting roles do not only have conflicting permissions. For example, the 'Manager' and 'Stock Controller' roles are conflicting, but both should still be allowed the 'view purchase order' permission. The conflicting permissions 'Edit Approval' and 'Edit Issuerec' may, however, also be assigned to the 'Manager' and 'Stock Controller' roles respectively.

6 Conclusion

This paper demonstrated the "conflicting entities" paradigm as a way of specifying SoD requirements. This paradigm uses the task abstraction to intuitively define separation of duty requirements that involve sequence of operations. It was shown that both Static and Dynamic SoD requirements can be formulated according to the "conflicting entities" paradigm in the SoDA prototype.

It was, furthermore, shown that the SoDA prototype enforces Static SoD requirements. By specifying one SoD requirement in three different ways, it was explained that equivalent results can be achieved.

It should be noted that Static SoD requirements are extremely restrictive on the organizations functioning. Consider, for example **(SoD1)**. To assume that a managers and a stock controller could never do the same job could be, especially for a small company, very restrictive. Dynamic SoD requirements addresses this issue by imposing the restrictions per process instance.

Other issues that could be of concern are the potential of a lock-out situation. A situation could arise that, for example no roles are available to assign to a task. This would immediately be noticable to the system administrator and he/she will have to rectify the situation manually. However, due the extremely strict restrictions imposed by static separation of duty, it is likely to be used sparingly. This makes the likelihood of a lock-out occurring extremely small and thus feasible for the adminstrator to manually correct. The issue of lock-out occuring due to dynamic SoD requirements are much more complex and state-of-the-art work regrading that may be found in [2].

References

[1] G-J. Ahn and R. S. Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the 4th ACM Workshop on Role-based Access Control*, pages 43 – 54, 28 – 29 Oct. 1999.

[2] E. Bertino, E. Ferrari, and V. Atluri. Specification and enforcement of authorization constraints in workflow management systems. *ACM*

Transactions on Information and System Security, 2(1):65–104, Feb 1999.

[3] R. A. Botha and J. H. P. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Systems Journal*, 40(3), 2001.

[4] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184 – 194, Apr. 1987.

[5] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transaction on Information and System Security*, 2(1):34 – 64, Feb. 1999.

[6] D. Hollingsworth. The workflow reference model. Technical Report TC-00-1003, Workflow Management Coalition, www.wfmc.org, Jan 1995.

[7] D. R. Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of the 2nd ACM Workshop on Role-based Access Control*, pages 23 – 30, Oct. 1997.

[8] M. Nyanchama and S. Osborn. The role-graph model and conflict of interest. *ACM Transactions on Information and System Security*, 2(1):3 – 33, Feb. 1999.

[9] S. Perelson and R. A. Botha. Conflict analysis as a means of enforcing static separation of duty requirements in workflow environments. *South African Computer Journal*, (26):212 – 216, Nov. 2000.

[10] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of IEEE*, 63(9):1278 – 1308, 1975.

[11] R. S. Sandhu, E. J. Coyne, H. L. Fenstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38 – 47, Feb 1996.

Received: 10/00, Accepted: 5/01

Appendix C. SoDA Prototype Scripts

These scripts create the second prototype described in chapter 8. This prototype demonstrates the feasibility of the SoDA model.

The scripts in these listings are used to create the database tables and to create the triggers for the Oracle DBMS as described in chapter 8. The trigger creation script begins on page 123 while the table creation script begins on page 136.

```
-------------------------------------------------------------------------------
-- SoDA integrity constraint rules
-- For Oracle DBMS
-- By Stephen Perelson
-- Copyright © 2001, Stephen Perelson and the Secure Workflow Research Group
-------------------------------------------------------------------------------

-- When inserting
--   *associations into the ua, pa and ta tables
--   *conflicts into the cr, cu, cp, and ct tables
--   *roles into a role network (rh table)
-- and when deleting
--   *role conflicts from the cr table
--   *role associations from the rh table
-- it becomes necessary to restrict them from occurring based upon
-- the SoDA integrity constraints.
-- The logic that is followed within these triggers can be found in
-- chapter 7.

-- First create the package with the variables for the triggers. This
-- is to prevent mutating table errors. This package contains all the
-- variables for every SoDA trigger.

CREATE OR REPLACE PACKAGE wf_mutvars_pkg
IS
   -- variables for the ua table (user to role association)
   TYPE arrua IS TABLE OF ua%ROWTYPE
      INDEX BY BINARY_INTEGER;
   uavalues       arrua;
   uaempty        arrua;

   -- variables for the pa table (permission to role association)
   TYPE arrpa IS TABLE OF pa%ROWTYPE
      INDEX BY BINARY_INTEGER;
   pavalues       arrpa;
   paempty        arrpa;

   -- variables for the ta table (task to role association)
   TYPE arrta IS TABLE OF ta%ROWTYPE
      INDEX BY BINARY_INTEGER;
   tavalues       arrta;
   taempty        arrta;

   -- variables for the cr table (role conflict)
   TYPE arrcr IS TABLE OF cr%ROWTYPE
      INDEX BY BINARY_INTEGER;
   crvalues       arrcr;
   crempty        arrcr;

   -- variables for the cu table (user conflict)
   TYPE arrcu IS TABLE OF cu%ROWTYPE
      INDEX BY BINARY_INTEGER;
   cuvalues       arrcu;
   cuempty        arrcu;

   -- variables for the cp table (permission conflict)
   TYPE arrcp IS TABLE OF cp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   cpvalues       arrcp;
   cpempty        arrcp;

   -- variables for the ct table (task conflict)
   TYPE arrct IS TABLE OF ct%ROWTYPE
      INDEX BY BINARY_INTEGER;
   ctvalues       arrct;
   ctempty        arrct;

   -- variables for the cr table (delete role conflict)
   TYPE arrdcr IS TABLE OF cr%ROWTYPE
      INDEX BY BINARY_INTEGER;
   dcrvalues       arrdcr;
   dcrempty        arrdcr;

   -- variables for the rh table (role network)
   TYPE arrrh IS TABLE OF rh%ROWTYPE
      INDEX BY BINARY_INTEGER;
```

```
   rhvalues       arrrh;
   rhempty        arrrh;

   -- variables for the rh table (delete role network)
   TYPE arrdrh IS TABLE OF rh%ROWTYPE
      INDEX BY BINARY_INTEGER;
   drhvalues       arrdrh;
   drhempty        arrdrh;
END;
/

--------------------------------------------------------------------------------
--
-- Triggers for the constraint of the user to role associations.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER uastatbef_trig
BEFORE INSERT ON ua
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.uavalues := wf_mutvars_pkg.uaempty;
END;
/

CREATE OR REPLACE TRIGGER uarowbef_trig
BEFORE INSERT ON ua
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.uavalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.uavalues (i).roleid := :new.roleid;
  wf_mutvars_pkg.uavalues (i).userid := :new.userid;
END;
/

CREATE OR REPLACE TRIGGER uastatafter_trig
AFTER INSERT ON ua
DECLARE
  -- Cursor to select all conflicting users for a particular user.
  CURSOR conusercur (v_userid NUMBER) IS
    SELECT userid FROM users
    WHERE userid IN
        (SELECT userid2 FROM cu WHERE userid1 = v_userid
         UNION
         SELECT userid1 FROM cu WHERE userid2 = v_userid);
  -- Get the conflicting roles for the roles associated with the
  -- conflicting users.
  CURSOR assrolecur (v_userid NUMBER, v_roleid NUMBER) IS
    SELECT roleid FROM ua, cr WHERE ua.userid = v_userid
    AND ((roleid = cr.roleid1) OR (roleid = cr.roleid2))
    AND ((cr.roleid1 = v_roleid) OR (cr.roleid2 = v_roleid));
  TYPE arrcu IS TABLE OF users.userid%TYPE
    INDEX BY BINARY_INTEGER;
  conuser arrcu;
  TYPE arrassroles IS TABLE OF ua.roleid%TYPE
    INDEX BY BINARY_INTEGER;
  assroles arrassroles;
  vuserid ua.userid%TYPE;
  vroleid ua.roleid%TYPE;
  v_count NUMBER(38);
  v_stop BOOLEAN := false;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.uavalues.COUNT
  LOOP
    -- Do the checks required here for each record inserted.
    vuserid := wf_mutvars_pkg.uavalues(i).userid;
    vroleid := wf_mutvars_pkg.uavalues(i).roleid;
    FOR conuserrec IN conusercur(vuserid) LOOP
      v_count := conuser.COUNT + 1;
      conuser (v_count) := conuserrec.userid;
    END LOOP;
    -- Move to the next step: check for conflicting roles.
    IF conuser.COUNT > 0 then
      FOR j IN 1..conuser.COUNT LOOP
        FOR assrolerec IN assrolecur(conuser(j), vroleid) LOOP
```

```
          -- This could have been done with a simple SELECT (count 1)
          -- SQL statement with an IF statement to check it, but this works.
          v_stop := true;
        END LOOP;
      END LOOP;
      -- Move to the next step.
      if v_stop = true then -- assroles.COUNT > 0 then
        -- There are conflicting roles already assigned to conflicting users.
        -- Therefore raise an error.
        raise_application_error(-20201,'Cannot insert user to role ' ||
                                       'association - a conflicting user ' ||
                                       'is already associated to a ' ||
                                       'conflicting role.');
      END IF;
    END IF;
    -- Insert the record (or in this case don't do anything because it is
    -- already inserted).
  END LOOP;
END;
/

--------------------------------------------------------------------------------
--
-- Triggers for the constraint of the permission to role associations.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER pastatbef_trig
BEFORE INSERT ON pa
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.pavalues := wf_mutvars_pkg.paempty;
END;
/

CREATE OR REPLACE TRIGGER parowbef_trig
BEFORE INSERT ON pa
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.pavalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.pavalues (i).roleid := :new.roleid;
  wf_mutvars_pkg.pavalues (i).permid := :new.permid;
END;
/

CREATE OR REPLACE TRIGGER pastatafter_trig
AFTER INSERT ON pa
DECLARE
  vpermid pa.permid%TYPE;
  vroleid pa.roleid%TYPE;
  v_cannot NUMBER(1) := 0;
  v_can NUMBER(38) := 0;
  v_none NUMBER(1) := 0;
  v_tally NUMBER(38) := 0;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.pavalues.COUNT
  LOOP
    -- Do the checks required here for each record inserted.
    v_tally := wf_mutvars_pkg.pavalues.COUNT;
    SELECT count(*) - v_tally INTO v_none FROM pa;
    IF (v_none <> 0) THEN
      vpermid := wf_mutvars_pkg.pavalues(i).permid;
      vroleid := wf_mutvars_pkg.pavalues(i).roleid;
      -- Count the conflicting permissions.
      SELECT count(*) INTO v_can
      FROM perms
      WHERE permid IN (SELECT permid2 FROM cp WHERE permid1 = vpermid
                       UNION
                       SELECT permid1 FROM cp WHERE permid2 = vpermid);
      IF (v_can <> 0) THEN
        -- Select all the roles associated to the conflicting permissions.
        -- If they are not all conflicting with the inserted role then it
        -- should raise an error.
        SELECT count(1) INTO v_cannot
        FROM DUAL
```

```
        WHERE NOT EXISTS
          (SELECT distinct(roleid) FROM roles
           WHERE roleid in
             (SELECT distinct(roleid) FROM pa
              WHERE permid in (SELECT permid2 FROM cp WHERE permid1 = vpermid
                               UNION
                               SELECT permid1 FROM cp WHERE permid2 = vpermid))
              MINUS
              (SELECT roleid2 FROM cr WHERE roleid1 = vroleid
               UNION
               SELECT roleid1 FROM cr WHERE roleid2 = vroleid))
        AND EXISTS
          (SELECT distinct(roleid) FROM pa
           WHERE permid in (SELECT permid2 FROM cp WHERE permid1 = vpermid
                            UNION
                            SELECT permid1 FROM cp WHERE permid2 = vpermid));
        IF v_cannot = 0 then
          -- There are conflicting roles already assigned to conflicting
          -- permissions. Therefore raise an error.
          raise_application_error(-20202, 'Cannot insert permission to ' ||
                                          'role association - a conflicting ' ||
                                          'permission must be assigned to a ' ||
                                          'conflicting role.');
        END IF;
      END IF;
    END IF;
    -- Insert the record (or in this case don't do anything because it is
    -- already inserted).
  END LOOP;
END;
/

--------------------------------------------------------------------------------
--
-- Triggers for the constraint of the task to role associations.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER tastatbef_trig
BEFORE INSERT ON ta
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.tavalues := wf_mutvars_pkg.taempty;
END;
/

CREATE OR REPLACE TRIGGER tarowbef_trig
BEFORE INSERT ON ta
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.tavalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.tavalues (i).roleid := :new.roleid;
  wf_mutvars_pkg.tavalues (i).taskid := :new.taskid;
END;
/

CREATE OR REPLACE TRIGGER tastatafter_trig
AFTER INSERT ON ta
DECLARE
  vtaskid ta.taskid%TYPE;
  vroleid ta.roleid%TYPE;
  v_cannot NUMBER(1) := 0;
  v_can NUMBER(38) := 0;
  v_none NUMBER(38) := 0;
  v_tally NUMBER(38) := 0;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.tavalues.COUNT
  LOOP
    -- Do the checks required here for each record inserted.
    v_tally := wf_mutvars_pkg.tavalues.COUNT;
    SELECT count(*) - v_tally INTO v_none FROM ta;
    IF (v_none <> 0) THEN
    BEGIN
      vtaskid := wf_mutvars_pkg.tavalues(i).taskid;
      vroleid := wf_mutvars_pkg.tavalues(i).roleid;
```

```
      -- Count the conflicting tasks.
      SELECT count(*) INTO v_can
      FROM tasks
      WHERE taskid IN (SELECT taskid2 FROM ct WHERE taskid1 = vtaskid
                       UNION
                       SELECT taskid1 FROM ct WHERE taskid2 = vtaskid);
      IF (v_can <> 0) THEN
        -- Select all the roles associated to the conflicting tasks. If
        -- they are not all conflicting with the inserted role then it
        -- should raise an error.
        SELECT count(1) INTO v_cannot
        FROM DUAL
        WHERE NOT EXISTS
          (SELECT distinct(roleid) FROM roles
           WHERE roleid in
             (SELECT distinct(roleid) FROM ta
              WHERE taskid in (SELECT taskid2 FROM ct WHERE taskid1 = vtaskid
                               UNION
                               SELECT taskid1 FROM ct WHERE taskid2 = vtaskid))
              MINUS
              (SELECT roleid2 FROM cr WHERE roleid1 = vroleid
               UNION
               SELECT roleid1 FROM cr WHERE roleid2 = vroleid))
        AND EXISTS
          (SELECT distinct(roleid) FROM ta
           WHERE taskid in (SELECT taskid2 FROM ct WHERE taskid1 = vtaskid
                            UNION
                            SELECT taskid1 FROM ct WHERE taskid2 = vtaskid));
        IF v_cannot = 0 then -- assroles.COUNT > 0 then
          -- There are conflicting roles already assigned to conflicting users.
          -- Therefore raise an error.
          raise_application_error(-20203, 'Cannot insert task to role ' ||
                                          'association - a conflicting ' ||
                                          'task must be assigned to a ' ||
                                          'conflicting role.');
        END IF;
      END IF;
    END;
    END IF;
    -- Insert the record (or in this case don't do anything because it is
    -- already inserted).
  END LOOP;
END;
/

--------------------------------------------------------------------------------
--
-- Triggers for the constraint of the role conflict assignment.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER crstatbef_trig
BEFORE INSERT ON cr
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.crvalues := wf_mutvars_pkg.crempty;
END;
/

CREATE OR REPLACE TRIGGER crrowbef_trig
BEFORE INSERT ON cr
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.crvalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.crvalues (i).roleid1 := :new.roleid1;
  wf_mutvars_pkg.crvalues (i).roleid2 := :new.roleid2;
END;
/

CREATE OR REPLACE TRIGGER crstatafter_trig
AFTER INSERT ON cr
DECLARE
  vroleid1 cr.roleid1%TYPE;
  vroleid2 cr.roleid2%TYPE;
  v_exists NUMBER(38) := 0;
```

```
  v_asso NUMBER(38) := 0;
  v_conf NUMBER(38) := 0;
  v_rolenet NUMBER(38) := 0;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.crvalues.COUNT
  LOOP
    vroleid1 := wf_mutvars_pkg.crvalues (i).roleid1;
    vroleid2 := wf_mutvars_pkg.crvalues (i).roleid2;
    -- Check whether the roles are the same.
    IF vroleid1 = vroleid2 THEN
      raise_application_error(-20218,'Cannot insert role conflict - ' ||
                                     'both roles are identical.');
    END IF;
    -- Do they form part of a role network?
    SELECT COUNT(1) INTO v_rolenet FROM DUAL
    WHERE 2 <= ANY (SELECT COUNT(1) FROM rh
                    WHERE childid = vroleid1 OR childid = vroleid2
                    GROUP BY rolenetid);

    IF v_rolenet > 0 THEN -- Yes they do so raise an error
      raise_application_error(-20223,'Cannot insert role conflict - ' ||
                                     'both roles are already part of a ' ||
                                     'role network.');
    END IF;
    -- Are they already conflicting? We don't worry about checking whether
    -- id1 and id2 are in their respective places as the primary key
    -- constraints will prevent that from occuring.
    -- This also prevents the same role from conflicting with itself.
    SELECT count(1) INTO v_exists FROM cr
    WHERE roleid1 = vroleid2 AND roleid2 = vroleid1;
    IF (v_exists = 0) THEN
      -- Doesn't exist yet so we can continue checking.
      -- Check for user-role associations.
      SELECT count(1) INTO v_asso FROM ua
      WHERE roleid = vroleid1 OR roleid = vroleid2;
      IF (v_asso <> 0) THEN
        -- Found some associations so go to step 3.
        SELECT count(1) INTO v_conf FROM DUAL
        WHERE EXISTS ((SELECT userid FROM users
                       WHERE userid IN (SELECT userid2 FROM cu
                                        WHERE userid1 IN (SELECT userid FROM ua
                                                          WHERE roleid=vroleid1)
                                        UNION
                                        SELECT userid1 FROM cu
                                        WHERE userid2 IN (SELECT userid FROM ua
                                                          WHERE roleid=vroleid1)))
                       INTERSECT
                      (SELECT userid FROM ua
                       WHERE roleid = vroleid2));
        IF (v_conf <> 0) THEN
          raise_application_error(-20205,'Cannot insert role conflict - ' ||
                                         'both roles are already associated ' ||
                                         'to conflicting users.');
        END IF;
      END IF;
    ELSE
      raise_application_error(-20204,'Cannot insert role conflict - ' ||
                                     'the role conflict already exists.');
    END IF;
  END LOOP;
END;
/
```

```
--------------------------------------------------------------------------------
--
-- Triggers for the constraint of the user conflict assignment.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER custatbef_trig
BEFORE INSERT ON cu
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.cuvalues := wf_mutvars_pkg.cuempty;
END;
/

CREATE OR REPLACE TRIGGER curowbef_trig
BEFORE INSERT ON cu
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.cuvalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.cuvalues (i).userid1 := :new.userid1;
  wf_mutvars_pkg.cuvalues (i).userid2 := :new.userid2;
END;
/

CREATE OR REPLACE TRIGGER custatafter_trig
AFTER INSERT ON cu
DECLARE
  vuserid1 cu.userid1%TYPE;
  vuserid2 cu.userid2%TYPE;
  v_exists NUMBER(38) := 0;
  v_asso NUMBER(38) := 0;
  v_conf NUMBER(38) := 0;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.cuvalues.COUNT
  LOOP
    vuserid1 := wf_mutvars_pkg.cuvalues (i).userid1;
    vuserid2 := wf_mutvars_pkg.cuvalues (i).userid2;
    -- Check whether the users are the same.
    IF vuserid1 = vuserid2 THEN
      raise_application_error(-20219,'Cannot insert user conflict - ' ||
                                     'both users are identical.');
    END IF;
    -- Are they already conflicting? We don't worry about checking whether
    -- id1 and id2 are in their respective places as the primary key
    -- constraints will prevent that from occuring.
    -- This also prevents the same user from conflicting with itself.
    SELECT count(1) INTO v_exists FROM cu
    WHERE userid1 = vuserid2 AND userid2 = vuserid1;
    IF (v_exists = 0) THEN
      -- Doesn't exist yet so we can continue checking.
      -- Check for user-role associations.
      SELECT count(1) INTO v_asso FROM ua
      WHERE userid = vuserid1 OR userid = vuserid2;
      IF (v_asso <> 0) THEN
        -- Found some associations so go to step 3.
        SELECT count(1) INTO v_conf FROM DUAL
        WHERE EXISTS ((SELECT roleid FROM roles
                       WHERE roleid IN (SELECT roleid2 FROM cr
                                        WHERE roleid1 IN (SELECT roleid FROM ua
                                                          WHERE userid=vuserid1)
                                        UNION
                                        SELECT roleid1 FROM cr
                                        WHERE roleid2 IN (SELECT roleid FROM ua
                                                          WHERE userid=vuserid1)))
                       INTERSECT
                      (SELECT roleid FROM ua
                       WHERE userid = vuserid2));
        IF (v_conf <> 0) THEN
          raise_application_error(-20207,'Cannot insert user conflict - ' ||
                                         'both users are already associated ' ||
                                         'to conflicting roles.');
        END IF;
      END IF;
    ELSE
      raise_application_error(-20206,'Cannot insert user conflict - ' ||
```

```
                                    'the user conflict already exists.');
    END IF;
  END LOOP;
END;
/

--------------------------------------------------------------------------------
--
-- Triggers for the constraint of the permission conflict assignment.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER cpstatbef_trig
BEFORE INSERT ON cp
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.cpvalues := wf_mutvars_pkg.cpempty;
END;
/

CREATE OR REPLACE TRIGGER cprowbef_trig
BEFORE INSERT ON cp
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.cpvalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.cpvalues (i).permid1 := :new.permid1;
  wf_mutvars_pkg.cpvalues (i).permid2 := :new.permid2;
END;
/

CREATE OR REPLACE TRIGGER cpstatafter_trig
AFTER INSERT ON cp
DECLARE
  vpermid1 cp.permid1%TYPE;
  vpermid2 cp.permid2%TYPE;
  v_exists NUMBER(38) := 0;
  v_asso NUMBER(38) := 0;
  v_conf NUMBER(38) := 0;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.cpvalues.COUNT
  LOOP
    -- We need to check for an empty table?
    vpermid1 := wf_mutvars_pkg.cpvalues (i).permid1;
    vpermid2 := wf_mutvars_pkg.cpvalues (i).permid2;
    -- Check whether the permissions are the same.
    IF vpermid1 = vpermid2 THEN
      raise_application_error(-20220,'Cannot insert permission conflict - ' ||
                                    'both permissions are identical.');
    END IF;
    -- Are they already conflicting? We don't worry about checking whether
    -- id1 and id2 are in their respective places as the primary key
    -- constraints will prevent that from occuring.
    -- This also prevents the same permission from conflicting with itself.
    SELECT count(1) INTO v_exists FROM cp
    WHERE permid1 = vpermid2 AND permid2 = vpermid1;
    IF (v_exists = 0) THEN
      -- Doesn't exist yet so we can continue checking.
      -- Check for permission-role associations for both permissions only.
      -- If one of the permissions has no association yet then allow add.
      SELECT COUNT(1) INTO v_asso FROM pa p1, pa p2
      WHERE p1.permid = vpermid1 AND p2.permid = vpermid2;
      IF (v_asso <> 0) THEN
        -- Found some associations so go to step 3.
        SELECT count(1) INTO v_conf FROM DUAL
        WHERE EXISTS ((SELECT roleid FROM pa
                       WHERE permid = vpermid2)
                       MINUS
                       (SELECT roleid FROM roles
                        WHERE roleid IN (SELECT roleid2 FROM cr
                                         WHERE roleid1 IN
                                         (SELECT roleid FROM pa
                                          WHERE permid=vpermid1)
                                         UNION
                                         SELECT roleid1 FROM cr
                                         WHERE roleid2 IN
```

```
                                          (SELECT roleid FROM pa
                                           WHERE permid=vpermid1))));
        IF (v_conf <> 0) THEN
          raise_application_error(-20209,'Cannot insert permission ' ||
                                         'conflict - one or more of the ' ||
                                         'associated roles of the two ' ||
                                         'permissions are not conflicting.');
        END IF;
      END IF;
    ELSE
      raise_application_error(-20208,'Cannot insert permission conflict ' ||
                                     '- permission conflict already exists.');
    END IF;
  END LOOP;
END;
/

--------------------------------------------------------------------------------
--
-- Triggers for the constraint of the task conflict assignment.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER ctstatbef_trig
BEFORE INSERT ON ct
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.ctvalues := wf_mutvars_pkg.ctempty;
END;
/

CREATE OR REPLACE TRIGGER ctrowbef_trig
BEFORE INSERT ON ct
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.ctvalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.ctvalues (i).taskid1 := :new.taskid1;
  wf_mutvars_pkg.ctvalues (i).taskid2 := :new.taskid2;
END;
/

CREATE OR REPLACE TRIGGER ctstatafter_trig
AFTER INSERT ON ct
DECLARE
  vtaskid1 ct.taskid1%TYPE;
  vtaskid2 ct.taskid2%TYPE;
  v_exists NUMBER(38) := 0;
  v_asso NUMBER(38) := 0;
  v_conf NUMBER(38) := 0;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.ctvalues.COUNT
  LOOP
    -- We need to check for an empty table?
    vtaskid1 := wf_mutvars_pkg.ctvalues (i).taskid1;
    vtaskid2 := wf_mutvars_pkg.ctvalues (i).taskid2;
    -- Check whether the tasks are the same.
    IF vtaskid1 = vtaskid2 THEN
      raise_application_error(-20221,'Cannot insert task conflict - ' ||
                                     'both tasks are identical.');
    END IF;
    -- Are they already conflicting? We don't worry about checking whether
    -- id1 and id2 are in their respective places as the primary key
    -- constraints will prevent that from occuring.
    -- This also prevents the same task from conflicting with itself.
    SELECT count(1) INTO v_exists FROM ct
    WHERE taskid1 = vtaskid2 AND taskid2 = vtaskid1;
    IF (v_exists = 0) THEN
      -- Doesn't exist yet so we can continue checking.
      -- Check for permission-role associations for both tasks only.
      -- If one of the tasks has no association yet then allow add.
      SELECT COUNT(1) INTO v_asso FROM ta t1, ta t2
      WHERE t1.taskid = vtaskid1 AND t2.taskid = vtaskid2;
      IF (v_asso <> 0) THEN
        -- Found some associations so go to step 3.
        SELECT count(1) INTO v_conf FROM DUAL
```

```
        WHERE EXISTS ((SELECT roleid FROM ta
                           WHERE taskid = vtaskid2)
                           MINUS
                           (SELECT roleid FROM roles
                            WHERE roleid IN (SELECT roleid2 FROM cr
                                                WHERE roleid1 IN
                                                (SELECT roleid FROM ta
                                                 WHERE taskid=vtaskid1)
                                                UNION
                                                SELECT roleid1 FROM cr
                                                WHERE roleid2 IN
                                                (SELECT roleid FROM ta
                                                 WHERE taskid=vtaskid1))));
        IF (v_conf <> 0) THEN
          raise_application_error(-20211,'Cannot insert task conflict - ' ||
                                         'one or more of the associated ' ||
                                         'roles of the two tasks are not ' ||
                                         'conflicting.');
        END IF;
      END IF;
    ELSE
      raise_application_error(-20210,'Cannot insert task conflict - ' ||
                                     'task conflict already exists.');
    END IF;
  END LOOP;
END;
/

--------------------------------------------------------------------------------
--
-- Triggers for ensuring integrity when deleting role conflict assignments.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER dcrstatbef_trig
BEFORE DELETE ON cr
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.dcrvalues := wf_mutvars_pkg.dcrempty;
END;
/

CREATE OR REPLACE TRIGGER dcrrowbef_trig
BEFORE DELETE ON cr
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.dcrvalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.dcrvalues (i).roleid1 := :old.roleid1;
  wf_mutvars_pkg.dcrvalues (i).roleid2 := :old.roleid2;
END;
/

CREATE OR REPLACE TRIGGER dcrstatafter_trig
AFTER DELETE ON cr
DECLARE
  vroleid1 cr.roleid1%TYPE;
  vroleid2 cr.roleid2%TYPE;
  v_stop BOOLEAN := false;
  v_exists NUMBER(38) := 0;
  v_passo1 NUMBER(38) := 0;
  v_passo2 NUMBER(38) := 0;
  v_tasso1 NUMBER(38) := 0;
  v_tasso2 NUMBER(38) := 0;
  v_conf NUMBER(38) := 0;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.dcrvalues.COUNT
  LOOP
    vroleid1 := wf_mutvars_pkg.dcrvalues (i).roleid1;
    vroleid2 := wf_mutvars_pkg.dcrvalues (i).roleid2;
    -- Doesn't exist yet so we can continue checking.
    -- Check for user-role associations.
    SELECT count(1) INTO v_passo1 FROM pa
    WHERE roleid = vroleid1;
    SELECT count(1) INTO v_passo2 FROM pa
    WHERE roleid = vroleid2;
```

```
    SELECT count(1) INTO v_tasso1 FROM ta
    WHERE roleid = vroleid1;
    SELECT count(1) INTO v_tasso2 FROM ta
    WHERE roleid = vroleid2;
    IF ((v_passo1 <> 0) AND (v_passo2 <> 0))
       OR ((v_tasso1 <> 0) AND (v_tasso2 <> 0)) THEN
      -- Found some associations so go to step 3.
      IF ((v_passo1 <> 0) AND (v_passo2 <> 0)) THEN
      -- Check the permissions first.
        SELECT count(1) INTO v_conf FROM DUAL
        WHERE EXISTS ((SELECT permid FROM perms
                       WHERE permid IN (SELECT permid2 FROM cp
                                        WHERE permid1 IN (SELECT permid FROM pa
                                                          WHERE roleid=vroleid1)
                                        UNION
                                        SELECT permid1 FROM cp
                                        WHERE permid2 IN (SELECT permid FROM pa
                                                          WHERE roleid=vroleid1)))
                       INTERSECT
                       (SELECT permid FROM pa
                        WHERE roleid = vroleid2));
        IF (v_conf <> 0) THEN
          v_stop := true;
        END IF;
      END IF;
      v_conf := 0;
      IF ((v_tasso1 <> 0) AND (v_tasso2 <> 0)) THEN -- check the tasks next.
        SELECT count(1) INTO v_conf FROM DUAL
        WHERE EXISTS ((SELECT taskid FROM tasks
                       WHERE taskid IN (SELECT taskid2 FROM ct
                                        WHERE taskid1 IN (SELECT taskid FROM ta
                                                          WHERE roleid=vroleid1)
                                        UNION
                                        SELECT taskid1 FROM ct
                                        WHERE taskid2 IN (SELECT taskid FROM ta
                                                          WHERE roleid=vroleid1)))
                       INTERSECT
                       (SELECT taskid FROM ta
                        WHERE roleid = vroleid2));
        IF (v_conf <> 0) THEN
          v_stop := true;
        END IF;
      END IF;
      IF (v_stop = true) THEN
        raise_application_error(-20212,'Cannot delete role conflict - ' ||
                                       'the roles form part of associations ' ||
                                       'to tasks or permissions that will ' ||
                                       'become invalid if the role conflict ' ||
                                       'is deleted.');
      END IF;
    END IF;
  END LOOP;
END;
/

--------------------------------------------------------------------------------
--
-- Triggers for the constraint of the role network association.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER rhstatbef_trig
BEFORE INSERT ON rh
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.rhvalues := wf_mutvars_pkg.rhempty;
END;
/

CREATE OR REPLACE TRIGGER rhrowbef_trig
BEFORE INSERT ON rh
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.rhvalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.rhvalues (i).rolenetid := :new.rolenetid;
```

```
  wf_mutvars_pkg.rhvalues (i).parentid := :new.parentid;
  wf_mutvars_pkg.rhvalues (i).childid := :new.childid;
END;
/

CREATE OR REPLACE TRIGGER rhstatafter_trig
AFTER INSERT ON rh
DECLARE
  e_rolenet_loop EXCEPTION;
  PRAGMA EXCEPTION_INIT (
    e_rolenet_loop, -1436);
  vrolenetid rh.rolenetid%TYPE;
  vparentid rh.parentid%TYPE;
  vchildid rh.childid%TYPE;
  v_root1 NUMBER(38) := 0;
  v_root2 NUMBER(38) := 0;
  v_exists NUMBER(38) := 0;
  v_circ NUMBER(38) := 0;
  v_conf NUMBER(38) := 0;
  v_count NUMBER(38) := wf_mutvars_pkg.rhvalues.COUNT;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.rhvalues.COUNT
  LOOP
    vrolenetid := wf_mutvars_pkg.rhvalues (i).rolenetid;
    vparentid := wf_mutvars_pkg.rhvalues (i).parentid;
    vchildid := wf_mutvars_pkg.rhvalues (i).childid;
    -- If there is no root record for the current rolenet
    -- (count - current records) then we cannot add the record
    -- if the parentid is not null.
    IF (vparentid IS NOT NULL) THEN
      SELECT count(1) INTO v_root1 FROM rh
      WHERE rolenetid = vrolenetid AND parentid IS NULL;
      IF (v_root1 = 0) THEN
        raise_application_error(-20216,'Cannot insert role into role ' ||
                                       'network - root role is missing ' ||
                                       'for the particular role network');
      END IF;
    END IF;
    -- Check for an existing root role. If one is found for the particular
    -- role network then raise an error.
    IF (vparentid IS NULL) THEN
      -- Must subtract the count because the record is already in the table.
      -- This of course assumes that there will only ever be one record
      -- inserted at a time. Disable all triggers and constraints when doing
      -- batch transfers.
      SELECT count(1) - v_count INTO v_root2 FROM rh
      WHERE rolenetid = vrolenetid AND parentid IS NULL;
      IF (v_root2 > 0) THEN
        raise_application_error(-20217,'Cannot insert role into role ' ||
                                       'network - root role already ' ||
                                       'exists for the particular role network.');
      END IF;
    END IF;
    -- Is parent role in the role network as a child?
    -- If not then raise an error.
    SELECT count(1) INTO v_exists FROM rh
    WHERE rolenetid = rolenetid AND childid = vparentid;
    -- Added extra checks to make the error message that
    -- occurs better because the record is already in the
    -- table. This screws up this check in cases when the
    -- parentid = childid. Also check if it is root level.
    IF ((v_exists <> 0) AND (vparentid <> vchildid)) OR (vparentid IS NULL) THEN
      -- Will it cause a circular reference? Rely on an
      -- exception to be raised when the sql statement
      -- fails. If it doesn't fail then it will work just
      -- fine. This works because the record is already in
      -- the table when this trigger fires.
      SELECT count(childid) INTO v_circ
      FROM rh
      WHERE childid <> vparentid
      START WITH childid = vparentid AND rolenetid = vrolenetid
      CONNECT BY PRIOR parentid = childid AND rolenetid = vrolenetid;
      -- Don't worry about checking the result of the previous
      -- SQL statement as it will not get to the next statement
      -- if it failed.
      -- Check whether any of the records in the role network
```

```
      -- are conflicting with the child we wish to add. We only
      -- need to check the children in the role network.
      SELECT count(1) INTO v_conf FROM DUAL
      WHERE vchildid IN
      (SELECT roleid FROM roles WHERE roleid IN
        (SELECT roleid2 FROM cr WHERE roleid1 IN
          (SELECT childid FROM rh WHERE rolenetid = vrolenetid)
         UNION
         SELECT roleid1 FROM cr WHERE roleid2 IN
           (SELECT childid FROM rh WHERE rolenetid = vrolenetid)));
      IF (v_conf > 0) THEN
        raise_application_error(-20215,'Cannot insert role into role ' ||
                                       'network - conflicting role(s) ' ||
                                       'are already present in the role ' ||
                                       'network.');
      END IF;
    ELSE
      raise_application_error(-20213,'Cannot insert role into role ' ||
                                     'network - the parent role is not an ' ||
                                     'existing child role.');
    END IF;
  END LOOP;
EXCEPTION
  WHEN e_rolenet_loop THEN
    raise_application_error(-20214,'Cannot insert role into role ' ||
                                   'network - it will cause a circular ' ||
                                   'reference to occur.');
END;
/

--------------------------------------------------------------------------------
--
-- Triggers for ensuring the integrity when deleting role network associations.
--
--------------------------------------------------------------------------------
CREATE OR REPLACE TRIGGER drhstatbef_trig
BEFORE DELETE ON rh
BEGIN
  -- Empty the main PL/SQL table buffer.
  wf_mutvars_pkg.drhvalues := wf_mutvars_pkg.drhempty;
END;
/

CREATE OR REPLACE TRIGGER drhrowbef_trig
BEFORE DELETE ON rh
FOR EACH ROW
DECLARE
  i NUMBER := wf_mutvars_pkg.drhvalues.COUNT + 1;
BEGIN
  -- Copy the row's values across to the PL/SQL table.
  wf_mutvars_pkg.drhvalues (i).rolenetid := :old.rolenetid;
  wf_mutvars_pkg.drhvalues (i).parentid := :old.parentid;
  wf_mutvars_pkg.drhvalues (i).childid := :old.childid;
END;
/

CREATE OR REPLACE TRIGGER drhstatafter_trig
AFTER DELETE ON rh
DECLARE
  vrolenetid rh.rolenetid%TYPE;
  vparentid rh.parentid%TYPE;
  vchildid rh.childid%TYPE;
  v_child NUMBER(38) := 0;
BEGIN
  FOR i IN 1..wf_mutvars_pkg.drhvalues.COUNT
  LOOP
    vrolenetid := wf_mutvars_pkg.drhvalues (i).rolenetid;
    vparentid := wf_mutvars_pkg.drhvalues (i).parentid;
    vchildid := wf_mutvars_pkg.drhvalues (i).childid;
    -- Does it have children?
    SELECT COUNT(parentid) INTO v_child
    FROM rh
    WHERE parentid = vchildid AND rolenetid = vrolenetid;
    IF (v_child > 0) THEN --It has children so prevent it from been deleted.
      raise_application_error(-20222,'Cannot delete role association - ' ||
                                     'children associations already ' ||
```

```
                                    'exist for the role been deleted.');
    END IF;
  END LOOP;
END;
/

--------------------------------------------------------------------------------
--
-- End of SoDA integrity constraint rules.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- SoDA Prototype Database Tables
--
--------------------------------------------------------------------------------

--Drop all constraints

-- FK's for ua
ALTER TABLE ua
      DROP CONSTRAINT fk_userroleid;

ALTER TABLE ua
      DROP CONSTRAINT fk_useruserid;

-- FK's for pa
ALTER TABLE pa
      DROP CONSTRAINT fk_permroleid;

ALTER TABLE pa
      DROP CONSTRAINT fk_permpermid;

-- FK's for ta
ALTER TABLE ta
      DROP CONSTRAINT fk_taskroleid;

ALTER TABLE ta
      DROP CONSTRAINT fk_tasktaskid;

-- FK's for cu
ALTER TABLE cu
      DROP CONSTRAINT fk_user1userid;

ALTER TABLE cu
      DROP CONSTRAINT fk_user2userid;

-- FK's for cr
ALTER TABLE cr
      DROP CONSTRAINT fk_role1roleid;

ALTER TABLE cr
      DROP CONSTRAINT fk_role2roleid;

-- FK's for cp
ALTER TABLE cp
      DROP CONSTRAINT fk_perm1permid;

ALTER TABLE cp
      DROP CONSTRAINT fk_perm2permid;

-- FK's for ct
ALTER TABLE ct
      DROP CONSTRAINT fk_task1taskid;

ALTER TABLE ct
      DROP CONSTRAINT fk_task2taskid;

-- FK's for rh
ALTER TABLE rh
      DROP CONSTRAINT fk_rolenetrolenetid;

ALTER TABLE rh
      DROP CONSTRAINT fk_parentroleid;

ALTER TABLE rh
```

```
      DROP CONSTRAINT fk_childroleid;


Drop Table tasks;
Drop Table roles;
Drop Table users;
Drop Table perms;
Drop Table rolenet;
Drop Table rh;
Drop Table cu;
Drop Table cr;
Drop Table cp;
Drop Table ct;
Drop Table ua;
Drop Table pa;
Drop Table ta;
--End of Dropping SoDATables

CREATE TABLE ta
(      roleid       Number(38),
       taskid       Number(38),
       CONSTRAINT pk_ta PRIMARY KEY (roleid, taskid)
);

CREATE TABLE pa
(      roleid       Number(38),
       permid       Number(38),
       CONSTRAINT pk_pa PRIMARY KEY (roleid, permid)
);

CREATE TABLE ua
(      roleid       Number(38),
       userid       Number(38),
       CONSTRAINT pk_ua PRIMARY KEY (roleid, userid)
);

CREATE TABLE ct
(      taskid1      Number(38),
       taskid2      Number(38),
       CONSTRAINT pk_ct PRIMARY KEY (taskid1, taskid2)
);

CREATE TABLE cp
(      permid1      Number(38),
       permid2      Number(38),
       CONSTRAINT pk_cp PRIMARY KEY (permid1, permid2)
);

CREATE TABLE cu
(      userid1      Number(38),
       userid2      Number(38),
       CONSTRAINT pk_cu PRIMARY KEY (userid1, userid2)
);

CREATE TABLE cr
(      roleid1      Number(38),
       roleid2      Number(38),
       CONSTRAINT pk_cr PRIMARY KEY (roleid1, roleid2)
);

CREATE TABLE rh
(      rolenetid    Number(38) NOT NULL,
       parentid     Number(38),
       childid      Number(38) NOT NULL,
       CONSTRAINT pk_rh UNIQUE (rolenetid, parentid, childid)
-- Caused bad things:
-- CONSTRAINT pk_rh PRIMARY KEY (rolenetid, parentid, childid)
);

CREATE TABLE rolenet
(      rolenetid    Number(38),
       description  Varchar2(128),
       CONSTRAINT pk_rolenet PRIMARY KEY (rolenetid)
);

CREATE TABLE perms
```

```
(      permid       Number(38),
       description  Varchar2(128),
       CONSTRAINT pk_perms PRIMARY KEY (permid)
);

CREATE TABLE users
(      userid       Number(38),
       description  Varchar2(128),
       CONSTRAINT pk_users PRIMARY KEY (userid)
);

CREATE TABLE roles
(      roleid       Number(38),
       description  Varchar2(128),
       CONSTRAINT pk_roles PRIMARY KEY (roleid)
);

CREATE TABLE tasks
(      taskid       Number(38),
       description  Varchar2(128),
       CONSTRAINT pk_tasks PRIMARY KEY (taskid)
);

-- End of create tables

--Beginning of Foreign Key constraints

-- FK's for ua
ALTER TABLE ua
      ADD CONSTRAINT fk_userroleid FOREIGN KEY (roleid)
          REFERENCES roles(roleid);

ALTER TABLE ua
      ADD CONSTRAINT fk_useruserid FOREIGN KEY (userid)
          REFERENCES users(userid);

-- FK's for pa
ALTER TABLE pa
      ADD CONSTRAINT fk_permroleid FOREIGN KEY (roleid)
          REFERENCES roles(roleid);

ALTER TABLE pa
      ADD CONSTRAINT fk_permpermid FOREIGN KEY (permid)
          REFERENCES perms(permid);

-- FK's for ta
ALTER TABLE ta
      ADD CONSTRAINT fk_taskroleid FOREIGN KEY (roleid)
          REFERENCES roles(roleid);

ALTER TABLE ta
      ADD CONSTRAINT fk_tasktaskid FOREIGN KEY (taskid)
          REFERENCES tasks(taskid);

-- FK's for cu
ALTER TABLE cu
      ADD CONSTRAINT fk_user1userid FOREIGN KEY (userid1)
          REFERENCES users(userid)
          ON DELETE CASCADE;

ALTER TABLE cu
      ADD CONSTRAINT fk_user2userid FOREIGN KEY (userid2)
          REFERENCES users(userid)
          ON DELETE CASCADE;

-- FK's for cr
ALTER TABLE cr
      ADD CONSTRAINT fk_role1roleid FOREIGN KEY (roleid1)
          REFERENCES roles(roleid)
          ON DELETE CASCADE;

ALTER TABLE cr
      ADD CONSTRAINT fk_role2roleid FOREIGN KEY (roleid2)
          REFERENCES roles(roleid)
          ON DELETE CASCADE;
```

```
-- FK's for cp
ALTER TABLE cp
      ADD CONSTRAINT fk_perm1permid FOREIGN KEY (permid1)
          REFERENCES perms(permid)
          ON DELETE CASCADE;

ALTER TABLE cp
      ADD CONSTRAINT fk_perm2permid FOREIGN KEY (permid2)
          REFERENCES perms(permid)
          ON DELETE CASCADE;

-- FK's for ct
ALTER TABLE ct
      ADD CONSTRAINT fk_task1taskid FOREIGN KEY (taskid1)
          REFERENCES tasks(taskid)
          ON DELETE CASCADE;

ALTER TABLE ct
      ADD CONSTRAINT fk_task2taskid FOREIGN KEY (taskid2)
          REFERENCES tasks(taskid)
          ON DELETE CASCADE;

-- FK's for rh
ALTER TABLE rh
      ADD CONSTRAINT fk_rolenetrolenetid FOREIGN KEY (rolenetid)
          REFERENCES rolenet(rolenetid);

ALTER TABLE rh
      ADD CONSTRAINT fk_parentroleid FOREIGN KEY (parentid)
          REFERENCES roles(roleid);

ALTER TABLE rh
      ADD CONSTRAINT fk_childroleid FOREIGN KEY (childid)
          REFERENCES roles(roleid);

-- End of Foreign Key constraints

-- Beginning of Sequence drop and create

Drop Sequence seq_rolenet;
Drop Sequence seq_perms;
Drop Sequence seq_users;
Drop Sequence seq_roles;
Drop Sequence seq_tasks;

CREATE SEQUENCE seq_rolenet;
CREATE SEQUENCE seq_perms;
CREATE SEQUENCE seq_users;
CREATE SEQUENCE seq_roles;
CREATE SEQUENCE seq_tasks;

commit -- End of Create Sequences
/
```